

MECHANIZED SUPPORT FOR RETRENCHMENT

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2008

By
Simon Fraser
School of Computer Science

Contents

Abstract	19
Declaration	21
Copyright	23
The Author	25
Acknowledgements	27
1 Introduction	29
1.1 An Overview of the Motivation and Goals	30
1.2 An Introduction to this Thesis	34
1.2.1 Thesis Structure	34
1.2.2 Notations and Conventions Used in this Thesis	38
1.2.3 A Note on Changes to Z's International Standard	38
2 Refinement and Retrenchment	39
2.1 Introduction	40
2.2 Refinement	41
2.2.1 Simulation	42
2.2.2 Operations with Inputs and Outputs	47
2.3 Refinement in Z	47
2.4 Limitations of Refinement	60
2.5 Retrenchment	65
2.5.1 Output Retrenchment	69

2.5.2	Multiple Retrenchment	69
2.5.3	Other Forms of Retrenchment	71
2.5.4	Other Techniques to Solve Refinement's Limitations	72
2.6	Retrenchment in Z	74
2.7	Concluding Remarks	81
3	Mechanical Support	83
3.1	Introduction	84
3.1.1	Specification and Syntax Checking	87
3.1.2	Type Checking	89
3.1.3	Animation – Validation	90
3.1.4	Model Checking – Verification	91
3.1.5	Deductive Reasoning (Proof) – Verification	92
3.1.6	Test Case Generation	93
3.2	Options for Retrenchment	94
3.2.1	Extending an Existing Toolkit	94
3.2.2	Theorem Provers	108
3.2.3	Creating a New Toolkit	112
3.3	Evaluating the Options	115
3.4	Frog: A New Toolkit to Support Retrenchment	117
3.5	Concluding Remarks	121
4	Parsing The Z Notation	123
4.1	Introduction	124
4.2	Existing Z Tools	125
4.2.1	CADiZ	125
4.2.2	ZTC/ZANS	126
4.2.3	<i>fuzz</i>	126
4.2.4	Z/EVES	127
4.2.5	CZT (Common Z Tools)	127
4.2.6	Summary	128
4.3	An Overview of Parsing a Z Specification	128
4.4	Handling Special Sections	132

4.4.1	The Prelude Section	132
4.4.2	The Mathematical Toolkit	133
4.5	Ordering	135
4.6	Syntax Checking	136
4.6.1	A Z Notation	136
4.6.2	ANTLR	138
4.6.3	Lexing	142
4.6.4	Parsing and Syntax Checking	154
4.6.5	Syntax Transformation	170
4.6.6	Type Checking	190
4.7	Semantic Translation	220
4.8	Evaluation: A Comparison of the Frog Parser and CZT	220
4.9	Concluding Remarks	222
5	Modelling in the Z Notation	225
5.1	Introduction	226
5.2	Frog-CCL	229
5.2.1	Syntax for Frog-CCL	230
5.2.2	Generating Proof Obligations	237
5.3	Incorporating Constructs	247
5.3.1	Extending the Z Syntax	247
5.3.2	Parsing the Extended Grammar	250
5.3.3	Syntax Transformation for the Extended Grammar	252
5.4	Configuring and Specifying a Machine	257
5.5	Configuring and Specifying a Refinement	272
5.6	Configuring and Specifying a Retrenchment	281
5.7	Machine Inclusion	293
5.8	Concluding Remarks	298
6	Generating Proof Obligations	299
6.1	Introduction	300
6.2	Producing a Semantic Model	303
6.2.1	The ZF Language	304

6.2.2	Semantic Bindings	312
6.3	Instantiation of Generic Proof Obligations	369
6.4	Theorem Provers	386
6.4.1	Otter	388
6.4.2	Vampire	389
6.4.3	SPASS	389
6.4.4	PVS	390
6.4.5	KIV	391
6.4.6	Coq/LogiCal	391
6.4.7	HOL	392
6.4.8	Isabelle/ZF	393
6.4.9	Z-Specific Theorem Provers	394
6.4.10	Summary	395
6.4.11	Evaluating our Choice	396
6.5	Discharging Proof Obligations	397
6.5.1	Translation	397
6.5.2	Proving in Isabelle/ZF	401
6.6	Concluding Remarks	414
7	Frog	415
7.1	Introduction	416
7.2	Architecture	417
7.3	The User View	421
7.3.1	Configuration	421
7.3.2	The Development Process	423
7.3.3	The Proof Process	429
7.3.4	Altering and Removing Files: Maintaining Integrity . .	433
7.3.5	Other Tools	436
7.4	Implementation	439
7.4.1	GUI and Construct Management	443
7.4.2	Configuration	447
7.4.3	Parsing and Syntax Checking	449
7.4.4	Generating Proof Obligations	454

7.4.5	Interfacing with Theorem Provers	456
7.4.6	Error Handling	458
7.5	Concluding Remarks	460
8	Case Study	461
8.1	Introduction	462
8.2	Types and Operators	464
8.3	POTS	467
8.4	Introducing the Holding Service	470
8.4.1	Specifying the Extended Machine	470
8.4.2	Specifying the Relationship	474
8.4.3	Syntax Checking the Relationship	476
8.4.4	Verifying the Relationship	487
8.5	Introducing The Forwarding Service	499
8.5.1	Specifying the Extended Machine	499
8.5.2	Specifying the Relationship	503
8.5.3	Syntax Checking the Relationship	507
8.5.4	Verifying the Relationship	508
8.6	Integration	512
8.6.1	Specifying the Extended Machine	513
8.6.2	Specifying the Relationship: POTS with Holding Ser- vice to Integrated Machine	517
8.6.3	Syntax Checking the Relationship	521
8.6.4	Specifying the Relationship: POTS with Forwarding Service to Integrated Machine	528
8.6.5	Syntax Checking the Relationship	531
8.6.6	Verifying the Relationship	532
8.6.7	Specifying the Relationship: POTS to Integrated Ma- chine	539
8.6.8	Syntax Checking the Relationship	542
8.6.9	Verifying the Relationship	543
8.7	Concluding Remarks	548

9	Conclusions	549
9.1	Review	550
9.2	Evaluation, Conclusions and Future Research	553
9.2.1	Evaluating the Research	554
9.2.2	Conclusions and Opportunities for Further Research . .	556
A	A Complete Example	569
B	Formatting The Z Notation	583
B.1	Paragraphs	583
B.1.1	Given Types	583
B.1.2	Axiomatic Description	583
B.1.3	Schema Definition	584
B.1.4	Horizontal Definition	584
B.1.5	Generic Operator Definition	584
B.1.6	Free Type	585
B.1.7	Conjecture	585
B.1.8	Operator Template	585
B.2	Schema Text	585
B.3	Expressions	586
B.3.1	Conditional	586
B.3.2	Schema Composition	586
B.3.3	Schema Piping	587
B.3.4	Schema Hiding	587
B.3.5	Cartesian Product	587
B.3.6	Schema Projection	587
B.3.7	Schema Precondition	587
B.3.8	Binding Selection	588
B.3.9	Tuple Selection	588
B.3.10	Binding Construction	588
B.3.11	Schema Negation	588
B.3.12	Schema Conjunction	588
B.3.13	Schema Disjunction	589

B.3.14	Schema Equivalence	589
B.3.15	Schema Implication	589
B.3.16	Schema Existential Quantification	589
B.3.17	Schema Universal Quantification	590
B.3.18	Function Construction	590
B.3.19	Substitution	591
B.3.20	Definite Description	591
B.3.21	Characteristic Definite Description	592
B.3.22	Set Extension	592
B.3.23	Set Comprehension	592
B.3.24	Characteristic Set Comprehension	593
B.3.25	Tuple Extension	593
B.4	Predicates	594
B.4.1	Literals	594
B.4.2	Membership	594
B.4.3	Equality	594
B.4.4	Negation	594
B.4.5	Conjunction	594
B.4.6	Disjunction	595
B.4.7	Equivalence	595
B.4.8	Implication	595
B.4.9	Existential Quantification	595
B.4.10	Universal Quantification	596
B.5	The Mathematical Toolkits	597
B.5.1	Prelude	597
B.5.2	Set Toolkit	597
B.5.3	Relation Toolkit	598
B.5.4	Function Toolkit	599
B.5.5	Number Toolkit	599
B.5.6	Sequence Toolkit	600
B.5.7	Standard Toolkit	601

C	Z Operators	603
	C.1 Concatenation	603
	C.2 Domain	603
	C.3 Domain Subtraction	604
	C.4 Extraction	604
	C.5 Finite Sequence	604
	C.6 Generalized Intersection	605
	C.7 Identity	605
	C.8 Injective Sequence	605
	C.9 Partial Injection	606
	C.10 Power Set	606
	C.11 Range	606
	C.12 Range Subtraction	607
	C.13 Relation	607
	C.14 Relational Override	607
	C.15 Set Intersection	608
	C.16 Set Difference	608
	C.17 Set Union	608
	C.18 Total Function	609
	C.19 Transitive Closure	609
D	Axioms and Lemmas	611
	D.1 Existential Quantification One Point Rule	611
	D.2 Law of Excluded Middle	611
	D.3 Laws of Simplification	611
	D.4 Set Comprehension One Point Rule	612
	D.5 Universal Quantification One Point Rule	612
E	Annotated Lexical Grammar	613
F	Annotated Parsing Grammar	625
G	Tree Transformation Rules	645
	G.1 Paragraph Transformation Rules	645

G.2 Predicate Transformation Rules	648
G.3 Expression Transformation Rules	654
G.4 Schema Text Transformation Rules	665
G.5 Operator Name Transformation Rules	668
G.6 Generic Name Transformation Rules	670
H Isabelle/ZF Proof Obligation	673
Bibliography	685

List of Tables

2.1	A summary of Z strokes	49
4.1	ANTLR's Metalanguage	139
4.2	User defined token types	148
4.3	Token streams that can require precedence resolution	161
4.4	Definition of Z Type Classes	193
4.5	Generic instantiation rules	206
4.6	Rules for determining implicit parameters	210
5.1	Logical rules for handling undefined clauses	246
6.1	ZF Language Classes	305
6.2	Summary of functions used in the creation of semantic bindings and their environments	316
6.3	Examples of differences between Isabelle/ZF syntax and Z syntax	398
6.4	Tactics of Isabelle/ZF	402
8.1	Possible status reports when connecting a call	466
8.2	Behaviour of connect operation in <i>POTSPlusHoldAndForward</i>	518
E.1	The Literals Symbol Table	622

List of Figures

2.1	Stepwise refinement	42
2.2	Forward simulation	45
2.3	Retrenchment's role	66
2.4	Transitions in Refinement and Retrenchment	68
2.5	Multiple Retrenchment	71
3.1	Standard B-Toolkit Development Structure	96
3.2	Extended B-Toolkit Development Structure	97
4.1	Interaction with existing Z parser	129
4.2	Contents of a Frog specification	130
4.3	Parsing a Frog file	131
4.4	Parent relation between sections of the mathematical toolkit (from [ISO02])	134
4.5	Excerpt from sample character dictionary configuration file . .	145
4.6	Dictionary classes	146
4.7	Trees created from Z fragment	155
4.8	Trees created from matching set expressions	158
4.9	Initial parse tree for $a \cup b \cap c$	162
4.10	Precedence and associativity transformation rules	164
4.11	Resolved parse tree for $a \cup b \cap c$	165
4.12	Overview of AST for train yard specification	166
4.13	AST for <i>trainEntersYard</i>	168
4.14	Syntactic transformation rules for equivalence and implication predicates	171
4.15	Syntactic transformation rules for schema definition paragraphs	177

4.16	Syntactic transformation rules for horizontal definition paragraphs	177
4.17	Syntactic transformation rule for generic operator definition paragraph	179
4.18	Syntactic transformation rule for lambda function construction	180
4.19	Syntactic transformation rule for characteristic set comprehension	181
4.20	Syntactic transformation rule for characteristic definite description	181
4.21	Applying the transformation rules to the declaration part of a schema text	183
4.22	Syntactic transformation rule for cartesian products	184
4.23	Creating the transformed AST for <i>trainEntersYard</i> (1)	185
4.24	Creating the transformed AST for <i>trainEntersYard</i> (2)	186
4.25	Creating the transformed AST for <i>trainEntersYard</i> (3)	186
4.26	Creating the transformed AST for <i>trainEntersYard</i> (4)	187
4.27	Creating the transformed AST for <i>trainEntersYard</i> (5)	188
4.28	Final AST for <i>trainEntersYard</i>	189
4.29	Z Type Classes	192
4.30	Type Environment class	194
4.31	Type Environments for <i>mySection</i>	195
4.32	Typing transformation rule for axiomatic description paragraph	198
4.33	Example of two schema type trees prior to conjunction	200
4.34	Example of schema type tree following conjunction	201
4.35	Typing transformation rule for schema conjunction expression	201
4.36	Typing transformation rule for membership predicate	203
4.37	Type tree for relation generic operator	205
4.38	Type trees required for implicit instantiation of emptyset	208
4.39	Typed AST for <i>trainEntersYard</i> (1)	216
4.40	Typed AST for <i>trainEntersYard</i> (2)	217
4.41	Typed AST for <i>trainEntersYard</i> (3)	218
5.1	Contents of a Frog Specification with Constructs	228

5.2	Configuration of a machine initialization clause	232
5.3	Configuration of an operation environment for machine operations	234
5.4	Configuration of proof obligation for machine initialization . .	236
6.1	Proving a Construct	301
6.2	Example of ZFObject tree for universal quantification	320
6.3	Tree for Initialization Proof Obligation of Machine Configuration	370
6.4	Tree for Operation Proof Obligation of Machine Configuration	379
7.1	Overview of Frog Architecture	418
7.2	The construct configuration browser	423
7.3	Editing files in a specification	425
7.4	Viewing the ASTs belonging to a file	427
7.5	The HTML representation of paragraphs in the Z object view	428
7.6	The HTML representation of constructs in the construct view	429
7.7	Proof obligations	430
7.8	Isabelle/ZF proof options	431
7.9	Tactic file editor	432
7.10	Results of proof process	434
7.11	Dependency Browser	435
7.12	HTML Definition of <i>trainYard</i>	438
7.13	Help System	439
7.14	Object View for <i>mySection</i>	453
8.1	Overview of telephone system requirements	462
8.2	Overview of telephone system constructs	465

Abstract

Refinement is a long-established technique that is widely used in the rigorous development of software. It can be argued that refinement has limitations that prevent it being used effectively in a wide range of system implementations. These claims led to the introduction of a liberalized form of refinement known as retrenchment. Whilst, when using retrenchment, we lose some of refinement's guarantees, we are able to describe the construction of specifications in situations where refinement struggles to provide a clear and concise picture. It is hoped therefore, that the use of retrenchment – alongside refinement – will increase the scope of system developments to which formal methods can be successfully applied.

It has been generally recognized that it is not feasible to apply formal methods to the development of complex systems without suitable tool support. Following an attempt to integrate retrenchment into the B-Toolkit – where the inflexibility of the application made change difficult – we decided to create a new tool that was capable not only of supporting the specification and proof of refinement and retrenchment, but any similar relationship. Our aim was to make the notion of the model and the relationship between models fully configurable, allowing the user to specify and prove with existing formal techniques, but also to be able to experiment in the creation of new techniques.

We chose to use the Z notation as the principal syntax for expressing our models and relationships. As the international standard for Z had only recently been published, we were also required to create one of the first Z tools that conformed (loosely) to this standard.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of the School of Computer Science.

The Author

Simon Fraser gained his B.Sc. (Hons) and M.Eng. degree in Computer Science from the University of Manchester in 1999. Upon completion of his undergraduate degree he joined Lucent Technologies, where he worked for three years as a consultant in software development and deployment. In 2002 he joined the Retrenchment Group at the University of Manchester, spending a year seeking to integrate retrenchment with the B-Toolkit and earning an M.Phil degree. Following the completion of this project he began further work with retrenchment in the pursuit of a Ph.D. degree. The result of that work is described in this thesis.

Acknowledgements

Principally, I would like to thank Richard Banach, my supervisor, whose guidance has made this thesis possible. Thanks also go to everybody within the Retrenchment group, and my adviser, David Lester.

Outside of academia I would like to express my gratitude to my parents without whose long-standing support and love I would never have reached this point. I would also like to express my thanks to all my friends and family who have offered care and assistance throughout this challenging period of my life.

Chapter 1

Introduction

This chapter provides an introduction to the thesis, and begins to describe the research which we have undertaken. We begin by providing the background to the research, and outline the motivations for our work. We proceed to examine the goals of our research, and indicate the objectives we hoped to achieve. We then describe the structure of this document, providing a synopsis of each chapter. Finally, we detail any convention or notations upon which the remainder of the document relies.

1.1 An Overview of the Motivation and Goals

As every year passes, the reliance of humanity on computers and the software that operates those computers, increases. As this dependence expands, one would expect that software developers would seek ways to ensure that their systems were proportionally more reliable; yet a glance at the headlines can persuade even the most cynical observer that this is not the case. Even in areas in which one would expect the utmost care to be taken – such as space exploration¹ [Lio96], missile development² [Inf92, Inf02] and health³ [LT93] – bugs continue to haunt software developments. It has long been recognized that the traditional – almost trial and error – approach to developing software is a haphazard and bug prone process, and that a more rigorous approach is necessary [Abr83, Dij76, Jon80].

In response to this need a collection of mathematically based techniques – known collectively as formal methods – have been created for the specification, development and verification of software (and hardware) systems. The use of formal methods can eliminate bugs in an application (see [BDM98, Des98], where not a single error was found in over 87000 lines of train control software), and the focus of testing shifts from testing an application for bugs, to testing that we have the correct specification. Although the use of formal methods for the specification of complex software systems is becoming increasingly common, their use remains uncommon outside the most critical of developments, and the majority of applications continue to involve the,

¹The Ariane 5 software was written using specifications from a previous mission. Unfortunately, the flight path for this mission was significantly different, and a data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception. Pre-flight tests were never performed on the software in flight conditions simulated for Ariane 5, so the error was not discovered before launch. The rocket was destroyed after just 37 seconds following the start of the main engine.

²In 1991 an Iraqi missile hit the United States Army's barracks in Dharaan, Saudi Arabia, killing 28 soldiers. Investigations revealed that the missile was not intercepted due to a software failure which led to a Patriot missile's system clock drifting by one third of a second (causing the missile to miss the intercept by 600 metres). Far from learning from their mistakes, a similar error almost led to equally disastrous circumstances in the United States' national missile defence system nearly ten years later.

³A software system was created to deliver doses of radiation to cancer sufferers. However, a bug led to those doses being far too high, eventually resulting in the death of patients.

‘design, code, and test’ approach. Even in critical situations undependable software is avoided through – and reliance is placed on – engineered hardware. There exists therefore a need to create techniques that can extend the scope of developments in which formal methods can be used and therefore increase the accessibility of the rigorous approach.

A technique that has found particular success in the world of formal methods is refinement [BvW98, dRE98, Wir71]; this technique involves the stepwise integration of detail into an abstract specification (each model being considered a refinement of the last), until we reach a model of our system that can be readily converted into programming language instructions. Refinement’s key property is that all models in the chain – from abstract to concrete – can be formally related so that the behaviour of each model fulfils the requirements of the most abstract. Therefore, if the most abstract model satisfies our specification, the most concrete model must also. There are circumstances, however, where refinement can require us to include unnecessary complexity at the abstract level. Consider an operation performing a simple addition. In an abstract model, one member of the infinite set of natural numbers is added to another, and the sum is returned. Upon consideration of a more concrete model of this operation, it becomes clear that there is typically a limit on the size of a natural number in a computer system, and therefore, there is a need to model some way of representing failure if this size is exceeded. Clearly, these two operations are not behaving identically and in order to make one a refinement of the other, the abstract model must be extended to model the maximum number and record the success or failure of the operation. The abstract model for the operation would become significantly more complex. In a realistic development, these increases would be exponential. Furthermore, it is surely one of refinement’s principal advantages that implementation detail is added gradually through the development process. If we need to include all of the implementation detail in the most abstract model, what is the point in refining it at all?

It is possible – through the manipulation of the refinement technique – to effectively hide implementation requirements (such as the limitation on natural numbers above). However, consideration must be given to the point at

which we draw the line between a system requirement and an implementation requirement. A requirement of many software systems is that they are able to run on a particular piece of hardware, or alongside another piece of software. What we really need is two specifications: one with, and one without, the implementation detail. To include both of these specifications within a rigorous development requires a technique for validating their relationship, and we have already discovered that refinement is not such a technique.

These limitations of refinement made it essential that a more flexible technique of relating abstract and concrete model was proposed. Retrenchment [BP98, BPJS07] is such a technique, and can be seen as a robust, but more liberal version of refinement. A retrenchment step is not an oblique transition from an abstract model to a more concrete version of that same model, but a transparent definition of their relationship. This relationship is viewed alongside the two models, showing exactly how they interact (that is, where their behaviour is consistent, and also where it is not). Retrenchment not only allows the resolution of the issues described above, but allows existing abstract specifications – which provide only the high-level stipulations that results from the system’s behavioural requirements – to be used by those only needing an overview of the application’s behaviour. Meanwhile more concrete specifications – incorporating the implementation requirements – can be used by developers, whilst users of both can be safe in the knowledge that they are working to a specification that meets the system requirements.

Unlike refinement, the purpose of retrenchment is not to produce implementable code from a specification (refinement is already excellent in performing this task), but is to produce an implementable specification from a user’s more abstract requirements. Once such a specification has been produced, techniques – such as refinement – will be used to produce code in the normal manner. It can be clearly seen that retrenchment extends the circumstances in which refinement (or other formal techniques) can be used to formally produce software; retrenchment allows those models outside the scope of refinement to be formally related to those within it.

Detractors of formal methods frequently refer to the effort required in creating specifications, and proving their validity, as a major flaw of rigorous

techniques, citing the necessity to employ specialists who are able to understand the minutiae of their application as a significant cost. For the use of any formal technique to become widespread, therefore, it is essential that tools are developed in order to aid understanding, reduce the effort required to check and prove a specification, and generally assist a software engineer in the use of the technique.

The aim of the research presented in this thesis was to investigate the ways in which retrenchment could be supported mechanically, and examine what tools would be useful to a practitioner (or a researcher) of retrenchment. We have previously investigated the possibility of integrating retrenchment into existing formal methods toolkits [BF05, Fra05], particularly the B-Toolkit [Abr91, LH96, Wor96], but have found that most existing tools are designed to support a particular technique and prove to be quite resistant to the introduction of others. After much deliberation we felt it was wisest to recommence our work by creating a new toolkit which was capable of supporting not just existing techniques like refinement and retrenchment, but that was configurable and extendable. This was to be done in such a way that not only could new techniques be supported, but that allowed for the experimentation in the creation of new techniques. We sought, therefore, to produce a system where the shape of the models in a specification, and the relationships between those models, were fully manipulable.

Whilst we considered the possibility of continuing to work with the notation of the B-Method [Abr96, Sch01] (the notation used by the B-Toolkit), we decided to opt for the greater flexibility afforded by the Z notation [Cur99, Spi92b, WD96]. The creation of a new international standard for this notation [ISO02] provided us with a further challenge; the only parsers that were able to process the notation according to the rules in the standard did not provide access to their source code. In order to gain full control over the parsing process we were put into the position of having to create one of the first standard-conformant tools.

We could therefore, summarize our objectives as follows. We wished to create a tool that was able to parse the Z notation in a way that conformed to the international standard. We wished to use the Z notation in the creation

of models and relationships where we were freely able to experiment with the nature of those models and particularly with the relationships between them (with a slight emphasis on showing that it was possible to provide effective tool support for the retrenchment technique). We wished to be able to create proof obligations for our models and relationship in such a way that – not only were they configurable – but they could easily be translated into a format suitable for passing to a number of theorem provers. Finally, we wanted to show that the tool we created was capable of all of the above, through the reexamination of the case study presented in [BP02] (a case study which we had attempted to mechanize with the tool described in [BF05, Fra05]). The tool we created was to be known as Frog.

The aim of this thesis is to document the creation of Frog and the success or failure of that tool in meeting the objectives above. More importantly however, this thesis must record the knowledge that has been gained in this process of development, providing aid for any future attempt to provide automated assistance to the retrenchment process.

1.2 An Introduction to this Thesis

This section provides an overview of this document’s structure. We describe the content of each chapter, and then proceed to outline conventions and notes that should be considered whilst reading this thesis.

1.2.1 Thesis Structure

The current chapter provides an introduction to the research that we have undertaken and details the structure, notations and conventions that we will use within this document.

Chapter 2 begins by introducing the concept of model-oriented formal specification, and we define what we mean by a model. We then proceed to describe the advantages of creating a formal relationship between an abstract model and implementable code. Stepwise refinement is then introduced as a technique which can assist in the specification of such a relationship. We

then describe how simulation (particularly forward simulation) can be used to prove the validity of a refinement relationship. The Z notation is then used to present a simple example of refinement working well, but we then present a second example where some of its limitations are exposed. We then acquaint the reader with the retrenchment technique and show how it can be used in situations where refinement is not particularly suitable. We discuss a number of different forms of retrenchment and briefly examine some alternate solutions. Finally, we re-examine the example where we were unable to use refinement previously, and show how retrenchment can describe a more appropriate relationship.

Chapter 3 begins by describing some of the issues that are considered to be preventing the uptake of formal methods and highlights some of the main reasons that its detractors give for this. We then discuss how tool support has led to successful industrial application and describe the ways in which mechanical assistance can aid a formal development. Following this we begin to discuss the ways in which we can provide mechanized support for retrenchment. We present various alternatives including extending existing toolkits, embedding within a theorem prover, and creating a new tool set. Our options are evaluated, and we present our conclusions, outlining our proposal for supporting retrenchment mechanically.

Chapter 4 begins with an introduction to Z , briefly describing its history – from its inception to the recent creation of an international standard. We give a short introduction to some of Z 's features and the way in which its documents are structured. A number of existing tools that provide support for Z are then described, and we discuss why we felt it was necessary to create our own parser. We then give an overview of the syntax checking process – describing in detail each of the major phases (lexing, parsing, syntax transformation and type checking) – and provide a running example showing how we transform a \LaTeX specification into a typed abstract syntax tree. We then provide a short comparison between the parser we have generated and another tool, CZT, which has also been developed from the international standard and in parallel with ours.

Chapter 5 begins by introducing the concept of a construct and explains

how constructs can be used to encapsulate a model, or the relationship between two models. We define the terms machine and relationship – providing simple examples – and show how they can be integrated within a Z specification. We then detail Frog-CCL, a language in which we can configure exactly what we mean by a machine or relationship and illustrate how the clauses and proof obligations of a construct can be manipulated through configuration. The extension to the Z grammar that is required to parse these configured constructs is presented, and we expound how the augmented grammar can be processed with minimal changes to the existing Z syntax checker. Finally, we present sample configurations (which become the defaults in this thesis) for a machine, and both refinement and retrenchment relationships.

In chapter 6 we consider the automated generation of proof obligations for constructs and the ways in which we can attempt to discharge those obligations. We begin by outlining the proof process and explain how a semantic model needs to be created for a construct before proof obligations can be created. We proceed to describe the generation of such a semantic model; introducing a language based on Zermelo-Fraenkel set theory (that we use as an intermediary between the Z notation and the syntax of individual theorem provers) and describing the translation process. We introduce the concept of a construct’s semantic binding environment, and show how the information it contains can be used to instantiate the generic proof obligations specified in that construct’s configuration. We present a review of the theorem provers whose use we considered to discharge our proof obligations, and present our reasons for choosing Isabelle/ZF. We then show how we can translate our generic proof obligations into a format suitable for a specific theorem prover, and show how it is possible to discharge them using a combination of the automated and interactive tools provided.

Chapter 7 describes the development of the Frog tool and brings together the subjects discussed in the preceding chapters. The chapter starts with an introduction to the design philosophies and the requirements that we wished to incorporate in the implementation process. We proceed to present Frog’s architecture, detailing how the major components of the application interact. We then examine the user’s view of the tool and show how the tool works

from a practical perspective. We then provide an overview of the tool's implementation, highlighting design decisions of importance, but principally just to give a taste of the methods we have used.

In chapter 8 we examine a simple, but non-trivial, case study. Here we examine an existing case study that involves the construction of a specification for a telephone system (we begin with a plain system, and introduce a number of features). This case study will show that Frog is able to provide support for more complex constructs – and their interaction – demonstrating support for the retrenchment relationship.

Chapter 9 draws to a close the main body of the thesis, and evaluates the work we have presented. We examine the contribution made by the research presented in this document, and evaluate our success in meeting our goals. Finally, we summarize our conclusions, and outline the details of further research which this work has suggested could be fruitful.

Appendix A describes the processing of a simple example from beginning to end, taking a construct specification and demonstrating each step Frog uses in checking that specification. The appendix culminates in showing how Frog is able to generate the required proof obligations from the specification.

Appendix B provides details of the formatting conventions we have used when presenting examples in the Z notation.

Appendix C provides definitions for Z operators that we use within this document.

Appendix D presents details of lemmas that will be used in the proofs contained within this thesis.

Appendix E provides a fuller version of the lexical grammar required by our Z parser and provides notes that explain its use.

Similarly, appendix F gives a comprehensive, annotated version of the parser grammar and describes some of the details for which there was not space to describe in chapter 4

Appendix G presents a full list of the tree transformation rules used in the syntax transformation phase of the Z parser's syntax checking process. The information in these last three appendices is provided to make clear the distinctions between the rules that we have used, and those presented in the

international standard [ISO02].

Appendix H provides an illustration of a Frog proof obligation that has been passed to, and processed by, Isabelle/ZF.

1.2.2 Notations and Conventions Used in this Thesis

We have attempted to describe all Z notation as and when it is used. Should this not prove sufficient, a comprehensive description of the way in which we have formatted the Z notation within this thesis is provided in appendix B, and we present a more detailed definition of operators on which we rely in appendix C.

Throughout this thesis, diagrams that demonstrate the behaviour or structure of the tool are presented in the UML [Alh98, FS00, FEL97].

1.2.3 A Note on Changes to Z's International Standard

When we began work on this research the final version of Z's international standard [ISO02] was not in the public domain. Much of the work on the Z parser – that we created as part of our tool – was, therefore, based on the final committee draft of that document [ISO99]. Inevitably, there were changes made to the standard between the last draft, and ISO's official version. Where possible, we have indicated where we have based a design decision on an element of the draft version that has been updated in the official version. There may, however, be other places in this thesis where the definitions we present vary slightly from those in the official version of the standard document: the reader can be assured that these definitions will be based on the draft version and do not affect functionality.

Chapter 2

Refinement and Retrenchment

This chapter describes the formal techniques of refinement and retrenchment. We begin by introducing the way in which systems can be modelled through formal specification and then proceed to describe how the established technique of refinement is used to rigorously produce implementable instructions from abstract specifications. We then show how simulation can be used to validate a refinement step between models, focusing particularly on the rules for the forward simulation method. A simple example of refinement is presented in the Z notation, where we specify an abstract and a concrete model and the relationship between them. We then describe some limitations of refinement that have restricted its uptake, presenting an example that illustrates these limitations. Retrenchment is then introduced and we describe how the use of this technique allows more systems to be developed formally, through the liberalization of refinement. We examine a number of different forms of retrenchment, and present a brief discussion of alternative solutions to refinement's limitations. Finally, we return to the example we used to demonstrate a situation where refinement was not a suitable solution and show that retrenchment can be used to considerably greater effect.

2.1 Introduction

Writing formal specifications involves the description of a system using an unambiguous language. This description of the system can be considered a model of that system¹. A model must be representative of the system it models and must not only reflect the state of the system at a given point, but also model the ways in which that state can be changed.

model^a, n.

1. A small object, usually built to scale, that represents in detail another, often larger object.

^aExcerpt of definition from the Free Dictionary at thefreedictionary.com

In a formal specification we usually define a state space (that records the state of the model at a given point), and a number of operations – which may take inputs or produce outputs – that alter or query that state space. Typically, we define a special ‘initialization’ operation that specifies the state of the model when it comes into being.

While the creation of a formal specification has considerable value, in computer science we are often looking to use that specification as a starting point for the development of a computerized representation of the system it models. As such, we seek to progressively add more information to our specification until it can be implemented directly with programming language instructions. It is important, however, that the addition of this detail does not change the meaning of our specification. Formal techniques have been devised that allows us to prove that two models – where one is more abstract, and the other more concrete – produce equivalent behaviour. One of the most popular of these techniques is known as refinement.

¹In this thesis we will consider only the model-based approach to specification. Other forms of specification include property-based, history-based and transition-based specification. A useful overview of these alternate forms is presented in [vL00].

2.2 Refinement

Stepwise refinement was first proposed in [Wir71] where it was described as a method of program development in which design decisions are incrementally added to an abstract specification until that specification can be expressed entirely in some implementation language. Refinement can be thought of as the removal of non-determinism from an abstract specification. This removal may involve the resolution of deferred design decisions (for example, how to store a piece of data), or the elimination of non-deterministic data types (for example, a set).

Since Wirth's proposition, refinement has become a by-word for formal development and has been incorporated – through simulation – into the most common specification systems, such as B [Abr96, Sch01, Wor96], VDM [AI91, Jon90, She95], and Z [BSC94, Cur99, Kin90b].

An alternate formalization is the refinement calculus [BvW98, Mor87, Mor94]. The refinement calculus provides a logical framework for program construction. The behaviour of the specification is described by an abstract, possibly non-executable, program which is manipulated through a number of correctness-preserving transformations into an equivalent, executable program. Each step involves the application of a transformation law to a program that, given any appropriate proof obligations are discharged, will produce a new program that is automatically a refinement of the first.

In this thesis our investigations principally revolve around the simulation form of refinement. From this point forth, any ambiguous reference to refinement should be considered as a reference to the simulation formalization.

The common core of all interpretations of formal stepwise refinement, is the investigation of the relationship between a pair of models; typically those adjacent in the development process (for example models \mathcal{A} and \mathcal{C} in figure 2.1² below). In this pair of models, the more concrete model will typically substitute³ some non-deterministic behaviour in the abstract model,

² \sqsubseteq is the standard notation for a refinement.

³We use of the definition of substitution presented in [DB01] where the principle of substitutivity which states that “it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken

with behaviour that can more easily be implemented on a computer system. The more concrete model will be considered to be a refinement of the abstract model if it is both ‘applicable’ and ‘correct’.

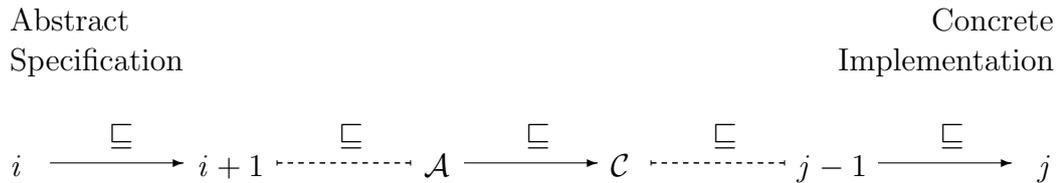


Figure 2.1: Stepwise refinement

We consider the relationship as applicable if whenever it is possible for a state transition to occur in the abstract model, it is also possible for a state transition to occur in the concrete model. We consider the relationship as correct if whenever the concrete model actually makes a move, that move can be simulated in the abstract model. We proceed now to discuss what we mean by simulation.

2.2.1 Simulation

Intuitively, a system simulates another system if when one system makes a move, the other also makes a move which, when substituted for the first, would only have differences that are undetectable (within the defined limitations of the simulation) to an observer.

simulation^a, n.

4. a. Imitation or representation, as of a potential situation or in experimental testing.
- b. Representation of the operation or features of one process or system through the use of another.

^aExcerpt of definition from the Free Dictionary at thefreedictionary.com

place”.

Simulation was first proposed as a method for the proof of refinement between state-based models by He, Hoare and Sanders in [HHS86] and augmented in [HHS87]. They propounded the use of a data type and then considered how to determine when one data type was a refinement of another. They determined that a relational approach could be used to show that one data type simulated the other (within the context of a specified ‘retrieve’ relation). If we were able to show that the models could be initialized in equivalent states and that every subsequent transition would lead to an equivalent state, we could conclude that one data type was a refinement of the other.

Woodcock and Davies [WD96] – building on the work of, He, Hoare and Sanders – describe how simulation can be used to prove correctness in the refinement of data types. For convenience, we will summarize the main points and demonstrate the proof methods we will use throughout this thesis.

In this thesis we are considering a model to comprise a state space, and an indexed set of operations⁴. We also declare that the creation of a model will initialize the state space in a specified manner. For completeness we will also declare a finalization that will return a model to the global state space.

For a given model, X , we use the quadruple (X, xi, xf, Xop) to represent the model’s state space, initialization, finalization and operations respectively.⁵ If we define the global state space as G , then we can define the following properties.

- initialization is a total relation between the global state space and our model’s state space — $xi \in G \leftrightarrow X$
- finalization is a total relation between our model’s state space and the global state space — $xf \in X \leftrightarrow G$
- our operations are indexed and collected into the set Xop , where each operation is a relation that may be either total or partial — $Xop ==$

⁴Our notion of model is thus equivalent to Woodcock and Davies’ notion of a data type.

⁵He, Hoare and Sanders require that the initialization, finalization and operations are all total. Woodcock and Davies also require initialization and finalization relation to be total, but allow partial relations to describe operations.

$$\{i : I \bullet xop_i\}$$

If we consider the state of our model over time we know that it will be initialized by xi , subsequently transformed by applications of our operations $xop_{0..n}$, and finalized by xf . We can consider the application of a sequence of operations as a program. A program begins and ends in the global state space, and as our model is encapsulated by initialization and finalization, we can consider an instance of the specific behaviour of our model over time as a program, $\mathcal{P}(X)$, where

$$\mathcal{P}(X) == xi \circ xop_0 \circ xop_1 \circ \dots \circ xop_{n-1} \circ xop_n \circ xf$$

Let us now reconsider the two models \mathcal{A} and \mathcal{C} . If two models share the same global state space, and they use the same index set for their operations then $\mathcal{P}(\mathcal{A})$ and $\mathcal{P}(\mathcal{C})$ are comparable. If for every program, \mathcal{P} , the effect of $\mathcal{P}(\mathcal{C})$ is defined whenever the effect of $\mathcal{P}(\mathcal{A})$ is defined, the behaviour of $\mathcal{P}(\mathcal{A})$ can be substituted for the behaviour of $\mathcal{P}(\mathcal{C})$, and $\mathcal{P}(\mathcal{C})$ resolves some of the non-determinism of $\mathcal{P}(\mathcal{A})$, then it is fair to consider \mathcal{C} a refinement of \mathcal{A} . We can, therefore, state the following.

$$\mathcal{A} \sqsubseteq \mathcal{C} \Leftrightarrow \mathcal{P}(\mathcal{C}) \subseteq \mathcal{P}(\mathcal{A})$$

Which is equivalent to saying the following.

$$\begin{aligned} \mathcal{A} \sqsubseteq \mathcal{C} \Leftrightarrow & ci \circ cf \subseteq ai \circ af \\ & \wedge ci \circ cop \circ cf \subseteq ai \circ aop \circ af \\ & \wedge ci \circ cop \circ cop \circ cf \subseteq ai \circ aop \circ aop \circ af \\ & \vdots \end{aligned}$$

Whilst this result gives us a method to show that one model is a refinement of another, it is clearly impractical for all but the simplest of model pairs. In order to make this method more useful, we break the process down. We know that the models have the same indexing set so it is possible to compare the model's programs on a step-by-step basis. To do this we define a relation between the state spaces of the abstract and concrete models. This relation

is called the retrieve (or simulation) relation and (assuming that the relation has type $\mathcal{A} \leftrightarrow \mathcal{C}$ and we are dealing with forward simulation⁶),) reduces the requirements to show that $\mathcal{A} \sqsubseteq \mathcal{C}$, to the following three requirements (this can also be seen in figure 2.2).

- $\mathbf{c}i \subseteq \mathbf{a}i \circ \alpha$
- $\alpha \circ \mathbf{c}f \subseteq \mathbf{a}f$
- $\forall i : I \bullet \alpha \circ \mathbf{c}op_i \subseteq \mathbf{a}op_i \circ \alpha$

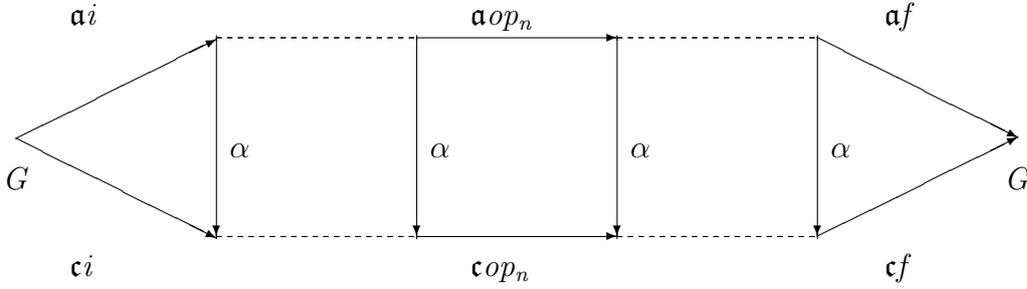


Figure 2.2: Forward simulation

These requirements assume that the operations of the model are total relations and we have already stated that operations may be partial relations. The solution to this is to totalize our model's operations. This is achieved through the definition of behaviour for those states not in the domain of the operation. Consider the following operation relation.

$$xop : X \leftrightarrow X$$

To totalize this relation we augment X by adding the distinguished, undefined element \perp , resulting in X^\perp . Elements that are in X^\perp , but not in the domain of xop are then paired with every element of X^\perp , and the pairs added to xop , giving the lifted and totalized relation $x\dot{o}p$. That is

⁶There are limitations to forward simulation and it can be necessary to use it in conjunction with backwards simulation. A discussion of backwards simulation is beyond the scope of this thesis and we will only make use of forward simulation.

$$x\dot{op} == xop \cup \{x : X^\perp; y : X^\perp \mid x \notin \text{dom } xop \bullet (x, y)\}$$

It is not necessary to totalize the retrieve relation, but it must be lifted so that any undefinedness is propagated. Imagine we begin with the following retrieve relation.

$$\alpha : X \leftrightarrow Y$$

In order to lift the relation we must add \perp paired with every element of the target type to the relation. That is

$$\begin{aligned} \dot{\alpha} &: X^\perp \leftrightarrow Y^\perp \\ \dot{\alpha} &== \alpha \cup \{\{\perp\} \times Y^\perp\} \end{aligned}$$

We now have forward simulation rules as follows.

- $\dot{c}i \subseteq \dot{a}i \wp \dot{\alpha}$
- $\dot{\alpha} \wp \dot{c}f \subseteq \dot{a}f$
- $\forall i : I \bullet \dot{\alpha} \wp \dot{c}op_i \subseteq \dot{a}op_i \wp \dot{\alpha}$

It is possible however, to create forward simulation rules that require neither lifting nor totalization. By considering domain and range restrictions we can obtain rules that give a more relaxed – and more automatable – framework for the proof of refinement. The process for obtaining these rules is described fully in [WD96]. Here, however, we shall simply present the rules for partial operations, which are as follows.

- $ci \subseteq ai \wp \alpha$
- $\alpha \wp cf \subseteq af$
- $\forall i : I \bullet (\text{dom } \mathbf{a}op_i) \triangleleft \alpha \wp \mathbf{c}op_i \subseteq \mathbf{a}op_i \wp \alpha$
- $\forall i : I \bullet \text{ran}((\text{dom } \mathbf{a}op_i) \triangleleft \alpha) \subseteq \text{dom } \mathbf{c}op_i$

A full discussion of simulation (forward and otherwise), and its usefulness in the proof of refinement is presented in [dRE98].

2.2.2 Operations with Inputs and Outputs

The forward simulation rules that we have described thus far do not allow for any operations with inputs and outputs. Woodcock and Davies extend their definition to show that the rules can incorporate operations with input and output components. Their solution requires that the inputs (and outputs) to an abstract machine are identical to those provided to a concrete machine. It is not possible to refine inputs and output when using their rules. Details of these rules and their derivation can be found on pages 250–255 of [WD96]. As the final rules rely on a number of constant and function definitions, we will not describe them further here.

Although these rules do not allow for refinement of inputs and outputs, they have since been extended to allow a more general handling of inputs and outputs. Cooper, Stepney and Woodcock [CSW00] and Boiten and Derrick [BD98] have propounded similar extensions that derive from Woodcock and Davies' work.

2.3 Refinement in Z

We have now introduced refinement and the forward simulation method for proving its applicability and correctness. We turn now to an examination of the practical application of refinement. We do this through a simple example, which conveniently allows us to provide a basic introduction to the Z notation.

Example 2.1

Consider the modelling of a train yard. We need to keep track of the trains that are in the yard, and to be able to model the effects of trains entering and leaving the yard.

In Z, the first thing we need to do is create a section in which to model the yard. This is done with the following statement.

```
section theTrainYard parents standard_toolkit
```

We would like to use the standard definitions included in Z 's mathematical toolkit (see annex B of [ISO02] or chapter 4 of [Spi92b]), so we declare our section to have a parent (*standard_toolkit*) from which those definitions are inherited. We then declare a type *trains* which represents the set of all trains in our world.

$$[trains]$$

Now we have declared the preliminaries we can define the train yard itself. We do this with a schema definition paragraph as follows.

$\begin{array}{l} \textit{trainYard} \\ \textit{trainsInYard} : \mathbb{P} \textit{trains} \end{array}$

This defines a schema called *trainYard* that has a single variable – *trainsInYard* – that is a subset of *trains* (a set of valid trains). Now we have specified our train yard we need to define the ways in which its states can be altered. Firstly, we consider the creation of a new train yard, in other words, the initial state of *trainYard*. Again we do this with a schema definition paragraph.

$\begin{array}{l} \textit{Init_trainYard} \\ \textit{trainYard} \\ \hline \textit{trainsInYard} = \emptyset \end{array}$

This defines a schema called *Init_trainYard*. We do not define any new variables in this schema, but import the *trainYard* schema. This makes all of the variables defined in *trainYard* available within *Init_trainYard*. Our constraining predicate states that in order to have a successful initialization, *trainsInYard* will be equal to the emptyset – that is the train yard contains no trains.

It is appropriate, at this point, to describe some of the strokes that are

used in the Z Notation to decorate variables names. Table 2.1 shows these strokes and explains their use.

Table 2.1: A summary of Z strokes

Stroke	Example	Use
'	$a' = a + 1$	When used on a variable this decoration allows us to refer to the post-transition value of a variable. When used on a schema, the post-transitional values of all variables in that schema's signature are made available.
?	$b? : \mathbb{N}$	This stroke is only used to decorate variables and indicates that the variable is an input to an operation.
!	$c! : \mathbb{N}$	This stroke is only used to decorate variables and indicates that the variable is an output to an operation.
Δ	ΔD	This stroke is only used to decorate schemas and is an abbreviation used to refer to the pre and post-transitional variables of a schema. For example, $\Delta D == [D; D']$.
Ξ	ΞE	This stroke is only used to decorate schemas and is an abbreviation used to refer to the pre and post-transitional variables of a schema, where the variables are unchanged (mostly used to imply that a schema is only queried, and thus, its state is not altered). For example, $\Xi E == [E; E' \mid \theta E = \theta E']$.

We now move on to consider the modelling of a train entering the yard. Again, we define a schema definition paragraph to specify the transition.

$\begin{array}{l} \textit{trainEntersYard} \\ \Delta \textit{trainYard} \\ \textit{trainEnteringYard?} : \textit{trains} \end{array}$
$\begin{array}{l} \textit{trainEnteringYard?} \notin \textit{trainsInYard} \\ \textit{trainsInYard}' = \textit{trainsInYard} \cup \{\textit{trainEnteringYard?}\} \end{array}$

This defines a schema called *trainEntersYard*. The notation $\Delta \textit{trainYard}$ allows us to refer to the pre/post-transition value of the *trainYard* schema (and its variables). We also define a new variable *trainEnteringYard?* to represent the new train entering the yard.

The first line of the constraining predicate of our schema ensures that the input variable does not belong to *trainsInYard*; that is the train entering the yard should not already be in it. The second line refers to the state of the train yard following the transition and specifies that the state of the train yard after the transition is equal to the state of the train yard before the transition unioned (see C.17) with the set containing the input variable; that is, the train yard after the transition will contain the same trains it did before the transition plus the train that entered the yard.

Finally, we consider the modelling of a train leaving the yard. Once again, we define a schema definition paragraph to specify the transition.

$\begin{array}{l} \textit{trainLeavesYard} \\ \Delta \textit{trainYard} \\ \textit{trainLeavingYard?} : \textit{trains} \end{array}$
$\begin{array}{l} \textit{trainLeavingYard?} \in \textit{trainsInYard} \\ \textit{trainsInYard}' = \textit{trainsInYard} \setminus \{\textit{trainLeavingYard?}\} \end{array}$

This defines a schema called *trainLeavesYard*. Again we refer to the pre/post-transition states of *trainYard*, and define an input variable – this time to represent the train leaving the yard. The first line of the constraining predicate of our schema ensures that the input variable belongs to the pre-transition

trainYard; that is, the train leaving the yard must actually be in the yard when the operation is performed. The second line of that predicate declares that the post-transition state of the train yard is equal to the pre-transition state of the train yard minus (C.16) the set containing the input variable; that is, the train yard after the transition will contain all the trains it did before the transition except the train that left the yard.

We now have a model of a train yard that allows us to initialize the train yard and show how trains enter and leave the yard. This model is clear and concise and is ideal for human consumption. It is also relatively simple to reason about this model's internal consistency.

Using the definition of a model provided in section 2.1 we can summarize our definition of the train yard as the quadruple $(\textit{trainYard}, \textit{Init_trainYard}, \emptyset, \langle \textit{trainEntersYard}, \textit{trainLeavesYard} \rangle)$. Although we defined the notion of finalization above, it is rarely used in practice, and our examples throughout this thesis will generally consist of just a state space, an initialization, and a set of operations.

Consider next, however, how we would implement our model. While a set is an ideal representation of our train yard, it is a mathematical concept and impossible to implement naturally within a computer due its inherent non-determinism. It is therefore necessary to create a new model of our train yard that uses a more deterministic representation.

Again, we begin with defining the trains in the yard and do this with a schema definition paragraph. To avoid confusion with our previous model, we suffix all our identifiers with an *R*.

$\textit{trainYardR}$ $\textit{trainsInYardR} : \textit{iseq } \textit{trains}$
--

This defines a schema called *trainYardR* that again has a single variable – *trainsInYardR*. Here, however, we define the train yard as a member of the set of all possible sequences of *trains* (where each element of *trains* can appear only once); that is, *trainYardR* is a sequence whose members all belong to

trains, but that contains no duplicates. As before, we now go on to define the initialization of the train yard.

$$\frac{\text{Init_trainYardR} \quad \text{trainYardR}}{\text{trainsInYardR} = \langle \rangle}$$

In the schema *Init_trainYardR* we again specify that when the train yard is created, it is empty. We do this simply by declaring that the train yard is initialized as the empty sequence. We consider next the addition of a train to the train yard.

$$\frac{\text{trainEntersYardR} \quad \Delta \text{trainYardR} \quad \text{trainEnteringYardR?} : \text{trains}}{\text{trainEnteringYardR?} \notin \text{ran } \text{trainsInYardR} \quad \text{trainsInYardR}' = \text{trainsInYardR} \hat{\ } \langle \text{trainEnteringYardR?} \rangle}$$

The similarities between this schema, and the equivalent – but more abstract – one defined previously are immediately apparent (which is to be expected given that we are modelling the same behaviour). However, as we are dealing with a sequence, when we state that the train is not already in the yard, we say that *trainEnteringYardR?* is not in the range of *trainsInYardR*, which is the same as saying that no index-object pair in the sequence has an object equal to *trainEnteringYardR?*. Similarly, the union operator makes no sense for a sequence, so when specifying the post-transition state of *trainsInYardR* we use the concatenation operator (see C.1).

$\frac{\text{trainLeavesYardR}}{\Delta \text{trainYardR}}$ $\text{trainLeavingYardR?} : \text{trains}$
$\text{trainLeavingYardR?} \in \text{ran } \text{trainsInYardR}$ $\exists x \in \mathbb{N}; Y \in \mathbb{P}\mathbb{N}$ <ul style="list-style-type: none"> • $(x, \text{trainLeavingYardR?}) \in \text{trainsInYardR}$ $\wedge Y = (\text{dom } \text{trainsInYardR} \setminus \{x\})$ $\wedge \text{trainsInYardR}' = Y \upharpoonright \text{trainsInYardR}$

Finally we consider the modelling of a train leaving the yard. The declarations in our schema are equivalent to those in the more abstract model, and again, the first part of our predicate simply ensures that the train to leave the yard is actually in the yard to begin with. The second part of the constraining predicate appears to be more complicated, but is simply stating that the post-transaction value of *trainsInYardR* will be the same as the pre-transaction value with the leaving train removed. (The condition is actually stating that there exists a natural number, x , and a set of natural numbers, Y , such that x is the index of *trainLeavingYardR?* in the pre-transition state of *trainYardR*, and Y is the set of all indexes in the pre-transition state of *trainYardR* excluding x , and the post-transition state of *trainYardR* is equal to the extraction (see C.4) of the indexes contained in Y from *trainYardR*.)

Again, we can use the definition of a model provided in section 2.1 to summarize our definition of our concrete model as the quadruple $(\text{trainYardR}, \text{Init_trainYardR}, \emptyset, \langle \text{trainEntersYardR}, \text{trainLeavesYardR} \rangle)$.

We now have a more concrete model of our train yard, and the next step is to ensure that our models and the relationship between them, is correct. In this example, we will assume that we have proved the internal consistency of our models, and that our specification matches our requirements. This leaves us free to concentrate on the validity of the refinement relationship.

The first step in proving that our concrete model is a valid refinement of our abstract model, is to specify the relationship between the states of the two models. The next step is proving that any change to the train yard is

reflected in both models identically, and the relationship is maintained. As described above this relationship is typically specified with a retrieve relation which maps states of the abstract model to equivalent states in the concrete model. From this point forwards we will use the symbol G to refer to the retrieve relation. A definition of the retrieve relation for our example is shown below.

$$\boxed{\begin{array}{l} G_{trainYard} \\ \text{trainYard} \\ \text{trainYardR} \\ \hline \text{trainsInYard} = \text{ran } \text{trainsInYardR} \end{array}}$$

We have defined a schema, $G_{trainYard}$, that operates over the state of the abstract and concrete models. In essence, we are stating that if our models are equal then the set in the state of our abstract model will be identical to the set produced from the application of the range operator to the sequence in the state of our concrete model.

Now we again turn to Woodcock and Davies' [WD96] to examine the proof of our refinement. In section 17.2 they show how the relaxed and unwound rules for forward simulation can be expressed for schema, and produce the following conditions to show refinement (for an abstract model, \mathcal{A} , and concrete model, \mathcal{C} such the \mathcal{AI} and \mathcal{CI} are the initializations of the appropriate machines, and \mathcal{AOp} and \mathcal{COp} refer to their operations in turn).

- $\forall \mathcal{C} \bullet \mathcal{CI} \Rightarrow \exists \mathcal{A} \bullet \mathcal{AI} \wedge G$
- $\forall \mathcal{A}; \mathcal{C} \bullet \text{pre } \mathcal{AOp} \wedge G \Rightarrow \text{pre } \mathcal{COp}$
- $\forall \mathcal{A}; \mathcal{C}; \mathcal{C}' \bullet \text{pre } \mathcal{AOp} \wedge G \wedge \mathcal{COp} \Rightarrow \exists \mathcal{A}' \bullet \mathcal{AOp} \wedge G'$

These rules use Z 's schema language. A schema can, however, be considered an abstraction of a simple set comprehension expression, where.

$$[\text{declaration} \mid \text{predicate}] == \{\text{declaration} \mid \text{predicate} \bullet \theta \text{declaration}\}$$

Woodcock and Davies explain how an operational schema can be considered a relaxed and unwound relational specification, where the schema

$$[\Delta state; input?; output! \mid predicate]$$

can be expressed as a relation as follows⁷.

$$\begin{aligned} & \{ \Delta state; input?; output! \mid predicate \\ & \bullet (\theta state, input?) \mapsto (\theta state', output!) \} \end{aligned}$$

We prefer to maintain this relational approach so we restate our forward simulation rules one last time⁸, giving the following conditions which we will henceforth refer to as the refinement proof obligations. (Note that we have also introduced inputs and outputs to operations in our proof obligation, we have however, restricted these to be equal in the operations of abstract and concrete models; that is we do not allow refinement of inputs or outputs.)

- $\forall v : \mathcal{C} \mid v \in \mathcal{CI} \bullet \exists u : \mathcal{A} \mid u \in \mathcal{AI} \bullet (u, v) \in G$
- $\forall u : \mathcal{A}; v : \mathcal{C}; i : inputs$
 $\mid (u, i) \in \text{pre } \mathcal{AOp} \wedge (u, v) \in G$
 $\bullet (v, i) \in \text{pre } \mathcal{COp}$
- $\forall u : \mathcal{A}; v, v' : \mathcal{C}, i : inputs, p : outputs$
 $\mid (u, i) \in \text{pre } \mathcal{AOp} \wedge ((v, i) \mapsto (v', p)) \in \mathcal{COp} \wedge (u, v) \in G$
 $\bullet \exists u' : \mathcal{A}; o : outputs$
 $\mid ((u, i) \mapsto (u', o)) \in \mathcal{AOp}$
 $\bullet (u', v') \in G \wedge (o, p) \in \text{id}_{outputs}$

Now we have confirmed the rules we will use, we can proceed to examine the proof of the refinement relationship between our train yard models. We

⁷Note that, the order of the declaration in a schema is unimportant, it is therefore, essential that we use our final expression to specify the order of the members of our relations. For convenience however, where the order of the declaration is equal to the order of the members in the relation, we will not provide a redundant declaration.

⁸As we intend to pass our proof obligations to a theorem prover, it is an advantage to have our proof obligations constructed in the mathematical language used by such programs.

consider first the initialization proof obligation. Substituting our schema definitions into the above obligation gives the following theorem.

$$\begin{aligned} \vdash? \forall v : \{ & \text{trainsInYardR} : \text{iseq } \text{trains} \mid \text{true} \} \\ & \mid v \in \{ \text{trainsInYardR} : \text{iseq } \text{trains} \mid \text{trainsInYardR} = \langle \rangle \} \\ & \bullet \exists u : \{ \text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{true} \} \\ & \bullet u \in \{ \text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{trainsInYard} = \emptyset \} \\ & \wedge (u, v) \in \{ \text{trainYard} : \mathbb{P} \text{trains}; \text{trainYardR} : \text{iseq } \text{trains} \\ & \quad \mid \text{trainYard} = \text{ran } \text{trainYardR} \} \end{aligned}$$

Through the application of the set comprehension one-point rule (see D.4) to the sets representing our schema definitions, we can reduce the theorem as follows. (We use simplification (see D.3) to remove redundant terms.)

$$\vdash? \forall v : \text{iseq } \text{trains} \mid v = \langle \rangle \bullet \exists u : \mathbb{P} \text{trains} \bullet u = \emptyset \wedge u = \text{ran } v$$

The application of the existential quantification one-point rule (see D.1) reduces our theorem still further.

$$\vdash? \forall v : \text{iseq } \text{trains} \mid v = \langle \rangle \bullet \emptyset = \text{ran } v$$

The universal quantification one-point rule (see D.5) is then applied, giving the following theorem.

$$\vdash? \emptyset = \text{ran} \langle \rangle$$

We are left with a simple theorem which we can deduce is true from the definition of the range operator (see C.11).

We now progress to the proof obligations for our operations. We consider first the operation which describes the addition of a train to the yard. Firstly, it is necessary to calculate the precondition for the abstract and concrete versions of these operations. We do this by the method detailed in chapter 14 of [WD96], giving the following definitions.

$\text{pre } \text{trainEntersYard} \text{ —————}$ trainYard $\text{trainEnteringYard?} : \text{trains}$
$\text{trainEnteringYard?} \notin \text{trainsInYard}$

$\text{pre } \text{trainEntersYardR} \text{ —————}$ trainYardR $\text{trainEntersYardR?} : \text{trains}$
$\text{trainEntersYardR?} \notin \text{ran } \text{trainsInYardR}$

We can substitute these definitions into our termination proof obligation, giving the following theorem.

$$\begin{aligned} &\vdash? \forall u : \{ \text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{true} \}; \\ &\quad v : \{ \text{trainsInYardR} : \text{iseq } \text{trains} \mid \text{true} \}; \quad i : \text{trains} \\ &\quad \mid (u, i) \in \{ \text{trainsInYard} : \mathbb{P} \text{trains}; \text{trainEnteringYard?} : \text{trains} \\ &\quad \quad \mid \text{trainEnteringYard?} \notin \text{trainsInYard} \} \\ &\quad \wedge (u, v) \in \{ \text{trainsInYard} : \mathbb{P} \text{trains}; \text{trainsInYardR} : \text{iseq } \text{trains} \\ &\quad \quad \mid \text{trainsInYard} = \text{ran } \text{trainsInYardR} \} \\ &\quad \bullet (v, i) \in \{ \text{trainsInYardR} : \mathbb{P} \text{trains}; \text{trainEntersYardR?} : \text{trains} \\ &\quad \quad \mid \text{trainEntersYardR?} \notin \text{ran } \text{trainsInYardR} \} \end{aligned}$$

Firstly, we apply the set comprehension one-point rule (see D.4) to the sets representing our schema definitions and reduce the theorem to the following. (We use simplification (see D.3) to remove redundant terms.)

$$\begin{aligned} &\vdash? \forall u : \mathbb{P} \text{trains}; \quad v : \text{iseq } \text{trains}; \quad i : \text{trains} \\ &\quad \mid i \notin u \wedge u = \text{ran } v \\ &\quad \bullet i \notin \text{ran } v \end{aligned}$$

The universal quantification one-point rule (see D.5) is then applied to eliminate u , giving the following theorem.

$$\begin{aligned} &\vdash? \forall v : \text{iseq } \mathit{trains}; i : \mathit{trains} \\ &\quad | i \notin \text{ran } v \\ &\quad \bullet i \notin \text{ran } v \end{aligned}$$

This proof obligation can now be discharged trivially.

Next we consider the correctness proof obligation. Again, we substitute our schema definitions into the refinement proof obligation presented above. As we do not have any outputs we will not consider their relationship in our proof obligation. (Note that while previously we have written the relations produced from the operation schemas in the form $((v, i) \mapsto (v', o))$, we will now use the equivalent – shorter – notation (v, i, v', o) . Similarly, to reduce the need for a characteristic tuple declaration at the end of each set comprehension, we will use the most convenient order for the members of the tuple produced from each set. While this strictly means that we are not producing relations, the sets of tuples are more than sufficient in this context.)

$$\begin{aligned} &\vdash? \forall u : \{ \mathit{trainsInYard} : \mathbb{P} \mathit{trains} \mid \text{true} \}; \\ &\quad v, v' : \{ \mathit{trainsInYardR} : \text{iseq } \mathit{trains} \mid \text{true} \}; i : \mathit{trains} \\ &\quad | (u, i) \in \{ \mathit{trainsInYard} : \mathbb{P} \mathit{trains}; \mathit{trainEnteringYard}? : \mathit{trains} \\ &\quad \quad | \mathit{trainEnteringYard}? \notin \mathit{trainsInYard} \} \\ &\quad \wedge (u, v) \in \{ \mathit{trainYard} : \mathbb{P} \mathit{trains}; \mathit{trainYardR} : \text{iseq } \mathit{trains} \\ &\quad \quad | \mathit{trainYard} = \text{ran } \mathit{trainYardR} \} \\ &\quad \wedge (v, v', i) \in \{ \mathit{trainsInYardR}, \mathit{trainsInYardR}' : \text{iseq } \mathit{trains}; \\ &\quad \quad \mathit{trainEnteringYardR}? : \mathit{trains} \\ &\quad \quad | \mathit{trainEnteringYardR}? \notin \text{ran } \mathit{trainsInYardR} \\ &\quad \quad \wedge \mathit{trainsInYardR}' \\ &\quad \quad = \mathit{trainsInYardR} \frown \langle \mathit{trainEnteringYardR}? \rangle \} \\ &\quad \bullet \exists u' : \{ \mathit{trainsInYard} : \mathbb{P} \mathit{trains} \mid \text{true} \} \\ &\quad | (u, u', i) \in \{ \mathit{trainsInYard}, \mathit{trainsInYard}' : \mathbb{P} \mathit{trains}; \\ &\quad \quad \mathit{trainEnteringYard}? : \mathit{trains} \\ &\quad \quad | \mathit{trainEnteringYard}? \notin \mathit{trainsInYard} \\ &\quad \quad \wedge \mathit{trainsInYard}' \\ &\quad \quad = \mathit{trainsInYard} \cup \{ \mathit{trainEnteringYard}? \} \} \\ &\quad \wedge (u', v') \in \{ \mathit{trainYard} : \mathbb{P} \mathit{trains}; \mathit{trainYardR} : \text{iseq } \mathit{trains} \\ &\quad \quad | \mathit{trainYard} = \text{ran } \mathit{trainYardR} \} \end{aligned}$$

Firstly, we apply the set comprehension one-point rule (see D.4) to the sets

representing our schema definitions and reduce the theorem to the following. (We use simplification (see D.3) to remove redundant terms.)

$$\begin{aligned} \vdash? \forall u : \mathbb{P} \text{trains}; v, v' : \text{iseq trains}; i : \text{trains} \\ | i \notin u \wedge u = \text{ran } v \wedge i \notin \text{ran } v \wedge v' = v \hat{\ } \langle i \rangle \\ \bullet \exists u' : \mathbb{P} \text{trains} \\ \bullet i \notin u \wedge u' = u \cup \{i\} \wedge u' = \text{ran } v' \end{aligned}$$

The application of the existential quantification one-point rule (see D.1) reduces our theorem still further.

$$\begin{aligned} \vdash? \forall u : \mathbb{P} \text{trains}; v, v' : \text{iseq trains}; i : \text{trains} \\ | i \notin u \wedge u = \text{ran } v \wedge i \notin \text{ran } v \wedge v' = v \hat{\ } \langle i \rangle \\ \bullet i \notin u \wedge \text{ran } v' = u \cup \{i\} \end{aligned}$$

The universal quantification one-point rule (see D.5) is then applied to eliminate u , giving the following theorem.

$$\begin{aligned} \vdash? \forall v, v' : \text{iseq trains}; i : \text{trains} \\ | i \notin \text{ran } v \wedge v' = v \hat{\ } \langle i \rangle \\ \bullet i \notin \text{ran } v \wedge \text{ran } v' = \text{ran } v \cup \{i\} \end{aligned}$$

Again, the universal quantification one-point rule is applied, this time eliminating v' , giving the following theorem.

$$\begin{aligned} \vdash? \forall v : \text{iseq trains}; i : \text{trains} \\ | i \notin \text{ran } v \\ \bullet \text{ran}(v \hat{\ } \langle i \rangle) = \text{ran } v \cup \{i\} \end{aligned}$$

Which we can see is true from the definitions of the injective sequence, and the range and concatenation operators (see C.8, C.11 and C.1).

A similar proof can be presented for the remaining operation, and with that it is possible to conclude that our concrete model is a refinement of our abstract model.

2.4 Limitations of Refinement

The benefits of refinement are clear to see, for example – and perhaps most noticeably – in the implementation of the Mondex purse [SCW98, SCW00], and in the development of software for the French railways system [BBFM99]. Anyone who has experience in a development environment however, will know that it is still not used in the majority of systems. Instead, development still relies on a trial-and-error, design-and-test regime. There are a number of reasons why the adoption of refinement is not widespread. This section will focus on a common eventuality that has led to the misuse of refinement. While this misuse is not strictly a limitation of refinement, (more a case of the wrong tool for the wrong job), it does highlight the need for newer techniques that can be used alongside refinement in the formal construction of software. The following example highlights the fact that we don't just need formal techniques to create code from specification, but also techniques which allow us to create specifications in the first place.

Example 2.2

This example will be very similar to the last, however, in this instance we will model the stars in the sky rather than the trains in a yard.

As before, the specification begins with a declaration of the section and the inheritance of the standard toolkit, which allows us to make use of Z 's standard mathematical definitions.

```
section theStarsInTheSky parents standard_toolkit
```

We then define the given type, *stars*, which is a set that contains all the stars in the universe.

```
[stars]
```

Using a schema definition paragraph, we then specify our simplistic view of the sky.

$$\frac{\textit{sky}}{\textit{starsInSky} : \mathbb{P} \textit{stars}}$$

We then define the initialization of our sky, where for simplicity we will assume that the sky begins empty. (Let us presume that no stars have yet been observed!)

$$\frac{\textit{Init_sky}}{\textit{sky}} \\ \textit{starsInSky} = \emptyset$$

Next, we define the operation we will use to record the discovery of a star. As before this simply involves checking that the new star is not already in the sky, and if not, then adding the star to our record of the sky's stars.

$$\frac{\textit{discoverStar}}{\Delta \textit{sky}} \\ \textit{newStar?} : \textit{stars} \\ \textit{newStar?} \notin \textit{starsInSky} \\ \textit{starsInSky}' = \textit{starsInSky} \cup \{\textit{newStar?}\}$$

Consider now the practical implementation of this model. Our first step in producing an implementable model would probably be to make our set of stars, a sequence (as in the train yard example). When we come to implement a sequence in a programming language we typically use an array structure which has similar behaviour, but usually a finite – and specified – size⁹. In our proposed refinement of the model we will skip the specification of the model with a sequence and move straight to a model involving arrays.

Again, we define a schema definition paragraph with which to represent

⁹In this example we shall consider an array to simply be a special type of sequence, where the only difference is the fixed limitation on the size of the array.

the state of our model. However, instead of declaring our known stars as a set, we define an array with a specified size, *upperlimit*¹⁰.

$\frac{\textit{skyR}}{\textit{starsInSkyR} : \text{array}[\textit{upperlimit}] \textit{stars}}$

Initialization proceeds as expected.

$\frac{\textit{Init_skyR}}{\textit{skyR}}$
$\textit{starsInSkyR} = []$

However, when we come to consider the operation that models the discovery of a star, the resolution of non-determinism forces us to make some changes.

$$\textit{MESSAGES} ::= \textit{starAdded} \mid \textit{starArrayFull}$$

$\frac{\textit{discoverStarR}}{\Delta \textit{skyR}}$
$\textit{newStarR?} : \textit{stars}$
$\textit{message!} : \textit{MESSAGES}$
$\textit{newStarR?} \notin \text{ran } \textit{starsInSkyR}$
$\# \textit{starsInSkyR} < \textit{upperlimit}$
$\Rightarrow \textit{starsInSkyR}' = \textit{starsInSkyR} \hat{\ } [\textit{newStarR?}]$
$\wedge \textit{message!} = \textit{starAdded}$
$\# \textit{starsInSkyR} \geq \textit{upperlimit}$
$\Rightarrow \textit{starsInSkyR}' = \textit{starsInSkyR}$
$\wedge \textit{message!} = \textit{starArrayFull}$

¹⁰In this instance we shall assume that *upperlimit* is a constant that has been defined elsewhere.

Firstly, we need to state that we can only add the new star to our collection if our array is not already full. Secondly, we need some way of reporting the success – or otherwise – of the operation.

Now we have our two models, the latter of which can be considered a more concrete version of the first. Can it be considered a refinement however? Ideally, we would have a retrieve relation as follows.

$ \begin{array}{l} G_{sky} \\ sky \\ skyR \end{array} $
$starsInSky = \text{ran } starsInSkyR$

Upon examination, the concrete model can clearly be seen to not be a refinement of the abstract; the behaviour of the two models differs. Firstly, the concrete model produces an output, where the abstract model does not. Some forms of refinement allow outputs to be refined so this could be worked around. Of more importance is the fact that when the array is full, the retrieve relation would not hold; the abstract model would have a set of stars containing one more star than in the range of the concrete model's array.

The only way in which we can now relate our two models is to allow the concrete implementation details to propagate up the chain and show them in our abstract specification. This would give us an abstract version of the operation as follows.

$\frac{\textit{discoverStar}}{\Delta sky}$ $newStar? : stars$ $message! : message$ <hr/> $newStar? \notin sky$ $\#starsInSky < upperlimit$ $\Rightarrow starsInSky' = starsInSky \cup \{newStar?\}$ $\wedge message! = starAdded$ $\#starsInSky \geq upperlimit$ $\Rightarrow starsInSky' = starsInSky$ $\wedge message! = starArrayFull$

Surely, however, the point of refinement is to *introduce* implementation details as we approach the most concrete version of our model. Now, while our concrete model can be considered a refinement of the abstract model, the abstract model is cluttered with design details that should not be necessary in an abstraction. When we consider the simplicity of our original model, it can be clearly seen that these details obfuscate the meaning of the specification, and in a practical example the implementation details required would be considerably more complex.

Hold on, you might say, surely this implementation detail would also be required in our train yard example. This is certainly true, however there is one big difference between the models that illustrates the incorrectness of implementation details being filtered to the abstract specification. A train yard is a physical entity and as such has a finite size – there is a real world limit. The imposition of an upper limit on the size of the train yard therefore is required in order to model it correctly; the lack of this in our model was a specification error as we have simply omitted a genuine requirement. However, the number of stars in the sky is unknown, and for all practical purpose is infinite; the limit we impose, therefore, is purely arbitrary and the result of a specific implementation. We have not omitted a system requirement, but have had a requirement imposed upon us through our implementation choices. If we were to examine the abstract specifications of our two models

we should be able to tell which included the limit as a system requirement and which required it in order to be able perform a specific implementation.

What we really require is to have two specifications for each of our systems. One which provides an abstract model of what happens in reality, and a second which provides an alternate perspective incorporating the restrictions imposed by the implementation. Clearly, we are unable to use refinement to show that these specifications mean the same thing, so what we require is a different technique that allows us to show a rigorous relationship between specifications with definable differences in behaviour.

2.5 Retrenchment

Retrenchment – first introduced in [BP98] – is a technique designed to overcome the limitations described in the previous section. The goal is principally to increase the number of systems to which formal development methodologies can be successfully applied. Retrenchment encourages the rigorous evolution of specifications. Using retrenchment we can formally relate abstract models of real world systems to implementable specifications of those systems. Once we have an implementable specification, existing techniques – particularly refinement – can be used to formally create implementations.

Much work has been carried out into the nature of retrenchment and its relationship to refinement. A consolidated view of the current retrenchment position can be found in [Ban03, BJF⁺04, BPJS07]¹¹.

Figure 2.3 on the next page shows how retrenchment fits in the development lifecycle. The abstract specification is first created from an analysis of the required system. New versions of that specification are then created which incrementally introduce gradually more limitations imposed on the abstract specification by the implementation. Retrenchment relationships (ρ) are then created that relate these new models to their more abstract equivalents. This process continues until all of the limitations are incorporated within our model. We refer to this model as the ‘contracted model’; that

¹¹A tutorial providing an introduction to retrenchment is available online, see [Ban].

is, it is the model with which the final implementation of the system will be compared when checking its validity. This model is named this way as it is intended to form the contract between the specifier and the developer of the system. The specifier guarantees that the model is a correct representation of the real-world system, and the developer guarantees that the system they produce will be a correct implementation of that model. Once we have the contracted model, refinement is used to create code that matches the specification in the normal way.

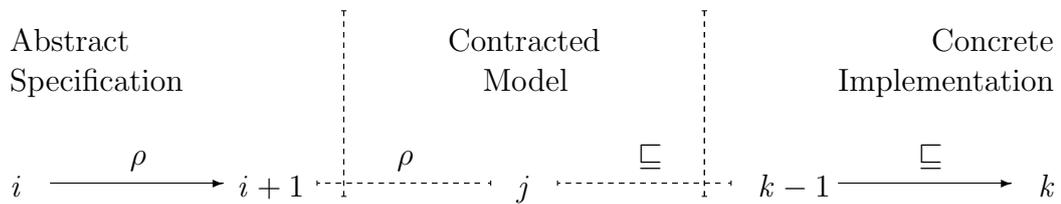


Figure 2.3: Retrenchment's role

Retrenchment is not dissimilar to refinement, and indeed, seeks to take advantage of refinement's strengths whilst weakening the inflexibility of its proof obligations in a controlled manner. In effect, retrenchment is a trade-off; we lose some of the formality of refinement (the change to the proof obligations prevents us being able to take advantage of the substitutivity of models), but greatly extend the range of system developments to which formal techniques can be applied.

Refinement can be seen as a 'black box' technique, as the property of substitutivity ensures that users are unable to tell whether they are using the most abstract model, the most concrete model, nor any model in between. When using refinement a user is unaware of the way in which a particular model has been specified (other than the most abstract), and nor are they aware of the specific nature of the relationships between the models. Retrenchment, on the other hand, can be seen as a 'glass box' technique, one model does not simply represent an equivalent of another. As the implementation details are gradually added to the abstract specification, the user must examine the abstract model, the concrete model and the relationship between them.

The starting point for retrenchment can be seen as the forward simulation rules for refinement. These rules are then weakened to extend the scope of models that can be related. This weakening occurs through the introduction of two relations that supplement the retrieve relation used in refinement. The first of these is called the ‘within’ relation. The within relation is used to reduce the set of abstract-concrete, pre-transition state pairs over which the relationship between the models is defined. The second relation is the ‘concedes’ relation, and this allows us to extend the set of valid abstract-concrete, post-transition state pairs for the relationship. This can be seen more clearly in figure 2.4 on the following page. If the set of all possible abstract-concrete state pairs is U , then the retrieve relation, G , is a subset of U , and refinement guarantees that if the abstract-concrete state pair is a member of the retrieve relation prior to the transition, then the state pair that results from the transition will also be a member of the retrieve relation. Retrenchment, however, only guarantees certain behaviour when the pre-transition, abstract-concrete state pair is a member of the retrieve relation, *and* the within relation, W . Furthermore, retrenchment does not guarantee that – even under these conditions – the post-transition, abstract-concrete state pair will belong to the retrieve relation, only that it will belong to *either* the retrieve relation, *or* the concedes relation, C .

The Z forward simulation rules for retrenchment for an abstract model, \mathcal{A} , and a concrete model, \mathcal{C} can be summarized as follows (that is the rules equivalent to the refinement rules presented in section 17.2 of [WD96]).

$$\triangleright \forall \mathcal{C} \bullet \mathcal{C}I \Rightarrow \exists \mathcal{A} \bullet \mathcal{A}I \wedge G$$

$$\triangleright \forall \mathcal{A}; \mathcal{C} \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \Rightarrow \text{pre } \mathcal{C}Op$$

$$\triangleright \forall \mathcal{A}; \mathcal{C}; \mathcal{C}' \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \wedge \mathcal{C}Op \Rightarrow \exists \mathcal{A}' \bullet \mathcal{A}Op \wedge (G' \vee C)$$

It should be noted that the operation proof obligations presented above form the definition of retrenchment. This is unlike refinement, which can be defined by the property of substitutivity from which its proof obligations are then derived.

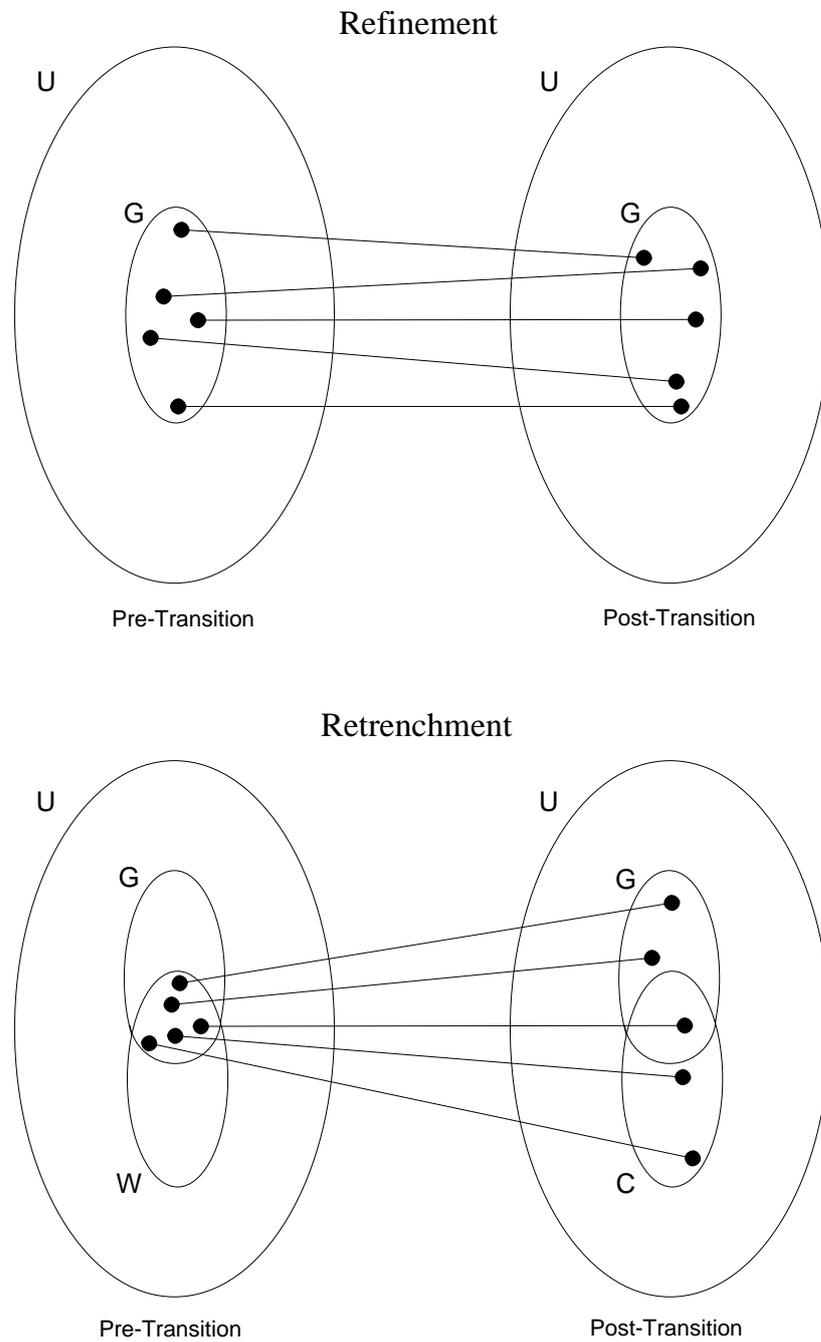


Figure 2.4: Transitions in Refinement and Retrenchment

A number of case studies have been performed with retrenchment. Most

notably the implementation of the Mondex purse (cited earlier as a success story of refinement) has been re-examined, – see [BPJS05a, BPJS05b, BJPS06a], and chapter 10 of [Jes05] – and a critical eye has exposed areas of the development where retrenchment has been able to add clarity to the process and permitted the creation of a concrete model that more accurately reflected the actual purse design.

2.5.1 Output Retrenchment

Since retrenchment was first proposed, the proof obligations presented above have been further refined. Output retrenchment permits the augmentation of the retrieve relation, with an operation specific relation – called the output relation – that allows a strengthening of the relationship between the outputs of the operation. That is, retrenchment will now guarantee that, should the pre-transition, abstract-concrete state pair belong to the retrieve relation and the within relation, then the post-transition, abstract-concrete state pair will belong to either the retrieve relation *and* the output relation or the concedes relation. The Z forward simulation rules for output retrenchment given an abstract model, \mathcal{A} , and a concrete model, \mathcal{C} are as follows.

- $\forall \mathcal{C} \bullet \mathcal{C}I \Rightarrow \exists \mathcal{A} \bullet \mathcal{A}I \wedge G$
- $\forall \mathcal{A}; \mathcal{C} \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \Rightarrow \text{pre } \mathcal{C}Op$
- $\forall \mathcal{A}; \mathcal{C}; \mathcal{C}' \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \wedge \mathcal{C}Op$
 $\Rightarrow \exists \mathcal{A}' \bullet \mathcal{A}Op \wedge ((G' \wedge O) \vee C)$

Note that the version of retrenchment specified in the previous section is now typically referred to as ‘primitive retrenchment’, and output retrenchment is considered to be the default definition of the retrenchment relationship.

2.5.2 Multiple Retrenchment

The loss of refinement’s substitutivity allows us to alter its proof obligations and define a retrenchment. This change also has other advantages. Whereas

an abstract and concrete model involved in a refinement relationship must have an identical set of operations, retrenchment requires only that the set of operations in the abstract model is a subset of the concrete model's operations. This means that a concrete model can be considered a retrenchment of two considerably different abstract models. For example, we could have two abstract models: one that defines addition, and another subtraction. We could then have a single concrete model that provides both these functions. Furthermore, we may have two abstract models that define the same operation. The behaviour of the operation in the two models may, however, be very different. With retrenchment, we are able to specify an operation that integrates those behaviours. For instance, we may have existing abstract models that describe ferry docking procedures in Britain and France, and are seeking to specify a European standard. With retrenchment we are able to specify a concrete model that defines a *dockFerry* operation that builds upon and integrates the equivalent British and French operations, and the retrenchments themselves provide a specification of the relationship between the new and existing standards. Additionally, retrenchment gives us the advantage that this relationship can be proved, with the discharge of the necessary proof obligations.

We now reconsider the role of retrenchment in development lifecycle, and can replace figure 2.3 with figure 2.5 on the next page. Now existing abstract specifications are taken where available, and new specifications created to describe new aspects of our system (for example, *A*, *B*, *C*, *D*, *E* and *F* in the figure). More concrete versions of these specifications are then created that not only introduce implementation specific detail, but also integrate the more abstract specifications. Each model is related to its more abstract equivalents through retrenchment relationships (ρ). This process continues until we reach a contracted model (*ABCDEF*) that contains all the functionality of our new system and contains all of the implementation specific detail. This contracted model is then refined to produce an implementation in the usual way.

[BP02] presents a case study that takes advantage of multiple retrenchment. The study begins with an abstract specification of the plain old telephone systems; at the next deepest level of abstraction, specifications are

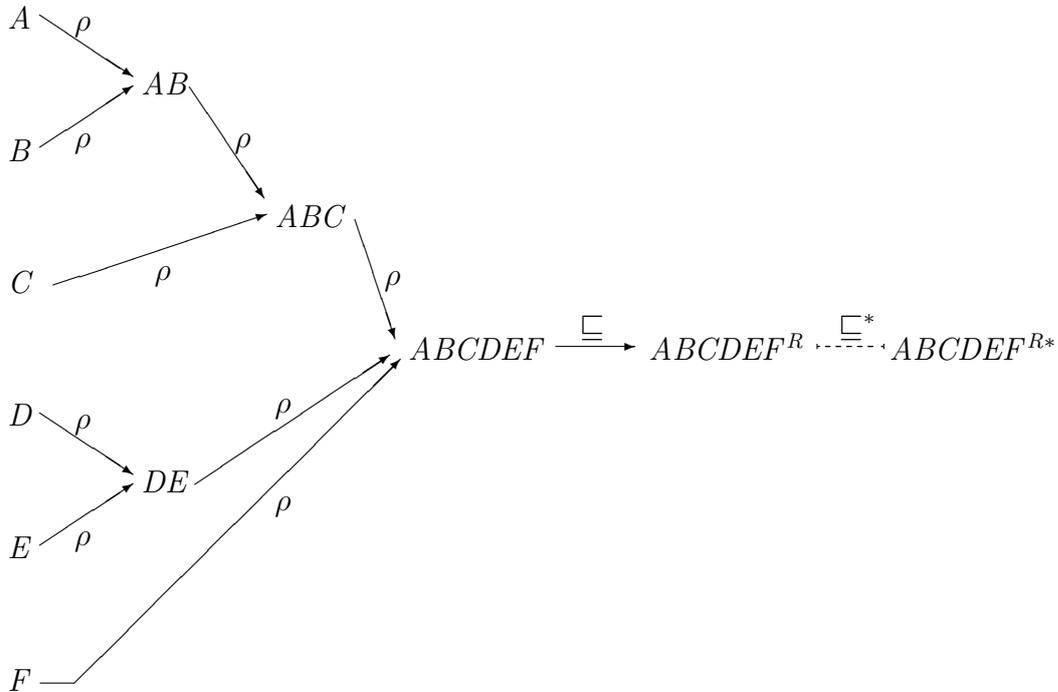
Abstract
SpecificationConcrete
Implementation

Figure 2.5: Multiple Retrenchment

created that add a number of services to this telephone system, and each is shown to be a retrenchment of the original system. A final model is created that integrates all of the introduced services, and retrenchment is used to show a relationship between each of the individual service specifications, and also the original model. In chapter 8, we revisit this case study and seek to show its validity within our developed tool.

2.5.3 Other Forms of Retrenchment

There are other forms of retrenchment than the ones described above. For instance, sharp retrenchment was proposed in [BP99]. Here the existing proof obligation is supplemented with an additional ‘nevertheless’ relation designed

to guarantee a specific relationship between the post-transition states of the abstract and concrete models. A valid retrenchment will therefore, guarantee that if the pre-transition abstract-concrete state pair belongs to both the retrieve and within relation, then not only will the post-transition state pair belong to the retrieve or concedes relation, but it will also belong to the nevertheless relation (regardless of whether the pair is a member of retrieve or concedes relation; allowing us to specify behaviour that will *always* be exhibited). The forward simulation rules for sharp retrenchment are as follows.

$$\begin{aligned}
&\triangleright \forall \mathcal{C} \bullet \mathcal{C}I \Rightarrow \exists \mathcal{A} \bullet \mathcal{A}I \wedge G \\
&\triangleright \forall \mathcal{A}; \mathcal{C} \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \Rightarrow \text{pre } \mathcal{C}Op \\
&\triangleright \forall \mathcal{A}; \mathcal{C}; \mathcal{C}' \bullet \text{pre } \mathcal{A}Op \wedge G \wedge W \wedge \mathcal{C}Op \\
&\quad \Rightarrow \exists \mathcal{A}' \bullet \mathcal{A}Op \wedge ((G' \vee C) \wedge N)
\end{aligned}$$

Of course, it is also possible to combine sharp retrenchment and output retrenchment to give sharp output retrenchment. Furthermore, as retrenchment is a relatively new technique, and work continues to further augment its usefulness, there is a strong possibility that new and different forms of retrenchment will continue to be proposed. These new forms combined with the existing techniques could potentially provide limitless numbers of different formats of proof obligations. It is essential, therefore, that any mechanical support for retrenchment be completely flexible in order to provide support not only for the retrenchment flavours available today, but also those available in the future.

2.5.4 Other Techniques to Solve Refinement's Limitations

Retrenchment is not the only solution that has been suggested to tackle the limitations of refinement identified above. A number of other techniques to liberalize refinement, and make specification construction a transparent, formal process have been proposed.

Boiten and Derrick propounded the use of approximations [BD05] to supplement a formal program development. The technique involves the use of chains of specification in place of a single specification. The specification is considered to be equivalent to its limit, the limit being the single specification replaced. For example, the limit of a specification of a natural number will be a specification of a number that belongs to the unbounded natural numbers. As the chain is traversed more implementation detail (that is the detail that would be added through retrenchment) is incorporated. When developing a specification there are four options open to the specifier: *element-wise refinement* where one specification is refined to another in the normal way; *sequence introduction* where a specification is replaced by a chain of specification; *sequence replacement* where a chain of specification is replaced by another (such that the limit of one is the refinement of the other's limit); and *compromise* where a chain of specification is replaced by one of its members. The closeness of a specification in a chain to its limit is indicated by a metric. Different metrics are used depending upon the nature of the specification. Jeske [Jes05] notes that metrics can intrinsically only focus on particular properties of a chained specification's relationship with its limit. He points out that there is no guarantee that incorrect behaviour will not be introduced with a specification's replacement, unlike retrenchment where every aspect of the concrete model's relationship to the abstract must be explicitly declared and verified with the retrenchment proof obligations.

Realization [Smi00b] is a technique that relates an abstract and concrete specification. Unlike refinement (but like retrenchment) the more concrete specification is only required to maintain an approximation of the functionality of its more abstract counterpart. Realization is based on an adaptation of Morgan's refinement calculus [Mor94], known as the timed refinement calculus [MH92]. The timed refinement calculus replaces preconditions and postconditions that hold at the beginning and end of a program's run, with an assumption and an effect which hold over all time. Refinement allows the assumption to be strengthened or the effect weakened. Realization extends the allowable transformations through the introduction of three additional

rules. These rules are known as ‘add assumption’, ‘modify inputs’ and ‘modify outputs’. The rules are said to be complete if the specification is feasible and the associated assumption is satisfiable. Like retrenchment, the onus is on the specifier to use the expressive power wisely and ensure that approximations are ‘acceptable’.

Evolution [Liu97, Liu99] is a technique that seeks to capture all possible software development activities in the process of producing a program from a set of abstract requirements. Liu determines that all software development steps can be categorized as either an improvement, an extension or a modification. An improvement is a standard refinement. An extension is a step that preserves the original behaviour, but enhances it in some way. A modification is a step that does not preserve the original behaviour. Of interest to us is the extension step which is designed to perform the function of retrenchment (in the evolution framework). An operation Q can be considered an extension of the operation P if the semantic equivalent of $\text{pre}(P)$ occurs as a component within $\text{pre}(Q)$ and the semantic equivalent of $\text{post}(P)$ occurs as a component within $\text{post}(Q)$. However, as Banach notes in [Ban00] S is equivalent to $(S \wedge T) \vee (S \wedge \neg T)$ and therefore, any operation Q can be considered an extension of any operation P . Of course, retrenchment also allows a relationship between any two operations. However, the nature of that relationship must be explicitly and transparently defined using retrenchment’s *concedes*, *output* and *within* relations.

2.6 Retrenchment in \mathbf{Z}

We now return to example 2.2 and examine how we can formally relate our abstract specification to that containing the implementation detail. We consider the abstract version of the discover star operation to be the following.

$$\begin{array}{l}
\textit{discoverStar} \\
\hline
\Delta sky \\
newStar? : stars \\
\hline
newStar? \notin starsInSky \\
starsInSky' = starsInSky \cup \{newStar?\}
\end{array}$$

The operation of the contracted model, that – while still a specification – contains the limitations imposed by the implementation is given by the following schema.

$$\begin{array}{l}
\textit{discoverStarC} \\
\hline
\Delta skyC \\
newStarC? : stars \\
message! : message \\
\hline
newStarC? \notin starsInSkyC \\
\#starsInSkyC < upperlimit \\
\quad \Rightarrow starsInSkyC' = starsInSkyC \cup \{newStarC?\} \\
\quad \quad \wedge message! = starAdded \\
\#starsInSkyC \geq upperlimit \\
\quad \Rightarrow starsInSkyC' = starsInSkyC \\
\quad \quad \wedge message! = starArrayFull
\end{array}$$

We now need to define the relations between the states of our models. We begin by describing the ideal situation through the retrieve relation.

$$\begin{array}{l}
G_{sky} \\
\hline
sky \\
skyC \\
\hline
starsInSky = starsInSkyC
\end{array}$$

That is, the set of stars in the abstract specification should normally equal the set of stars in the contracted model. We then define the operation-specific

relations between the models (collectively we refer to these as an operation's 'ramifications'). Consider first the within relation, that is used to add further constraints to the pre-transitional states of the models. Our within relation is specified below.

$$\begin{array}{|l}
 \hline
 W_{discoverStar} \\
 \text{pre } discoverStar \\
 \text{pre } discoverStarC \\
 \hline
 newStar? = newStarC? \\
 \hline
 \end{array}$$

With our definition of refinement, the inputs to the two operations are implicitly identical; in retrenchment however, it is possible to have stronger or weaker links between the inputs of the two operations. Therefore it is necessary to explicitly state any relationship. In this instance, we require each operation to receive the same input, and we specify this using our within relation.

Note that, only the pre-transitional components (inputs and unprimed state components) of the involved operations can be used when defining the within relation. We can use the precondition of each operation's schema to indicate the allowed components.

It is also possible to have varying relationships between the outputs of the two operations and we must specify the required relationship. For this we define the output relation which relates the outputs of the two models' operations. In this instance, only one of our operations has an output, but the output relation allows us to strengthen the retrieve relation locally to an operation. The output relation for our example is specified below.

$$\begin{array}{|l}
 \hline
 O_{discoverStar} \\
 discoverStar \\
 discoverStarC \\
 \hline
 message! = starAdded \\
 \hline
 \end{array}$$

This local strengthening is what is required in our operation so that we can guarantee that whenever the retrieve relation holds, the output from the concrete model's operation will produce a message that indicate the success in adding the star.

Finally, we define the concedes relation between the two versions of our operation. The concedes relation allows us to extend the set of valid post-transitional states of our models. The concedes relation for this example follows.

$$\begin{array}{l}
 \frac{C_{discoverStar} \quad \text{discoverStar}}{\text{discoverStar}\mathcal{C}} \\
 \hline
 \#starsInSky\mathcal{C} \geq upperlimit \\
 starsInSky' = starsInSky\mathcal{C}' \cup \{newStar\mathcal{C}'?\} \\
 message! = starArrayFull
 \end{array}$$

The concedes relation allows us to specify the conditions where the behaviour of our two models differs, and to formally relate the behaviour of the models in such a circumstance. In this instance, the concrete model will behave differently when it has stored as many stars as it possibly can. Hence, we indicate that the concedes relation can only hold when the upper limit has been reached, and in that situation the state of the concrete model will be equal to that of the abstract model without the new star and an 'array full' message will be generated.

We consider now the proof of the retrenchment relationship between the two models. The relational version of the retrenchment proof obligations are presented below.

- $\forall v : \mathcal{C} \mid v \in \mathcal{C}I \bullet \exists u : \mathcal{A} \mid u \in \mathcal{A}I \bullet (u, v) \in G$
- $\forall u : \mathcal{A}; v : \mathcal{C}; i, j : inputs$
 $\mid (v, i) \in \text{pre}\mathcal{A}Op \wedge (u, i, v, j) \in W_{Op} \wedge (u, v) \in G$
 $\bullet (u, j) \in \text{pre}\mathcal{C}Op$

- $\forall u : \mathcal{A}; v, v' : \mathcal{C}; i, j : \text{inputs}; p : \text{outputs}$
 $| (u, i) \in \text{pre } \mathcal{A}Op \wedge (v, j, v', p) \in \mathcal{C}Op \wedge (u, v) \in G$
 $\quad \wedge (u, i, v, j) \in W_{Op}$
- $\exists u' : \mathcal{A}; o : \text{outputs}$
 $| (u, i, u', o) \in \mathcal{A}Op$
- $((u', v') \in G \wedge (u, i, u', o, v, j, v', p) \in O_{Op})$
 $\quad \vee (u, i, u', o, v, j, v', p) \in C_{Op}$

The initialization proof obligation for retrenchment is exactly the same as for refinement, and as our current model is very similar to the train yard model, we will not illustrate the discharge of this trivial obligation for our example. Similarly, the termination proof obligation, whilst not identical to that of refinement, can be discharged trivially and in a manner very like that of the previous example. We shall therefore concentrate on the applicability proof obligation where the main differences between refinement and retrenchment arise.

We define first the precondition of *discoverStarC* (again derived using the method in [WD96]).

$$\frac{\text{pre } \text{discoverStarC} \quad \text{skyC} \quad \text{newStarC?} : \text{stars}}{\text{newStarC?} \notin \text{starsInSkyC}}$$

We have now defined all the required terms for the proof obligation and can substitute them to produce the following theorem.

$$\begin{aligned}
& \vdash? \forall u : \{starsInSky : \mathbb{P} stars \mid \text{true}\}; \\
& \quad v, v' : \{starsInSkyC : \mathbb{P} stars \mid \text{true}\}; i, j : stars; p : MESSAGES \\
& \quad | (u, i) \in \{starsInSky : \mathbb{P} stars; newStar? : stars \\
& \quad \quad | newStar? \notin starsInSky\} \\
& \quad \wedge (u, v) \in \{starsInSky, starsInSkyC : \mathbb{P} stars; \\
& \quad \quad | starsInSky = starsInSkyC\} \\
& \quad \wedge (v, j, v', p) \in \\
& \quad \quad \{starsInSkyC : \mathbb{P} stars; newStarC? : stars; \\
& \quad \quad starsInSkyC' : \mathbb{P} stars; message! : MESSAGES \\
& \quad \quad | newStarC? \notin starsInSkyC \\
& \quad \quad \wedge (\#starsInSkyC < upperlimit \\
& \quad \quad \Rightarrow starsInSkyC' = starsInSkyC \cup \{newStarC?\} \\
& \quad \quad \wedge message! = starAdded) \\
& \quad \quad \wedge (\#starsInSkyC \geq upperlimit \\
& \quad \quad \Rightarrow starsInSkyC' = starsInSkyC \\
& \quad \quad \wedge message! = starArrayFull)\} \\
& \quad \wedge (u, i, v, j) \in \{starsInSky : \mathbb{P} stars; newStar? : stars; \\
& \quad \quad starsInSkyC' : \mathbb{P} stars; newStarC? : stars \\
& \quad \quad | newStar? = newStarC?\} \\
& \bullet \exists u' : \{starsInSky : \mathbb{P} stars \mid \text{true}\} \\
& \bullet ((u, u', i) \in \{starsInSky, starsInSky' : \mathbb{P} stars; newStar? : stars \\
& \quad | newStar? \notin starsInSky \\
& \quad \wedge starsInSky' = starsInSky \cup \{newStar?\}\}) \\
& \quad \wedge ((u', v') \in \{starsInSky, starsInSkyC : \mathbb{P} stars \\
& \quad | starsInSky = starsInSkyC\} \\
& \quad \vee (u, i, u', v, j, v', p) \in \\
& \quad \quad \{starsInSky : \mathbb{P} stars; newStar? : stars; \\
& \quad \quad starsInSky' : \mathbb{P} stars; starsInSkyC : \mathbb{P} stars; \\
& \quad \quad newStarC? : stars; starsInSkyC' : \mathbb{P} stars; \\
& \quad \quad message! : MESSAGES \\
& \quad \quad | message! = starArrayFull\}) \\
& \quad \vee (u, i, u', v, j, v', p) \in \\
& \quad \quad \{starsInSky : \mathbb{P} stars; newStar? : stars; \\
& \quad \quad starsInSky' : \mathbb{P} stars; starsInSkyC : \mathbb{P} stars; \\
& \quad \quad newStarC? : stars; starsInSkyC' : \mathbb{P} stars; \\
& \quad \quad message! : MESSAGES \\
& \quad \quad | \#starsInSkyC \geq upperlimit \\
& \quad \quad \wedge starsInSky' = starsInSkyC' \cup \{newStarC?\} \\
& \quad \quad \wedge message! = starArrayFull\})
\end{aligned}$$

Firstly, we apply the set comprehension one-point rule (see D.4) to the sets representing our schema definitions and reduce the theorem to the following. (We use simplification (see D.3) to remove redundant terms.)

$$\begin{aligned}
& \vdash? \forall u, v, v' : \mathbb{P} \text{ stars}; i, j : \text{stars}; p : \text{MESSAGES} \\
& \quad | j \notin v \wedge u = v \\
& \quad \wedge (\#v < \text{upperlimit} \Rightarrow v' = v \cup \{j\} \wedge p = \text{starAdded}) \\
& \quad \wedge (\#v \geq \text{upperlimit} \Rightarrow v' = v \wedge p = \text{starArrayFull}) \wedge i = j \\
& \quad \bullet \exists u' : \mathbb{P} \text{ stars} \\
& \quad \quad \bullet i \notin u \wedge u' = u \cup \{i\} \wedge ((u' = v' \wedge p = \text{starAdded}) \\
& \quad \quad \quad \vee (\#v \geq \text{upperlimit} \wedge u' = v' \cup \{j\} \wedge p = \text{starArrayFull}))
\end{aligned}$$

We can then apply the existential quantification one point rule (see D.1) to eliminate u' , giving the following.

$$\begin{aligned}
& \vdash? \forall u, v, v' : \mathbb{P} \text{ stars}; i, j : \text{stars}; p : \text{MESSAGES} \\
& \quad | j \notin v \wedge u = v \\
& \quad \wedge (\#v < \text{upperlimit} \Rightarrow v' = v \cup \{j\} \wedge p = \text{starAdded}) \\
& \quad \wedge (\#v \geq \text{upperlimit} \Rightarrow v' = v \wedge p = \text{starArrayFull}) \wedge i = j \\
& \quad \bullet i \notin u \wedge ((u \cup \{i\} = v' \wedge p = \text{starAdded}) \\
& \quad \quad \vee (\#v \geq \text{upperlimit} \wedge u \cup \{i\} = v' \cup \{j\} \wedge p = \text{starArrayFull}))
\end{aligned}$$

We can then apply the universal quantification one point rule (see D.5) to eliminate i and u , and apply our laws of simplification (see D.3) to attain the following (that is, we simply assume the post-transitional abstract-concrete state pair does not belong to the retrieve relation, and therefore must belong to the concedes relation).

$$\begin{aligned}
& \vdash? \forall v, v' : \mathbb{P} \text{ stars}; j : \text{stars}; p : \text{MESSAGES} \\
& \quad | j \notin v \wedge (\#v < \text{upperlimit} \Rightarrow v' = v \cup \{j\} \wedge p = \text{starAdded}) \\
& \quad \wedge (\#v \geq \text{upperlimit} \Rightarrow v' = v \wedge p = \text{starArrayFull}) \\
& \quad \bullet p = \text{starAdded} \\
& \quad \quad \vee (\#v \geq \text{upperlimit} \wedge v \cup \{j\} = v' \cup \{j\} \wedge p = \text{starArrayFull})
\end{aligned}$$

We are left with a trivial obligation. When the set is smaller than the required size the new star is added (producing a success message in the concrete model), and when the maximum size has been reached, it is not (producing

a failure message in the concrete model). Our ramifications, have therefore, allowed us to show that our contracted model is a valid retrenchment of our abstract specification. This allows different members of a team to use the specification at a level of abstraction suitable for their role in the system's development, and to know exactly how that specification is related to others used within the development.

2.7 Concluding Remarks

In this chapter we have introduced the formal techniques of refinement and retrenchment. Refinement is a well established technique that has been put into practice in a number of industrial situations. However, refinement has limitations that reduce the scope of its applicability. Retrenchment is a newer technique designed to overcome these limitations and extend the scope for formal development. When retrenchment and refinement are used together they provide a powerful tool in the rigorous development of systems that provide a formal approach to all stages of the development lifecycle, from requirements gathering to implementation.

Chapter 3

Mechanical Support for Formal Methods

This chapter describes the necessity to provide effective, mechanical support to developers using formal methods. We begin by examining some of the debates regarding the practicability of using formal techniques in a standard commercial development. We show that suitable tool support is considered to be one of the most critical factors in the improvement of formal methods' image, and therefore its uptake. We then consider the individual tasks involved in a formal development, and consider the extent to which mechanization can be useful in each. We also examine some of the more common formal specification methodologies, and briefly detail the tool support available for each in the tasks we have outlined. We then proceed to describe the options we have considered when looking to provide mechanized support for retrenchment. We examine existing toolkits and their potential for extension; we look at theorem provers and the possibility of embedding a retrenchment concept within one; we also look at the advantages and disadvantages of creating a new stand-alone tool. Finally, we evaluate the options, and detail the choices that we have made.

3.1 Introduction

The debate over the benefits of using formal methods (and the most effective ways to do so) continues to rage [BH95b, BH06, CWA⁺96, LG97]. It is not our intent to use this thesis to justify the use of formal methods in system development, as to us, the benefits are clear. We need to consider, however, what it is about formal methods that provokes such debate, and why some parties are so opposed to their use.

Many of the claims that formal methods' detractors make are detailed in [Hal90] and [BH95a], and it appears from the literature that the 'myths' presented in these papers continue to haunt the use of formal methods. We consider some of these perceived problems and show that suitable tool support can be a great aid in 'solving' these issues.

- *Formal methods involve complex mathematics* — Tools can reduce the complexity in creating specifications. Applications can be created that are similar to the integrated development environments of programming languages; providing an explanation of notation as its used and generating simpler representations of that notation. Furthermore, the increasing power of model checkers, automated theorem provers and interactive proof assistants can make the verification of specifications a semi-automated task.
- *Formal methods increase the cost of development* — The use of tools in a formal development can automate many of the tasks that consume development budgets. Of particular note are model checkers, which can test every possible state in a finite system (eliminating the need for manual testing) and test case generators (in instances where model checkers can't be used) where tools can automatically produce testing scripts from a specification.
- *Formal methods are incomprehensible to clients* — Tools make the translation of a formal specification into more suitable notations an automated – and therefore, cost free – task. Tools are already being

developed that can translate between formal notations, and ubiquitous notations such as UML. The very nature of formal methods' rigid structures, make the interpretation of its notations into less formal representations, an easily automatable process.

- *Formal methods delay the development process* — As we have mentioned previously, tools supporting formal methods are able to significantly reduce the time required for testing. Furthermore, the use of formal methods' tools can make the re-use of components – within a system – a simpler and more traceable process.
- *Formal methods are not supported by tools* — The increasing availability of suitable tools will debunk this myth.
- *Formal methods mean forsaking traditional engineering design methods* — Increasingly formal methods tools are designed to work alongside traditional methods, providing interfaces between the specifications of those systems components that are specified formally and informally. Tools also make the adaptation of a formal specification – due to changing requirements – a considerably more feasible option. In some cases changes can be propagated through a specification automatically, with any inconsistencies being highlighted immediately.

We believe from this, that we are able to conclude that increased tool support will be able to lessen the impact of these issues, and therefore make formal methods more accessible. Can we conclude, however, that the use of tools makes an impact on the creation of successful industrial software? In [BH97], Bowen and Hinchey present a number of examples of formal methods being used in industrial applications. When we examine this list of studies we can quickly determine that the common feature to these successful developments has been the use of a toolkit. For example, the design of a Storm Surge Barrier Control System in the Netherlands used tools to create a formal specification, and then a model checker was used to automatically verify that all of its interfaces were correct. Elsewhere, the whole of a development for the French Population Census of 1990 was performed with formal

methods, and supported by tools at all stages – from specification through to implementation – and then verified and validated with those same tools. We could argue, therefore, that the evidence suggests that tool support can significantly increase the likelihood of success in applying formal methods to an industrial development.

Heitmeyer [Hei97] goes even further, indicating that automated assistants have been used to re-examine specifications for an Operational Flight Program that had previously been believed to be valid; this process uncovered a number of typing errors, as well as revealing six cases of undesired non-determinism. These errors could have been disastrous, and this evidence indicates that tool support not only makes the specification and verification process easier, but also reveals errors in the most thoroughly checked specifications. Heitmeyer goes on to describe further examples (torpedo control, aircraft collision avoidance) where automated methods have caught significant errors that have not been exposed despite extensive manual inspection.

Despite Bowen and Hinchey, and Heitmeyer, reporting the success of tools in real world projects, they both conclude that while mechanical assistance is already proving of use in the industrial world, improved (particularly more user-friendly) tool support is vital if formal methods are going to be increasingly used. One may argue that these papers were published some time ago, but Heitmeyer has recently revisited her examination of formal methods in the industrial world [Hei05], and found that while mechanized assistance has become more widely available, there is still a long way to go before it is able to be used on an everyday basis by ordinary software engineers. If anything, the need for suitable tool support is growing.

Now that we have established that satisfactory tool support is widely considered to be a necessity for increasing uptake of formal methods, we consider those areas in which mechanical assistance can be a boon. Formal methods can be used to various degrees within a development: from simply creating an unambiguous specification, to rigorously transforming all of a system's requirements into code. Similarly therefore, tools have been created that support formal methods to different extents. In the remainder of this section we break the formal development process into a number of blocks,

and consider the tool support available for each.

3.1.1 Specification and Syntax Checking

Any formal method relies on the use of a formal syntax and semantics that can be used to describe a system's behaviour. Typically, when we talk about formal specification, we refer to the use of a syntax based on mathematics rather than natural language. Creating a formal specification is considered to be a worthwhile process (even without validation and verification) as it forces a specifier to think about the entire behaviour of each component within their system, and its interactions with the other components that belong to that system. This necessity to consider the entirety of the behaviour often reveals errors that may have been missed with a less formal specification. Typically these errors are due to ambiguity, contradiction and incompleteness, all of which can be more easily exposed with a rigorous approach.

A number of notations have been created that allow a specification to be expressed formally. We discuss a few of the most popular below.

The B-Method [Abr96, Sch01] is a formal method based around the human-readable abstract machine notation (AMN), and machine readable generalized substitution language (GSL). The B-Method is particularly adept at describing large and complex systems. This is due to its focused, model-oriented approach where small abstract machines with a very narrow purpose are pieced together to form increasingly larger abstract machines until they describe vast and intricate systems. This not only makes the initial development easier, but makes the entire system maintainable, and encourages the re-use of individual machines. The B-Method's heavily structured approach and its good tool support, has led to it being of great interest in the commercial world. Of particular note, is its use in the development of the critical elements of a control system for the fully automated Paris Metro Line 14 [Des98, BDM98].

Z [Cur99, Spi92b, WD96] is the name for both a formal specification methodology and its notation (an alternative, object oriented version of the notation is known as Object-Z [Smi00a]). In 2002 the notation for Z was

standardized in [ISO02]. A specification in Z involves formal, mathematical statements mixed with informal descriptions of those statements. Z is based on Zermelo-Fraenkel set theory and is strongly typed. The distinguishing feature of Z is the use of schemas, and the associated schema calculus. Schemas in Z, form the basic building blocks of specifications. Unlike the abstract machines of B however, the schema does not represent a model (although it may do so), and may be used to indicate the state of a model, an individual operation, or an entire system. This lack of rigid structure, and its flexible notation have made Z ideal for research into the nature of formal methods, and it is commonly used in academia not just for specification, but in the development of new techniques (for example, automatic translation from UML to formal specification [BF98, EC98]), and to illustrate existing techniques (for example, refinement's strengths were shown through the specification of the Mondex purse [SCW98, SCW00]). Having said this, Z has also been applied in industrial situations, and it is probably fair to say that it is currently the mostly widely used specification language.

The Vienna Development Method (VDM) [AI91, Jon90, She95] is a formal method that allows formal specification with VDM-SL [Daw91], or the object oriented VDM++ [FLM⁺05]. Like Z, VDM-SL has been standardized, and achieved ISO standardization in 1996 [VDM]. VDM is one of the oldest formal specification notations, and has been widely used in academia and industry since the 1980s. As with Z, the specification is broken down into informal description, and formal statements; in VDM, however, these formal statements – whilst still based on typed set theory – are closer to the syntax of programming languages. All of the definitions in a VDM model (both formal and informal) are gathered into a module which – since the introduction of VDM-SL – can be structured to form a complete specification (it can be argued that formation of modules is very similar to the construction of classes in object-oriented programming languages). Data reification is VDM's alternative to data refinement (in practice, only a nominal alternative), and involves finding a more concrete representation of the abstract data types used in a specification. This technique can be used to reduce an abstract specification into an implementable definition in the same way as

using refinement with B or Z. In fact, the principal difference between VDM and the other methodologies we have discussed, is its lack of a strong typing system. This leads from the fact that it does not consider all types to be sets, and hence it is not always possible to assign every expression a unique type.

Writing a specification using any of these notations can be a tricky business, and syntactical errors are likely to encroach unless we are either very careful, or use some form of tool to check the consistency of our syntax. Fortunately, a multitude of tools exist for each of these notations (we will discuss some examples of these tools in section 3.2.1), and these assist in specification – providing a specification equivalent of an integrated development environment (IDE) – and syntax checking – ensuring that the specification belongs to the language defined by the particular notation’s grammar.

3.1.2 Type Checking

Type checking a specification provides an extra ability to expose simple errors in a formal development. Typing a specification typically reveals inconsistencies in that specification that cannot be normally revealed through syntax checking alone. It is particularly useful in showing mistakes that cause syntactically correct specifications to have no – or inconsistent – semantics.

Both the B-Method and Z are strongly typed, and their definitions in [Abr96] and [ISO02] respectively provide strict rules for type check specifications in the two notations. As such automating the typing process is a relatively simple task, and type checking is provided by most of the tools which are capable of syntax checking the notations.

With VDM the lack of strong typing makes type checking a more difficult proposition. It is impossible for a machine to guarantee to be able to type a VDM specification statically as we do not have the basic premise that all types are sets, this makes it extremely difficult to automate the comparison of union types. To overcome this, we can instead classify models to be either definitely correct, definitely wrong, or undeterminable, and when we have a model that is undeterminable we must create extra proof obligations to ensure

our model is correctly typed. Despite this more complicated approach, most of the VDM tools we detail below handle type-checking, but in this instance we are not necessarily able to reveal bugs in the specification until we attempt to discharge the typing proof obligations.

3.1.3 Animation – Validation

Boehm [Boe81] defines verification, and validation as follows. We will use these definitions in this thesis.

Verification: “Are we building the product right?”

Validation: “Are we building the right product?”

With a lot of formal methods, the focus is on the verification of a specification, but while this ensures that our application will be a correct implementation of that specification, it does not ensure that the specification actually meets the original requirements. Users of formal methods need therefore, a way to check that a specification actually behaves in the way that the specifiers believe it does.

Animation [BDMW97, PLT00] is a technique that allows us to validate a specification, and ensure that any application developed rigorously from that specification will meet our original requirements. The process enables a specification to be brought to life so that we are able to test that it behaves as we expect. Animation can usually be performed interactively or through the use of a script. For example, we may prepare a test script from the requirements of our model, where we describe a valid initial state and then state the changes that should be made to the state when various combinations of operations are invoked upon it. Once our script has been prepared, and our expected results outlined, we apply the logic of our specification to determine whether the same results would be achieved. Of course, like all testing methodologies, animation can only validate that the specification meets our requirements within the scope of situations in which it is performed. Despite this, animation allows us to eliminate errors and inconsistencies in the specification at an early stage, to provide an immediate understanding of

the specification's behaviour and certainly provides some assurance when bridging the gap between informal requirements and formal specification.

Obviously, animating a specification by hand can become an incredibly complex task, and there have been many tools created that provide assistance in the process. For example, animation tools exist that support the B-Method (the B-Toolkit [WB98]), Z (PiZA [HOS97] and Jaza [Led06]), and VDM (KIDS/VDM [Led95]).

3.1.4 Model Checking – Verification

Model checking [CGP99, CS01] is used to determine whether a finite system satisfies a certain property. Originally, it was principally used in the development of hardware, but due to its success it is increasingly used in the verification of software. Given a model of a system and a specification of its properties, a model checker should be able to verify that the properties hold in every single state of that system, and if not provide a counter-example to show a state where the properties fail. An obvious advantage of the model checking system is that theoretically – provided you can get the model and properties into a suitable format – the process is fully automatable; a model checker can simply apply an exhaustive check on every possible state of the system. There are however, some limits to this automation as not all properties can be determined through the application of an algorithm, and we can only apply the model checker to a finite model (or some subset of an infinite model).

Attempts have been made to use all of the specification languages we are considering to express the system properties input to model checkers. ProB [LB03] is the most complete of these and comprises both an animation and model checking tool supporting the majority of the B-Method's notation. Jackson's Nitpick [Jac96, JNW99] and Alcoa [JSS00] are tools that enable partial specification and automatic model checking, but which sacrifice breadth of coverage and expressive power by only allowing specification in a – relatively small – subset of Z. Jackson has also investigated using VDM [Jac94], but found the same problems encountered with Z and chose to use

that notation in his tools.

3.1.5 Deductive Reasoning (Proof) – Verification

Deductive verification systems use axioms and proof rules to prove the correctness of a model. Initially such proofs were created by hand, increasingly however, automated theorem provers and interactive proof assistants (for example, HOL [GM93], PVS [SCR96] and Vampire [RV01]) have been able to reduce the work required. An advantage of the deductive system is that it is possible to reason over infinite state spaces, whilst model checkers are clearly limited to a finite state space.

The creation of proof obligations for specifications using the B-Method is extremely structured and relatively simple. The proof obligations for a B-Method construct are generated using the process of ‘substitution’ – based on the weakest precondition calculus – and produce much simpler statements than the before-after predicates used in our other specification languages. Proof obligations can be derived from a GSL definition of a machine, and as that can be derived from the AMN definition of the same machine, the process of creating proof obligations can be easily automated (whether for machine consistency or refinement). As such, most B tools provide the ability, to generate proof obligations, and most also offer simple proof assistants (for example the B-Toolkit’s, B-Tool [Abr91], and Atelier-B and B4Free’s, Click’n’Prove [AC03]) that allow the user to tackle those obligations.

Unlike the B-Method, Z does not provide any set of standard proof obligations in fact there is very little guidance – associated directly with Z – that conveys what needs to be proved in order to verify a specification. Z’s flexibility has allowed it to be used in a variety of ways and as such the generation of proof obligations is associated with the specific use and not the methodology as a whole. The notation does allow a specifier to incorporate conjectures directly within a specification, but the nature of these conjectures is left entirely to the specifier. This lack of structure has led to little tool support for deductive reasoning over Z specifications. The now defunct Z/EVES tool (see section 4.2.4) was probably the most advanced, but

currently the CADiZ tool [Toy] and ProofPower [Art91, KA96] provide the most useful support (CADiZ is able to reason about any conjecture within a Z specification).

In VDM it is necessary to show that every function operation is both implementable and satisfiable. Similar proof obligations exist to show that pairs of reified function operations behaved identically. Although these proof obligations seem relatively easy to derive from the precondition and post-condition of each operation, like Z, there is little mechanized support for deduction reasoning using VDM. The Mural proof assistant [FEG92, JJLM91], does however, contain a VDM support tool that allows a user to reason about specifications and also reifications between specifications, automatically generating the required proof obligations, and providing a fairly powerful proof assistant that can be used to discharge those obligations.

3.1.6 Test Case Generation

Where the more formal verifications of model checking and deductive reasoning are not used in a formal development, it will be especially necessary to test the software developed from a formal specification (of course, even a formally verified specification is typically subject to some extent of testing). These tests usually involve the creation of test cases from the specification, each of which contain a script and the expected results from the application. While it is possible for these test cases to be generated manually, it is preferable to be able to generate them automatically. Not only does an automated approach allow the creation of more and deeper tests (given a constant overhead), but the generation of test cases can also expose errors in the specification itself. When test cases are generated manually, any misunderstanding of the specification will be misrepresented in the test cases; when the test cases are generated automatically they will produce tests that expect the results exactly as they are detailed in the specification, ensuring that the same misunderstandings that have passed into the application's code do not also pass into its test cases.

Somewhat surprisingly none of the major B-Method tools currently support test case generation. Various tools exist that can generate test cases from Z specifications including the one described in [HNS97]. Similarly, tools exist that can automatically create test cases from a VDM specification including the one presented in [Dro99].

3.2 Options for Providing Retrenchment's Mechanized Support

In the previous section we established that for any formal technique to be practicable, tool support is required. We have also described a number of areas in which mechanical assistance can be most useful. We consider next the best methods of providing tool support for retrenchment, and examine the different ways in which this support could have been provided. As we have indicated above, there are many tools already available that support refinement and other formal techniques, and we considered the implications of extending one of these. However, if our focus was on experimenting with retrenchment itself – and its relationship with other formal techniques – then perhaps we could have considered embedding a system directly within a theorem prover. Finally, we needed to consider that if neither of these solutions would fulfil all our needs, and we were left with no alternative but to create our own tool from the ground-up, then what form should a configurable and extendable tool take? We present the results of our research in the following section.

3.2.1 Extending an Existing Toolkit

We begin by presenting details of some popular existing toolkits that provide support for formal methods, and we consider the potential of extending them to incorporate retrenchment. Our focus is on tools that provide both formal specification and verification, rather than those that deliver what is known as

‘formal methods light’ [Jon96]¹ where the focus is primarily on specification (see 3.1.1 above).

The B-Toolkit

The B-Toolkit [Abr91, LH96, Wor96] is a collection of tools created to provide support for the B-Method, and is a proprietary piece of software developed by B-Core (UK) Ltd. The toolkit provides a single interface from which the user is able to pursue the usual activities of software engineering (notably specification, type-checking, animation and proof). The kernel of the toolkit is the B-Platform (also known as the B-Tool) which is an inference engine, that is used to perform syntax checking, type checking and even theorem proving (it can also produce a variety of side effects; for example it has the ability to write data to files). The B-Platform does not maintain its own state, and a separate construct manager keeps track of the various machines and the progress the user has made towards developing them.

An initial foray to provide mechanized support retrenchment was made by the author, and we attempted to extend the B-Toolkit in order to provide support for retrenchment. While this process is described in detail in [Fra05, BF05], we will present a short outline of the project and the principal problems faced in the remainder of this section.

In the standard version of the B-Toolkit the basic idea is that, given a specification, we create one or more abstract machines (AM) to define the behaviour of our model. For each of these machines the development proceeds linearly, and we gradually add more and more detail creating a chain of refinement machines (RM). When the level of detail in the model is sufficient we create an implementation machine (IM) – which is a refinement of our most concrete refinement machine – which completes our development chain, and can be easily converted to programming language instructions (see figure 3.1 on the following page).

In order to incorporate retrenchment, the syntax of the B-Method was

¹Also ‘lightweight formal methods’ or ‘formal methods lite’.

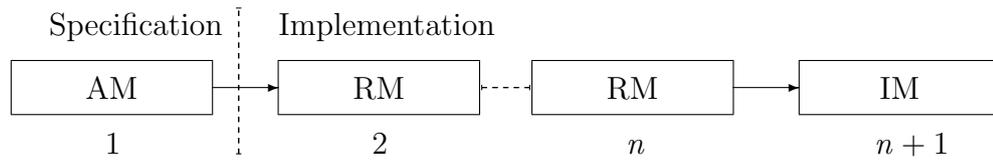


Figure 3.1: Standard B-Toolkit Development Structure

augmented with the definition of a new construct, the retrenchment relationship (RR). Initially we considered the use of a retrenchment machine similar to the refinement machine, but there were significant disadvantages to this approach, principally the reduction in clarity. Unlike a refinement machine, a retrenchment machine could have been used as a top level specification by a developer, and the presence of retrenchment details would have caused needless confusion. We therefore felt that all specifications belonged in abstract machines, and that the specification of the retrenchment belonged in a distinct construct.

The consequences of this decision were significant in our attempts to manipulate the B-Toolkit. The standard B-Toolkit views a machine development as a sequence that must contain only a single abstract machine (at the start), and a single implementation machine at the end. Unfortunately, the whole of the system is based around this assumption, and the tool proved most resistant to change. After much effort, we were able to incorporate multiple abstract machines within a single machine development and introduce the concept of a relationship that was distinct from a machine, but it was not feasible – with our resource constraints – to challenge the linear approach to development (where a rewriting of the construct manager, and much of the toolkit binaries would have been necessary).

This left us with the development structure shown in figure 3.2 on the next page. A development involving retrenchment has two or more abstract machines that are linked by retrenchment relationship, where we move from an abstract ‘idealized’ specification, to a more concrete specification that incorporates our implementation requirements. When that concrete specification is reached, the standard process – refining until an implementation

machine is produced – commences. Of course, this meant that we were unable to explore some of the more interesting aspects of retrenchment; for example, it was not possible to show how multiple retrenchment could be used to relate the specifications of two distinct features and their interaction; nor the inverse where two sets of implementation specifications could be derived from a single abstract specification.

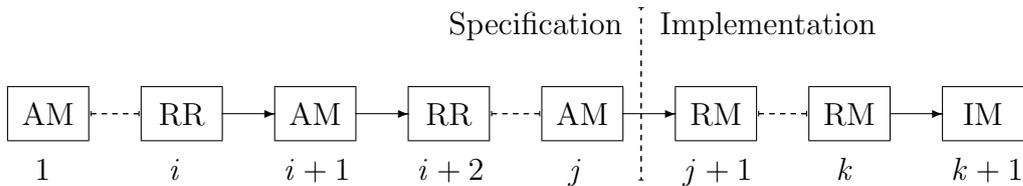


Figure 3.2: Extended B-Toolkit Development Structure

Whilst appearing to be only a minor alteration, this change to the development structure required major changes at every point, and a considerable proportion of our time was required to update the construct manager’s referential integrity facilities (where a change to one machine, would cause state changes in others). The changes required to the syntax checking process and the proof obligation generator were minor in comparison and were made with relative ease.

Once development was complete the extended tool was more than capable of handling the specification of retrenchment relationships (obviously, within the above constraints). We were able to check the syntax and typing of constructs and generate the correct retrenchment proof obligations. When we proceeded to examine a less trivial case study however, the limitations of the tool became apparent.

We sought to mechanize the retrenchment case study presented in [BP02]. In this paper, Banach and Poppleton outline a simple machine describing the behaviour of a telephone system and then introduce features, before showing how the specifications incorporating the features were retrenchments of the original plain system² (and a specification containing all of the features

²It was necessary, of course, to perform each of these in separate projects, as we were unable to involve a single abstract machine in more than one retrenchment relationship.

was retrenchment of all three other specifications). The specifications, and retrenchments were translated into the B syntax, and we attempted to prove them with the B-Toolkit's proof facilities. Unfortunately, the prover did not appear to be up to the task, and the proof process – using the power of a standard desktop machine – stretched the capabilities of the system beyond its limits in all but the simplest of the retrenchments. We tracked this problem and discovered that the source was the B-Platform's naïve approach to disjunction. Whilst disjunction may be considered fairly common in refinement proofs, it is, of course, ubiquitous in retrenchment proofs, and this was a significant issue.

In summary, we can conclude that while it was possible to incorporate some aspects of retrenchment within the B-Toolkit, we were unable to take advantage of its differences to refinement, and we were certainly unable to experiment with the relationship itself. The architecture of the B-Toolkit was heavily linked to monolithic refinement – which was of course its original objective (indeed it could be said that the fact that a B-Method retrenchment machine can only refine a single abstract machine imposes this view) – and was not particularly suited for this sort of extension. Furthermore, the associated theorem prover was unable to discharge any but the simplest of retrenchment proof obligations.

When considering another attempt to extend the B-Toolkit we must remember to take into consideration that it is proprietary software, and that whilst B-Core were very kind to allow us to examine its potential for extension, this fact will restrict the use of any tool resulting from an extension. These restrictions are twofold, the first is the obvious restriction that B-Core are not going to want us to distribute their extended toolkit without collecting their usual licence fee as there is no way in which we would be able to separate our functionality from the toolkit's core functionality. The second restriction leads from the changes that we have made to the toolkit. We were not able to make a modular extension to the toolkit so in some instances we have made alterations to its kernel, and throughout its code. However, there is no guarantee that the toolkit will not be developed further by B-Core so our extension would need to be either brought into the core view of the tool,

or a branch created. Neither of these alternatives would be particularly attractive: B-Core are unlikely to maintain code which they have not created and will not presently provide them with additional revenue, neither is it likely to be possible for us to maintain the parallels between our extended toolkit and the original toolkit, once that original toolkit has been subject to further development. Since we completed this project, for example, B-Core have extended the B-Toolkit so that it is possible to view a diagrammatic overview of constructs within a project. Obviously, B-Core would not desire the extra work required to incorporate retrenchment relationships within this view, and we would have difficulties replicating their functionality.

Atelier-B and B4Free

Atelier-B [Cle, Buc98, ML02] is a set of tools – supporting the B-Method – that are very similar to those provided by the B-Toolkit. Its use has traditionally been restricted to the French market, with the B-Toolkit being used in other territories. The development of Atelier-B is performed by ClearSy, and is supported by RATP, SNCF and INRETS, and the tool thrives through their co-operation which also leads to its application in the development of many French rail systems. Originally the tool was produced by Digilog and GEC Alstom Transport in co-operation with the creator of the B-Method, Jean-Raymond Abrial.

Like the B-Toolkit, specification, type checking, animation and proof facilities are provided, amongst other project management tools. It is difficult to determine the internal structure of Atelier-B, or its suitability for extension as it is a proprietary application, and special permission would be required to examine its workings. It is imagined, however, that the structure of Atelier-B will inevitably be very similar to that of the B-Toolkit, and we would likely face the same problems with its extension as we did in the above work. It was felt, therefore, it was not worth pursuing an investigation of Atelier-B; this view was enhanced by the fact that as the tool was closed-source, we would be unable to release the extended tool other than to existing customers of Atelier-B, and we would also be exposed to the same branching problems we

have discussed above.

B4Free [B4F] is a version of Atelier-B that is provided free by ClearSy for academic use. B4Free provides a similar tool set to Atelier-B, and includes access to its proof assistant, Click'n'Prove [AC03]. Whilst use of this tool is free, access to its source code is still restricted, and the terms of use³ make clear that this is unlikely to change in the near future. It was felt, therefore, that the possibilities for extending this toolkit were extremely limited.

RODIN

RODIN [CJO⁺05] (Rigorous Open Development Environment for Complex Systems) is a project to produce methods and open-source tools for handling the formal specification and design of complex systems. The project seeks to tackle complexity from two angles. Firstly, formal methods will be used to reduce complexity through clear thinking and rigorous verification and validation. Secondly, fault tolerance design techniques will be used to design architectures that are capable of handling unpredictable changes to their environments.

One of the deliverables of the projects is a tool set (also referred to as RODIN) that supports the ‘unified methodology’ derived from the combination of formal methods and fault tolerance design. This toolkit [Voi06] will be comprised of a number of plug-ins to the Eclipse integrated development environment [Ecla, Eclb], and will contain tools that are capable of model construction, model simulation, model checking, verification, testing and code generation. The notation and methodology used within these tools is known as Event-B.

Event-B [MAV05, Lec02] is an extension to the B-Method that has been created for system modelling, and distributed system development⁴. An Event-B specification replaces the abstract machine of the traditional B-Method with a model that is split into a static part (for example, containing

³“This software is confidential and copyrighted. ... You may not modify, decompile, or reverse engineer software. ...” <http://www.b4free.com/public/licence.php>

⁴In many ways Event-B is a formalism that changes the perspective on a specification in the same way that Back’s [BKS88] action systems provide a different view of distributed implementation.

constants, and properties) called the context, and a dynamic part called the machine (for example, containing state variables and operations). Unlike a standard B-Method machine where operations are invoked by other machines, here operations are invoked through events. An Event-B model can be refined in a similar way to a normal abstract machine, although it is possible to refine the context and machine separately. Event-B also allows a machine to refine more than one other machine, and also to be refined by more than one other machine.

Event-B handles refinement somewhat differently to other systems – that we have discussed – in that the refinement relationship is actually being used in the construction of a specification, and that the events added as part of a refinement are not necessarily hidden and may actually be considered genuine system requirements. In chapter 2, we discussed whether using refinement in this way was the best approach to system development. Indeed, there is some discussion of this point within the RODIN documentation itself (see section 4.6.2 of [HJJ⁺04]), where the use of refinement as the sole formal relationship is questioned, and the use of retrenchment as an additional method for expressing formal relationships is subject to further investigation.

In many ways, the RODIN tool sounds ideal for extension to support retrenchment; it is open-source and uses a plug-in approach giving the sort of flexibility that other toolkits do not offer. Unfortunately, the RODIN project was only begun in 2004, and even as this thesis is written, the tool associated with the project is in the ‘alpha’ stage of development and not yet sufficiently mature for adaptation to retrenchment. We hope, however, that given a modular approach to our current development we would be able to re-use much of our work in the future creation of a RODIN plug-in.

ProB

ProB [LB03, LT05, LBP05] is an animator and model checker that supports a large subset of the B-Method’s notation. It can fully animate many B specifications and can be used to systematically check those specifications for errors. Furthermore, refinement machines can be specified and animated,

and the refinement relationship can be made subject to the same exhaustive correctness examination as a specification. As with all model checkers, only a finite model can be examined, so it is necessary either to use finite given sets or to specify ranges when using infinite sets. Every state that violates a constraint will be highlighted by the ProB tool, and the counter-example presented in detail.

ProB was initially developed using SICStus Prolog (based on a previous CSP⁵ animator [Leu01]), but a version written for Java is currently being implemented. Similarly, work is in progress on a version of the tool that is capable of animating and checking specifications in the Z notation.

There exists potential to incorporate retrenchment within ProB, and provide equivalent animation and checking facilities that are currently available for refinement. ProB is freely available for non-commercial and academic use, but the application is closed-source, which again adds complications to an extension for retrenchment. Furthermore, whilst ProB is capable of verifying simple models with finite state spaces, it is not able to prove the correctness of models with large state spaces never mind those with infinite state spaces. Many of the applications that have been touted for the retrenchment technique have involved the relationships between continuous, infinite systems and their discrete, finite (and implementable) models. It could be argued that a tool only capable of reasoning over a small set of states will have limited practical use in projects involving retrenchment.

CADiZ

CADiZ [JMT91, TM95, Toy96] is a tool that aims to give support to the Z methodology, and is the first tool providing support for the ISO standard notation [ISO02]. The toolkit provides specification editing, scope and syntax checking and supports deductive reasoning over a specification's conjectures. A model checker can be used with CADiZ's theorem prover, using a tactic that can decide predicates by exhaustively considering all possible combinations of values for its local declarations. The model checking is actually

⁵Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent systems.

performed with the NuSMV toolkit [CCGR00], to which CADiZ interfaces.

CADiZ was initially marketed as a commercial product – with considerable success – by York Software Engineering Ltd., being deployed in numerous sites across the world, and was the winner of a British Computer Society award for outstanding technological achievement in the computing field. More recently, CADiZ has been made freely available; however, it is still a closed-source tool, and although there are indications that this may change, currently we would face the same problems as with a proprietary tool.

CADiZ offers some ability to specify and prove refinement relationships, as its theorem prover allows a user to state, and attempt to prove, theorems relating to their specifications. However, its focus is on specification and the verification of that specification. In fact, we have seen in our research that the majority of tools that support the Z notation are lightweight formal method tools. CADiZ is one of the few that offers any verification support, and one of even fewer that is capable of rigorous development from a specification.

When we consider the potential for using CADiZ to support retrenchment, we are again hindered by the closed nature of the source code. The way in which CADiZ is designed reflects its specification focus and whilst it is possible to specify conjectures that define a refinement or retrenchment relationship, it is necessary for the user to define that relationship upon each occasion. In order to extend the CADiZ toolkit so that it could support retrenchment constructs, and automatically generate the required proof obligations, it would be necessary to have access to the source code or create a custom tool that wrapped the CADiZ kernel.

ZETA

ZETA [Gri00b, BG00b, GL00] is another tool set that supports the Z methodology. ZETA has a modular design which comprises the following.

- A \LaTeX adaptor allowing specifications in an approximation of the notation defined in the Z ISO standard.
- The ESZ parser and type checker.

- The ZAP compiler [Gri00a] which is an execution engine for Z based on concurrent constraint resolution and which provides animation facilities to the tool.
- Adaptors that connect to HOL [GM93] and SMV [K.L92]. Note that whilst these adaptors have been created for ZETA, they have not been ported to the most recent version of the tool.

When the most recent revision of ZETA (1.5) was released in 2000, the source code was placed in the public domain subject to the GNU Public Licence. At this time, however, work also ceased on the project and no further development has taken place. At first glance it may appear that such a tool would be exactly what we needed – providing a solid basis for the mechanized support of retrenchment.

Unfortunately ZETA was not only written using a deprecated version of Java, but also relies on a number of external tools which have also fallen out of use. For example, the majority of the ‘Java’ code is in fact written using the Pizza language [OW97], which is an extension to Java that introduces generics, function pointers, class cases and pattern matching. Unfortunately, work stopped on Pizza in 2002, and any attempt to use the ZETA source would require extensive updates to the tool code⁶ and the sourcing of replacements for the tools it relied upon.

A further disadvantage is that ZETA does not currently support (specifically) the use of refinement. Any refinements between Z models would need to become part of the specification itself and conjectures created to show that the relationship holds. That is, there is no notion of proof obligation generation, and while it is possible to specify relationships with the tool, it would need to be done explicitly by the user on every occasion.

Jaza

Jaza [Led06, Utt00, Utt05] (Just Another Z Animator) is an open source animator that supports the Z notation. The notation of [Spi92b] is used,

⁶Generics were introduced to core Java with the release of J2SE 5.0 in 2005, this facility would have made the use of ZETA source code far easier.

but generics, bags and axiomatic definitions are not supported. Jaza aims to validate Z proofs through execution and testing (both forms of animation). Execution involves providing the inputs to an operation from which Jaza will generate the outputs. Unfortunately, execution can only be used on Z specifications that happen to be wholly written with statements that can be executed and – as it is commonly held that it is better style to write a specification in a non-executable fashion – this is rarely the case. Testing, in this instance, refers to the process of providing Jaza with both an operation’s inputs and an expected output; the tool will then try and determine whether it is possible for the operation to produce that output with the given inputs (which is a far easier task, and implementable even on non-executable specifications).

Jaza is written in Haskell and distributed under the GNU public licence, which would allow for extension for retrenchment. Like ZETA, however, Jaza has no support for refinement, and it is hard to see how retrenchments could be animated with the techniques it currently employs. Furthermore, we can question the immediate value of animators in system developments supporting retrenchment. In such a project, the most abstract specification could be considered to be a formalization of the requirements themselves and validation by inspection should be relatively simple. Whilst we would like to provide as many tools as possible that support retrenchment, we believe that an animator is not a top priority.

VDMTools

The IFAD VDM-SL/VDM++ Toolboxes [AL99, ELL94, Lar01] (collectively known as VDMTools) have been some of the most widely used formal methods tools in both industry and academia. The tools support the standard VDM notations (including the ISO standard for VDM-SL [VDM]), they provide syntax checking, extended type and semantic checking, and validation through animation tools that take advantage of the VDM-SL’s largely executable behaviour. Although the tools do not explicitly allow for the verification of models, there have been several projects that ‘plug-in’ this behaviour

to the tool through modular extensions (for instance, see [AL97] which describes an extension to generate proof obligations to complete type checking, and to show that a model is internally consistent). The developers of VDM-Tools were proponents of lightweight formal methods [FL98], and as such the tool has limited support for data reification.

At the end of 2001 Peter Gorm Larsen left IFAD after more than a decade working on the development and marketing of the VDMTools product suite. IFAD ceased investment in VDMTools and the active VDM community declined substantially. The intellectual property rights for VDMTools were acquired by one of the tool set's most serious users, CSK Corporation in Japan.

This period of upheaval was occurring at the time when we were examining the tools in which we could possibly integrate retrenchment. As the future of the tools was uncertain – and the tools were likely to stay out of the public domain – we felt it best to consider other alternatives.

SpecBox/Mural

SpecBox [BFM89, FMB89, MF91] and Mural [FEG92, JJLM91] are a pair of tools that can provide a complete specification and verification system supporting the VDM. SpecBox provides the ability to specify models in a version of the VDM-SL notation based on the draft version of the international standard (a few changes were made in the definitive version [VDM]). SpecBox provides syntax checking, and some limited type checking (subject to the restrictions we have discussed above). Once a specification is found to be syntactically sound, SpecBox can translate it into a format suitable for Mural's VDM support tool, which can then generate and discharge the appropriate proof obligations. Data reification is an integral part of the VDM, and as such these tools provide support for its use.

Like the majority of VDM tools, SpecBox and Mural have suffered from the decline in the VDM's popularity and neither appears to have been updated for some time. Recently, the SpecBox tool has been made freely available for non-commercial use. Despite this, however, Adelard – the company

who created SpecBox – maintain copyright over the application, and the source code is not publicly available.

The possibilities for extending this tool to support retrenchment seem limited. SpecBox is designed to run on MS-DOS, and even were the source for the tool available, porting it to a modern operating system would prove a large task in itself.

Overture

Perhaps in answer to the dearth of VDM tools that are currently active, 2004 saw the proposal of a new open-source project to create a tool set that supported the VDM++ notation. This project is known as Overture [vdS04, LP05, NH05], and its principal aims are to create an industrial strength, open tool that supports the VDM allowing other researchers and interested parties to experiment with modifications and enhancements to the tool. While the original intent was to support the VDM++ notation, Overture now has its own notation, OML (Overture Modelling Language) that is similar, but not identical, to that of VDM++.

Like RODIN, the Overture tool will be created as a plug-in to the Eclipse integrated development environment. There will be a central kernel to perform the most basic core tasks, and all other functionality will be provided in the form of further plug-ins. For example, there will be a type checking plug-in and a plug-in to perform proof obligation generation. As the project is in its infancy the range of plug-ins, and hence the functionality of the tool, has yet to be finalized.

Again, it is unfortunate that the proposal of this tool was announced after our tool analysis period was complete, but in any case – as with RODIN – this tool would not have been sufficiently mature for use in our immediate work in the mechanization of retrenchment. In the future, however, the flexibility of the tool being created through the Overture project should hopefully give us another method of providing tool support for retrenchment, with the advantage of being able to use much of our existing work.

Perfect Developer

As well as those tools that support the major formal specification languages, there are other, commercial tools aimed particularly at the industrial market. One of these is Perfect Developer [CMM05, Cro03a, Cro03b] produced by Escher Technologies. Perfect Developer uses its own notation, the Perfect Language, which is intended to resemble object oriented programming language instructions rather than the mathematical language of other notations. This language is used for expression requirements, specifications and code. As well as providing syntax checking, the language is strongly typed, and proof obligations are generated to show both internal consistency of models and refinement. The tool's makers claim that the tool can typically discharge more than ninety five per cent of correct proof obligations without any user intervention.

The power and simplicity of Perfect Developer make it likely to succeed in an industrial environment. However, although Perfect Developer has been introduced to some academic institutions as a teaching aid in formal methods, it has not yet been used to any great extent in academic research. While our goal is to provide mechanized support for retrenchment, we must recognize that our first priority is support for its use in research, and as such extending toolkits such as Perfect Developer, should not be our primary target. When we consider also, that as tools such as Perfect Developer are aimed at an industrial market – and they therefore are reliant on sales of their tools – they are unlikely to allow us to experiment with their tool, nor make available an extended version that supports retrenchment.

3.2.2 Supporting Directly within a Theorem Prover

In the previous section we have considered the prospect of extending an existing toolkit to support retrenchment. Another alternative is to embed support directly within an existing theorem prover. There are two approaches we could take to this. Firstly, we could consider embedding support for one of the common specification languages, and then manipulate that to produce support for retrenchment. A second approach is to define a language

specifically for expressing retrenchment by directly using the syntax of the theorem prover itself. We consider both alternatives in the following section.

Embedding an Existing Notation

Rather than creating a stand alone program to give mechanized support for a formal methodology, it is possible to embed the logic of that methodology directly within a theorem prover. This approach has been attempted with the Z methodology and the HOL [GM93] theorem prover, initially by Bowen and Gordon [BG94, BG95], and more recently by Brucker et al [BRW03]. Bowen and Gordon's work involved a shallow embedding of the Z notation. With shallow embedding the language constructs of the notation are mapped to their semantic equivalents in the theorem prover's language. A deep embedding involves defining the language constructs of the notation directly within the theorem prover. Constants, functions and rules were defined in HOL that allowed a Z schema (and associated schema operations) to be expressed directly within the theorem proving environment. Proof obligations are not created automatically (as is typical for a Z specification), but users are free to express any conjectures about the schemas they have defined using the Z meta-language and any of HOL's existing set theoretical rules and tactics. Brucker's work extended that of Bowen and Gordon and incorporated the HOL-Z tool that they had developed within a larger toolkit (the ZETA toolkit described above). This work allowed the specifier to create a model using the specification, syntax checking and type checking facilities of ZETA, and then translate the specification to the meta-language used by HOL-Z. A tool to automatically generate proof obligations for consistency and refinements was also provided.

The principal advantage of the embedded approach is that the developed tool should be considerably more reliable than a stand alone tool (assuming that we use an established and well tested theorem prover), as the only place where errors can encroach is in the implementation of the meta-language used to express the specification language's notation. A second benefit, is that a shallow embedding allows the use of extensive, existing libraries whose

use is well established in proving theorems. With a deep embedding special tactics have to be developed that are specifically able to handle the language constructs of the embedded notation; clearly, these tactics will not have the same level of optimization or support as the core tactics of the theorem prover.

A restriction of this method is that there is usually a necessity to develop a meta-language that forms a bridge between the specification language and the language of the theorem prover. This can place limits on the range of the specification language used. If the specifier is to create their specification directly in the meta-language, we lose some of the benefit of using an established language. If an automatic translation is provided, we still run into trouble in the reporting of errors and the format of proof obligations and proofs (which will be in the language of the theorem prover).

Other disadvantages – highlighted by Brucker – include the inability to reason about the specification language itself (when using a shallow embedding), difficulty in structuring specifications, and the occasional incompatibility of a theorem prover’s tactics with the concepts included in the specification language. Brucker’s work, however, has shown that it is possible to incorporate a logical embedding as part of a larger tool set and that there are conceivable solutions to these problems.

Describing Retrenchment using a Theorem Prover’s Language

As well as embedding a formal methods’ notation in a theorem prover, there exists the possibility to use the language of the theorem prover to describe retrenchment directly.⁷ In [Kna96], Knappmann describes an attempt to build a tool that supports refinement of programs using the PVS [SCR96] theorem prover. Knappmann’s work involved the creation of a wrapper language that represented high-level programming language instructions, which translated directly into PVS functions (note that, the functions were created first and

⁷Note that, we consider this approach to involve the definition of a semantics in the theorem prover, which is then translated outwards through the creation of a meta-language. We draw a distinction between this, and creating specifications outside a theorem prover and then translating the proof obligations generated from that specification into the language of the theorem prover.

then the instructions based upon them – there is no interpretation between syntaxes just a one-to-one translation). Refinement rules were created and organized into strategies that were able to automatically discharge the typical proof obligations generated by each step. Where an automated discharge could not be achieved, these rules and PVS’s libraries were available to a user to complete a proof. Note that, Knappmann’s tool was intended to iteratively refine a single abstract specification into a single concrete program, and there was no support for other software engineering activities.

An advantage of this approach is that there is no ambiguity over the definitions of relations and functions, as we only ever use the inbuilt definitions of the theorem prover. We effectively have a deep embedding of our high-level programming language instructions. Another benefit of this method, is that as we would be using a novel language we would be free to use any syntax that meets our needs (assuming we are able to express the translation of that syntax directly within PVS).

The principal disadvantage of this approach is that while using an original notation provides us with a syntax adaptable to our purpose, it is not immediately accessible to a new user of the tool. As one of the goals we have outlined for tool support is to make formal methods comprehensible to an average software engineer, we would like to use common notations for which existing literature provides a great deal of support. It could be argued, that a deep embedding of one of the common notations could be attempted, we felt however that, while interesting, such a task would be a research project in itself, and would distract from our effort to provide mechanized support for retrenchment.

Another limitation that results from the pursuit of this route, is that the models and relationship involved in a retrenchment (for example) step must all be passed to the theorem prover where the proof obligations are created and discharged. This not only makes keeping track of models in a large system more complex (especially where each model is involved in more than one relationship), but also means that the proof obligations (and their proofs) will only ever be presented in the underlying notation of the theorem prover, which would make them difficult to be understood by all but experts.

3.2.3 Creating a New Toolkit

We have now examined some existing toolkits, and investigated whether it is possible to support retrenchment directly within a theorem prover. We now turn to our final option which is to create a new tool that is capable of giving the utmost flexibility whilst providing support for retrenchment. What are the advantages and disadvantages of such an approach, and in what ways did we envisage that taking this approach would better fulfil the requirements of providing mechanized support for retrenchment?

Clearly, there are the usual advantages and disadvantages to creating a new toolkit over adapting an existing application. The question being whether the additional flexibility available in the creation of a new tool is worth the ‘reinvention of the wheel’. The use of an established toolkit as a foundation, provides not only the toolkit, but documentation, case studies and an existing user base. On the other hand, creating a new toolkit allows us to examine the best way in which to provide support for retrenchment, rather than finding a method that *can* work with an existing toolkit.

Building upon an established tool does not necessarily provide a panacea to our development, in fact the use of a base-tool can often lead to as much work as the creation of a brand new tool. The source code of existing toolkits is very rarely well documented, and it is usually necessary to reverse-engineer the implementation to determine exactly what occurs. This process can be time consuming and can easily lead to misunderstandings. Such code will also typically contain undocumented bugs. These bugs can fall into two categories. Firstly, there are bugs that are located in code such that they do not affect the behaviour of the existing tool, but give functionality that is relied upon in an extension; these are extremely infuriating for a developer, as they will often assume that the problem is in their extension rather than the original code. The second type of bug is one which does affect the behaviour of the existing toolkit, but is little used so has either not been discovered, or given such a low priority for repair that its presence is missing from documentation. Unfortunately, while the code containing these bugs may be little used in the existing toolkit, it is surprisingly common how

often the functionality is relied upon when augmenting that toolkit. This type of bug can be even more difficult to find, as the existing toolkit will often appear to be behaving correctly in an ‘identical’ situation. In the author’s experience this lack of documentation and the almost certain presence of undocumented bugs, proves to be one of the biggest disadvantages when extending an existing toolkit. As the creation of a new toolkit circumvents these problems entirely, the converse proves to be a great benefit of starting from scratch.

A similar advantage of creating a new toolkit, is that modern programming languages have a tendency to develop quickly, and support tools – which may have been used within an existing toolkit – tend to have a shorter shelf life. For example, in the last ten years huge swathes of the Java programming language have become deprecated; a particularly notable instance of this evolution, was the replacement of the Abstract Window Toolkit provided for GUI design in Java 1.1 with the Swing foundation classes of Java 1.2. This was not simple change in the syntax, but a complete re-implementation of the underlying classes, and implementing a GUI using AWT would now be considered unthinkable [LEW⁺02]. If we were to extend an existing toolkit implemented in an evolving programming languages, we would need to consider the implications of this. We would have two options available, either update the entire application to conform to a modern specification of the language, or continue to develop using deprecated instructions (which would, at some point, become unsupported). Obviously, neither of these options is ideal. In some ways the situation is even worse where the existing toolkits depend on other, supporting tools, as there will be a need to maintain synchronicity with the publicly available versions of these tools, and there will always be a threat that the supporting tool may become unavailable for use with our tool. We have seen how a toolkit (ZETA above) can become practically useless when it contains these dependencies and we have seen that this decline can occur relatively quickly (the technologies ZETA relied upon were current in 2000). It is important to consider this when creating a new toolkit to ensure that we do not rely on programming language functionality that is likely to become deprecated in the near future. We feel that any supporting tool used needs

to have a licence that will allow us to distribute that tool (in the appropriate version) as part of a created toolkit (rather than requiring a user to acquire such tools themselves).

A further benefit of creating a new toolkit is the control that can be retained over its development and the way in which it is released. If we used an existing toolkit we would be restricted by its licensing and that of any supporting tools that it used. As we wish our tool's use to be as widespread as possible, we were not keen for its distribution to be limited in such a way. Furthermore, the extension of any toolkit that is still being developed would require branching the source code, and again this is not an attractive option, as it would cause confusion between the users of the original and extended toolkits; of course, we may choose not to market the enhanced tool as an extension, but then we lose the aforementioned benefits of existing documentation, case studies and users.

As we investigated the requirement of a toolkit that supported retrenchment – and looking back on experiences adapting the B-Toolkit – we realized that the incorporation of retrenchment into an existing toolkit required the introduction of not one, but several relationships. We discussed in section 2.5.3, the existence of several forms of retrenchment and the need for any tool supporting retrenchment to be flexible enough to support all of these forms. We also stated that a tool would need to be able to adapt to potential new forms. Initially, we had considered that this flexibility may be achieved using a plug-in approach such as that we have discussed being used in RODIN and Overture above. If we were creating a new toolkit we would have much more control over the way in which models (and the relationships between models) could be specified and in fact we could make the notion of a formal relationship (or model) configurable. In this way, a researcher or interested party who was looking to add support for a relationship to our tool, could do so simply through the addition of a specification for that relationship to some configuration file. Hence, if a new form of retrenchment arose, or we wished to support one of the various mutations of refinement we would be able to do so within any development whatsoever. We therefore felt that a configuration-based approach would be a huge advantage of creating a new

toolkit.

Similarly, our previous experience has taught us that in order to work within an existing toolkit's framework, we would need to maintain the structure of its developments. If we consider a construct to be configurable, how can we impose a structure upon those constructs? The ability to design and use our own structure would therefore, be a considerable benefit of creating a new toolkit. We described above how the B-Toolkit used a linear model of development and how this severely restricted our ability to support retrenchment. If we were able to specify our own structure for a specification we could reduce our view to a very abstract model; for instance, a directed graph where each model was a vertex, and each relationship (between two models) an edge (between two vertices).

The other restricting issue that arose in our attempt to extend the B-Toolkit to support retrenchment was that the available proof assistant was not powerful enough to discharge the retrenchment proof obligations. A further gain in the creation of a new toolkit would be the possibility to use more than one established theorem provers in the discharge of these obligations. We could use theorem provers such as HOL and PVS either to embed some meta-language into which we could translate our specifications, or to simply attempt the discharge of our proof obligations.

3.3 Evaluating the Options

We have outlined the potential for extending an existing toolkit, embedding within a theorem prover, or creating a new toolkit to support retrenchment. We will now compare the strengths and weaknesses of these options and outline how we decided to proceed.

If we consider first the option of extending toolkit, we can see that the main issue with the tools that were available, when we came to make our choice, is that most of them were proprietary, and while some may have been freely available for non-commercial use, we did not have the potential to access the code without specific permission. Even if that permission had been obtained, the extended tool produced would have been subject to the same

licensing restrictions as the original. Our aim here is to create mechanized support for retrenchment that can be freely used, extended and modified, and we felt that using an existing, closed-source tool would severely limit this possibility.

Of the remaining tools where the source was freely available, there were numerous other restrictions that we felt prevented their use in providing all-round support for retrenchment. The RODIN and Overture projects had not been conceived when we were making our decision. ZETA is based on deprecated technology, and the effort required to bring it up to date may well have exceeded the effort in the creation of a new tool. It was felt that if we were making this effort we should choose the option with greatest flexibility. Jaza would have proved (and may still prove to be) a useful starting point if we had sought to create a retrenchment animator, but as we have previously discussed, the creation of an animator was not felt to be an urgent task. More importantly, it was felt that we needed to be able to provide support for the entire development process – from specification to verification – and extending Jaza would not have given us a tool capable of this. Having eliminated all of the tools that we had considered extending, we thought it was wisest to pursue an alternative approach to supporting retrenchment.

The option to embed a specification language within a theorem prover is an interesting one, but we felt that this approach – and the direct embedding of retrenchment within a theorem prover – would require us to devote more of our resources to the practicality of the embedding, rather than in providing tool support to retrenchment. Furthermore, the tactics or strategies that we would need to create within the theorem provers to support retrenchment, would be very specific and would not give us the option to support a configurable range of relationships. Nevertheless, we feel that at some point in the future we could consider an interface to HOL-Z (or another embedding) from our tool, that would provide alternatives in our verification process.

The final option was to create a brand new, stand-alone toolkit that supported the specification of models and retrenchment relationships. Given the problems with the alternate approaches discussed above, this proved to be

a much more suitable alternative, offering some distinct advantages. Prime among these, were the ability to create our own structure and the option to make constructs configurable. We can summarize this, by concluding that we felt the extra work required in the creation of a new toolkit (it is actually arguable whether the effort in creating a toolkit actually exceeds that needed to reverse-engineer and update an existing tool) was worth the added flexibility that such an approach would give us.

3.4 Frog: A New Toolkit to Support Retrenchment

We have seen in section 3.2.3 that there are many advantages to the creation of a new toolkit, and we have decided that the creation of a new tool set to support retrenchment will be the best way in which to provide it mechanical support. At this stage we made the arbitrary decision to refer to our new toolkit as Frog.

Having made the decision to create a brand new toolkit, the first step was to look at the requirements for such a toolkit. In what ways would a retrenchment tool need to be different from existing tools? Which aspects of tool support would we strive to provide? Should we use an existing notation or create a new one? We answer these questions in the remainder of this section.

We mentioned in the previous sections the need to make the concept of the model and the relationship configurable, and we considered this to be the main way in which Frog would be different to existing tools. It was necessary to create a tool that would allow the syntax and proof obligations of constructs to be altered dynamically (obviously within limitations – we would not want the configuration of a construct changing, if the original configuration were relied on elsewhere).

Similarly, we did not wish Frog to impose a rigid structure upon the constructs within a project. Unlike existing tools, we did not want to compel a specifier to use models and relationships in a particular way. For example,

we would not make a construct-level distinction between a model equivalent to the B-Method’s abstract machine and another equivalent to its implementation machine (indeed we would not even want to make a construct-level distinction between a refinement relationship and a retrenchment relationship). A specification created using Frog therefore, would consist – at an abstract level – of models with relationships between those models. A side effect of this, is that (as least initially) Frog would not enforce any language restrictions at a construct-level (again, unlike the B-Method), and also that the responsibility of maintaining a development’s structure would lie with the specifier.

We have considered previously the difference between the proof obligations of refinement and retrenchment, and how the proof assistant of the B-Toolkit had trouble when attempting to discharge those belonging to retrenchment. We decided therefore, that we would like the proof obligations generated by Frog to be theorem prover independent such that they could be translated into the specific languages of a number of different proof assistants. This would not only give us the greatest chance of discharging our obligations, but would also allow a user the flexibility to work with the theorem prover with which they were most familiar.

We discuss now, what facilities a retrenchment toolkit should provide. Obviously, it would have been ideal to provide all of the tools we outlined in section 3.1, but we had limited resources and there was a need to prioritize the aspects of support that we incorporated. We consider each of the possible tools in turn.

- *Specification and Syntax Checking* — The need to be able to specify a model, or the relationship between models, is clearly essential in a tool to support retrenchment. If our toolkit is to be able to understand those models, it must also be able to parse and syntax check the notation contained within.
- *Type Checking* — While a type checker may not strictly be necessary, we felt that it is of significant use in uncovering errors in specifications, and particularly when expressing a relationship (for instance, a

retrenchment) between two models. Errors of this sort are increasingly common where the declarations for the variables used in a relationship are included within the models it references, rather than the relationship itself.

- *Animation* — We have discussed above the need for animation within a tool supporting retrenchment. The most abstract model in a retrenchment will often prove to be nothing more than a formalization of the requirements, and whilst useful, an animator is not therefore our top priority.
- *Model Checking* — As with animation, we questioned earlier whether the production of a model checker would prove the best use of our resources at this time, and while we would like to include a model checker in the future, it is not necessary to include one in the first generation of our toolkit.
- *Deductive Reasoning* — We mentioned in section 2.5 that, unlike refinement, retrenchment was actually defined by its proof obligations. It was highly necessary, therefore, that our tool set provided the facilities to generate these proof obligations, and furthermore, interface with a tool which would enable a user to attempt the discharge of these obligations through deductive reasoning.
- *Test Case Generation* — We have mentioned that the development of Frog will be focused on providing an environment in which a user can experiment with retrenchment and its use in the development process. The need for test case generation would be greatest in the development of commercial products. Providing support for commercial development was not a motivator in the creation of our tool, so we did not feel that including test case generation would be the most effective use of our limited resources.

We move next to a consideration of notations. Firstly, we needed to decide whether it was best to use one of the notations already in use, or create one

specifically for use with retrenchment. We felt that as retrenchment was already a relatively new concept, we did not wish to muddy the waters further through the construction of a new language. Furthermore, we felt that this would have just been a case of ‘reinventing the wheel’; the notations available were capable of expressing the majority of our needs, and were flexible enough to incorporate any necessary changes. Once we had made this decision, it was necessary to consider which of the available notations to use. When we were considering this choice, the VDM was in a phase of significant decline (as we discussed above), and we felt that as such, its use in a new project would not have been sensible at that time. This left us with a choice principally between the B and Z notations. While the B-Method has a great track record in producing implementable code from a specification, in this instance we are more interested in experimenting with specifications, and Z is a more flexible and expressive specification language. Furthermore, given our desire to have a flexible notion of model and relationship we felt that the B-Method’s rather rigid structure of abstract machines, refinement machines and implementation machines would not give us the fluidity required to specify the notion of a construct dynamically⁸. We therefore concluded, that we would use the Z notation for expressing the body of our models and relationships, and would create a small meta-language (translatable to Z) that would allow us to encapsulate the notion of a construct. That is, instead of a Z specification consisting wholly of sections and paragraphs, we would incorporate the concept of models and relationships which would be delineated through the use of our meta-language. In keeping with our theme of flexibility, however, we felt that the design of toolkit should not be notation-focused, and leave open the option to use other notations at a future date.

⁸Although the above paragraphs may have suggested that this would be impossible in the B-Method, [Fra05] shows that we can work around the rigidity of the notation, while maintaining the ethos of the methodology. Of course, the necessity to circumvent the traditional notation provided another reason for not using it as the initial notation of our tool.

3.5 Concluding Remarks

In this chapter we have discussed the need for tool support to formal methods. Effective tool support is essential to the uptake of formal methods as it can significantly decrease the overheads incurred in their use (and thereby counteract the arguments of detractors). There are a variety of tasks in which mechanical assistance can be of great help, from syntax and type checking, to verification, validation and test case generation. When we introduce a new formal technique, such as retrenchment, it is vital that we are able to produce tools that provide assistance in its use. There are many toolkits that support formal methods, and we have discussed the suitability of extending one of these for retrenchment. We have also seen that it is possible to embed a tool supporting retrenchment directly within a theorem prover. Unfortunately, we have felt that we were unable to pursue either of these approaches, and we were left to consider the creation of a new tool set. Having come to this decision, we decided that our new toolkit should provide facilities to specify and – through the generation of proof obligations – allow the verification of retrenchment. We concluded also that the Z notation will be used to form the bulk of our specifications. It was felt that the hallmarks of the new tools should be configurability, flexibility and modularity; this applies not only to the design of the toolkit, but also to the way in which we handle the models and relationships of our specifications.

Chapter 4

Parsing The Z Notation

This chapter describes how it is possible to parse the Z notation mechanically. We begin by introducing the syntax of the Z notation and its structure. A number of existing tools that are already able to parse the Z notation are then presented, and our motivations for creating a new parser outlined. We then present an overview of the parsing process and clarify the concept of a specification and its structure. Z's special sections are then introduced, and the exceptional methods of handling these described. Following this, we commence our description of the actual parsing process, beginning with the syntax checking phase. This section focuses on providing a machine-readable version of the grammar presented in [ISO02], and the train yard example is revisited to illustrate abstract syntax tree (AST) generation. The next phase of the process – syntax transformation – is then described, and the train yard is again used to demonstrate how this phase proceeds. Finally, we examine the type checking phase. We describe how we create the final typed-AST for specifications in general and the train yard particularly. Generic type instantiation, and the creation of an object-based view of a specification are examined in detail.

4.1 Introduction

Z [Cur99, Spi92b, WD96] was first proposed around 1981 by the Programming Research Group at Oxford University as part of a project to establish a mathematical language that could express common programming concepts and which could be used in an industrial situation. Since that time Z has become one of the most popular specification languages and has been applied in the development of a wide range of systems in academia and industry.

A Z specification comprises two equally important parts: a precise, mathematical description of the system and an informal, natural language description that provides an abstract, but unambiguous explanation of that formal notation. The mathematical descriptions are formed from a combination of set theory and first order predicate logic. A particular feature of Z is that every expression within a specification can be typed. This typing is an important advantage when trying to uncover bugs in a specification. This typing process can be automated, meaning that errors can be quickly and easily spotted. A second feature of Z is the concept of a ‘schema’. Each schema comprises a series of declarations and a constraining predicate. These schemas can be used to structure a specification improving readability and understandability.

We have begun to introduce the mathematical language of the Z notation in previous chapters and we will continue to describe important concepts as we proceed with this chapter.

The structure of a document is also important when using the Z notation. Every Z statement must belong to a paragraph, of which there are various types. For instance, a schema is typically defined in a schema definition paragraph, and types are declared in a given type paragraph. Variables declared within a paragraph have a scope that depends upon the type of the paragraph. For example, variables declared in a schema definition paragraph can only be used directly within that paragraph. Variables declared in a given type paragraph (that is, the types themselves), however, are available anywhere within the section to which the paragraph belongs. Every paragraph

must belong to a section (where no section is explicitly created, an anonymous section is presumed), and the definitions of one paragraph are available to all other paragraphs within that section. Each section can declare one, or more, parent sections. Declaring a section as a parent makes all of the definitions in the parent section available throughout the child section. A Z specification always comprises one or more sections.

In recent years, the somewhat informal definition of the Z notation has been standardized through the production of a standard for the International Standards Organization [ISO02]. This standard provides a stable basis for the development of tools based on the Z notation. However, as this standard has been created relatively recently, many existing tools have become deprecated. For this reason and others – which we discuss below – we decided we would need to implement our own Z parser.

4.2 Existing Z Tools

There are many existing tools that are able to parse the Z notation, so why did we feel it was necessary to create our own parser? In this section we present a brief description of some of the major Z tools available and then explain why we chose to begin building our tool from scratch.

4.2.1 CADiZ

As we discussed in the last chapter, CADiZ [JMT91, TM95, Toy96] is a set of freely distributed tools, developed at the University of York, to support the use of Z. CADiZ has been developed by some of the major proponents of the ISO standard Z notation [ISO02], and as such was one of the first Z parsers to strive for compliance with the standard. Currently, CADiZ does not completely satisfy the standard, but the deviations it possesses are minor and well documented (see sections 3.7, 3.8 and 3.9 of [Toy]).

CADiZ provides tools that are able to check a Z specification for syntactical and type correctness. Also provided is an interactive theorem prover –

incorporating its own tactic language – that allows users to reason over conjectures in their specifications. Numerous other typesetting and browsing tools are also included.

Whilst at first glance CADiZ may appear to be an ideal solution to our parsing problem, the source of the program is not currently in the public domain, nor are the APIs¹ provided, sufficient to allow a tool developer to take full advantage of its facilities.

4.2.2 ZTC/ZANS

ZTC [Jiab] is a type checker for Z that takes a specification in either \LaTeX or ZSL format² and checks it for syntactical and typing errors. ZANS [Jia95, Jiaa] is a complementary prototype of a tool for animating Z specifications.

ZTC and ZANS accept Z \LaTeX specifications created with either the ‘oz’ [Kin90a] or ‘zed’ [Spi90] packages. The definition of Z used does not, however, conform to the ISO Z standard [ISO02].

Again, the main disadvantage of these tools is that – while they can be freely distributed – they are closed source and do not provide any output that could be used by a tool wishing to introduce additional functionality.

4.2.3 *f*UZZ

*f*UZZ [Spi92a] is a toolkit that provides syntax checking and type checking for Z specifications in the \LaTeX notation. Again, *f*UZZ uses the \LaTeX format outlined in [Spi90] and conforms to the Z definition of [Spi92b]. There are no plans to produce a version that will parse the ISO standard Z definition.

Originally *f*UZZ was a proprietary tool, but in 2000 the source code was released for all non-commercial use.

The principal disadvantage with using *f*UZZ was that it did not support the ISO standard. As it was first written in 1989, the parsing techniques used in its implementation were application-specific (rather than the more adaptable, generic parser generators available today), and we felt that the

¹Application Programming Interfaces

²ZSL is an ASCII based format developed by the creators of ZTC

alterations required to parse Z conforming to the ISO standard would be extremely time consuming.

4.2.4 Z/EVES

Z/EVES [Saa97], was a fully functional tool for creating Z specifications, syntax and type checking those specifications. The tool also included an interface to the integrated general theorem prover (EVES) that was able to reason effectively about Z specifications and was also able to perform precondition calculation (amongst other features). Z/EVES was a proprietary tool and its source was not available for modification. In July 2005 the distribution of both EVES and Z/EVES was ceased by its owners ORA Canada.

4.2.5 CZT (Common Z Tools)

In 2001, Andrew Martin proposed [Mar] the creation of an internet-based community project to build an open-source framework for the development of tools supporting Z. This proposal was made in the absence of any integration between existing Z tools and the closed nature of their source codes. In the following years the requirements for the project were determined, and eventually work began on the design and implementation of the core framework.

In 2003, the CZT project began in earnest with the creation of a project on the popular open-source software distribution site, Sourceforge. The initial results of the CZT project were reported in 2005 [MU05, MFMU05], and alpha versions of the core tools were made available.

CZT uses an abstract syntax tree (AST) comprised of Java interfaces and classes to represent a Z specification. These AST are created through the parsing of documents in a number of formats, particularly the ZML notation first proposed in [UTS⁺03]. CZT currently provides tools to type check these specifications, and to translate into a number of alternative notations.

There are many people contributing to CZT extending its core functionality and providing the plug-ins that allow integration with existing tools. One of the main advantages of CZT – and a motivation in its creation –

is that the lack of ties to any particular company, school or developer will ensure openness and longevity.

4.2.6 Summary

Ideally, we would have liked to be able to interface with an existing Z parser to produce activity such as that shown in figure 4.1 on the facing page. The user would create their specification in our new tool, Frog. This specification – following some pre-processing (such as that described in chapter 5) – would then be checked for correct syntax and typing by an external Z parser. Assuming there were no errors, that external Z parser would then return a typed abstract syntax tree (AST) which Frog could use for further processing (for example, the generation of proof obligations).

Unfortunately, there were restrictions preventing us from being able to re-use any part of the existing tools. CADiZ, ZTC/ZANS, and Z/EVES were closed tools that were not available for extension. *fUZZ* was open source but did not provide support for the ISO standard (nor had any intentions to do so). CZT – which would have provided an ideal solution – was not in development at the beginning of this period of research (one could argue that the tools created through the CZT project are still not stable enough to base the development of another tool upon). It was decided, therefore, that it would be necessary to create our own Z parser.

4.3 An Overview of Parsing a Z Specification

The first thing that we need to do is disambiguate the term ‘Z Specification’. [ISO02] provide no practical definition of a specification, merely stating that a specification is a collection of sections and paragraphs. We consider a specification to be the all encompassing unit for a project, that is, a Frog specification will contain all of the information required to model an entire system. A Frog specification, therefore, need not be contained within a single document.

In order that we are able to maintain changes on a file by file basis we

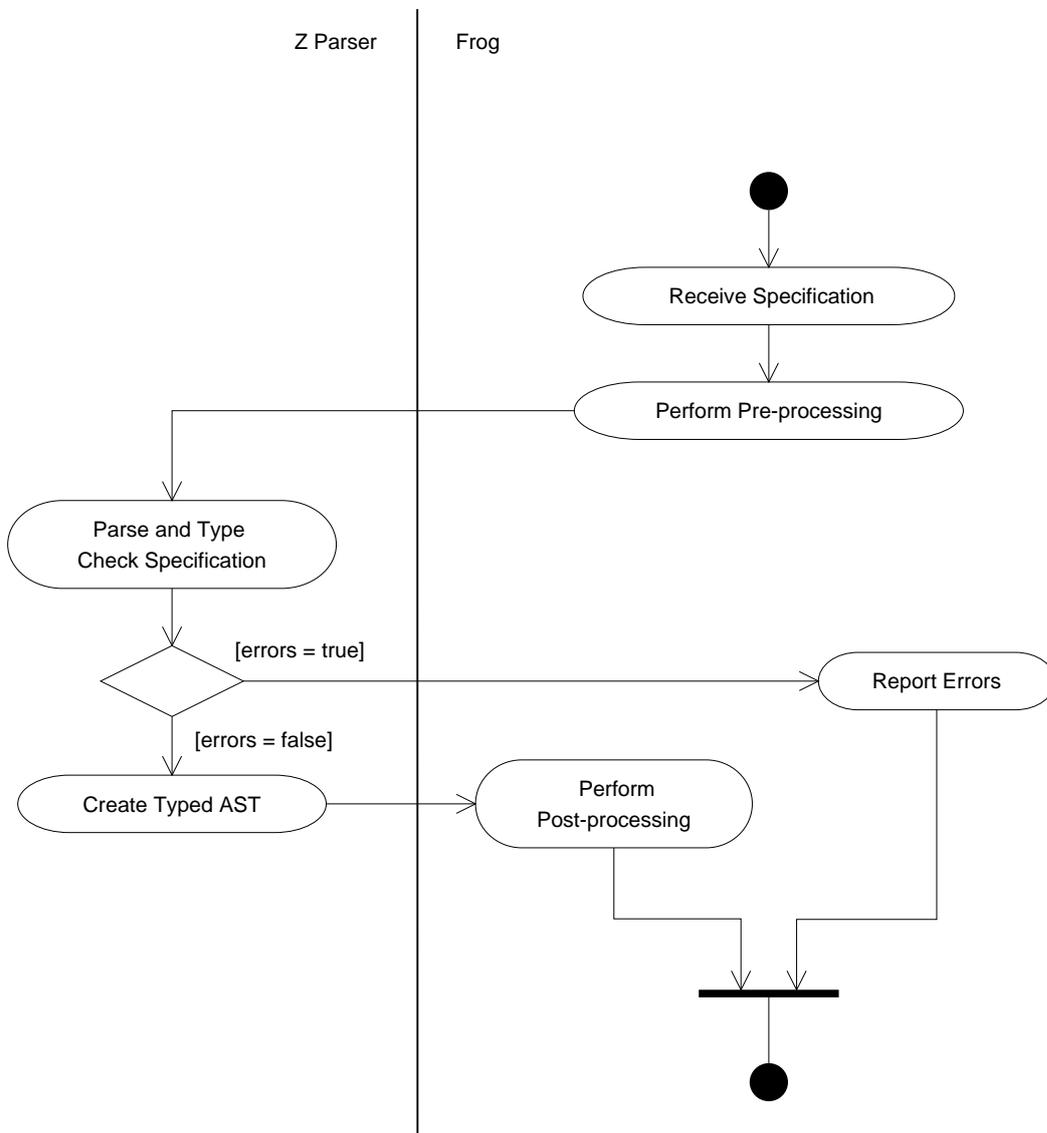


Figure 4.1: Interaction with existing Z parser

create the concept of a Frog file. The Frog file could be considered closer to the definition of a specification in [ISO02], that is, it is a document that contains a number of sections and paragraphs. A section defined in a particular file however, may specify any other section in the specification as a parent, and we consider each section to belong both to a Frog file and a Frog Specification.

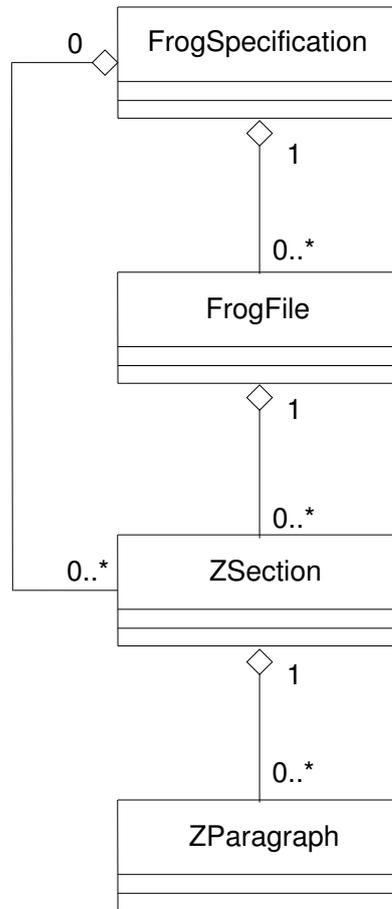


Figure 4.2: Contents of a Frog specification

All paragraphs must belong to a section; any paragraph that is not explicitly declared to belong to a section will belong to an anonymous section. Anonymous sections are handled on a file by file basis – that is, it is possible to have more than one anonymous section in a specification – so each anonymous section is identified by the name of the file in which it is defined.

Figure 4.2 shows the building blocks of a Frog specification and the nature of their inter-relation.

We now turn to the process by which a Frog specification is parsed. We consider the transformation of the specification into a type abstract syntax tree (AST) to be a single process, consisting of a number of phases. As noted above, each file in a specification is parsed individually and upon successful

Chapters 7 and 8 of [ISO02] provide the starting point for this process, giving a human-readable version of the required lexis and grammar. Section 4.6 describes how these processes work in practice.

Assuming there are no errors found at this stage, an AST is created providing a manipulable version of the Z syntax parsed. This tree is then transformed by the process described in 4.6.5 to produce an alternate version of our specification in a subset of the Z syntax. This subset of the Z syntax is specified in chapter 10 of [ISO02].

Finally, and again assuming no errors have been discovered, the transformed AST is passed to the type checker. The specification is then type-checked through the process described in 4.6.6. This process produces yet another AST that describes the typed version of our specification in the annotated syntax and is also the point at which an object view of our specification is created. The starting point for this process is the type inference rules described in chapter 13 of [ISO02].

4.4 Handling Special Sections

There are two exceptions to the rules for handling sections described above: the prelude section, and the sections belonging to the mathematical toolkit (each component toolkit equates to a Z section). In this section we describe the ways in which these special sections are handled differently to normal sections.

4.4.1 The Prelude Section

The prelude section is a special Z section that is the implicit parent of every other Z section. It contains definitions that are required in the syntactic transformation of number literals (see 4.6.5) and defines the token for the power set operator.

As every section must have a parent section, and we could find no other way to mark the prelude section as being such, we have reserved the identifier ‘prelude’. This identifier will only be used to refer specifically to the section

intended to be the actual prelude. Therefore, if the standard prelude section is replaced – through the user’s introduction of another section called prelude – then the replacing section will be used as if it were the standard prelude section.

Clearly this method of determining the prelude section is not ideal. It would be beneficial in tool development if there were some way to mark a section specifically as the prelude, perhaps providing an alternative to the redundant ‘SECTION’ token. For example,

```
\begin{section}
\SECTION prelude
\end{section}
```

could become the following (which, if required, could still be rendered in the existing manner).

```
\begin{section}
\PRELUDE
\end{section}
```

4.4.2 The Mathematical Toolkit

The core Z language can be used with just the definitions provided in the prelude section. There are a number of common definitions, however, that are often required in a formal specification. For convenience these definitions have been standardized, and are collectively known as the mathematical toolkit (first presented in [Spi92b], and standardized in [ISO02]).

The mathematical toolkit has a number of components each containing a number of definitions, and they can be used individually or in combination. The contents of each toolkit (the available toolkits are the set, relation, function, number, sequence and standard toolkits) are self-evident from their nomenclature. There is some dependence between the component sections which is illustrated in figure 4.4 on the next page. In fact, the standard toolkit contains no definitions of its own and simply provides an interface to

the entire toolkit allowing a section to include all the standard definitions through the declaration of a single parent section.

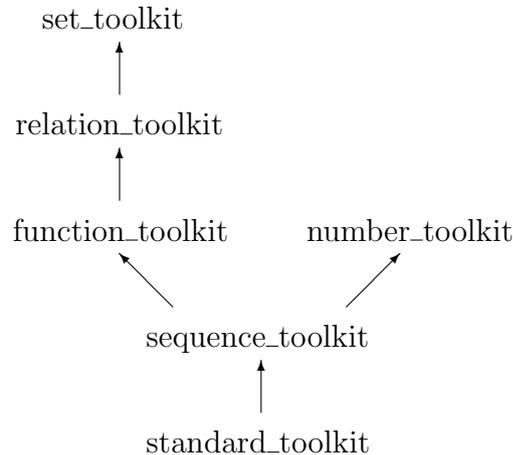


Figure 4.4: Parent relation between sections of the mathematical toolkit (from [ISO02])

A Z parser needs to ensure that the mathematical toolkits are available to a user. However, there can be no assumption that a user wishes to import the standard toolkit by default. A problem arises as we cannot reserve the names of the toolkits (for example, ‘*standard_toolkit*’) as a user must be able to specify their own toolkits and they may wish to use these names, or alter the existing toolkits. We solve this quandary by associating the toolkits available with a Frog specification. When the specification is created the user can choose which toolkits they wish to include by default, these toolkits are then available to be used as parent sections by any section within the specification. Of course, it is still possible to incorporate further toolkits, but the user must do so manually (that is, fetch the toolkit specification and parse it).

To provide additional flexibility, we also make the inclusion of the standard prelude optional. The user must define their own prelude section, however, as all sections are expected to have it as their parent. Certain definitions are expected in the prelude section, and if these definitions are missing then

behaviour cannot be guaranteed. It is, therefore, recommended that definitions are only added to the prelude section (making them available to every other section) and not removed.

Note that, should the definitions of toolkits change following the creation of a specification, we will assume that the required definition of a toolkit is the one available at creation time. Again, this also applies to the prelude section.

4.5 Ordering

One of the common difficulties found with parsing a *Z* specification is that the scope of an operator template includes the whole of the section in which it is defined. This means that it is possible to use an operator before it has been defined. This is a problem with *Z* as the definition of the operator template includes instructions on how to parse the operator. For example, the following operator template is used to define the relation operator.

```
generic 5 rightassoc(_ ↔ _)
```

This indicates that the operator is right associative, has a relative precedence level of five and is a binary, infix operator. There is no way that we can parse a use of the operator until we have this information. Similarly, there is no mandate on a specifier to include the section of a specification in such a way that ancestors will appear before their descendants.

Some tools, such as *fUZZ*, perform a preliminary pass of the specification and determine a dependency graph between paragraphs (and sections) and then reorder those paragraphs so that these problems will not occur when we proceed to the main parsing process. Other tools, such as *CZT*, require that paragraphs (and sections) are ordered so that definitions of operators appear before their use (or parents appear before their children). In [MU05], the authors argue that such a restriction helps to improve the human readability of a specification.

We decided that, initially, like *CZT*, we would work under the assumption

that sections and paragraphs are declared before references to them occur. We felt that the reordering of a file's contents may have an adverse effect on our attempts to incorporate constructs (see chapter 5). Having said this, given extra resources, we would like in the future to consider the possibility of incorporating an approach similar to that used by *fuzz*.

4.6 Syntax Checking

We use the term syntax checking to refer to the complete parsing process that involves transforming a specification into a typed abstract syntax tree. This section describes this process. Firstly we examine the representations of the Z notation that are available, and confirm which notation we will use for our specifications. We then provide an introduction to ANTLR – the parser generator which we use to build our parsers – as its use affects the style of the grammars we use to parse our specifications. We then provide a summary of each of the principal stages of syntax checking, beginning with lexing and parsing, before progressing to syntax transformation and type checking.

4.6.1 A Z Notation

There have been many representations of the Z notation that all attempt to provide a balance between something that is easily readable by humans, and something that can be easily processed by tools. A fully considered view of these representations, the translation between them and their transformation into a traditional rendering of Z is given in [TS02].

Initially, the most common representations of the notation were based on the notation of type setting tools such as \LaTeX [KD99, Lam94] and troff [Ker78]. These representations were easily rendered using the appropriate tools and were in a style already familiar to most of their users.

A number of ASCII representations have been created (such as ZTC's ZSL, or the email representation proposed in [ISO02]). These representations tend to be useful when creating a specification as the notation is usually concise. They can, however, be difficult to read as there is no way to render them

in the correct mathematical notation without translation to a suitable format. More recently, unicode versions of the Z notation have been proposed, but while these are much easier to read, given their immediate rendering of mathematical symbols, they still do not have the ability to render the boxes that define Z's structure.

ZML [SDLW02, UTS⁺03] is the most recent representation to gain popularity and is based on the XML standard. ZML is the central language used for the CZT platform, and is considered to be the natural successor to the \LaTeX representation defined in [ISO02]. An advantage of ZML is that it can be used not only to provide a representation of a specification, but also of a parsed and checked specification (that is, it has the flexibility to record how a specification has been parsed and typed). This means that tools using this format will not only be able to read each other's specifications, but also be able to import the typed ASTs that belong to those models. The principal disadvantage to using ZML, is that it is a standard that is still very much in its infancy and is still under constant evolution (there have been six specifications of ZML produced between 2003 and 2006). We felt that when we embarked upon our project to create a Z parser the ZML standard was too fluid to use as a basis for that parser.

In conclusion therefore, we felt that if we were going to create a parser for ISO standard Z, then we should parse a representation of that notation that has been fixed within that standard. This leaves us with a choice of either the \LaTeX or email representations. Hence, we decided that initially our tool would process the \LaTeX representation of the notation as it is most familiar.

Note that, once an AST has been created it is a relatively simple task to produce a copy of a specification in any of the above formats. Therefore, when we considered the representation to parse, we were selecting solely our initial input language. We would still be able to inter-operate with other tools through the creation of tree parsers that can operate on our typed ASTs and, therefore, produce specifications in the alternate representations.

4.6.2 ANTLR

ANTLR³ (ANother Tool for Language Recognition) [Mil05, PQ95, Par05] is a parser generator that takes a grammar and creates code for lexers, parsers and tree parsers in a number of programming languages. ANTLR was formerly known as PCCTS [Par96].

ANTLR creates table-based, top-down parsers, using a fixed arbitrary lookahead (k). ANTLR also allows the use of semantic and syntactic predicates that allow parsing decisions to be made based on the semantic environment, or through the use of extra lookahead. For these reasons, the parsers created by ANTLR are considered to be predicated LL(k) parsers.

The use of k greater than one, significantly reduces the need to left-factor grammar rules, and makes the design of the grammar much simpler. The initial research that led to the development of LL parsers with a lookahead greater than one, is described in [Par93].

The choice of ANTLR for the development of our tool was based very much on its flexibility, and the ease with which grammars could be created. The use of semantic and syntactic predicates allows Z 's inherent ambiguity to be resolved with much less difficulty than with traditional parser generators. The ability to generate parsers for abstract syntax trees was also a big advantage as it meant that once a tree was generated in the syntax checking phase, it could be transformed in situ during the remaining phases.

The ANTLR Metalanguage

ANTLR specifies its lexer and parser rules with an almost identical notation which itself is based on the syntax of Yacc [LMB92]. Supplementing the Yacc-like syntax, ANTLR introduces a number of constructs. ANTLR defines constructs that provide some equivalents to the metalanguage of the extended BackusNaur form and two further constructs that allow for predicated parsing. A list of the constructs we use in this thesis (along with their definition) can be found in table 4.1 on the facing page.

³Throughout this thesis, any reference to ANTLR refers to the stable, version two of the tool, and not the proposed version three in which a number of core features have been

Table 4.1: ANTLR's Metalanguage

Construct	Definition
$(expression)^*$	Match zero or more instances of the pattern matching expression indicated within the parentheses.
$(expression)^+$	Match one or more instances of the pattern matching expression indicated within the parentheses.
$(expression)?$	Match zero or one instances of the pattern matching expression indicated within the parentheses.
$expression1 \mid expression2$	Match either one of two pattern matching expressions.
"string"	Match the string contained within the quotation marks.
$(expression1) \Rightarrow expression2$	Syntactic predicate (see below).
$\{java_expression\}? expression2$	Semantic predicate (see below).

Syntactic Predicates

ANTLR's syntactic predicates allow us to make parsing decisions that cannot be reached with a finite lookahead. Consider the following ANTLR rule.

```
myRule : (A)+ B
        | (A)+ C;
```

The parser cannot guarantee to disambiguate between the two alternatives; each allows a unboundedly large number of A tokens before a decision can be made. With a finite lookahead, therefore, we can never know which alternative we require. It is possible to 'left-factor' our rule, as follows.

altered.

```
myRule : (A)+ (B | C);
```

While this allows the same set of token streams to be parsed, all but the simplest of grammars require embedded actions. While we have one set of actions for processing A tokens followed by a B token, we may have an entirely different set of actions for processing A tokens when they are followed by a C token.⁴

ANTLR's syntactic predicates allow us to use arbitrary lookahead at any point within a rule so that potentially unbounded strings can be used to predict the correct path. In reality, these predicates are a form of selective backtracking; the parser attempts to process the predicted predicate as normal, but without processing the associated actions. If a successful match is found we return to the location of the predicate and can continue processing (with the actions now activated). If no match is found then we skip the part of the rule associated with the predicate (that is, the production). The syntax for a syntactic predicate is as follows.

(prediction) => production

If we re-examine our initial rule, we can now produce an unambiguous grammar that recognizes the same language, as shown below.

```
myRule : ( (A)+ B ) => (A)+ B
        | (A)+ C;
```

Note that, the processing of alternatives within a rule in ANTLR is ordered from top to bottom, so we can guarantee that we only ever attempt to match our second branch when the syntactic predicate on the first cannot be matched.

A good example of syntactic predicates in action is in the provision of a solution to the 'dangling-else' problem (see section 4.3 of [ASU86] or section 7.1.1 of [Lou93]). This problem involves the parsing of a statement such as the following.

⁴Left-factored grammars are also generally considered to be significantly less readable – and hence less maintainable – than grammars using syntactic predicates.

```
if a then if b then c else d;
```

The question is whether the `else` belongs to the first `if` or the second. The default behaviour of ANTLR is to attach the `else` to the nearest `if` which is achieved with the first of the two rules presented below⁵. The second rule uses a syntactic predicate to produce the alternate behaviour where the `else` will be matched to the outermost `if`.

```
statement
```

```
: "if" LPAREN expression RPAREN "then" (statement)*
  ( "else" (statement)*)?
| ... ;
```

```
statement
```

```
: "if" LPAREN expression RPAREN "then" (statement)*
  ( ("else")=> "else" (statement)*)?
| ... ;
```

Semantic Predicates

ANTLR's semantic predicates allow us to make parsing decisions based on the context of the surrounding program. For example, suppose our program has a setting that either allow underscores to be included with names or not. Without semantic predicates we would need to write two grammars (or at least produce a considerably more complex grammar) in order to be able to alter the behaviour of our parser on the value of this setting.

A semantic predicate in ANTLR has the following syntax, and is placed immediately before the matching code for a rule alternative.

```
{java_expression}?
```

If the *java_expression* evaluates to true at run time then the alternative will

⁵ANTLR will produce ambiguity warnings if using this alternative, but these can be eliminated by explicitly stating that the matching of the `statement` in the `then` clause is greedy.

be processed, otherwise it will be skipped. So for the example described above, we may have the following rule for a name token.

```
NAME : { useUnderscores == TRUE }? ('a'..'z' | 'A'..'Z' | '_' )
      | { useUnderscores == FALSE }? ('a'..'z' | 'A'..'Z');
```

In more complex grammars it is possible that the value of the semantic predicate changes over time. So, using the example above, we may have a grammar that accepts names with underscores in one context and not in another.

Semantic predicates can also provide a function similar to that of assertions in Java (see chapter 6 of [Tra02]). In this instance, they are placed anywhere other than immediately before an alternative. If the condition in the semantic predicate is met, nothing happens; if, however, the condition is not met, an exception is thrown (note that, like assertions these ‘validating predicates’ should not be used to test conditions but to assert that something is expected to be true and that there is a bug present if it is not).

4.6.3 Lexing

The lexing phase involves transforming the stream of characters that form a user’s Z specification into a stream of tokens that can be understood by a parser. The lexing phase is notionally the first stage of syntax checking a Z specification. In actuality, the lexing and parsing phases are inter-dependent. There are points at which a token can be lexed differently according to its context. That is, the way in which a character string is assigned a token can depend upon the parsing of previous tokens. When using ANTLR we – in effect – perform the lexing and parsing phases in parallel. This means that we perform a single pass of the specification in the syntax checking process.

The grammar we use is equivalent to that described in section 7 of [ISO02]. The grammar presented there is in a human readable format, and we have had to make alterations to the exact specification in order to provide a machine implementable version. There are also a number of issues described in that section that describe desired behaviours of a valid lexer, but it leaves open

some of the details about how those behaviours should be implemented. We make clear our decisions in the remainder of this section. A full annotated version of the lexical grammar is provided in appendix E, we present only the interesting details below.

Token Disambiguation

There are a number of places in the grammar where additional, contextual rules are required to disambiguate between the type of the token, or the way in which it should be encapsulated.

A first example is the handling of new line characters. The new line character is intentionally ambiguous in Z, and can be used to conjoin a number of clauses or simply to break long lines. In the \LaTeX representation this becomes more interesting as there is a separation between the explicit new line character `\\` which forces a new line in the rendered document, and the new line character used to make the mark-up itself more readable – `\n`. [ISO02] has a discussion of these issues in sections 7.5 and A.2.7, with a simple set of rules provided. In summary, new line characters in the mark-up itself (`\n`) are considered ‘soft’ and are simply ignored. New line characters that form part of the specification are considered ‘hard’ and must be parsed as `NLCHAR` tokens. Explicit new line characters have a set of rules that enable us to characterize each instance as either ‘hard’ or ‘soft’. When we lex an explicit new line character, it is necessary for us to examine the token stream to determine the token preceding the new line character, and peek ahead to examine the token following the new line character. Each of these tokens is then placed into one of four categories that indicate whether it is possible for a preceding new line character to be soft, a following new line character to be soft, both or neither. If either the token preceding or following the new line character allows it to be soft then it will be so, otherwise the new line character is hard and is assigned a `NLCHAR` token. For example, examine the following excerpt from a \LaTeX specification.

```
a' = b + c \\
a > 1 \implies \\
```

`b' = c`

The first new line character is preceded and followed by a `NAME` token. `NAME` tokens are categorized as allowing soft new lines neither before nor after, so our first new line character must be parsed as a ‘hard’ `NLCHAR` token. The second new line character is followed by a `NAME`, but is preceded by an `IMPLIES` token. The `IMPLIES` is categorized as requiring soft new lines both before and after, so our new line character must be soft, and is simply skipped in the lexing process.

A similar issue arises when dealing with white space generally. Namely, it is important to make a distinction between the following two lines of a `LATEX` specification. (This issue is discussed in section A.2.7 of [ISO02].)

```
\Delta a \\
\Delta~a
```

The first should be lexed as a single `NAME` token and rendered ‘ Δa ’, whilst the second should be lexed as two `NAME` tokens and rendered ‘ Δa ’. In order to achieve this our grammar makes a distinction between explicit white space (for example ‘`~`’), and non-explicit white space (for example, ‘’). When we lex a Greek letter (that is a token of type `GREEK`) we consume all non-explicit white space following that letter, whilst explicit white space is unaffected. Note that, this rule only applies to Greek letters; for example, the following excerpt will be lexed (initially) as two `NAME` tokens.

```
\power A
```

The Character Dictionary

A method for introducing new characters into the `LATEX` mark up is described in appendix A of [ISO02], and in more detail in section 6 of [TS02]. These works describe the use of the ‘`%%ZChar`’ mark up directive to provide a translation between a latex directive and its unicode equivalent. For example, we may define the symbol, \mathbb{N} , with the following command.

```
%%ZChar \nat U+2115
```

We have decided not to use this method for specifying character translations. This is for a number of reasons. Firstly, we would like to provide a tool wide system so that the translations do not need to be defined specifically for every individual specification. Secondly, while we are currently using \LaTeX as the sole formatting for our Z specifications, the future may see us need to use other formats, and hence require alternate translation definitions. Finally, we would like to be able to specify all translation rules in a single place; the \LaTeX directives only provide a rule to translate between the \LaTeX notation and unicode, and we need rules to translate to a number of formats (for example, the notations of theorem provers).

Instead of the \LaTeX directives, therefore, we provide a single ‘characters’ file in which all translation rules are defined. An excerpt of a sample file is given in figure 4.5.

```

...
{DEFAULT=UNICODE, PLAIN="%N", LATEX="\nat",
  UNICODE="2115", HTML="&#8469;"}
{DEFAULT=UNICODE, PLAIN="%Delta%", LATEX="\Delta",
  UNICODE="0394", HTML="&#916;"}
{DEFAULT=LATEX, PLAIN="%arithmos", LATEX="\arithmos",
  HTML="<I>A</I>"}
{DEFAULT=LATEX, PLAIN="%P\v1^\", LATEX="\power_1",
  HTML="&#8473;<SUB><SMALL>1</SMALL></SUB>"}
{DEFAULT=LATEX, PLAIN="%E\v1^\", LATEX="\exists_1",
  HTML="&#8707;<SUB><SMALL>1</SMALL></SUB>", ISABELLE="EX!"}
{DEFAULT=LATEX, PLAIN="-||->>", LATEX="\ffun"}
{DEFAULT=LATEX, PLAIN=">-||->>", LATEX="\finj"}
{DEFAULT=UNICODE, PLAIN="%u", LATEX="\cup",
  UNICODE="222a", HTML="&#8746;", ISABELLE="\<union>"}
{DEFAULT=UNICODE, PLAIN="%n", LATEX="\cap",
  UNICODE="2229", HTML="&#8745;", ISABELLE="\<inter>"}
{DEFAULT=PLAIN, PLAIN="("}
...

```

Figure 4.5: Excerpt from sample character dictionary configuration file

The rules for each character are enclosed within a pair of braces. Within these braces are defined the various representations of the character, no particular representation is used to index the characters as we do not require the definition of a string for every representation. As each symbol may not be defined for each of the required representations we allow the specification of a default representation which will be used whenever a requested representation is not available. Note, that the unicode representation is given simply as the hexadecimal reference to the required characters, and that the HTML representation may be a single character or a sequence of characters and mark-up directives.

For example, the symbol, \mathbb{N} , can be represented by `%N` in an ASCII environment, by the L^AT_EX symbol `\nat`, by the unicode character `U+2115`, or by the special character `ℕ` in HTML. We do not define an Isabelle representation as the definition of the natural numbers in our specification may be different to that defined by Isabelle.

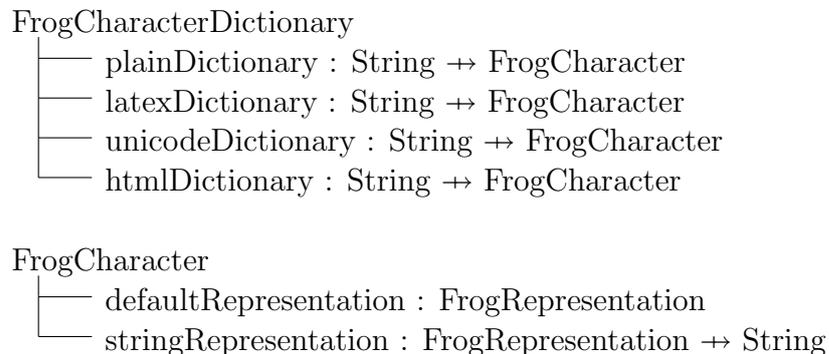


Figure 4.6: Dictionary classes

Figure 4.6, outlines the structure of the classes we use to index the information retrieved from the characters files. Each character defined within the file is an instance of the `FrogCharacter` class which contains a default representation, and a partial function between representation and the string that gives the rendering information for the character in that representation. The `FrogCharacterDictionary` class indexes the available characters allowing

a user to perform lookups on the plain, \LaTeX , unicode, and HTML representations. Unicity is ensured upon the population of the dictionary. It should be noted that the dictionary is populated on the creation of a specification and saved with it. Therefore, any changes to the characters file will not affect existing specifications. At present we do not consider it wise to allow changes to the way in which characters are parsed within an existing specification. We feel that allowing dynamic changes would make ensuring the integrity of the specification's existing files extremely difficult. We have considered allowing users to *add* translation rules to an existing specification through the tool, but have not presently implemented such functionality.

As well as being useful for translation between character representations we use the character dictionary in the lexing process. When lexing a \LaTeX character stream we will attempt to match any directive (that is any character sequence beginning with ‘\’) used within the Z delimiters. We then use a lookup on the character dictionary to determine whether that directive is a valid symbol descriptor, or is a misplaced \LaTeX directive. At present we reject all formatting (or other \LaTeX) directives, but we have considered allowing certain directives as their use could easily be regulated by the character dictionary. That is, we could have certain symbols – for example, `\hspace` – which users may find useful in formatting their specifications, that are not rejected by the lexer, but that are not passed to the parser.

Token Environments

The use of user-defined operators is described in section 7.4.4 of [ISO02]. We use token environments to administer the use of the tokens used within user-defined operators. Each token environment contains a partial function from names to the derived token type. The available token types are shown in table 4.2 on the following page (from [ISO02]).

Table 4.2: User defined token types

Type	Definition
PREP	Prefix unary relation
PRE	Prefix unary function or generic
POSTP	Postfix unary relation
POST	Postfix unary function or generic
IP	Infix binary relation
I	Infix binary function or generic
LP	Left bracket of non-unary relation
L	Left bracket of non-unary function or generic
ELP	First word preceded by expression of non-unary relation
EL	First word preceded by expression of non-unary function or generic
ERP	Right bracket preceded by expression of non-unary relation
ER	Right bracket preceded by expression of non-unary function or generic
SRP	Right bracket preceded by sequence of non-unary relation
SR	Right bracket preceded by sequence of non-unary function or generic
EREP	Last word followed by expression and preceded by expression of tertiary or higher relation
ERE	Last word followed by expression and preceded by expression of tertiary or higher function or generic
SREP	Last word followed by expression and preceded by sequence of tertiary or higher relation
SRE	Last word followed by expression and preceded by sequence of tertiary or higher function or generic
ES	Middle word preceded by expression of non-unary operator
SS	Middle word preceded by sequence of non-unary operator

When we begin parsing a Z specification we obviously do not know which strings will be used by the user in their user-defined operators. In order that we can associate words with one of the specified token types the user defines operator templates. For example, the user-defined operator relation (typically defined in the set toolkit) may have an operator template as follows.

```
generic 5 rightassoc (- ↔ -)
```

The first word indicates whether the operator is a relation, function or generic. The second – only used for function and generics – indicates the precedence of the operator. The associativity of the operator is then expressed (where necessary), and finally the tokens that are used to define the operator are specified. The template may contain a single token as above, or a number of token as in the specification of the template for relational image below.

```
function 90 (-(| - |))
```

When parsing the operator template we add the tokens to the current token environment with their associated types. A new token environment is created for each Z section, and the token environments of that section's parents are considered the parents of that token environment. If a section has more than one parent, and the token environments belonging to those parents contain tokens that clash we will report an error and cease processing (the exact rules for determining whether re-use of a token is a clash are explained in detail in [ISO02]).

We can split the operator template into one of four categories using the grammar specified in [ISO02]. As this grammar is inherently ambiguous we use ANTLR's syntactic predicates to peek ahead before committing ourselves to a particular route. The ANTLR grammar for the template is as follows (note that, by this point the keyword, precedence and associativity would already have been matched).

```
template : (prefix_template) => prefix_template
         | (postfix_template) => postfix_template
```

```

        | (infix_template) => infix_template
        | (nofix_template) => nofix_template
    ;

prefix_template : LEFTBR NAME ((UNDERSCORE | DBLCOMMA) NAME)*
                UNDERSCORE RIGHTBR;

postfix_template : LEFTBR UNDERSCORE NAME
                 ((UNDERSCORE | DBLCOMMA) NAME)* RIGHTBR;

infix_template  : LEFTBR UNDERSCORE NAME
                 ((UNDERSCORE | DBLCOMMA) NAME)* UNDERSCORE
                 RIGHTBR;

nofix_template  : LEFTBR NAME ((UNDERSCORE | DBLCOMMA) NAME)*
                 RIGHTBR;

```

In essence, the syntactic predicate allows us to check each option until we find a successful one. Whilst this is not particularly efficient, operator templates are unlikely to be very big, so we consider the advantage of maintaining a grammar very similar to the [ISO02] specification to be greater than the disadvantage of the performance loss.

Once we have determined the ‘fix’ of the operator template, we can assign the tokens their token types. For example, with a prefix template we first create a queue of the names enclosed in the template. If this queue has a single element then we know we have a unary operator, and can assign the single element to have a token type either `PREP` or `PRE` depending on whether we are dealing with a relation, function or generic. If the queue has more than one element, we process it iteratively. The first element being given a type `LP` or `L`, each next element (until the last) is given a type `ES` (if preceded by an `UNDERSCORE` token) or `SS` (if preceded by a `DBLCOMMA` token). The final element will be given the type `EREP`, `ERE`, `SREP`, or `SRE` again depending on the template type and whether it should be preceded by an expression or

a sequence. The token types for templates of other ‘fixes’ are calculated similarly.

For example, in the operator templates above, \leftrightarrow will be added to the current token environment with the token type I; (| with the type EL; and |) with the type ER.

When we are lexing NAMES, therefore, it is essential that before we finalize the token type as NAME, we check the current token environment to determine whether that NAME has been specified in an operator template, if so we retrieve the associated token type and use that instead of NAME. For example, if we lexed the following character stream.

$$a \leftrightarrow b$$

We would normally expect a resultant token stream of NAME NAME NAME. If, however, we had defined the operator templates above, the token stream produced should be NAME I NAME.

Example 4.1

We now begin a simple example to show how a typical Z section is processed. We will take this example through all the stages of the parsing process. In this initial part of the example we show how a L^AT_EX character stream is transformed into a token stream.

We revisit the model of the train yard we presented in example 2.1. The L^AT_EX representation syntax that matches the specification we presented previously is given below.

```
\begin{zsection}
\SECTION theTrainYard \parents standard\_toolkit
\end{zsection}
```

```
\begin{zed}
[trains]
\end{zed}
```

```
\begin{schema}{trainYard}
trainsInYard: \power trains
\end{schema}
```

```
\begin{schema}{Init\_trainYard}
trainYard
\where
trainsInYard = \emptyset
\end{schema}
```

```
\begin{schema}{trainEntersYard}
\Delta trainYard \\\
trainEnteringYard? : trains
\where
trainEnteringYard? \notin trainsInYard \\\
trainsInYard' = trainsInYard \cup \{ trainEnteringYard? \}
\end{schema}
```

```
\begin{schema}{trainLeavesYard}
\Delta trainYard \\\
trainLeavingYard? : trains
\where
trainLeavingYard? \in trainsInYard \\\
trainsInYard' = trainsInYard \setminus \{ trainLeavingYard? \}
\end{schema}
```

If we examine first the section declaration, we match the first character string – `\begin{zsection}` – with the rule `RESOLVE`⁶, which sets the type of the generated token to `BOXCHAR`. When this token is passed to the parser, however, its type indicates that it should be checked against the literals table, and the type of our token is refined to `SECTION`. A similar process proceeds

⁶See appendix E for the annotated lexical grammar.

for the remainder of the declaration, `\SECTION` and `\parents` are determined to be literal refinements of `KEYWORD` tokens, with `theTrainYard` and `standard_toolkit` lexed as `NAMES` (that is they matched no other `RESOLVE` alternatives. The final token – `\end{zsection}` – is simply matched as a token of type `END` as no distinction is made between terminating tokens, and therefore no literal lookup is required.

The given type paragraph and initialization paragraphs are lexed in much the same way. The third and fourth – schema definition – paragraphs are more interesting; we can see that the ‘`\`’ character is lexed as a hard new line in all instances as it precedes and follows a `NAME` token. Also, the ‘`\cup`’ and ‘`\setminus`’ operator tokens are both resolved by the `RESOLVE` rule as `NAMES`, and are only reassigned to the `I` type following a lookup on the character dictionary.

Note that ‘`\in`’ produces a literal token (`IN`) as it forms part of the core `Z` language rather than being introduced through an operator template. On the other hand the token type for ‘`\notin`’ is initially `NAME`, and changed to `IP` following reference to the character dictionary (the operator templates for `\cup` and `\notin` are included in the standard toolkit).

The final token stream produced for the train yard section is as follows. (We have just shown the token types and introduced new lines to separate paragraphs for convenience.)

```
SECTION KW_SECTION NAME PARENTS NAME END
```

```
ZED LEFTSQ NAME RIGHTSQ END
```

```
SCH LEFTCU NAME RIGHTCU NAME HALFPIPE NAME EQUALS NAME END
```

```
SCH LEFTCU NAME RIGHTCU NAME NLCHAR NAME COLON NAME HALFPIPE NAME
IP NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
END
```

```
SCH LEFTCU NAME RIGHTCU NAME NLCHAR NAME COLON NAME HALFPIPE NAME
```

```
IN NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
END
```

4.6.4 Parsing and Syntax Checking

The parsing process involves the transformation of the token stream produced by the lexing process into an abstract syntax tree (AST). This process also involves the elimination of Z sentences that are syntactically incorrect. As discussed in section 4.6.3, this process takes place concurrently with the lexing process.

The grammar used by this process is a machine-implementable version of the human-readable grammar presented in section 8 of [ISO02] (hereafter referred to as ‘the human-readable grammar’). A full, annotated version of our grammar is presented in appendix F. In the following section we describe how we have handled issues raised in [ISO02], and present solutions to other issues of interest that have arisen.

Disambiguating Expressions and Predicates

Section 8.4 of [ISO02] describes the ambiguity of the logical operators inherent in the human readable grammar presented therewithin. The logical operators, \wedge , \vee , \neg , \Rightarrow , \Leftrightarrow , \forall , \exists and \exists_1 , can all be used correctly in both predicates and expressions. However, when a predicate is expected it is possible to match an expression in its place. This leads to an ambiguous situation. For example, consider the following Z fragment (where a and b are valid expressions).

$$a \wedge b$$

The human readable grammar could generate either of the trees shown in figure 4.7 when parsing this fragment.

In essence, we have two expressions being used as predicates that are conjoined to form a single predicate, or a conjunction of expressions used to form a single expression that is then used as a predicate. As the meaning of

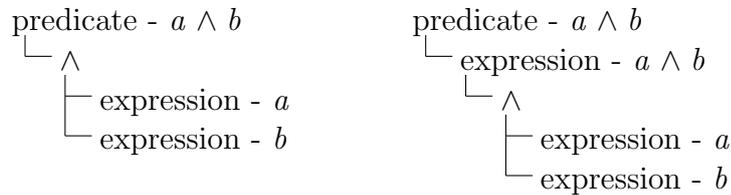


Figure 4.7: Trees created from Z fragment

both interpretations is equivalent the actual implementation is unimportant. In order for us to eliminate ambiguity in our machine-readable grammar, however, we have had to define restrictions that will ensure that we use a consistent approach to predicates formed in this way.

In order to ensure an invariant approach we use a combination of syntactic and semantic predicates, and also take advantage of ANTLR's ability to give grammar rules arguments. We use syntactic predicates to ensure that if we are expecting a predicate, and our token stream begins with a logical operator, we will first attempt to match a predicate. For example, the following sub-rule states that if the token stream begins with the universal quantification symbol (\forall), then we will attempt to match a predicate.

```

pred_quantification
: (FORALL) => FORALL^ schema_text AT pred_quantification
| ... ;
  
```

When parsing an expression, it would now be useful to know whether the current expression is nested within a predicate. We achieve this by providing an argument to the `expression` rule. This argument will have one of two values: `PART_OF_PREDICATE` or `NOT_PART_OF_PREDICATE`. At all points in the grammar where we attempt to match an expression, therefore, it is necessary to indicate whether – contextually – that match will be part of a predicate (in some cases the argument can be passed as it is received). For example, the following excerpt from the grammar shows a case where the expression is part of a predicate, is not part of a predicate, and also where the value of the argument is inherited⁷.

⁷Note that, an additional syntactic predicate is present on the expression alternative to

```

pred_atom
  : ...
  | (expression[PART_OF_PREDICATE])
    => expression[PART_OF_PREDICATE]
  | ...;

expr_fn_construction[boolean isPartOfPredicate]
  : LAMBDA^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
    | expr_definite_descr[isPartOfPredicate];

```

We use this argument to prevent logical operators being matched when processing the expression rule whilst within a predicate. This can be achieved through the use of semantic predicates, and can be seen in the following rule, which restricts access to the universal quantification alternative of an expression sub-rule.

```

expression[boolean isPartOfPredicate]
  : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    FORALL^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
  | ...;

```

Similar semantic predicates are placed on all logical operators within the expression rule, and therefore we ensure that the first of the two trees in figure 4.7 on the previous page will always be produced. That is, a predicate formed of expressions will always be matched as a series of expressions joined by logical operators, rather than a single expression.

It should be noted that, the presence of the semantic predicate actually makes the syntactic predicates on the quantification operators in the predicate rule redundant. However, the presence of these rules provides a performance advantage as ANTLR processes syntactic predicates considerably more efficiently than semantic predicates.

distinguish a predicate enclosed in parentheses from an expression enclosed in parentheses.

Disambiguating Set Comprehension and Set Extension

Note 1 of section 8.1 in [ISO02], highlights the ambiguity present in the human-readable grammar when trying to distinguish between the comma used in set comprehension, and set extension expressions. Furthermore, we note the ambiguity between these expressions, and characteristic set comprehension expressions. For example, we may have the following expressions to parse.

$$\begin{aligned} &\{a, b : \mathbb{N} \mid a - b = 5 \bullet a\} \\ &\{a, b\} \\ &\{a, b : \mathbb{N} \mid a - b = 5\} \end{aligned}$$

With a single token of lookahead it is impossible to distinguish which of the three possible expression-types we are parsing. If we increase the lookahead, we simply displace the problem (that is, we cannot place an upper limit on the number of expressions in the set). In order to resolve this ambiguity we take full advantage of ANTLR's syntactic predicate functionality. An excerpt from the 'expression atom' grammar rule (presented in full in appendix F) is shown below.

```

expr_atom
: ...
| (LEFTCURLY schema_text AT
  expression[NOT_PART_OF_PREDICATE] RIGHTCURLY)
=> LEFTCURLY! schema_text AT
    expression[NOT_PART_OF_PREDICATE] RIGHTCURLY!
    { setRoot(#expr_atom,SET_COMP); }
| (LEFTCURLY expression[NOT_PART_OF_PREDICATE])
=> LEFTCURLY!
    (expression[NOT_PART_OF_PREDICATE]
    (COMMA expression[NOT_PART_OF_PREDICATE])*
    )?
    RIGHTCURLY!
    { setRoot(#expr_atom,SET_EXT); }

```


expression involving a schema. For example, we may have the following expressions to parse.

$$e[a, b]$$

$$e[a, b : \mathbb{N}]$$

Again, it is impossible to distinguish with a single token of lookahead, and we use syntactic predicates to resolve the ambiguity. The relevant portion of the grammar is shown below.

```

expr_application
: (application) => application
| expr_decoration
  (expr_decoration
   { setRoot(#expr_application,APPLICATION); }
  )?;

expr_atom
: (ref_name)
  => ref_name
  ((LEFTSQ expression[NOT_PART_OF_PREDICATE]
   (COMMA expression[NOT_PART_OF_PREDICATE])* RIGHTSQ)
 => (LEFTSQ expression[NOT_PART_OF_PREDICATE]
   (COMMA expression[NOT_PART_OF_PREDICATE])*
   RIGHTSQ)
  )?
| ...;

```

One obvious difference between this and the human-readable grammar, is that the expression rule has been broken down into a number of sub-rules. This allows us to handle precedence more easily within ANTLR's grammatical syntax. The first of our sub-rules handles application. The first alternative covers function or generic operator application, whilst the second can cover the application of one expression to another. Note that, ANTLR allows us to place Java code within the grammar rules. For example, in

the second alternative of our first sub-rule, the code to set the tree root to be a token of type `APPLICATION` will only be called if the optional second `expr_decoration` is matched. (For function and generic operator application we use a special root token when creating a tree for the `application` rule, this is discussed further below.)

It may not be obvious at first where the disambiguation occurs in the rules specified above. The syntactic predicate involving the `application` rule allows us to distinguish – for example – between expressions, and expressions followed by postfix function or generic operators, but this is not what concerns us here. The ambiguity with which we are involved is resolved by the second syntactic predicate in the `expr_atom` rule. This syntactic predicate allows us to consider a generic instantiation as a single expression (which is, of course composed of other expressions); if at any point we match a `ref_name`, we check whether that name is actually part of a generic instantiation expression, and if so, we match the remainder of that expression wrapping up the result as a single match of the `expr_atom` sub-rule ensuring that when this expression propagates back up to the `expr_application` level it is maintained as single match of the `expr_decoration` sub-rule, and cannot be confused with the two matches of that rule required for an application.

Of course, these are not the only places in which it is necessary to use techniques to remove the ambiguity in the human-readable grammar presented in [ISO02]. We have described these instances here as the standard brings particular attention to them. For further details of the disambiguation techniques used please see appendix F.

Associativity and Precedence

Whilst handling the associativity and precedence of the Z core language is relatively simple – as the rules are implicitly enforced by a valid grammar – the correct determination for user-defined operators can be more complex. We will not handle user-defined operators through grammatical rules, as the operators may be defined in the same section in which they are used, and while

this may be handled through the use of dynamic grammar generators⁸, the development of such a tool is needlessly complicated. Instead, therefore, we choose to use a form of post-processing. After the use of a user-defined operator (for instance, in an application or relation), we first determine whether we are in a situation in which a precedence or associativity clash can occur. If so, we then determine the precedence and associativity of the operators involved and use those values to determine whether the tree is correctly structured, if not we transform it. The use of separate tokens for user-defined relation operators and for user-defined function and generic operators, simplifies the situation somewhat as we do not need to consider clashes between relations and (function) applications, nor those core notations whose precedences fall between the two.

The first stage in the process is to determine the situations in which a precedence or associativity clash can occur. Table 4.3 shows the token streams in which we will need to examine the operator precedences. Note that, for example, the term *prefix* will refer to a PRE token in applications, and a PREP token in relations.

Table 4.3: Token streams that can require precedence resolution

Token Stream	First Resolution	Second Resolution
<i>prefix e postfix</i>	<i>(prefix e) postfix</i>	<i>prefix (e postfix)</i>
<i>prefix e₁ infix e₂</i>	<i>(prefix e₁) infix e₂</i>	<i>prefix (e₁ infix e₂)</i>
<i>e₁ infix e₂ postfix</i>	<i>(e₁ infix e₂) postfix</i>	<i>e₁ infix (e₂ postfix)</i>
<i>e₁ infix e₂ infix e₃</i>	<i>(e₁ infix e₂) infix e₃</i>	<i>e₁ infix (e₂ infix e₃)</i>

Once we have determined that we are in a situation where a clash may occur, we must examine the relative precedences of the two operators. For example, we may have the following Z sentence.

⁸ANTLR has the flexibility that allows us to create parsers on the fly; hence it is theoretically possible to dynamically create grammars that incorporate the user-defined operators required.

$$a \cup b \cap c$$

From this our initial parsing will give us the tree shown in figure 4.9. Note that, when creating the root node for each infix application we make a note of the full operator name⁹ that we have derived from the arguments to that operator when creating the tree; this ensures that we do not need to re-parse a child operator tree to determine its full operator name when fetching that operator's precedence and associativity from the current operator environment.

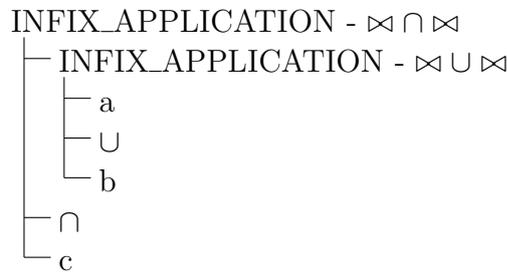


Figure 4.9: Initial parse tree for $a \cup b \cap c$

We check the precedences of the operators involved in our potential clash immediately after the construction of the tree. That is, we do not wait until we have created a tree for an entire Z file and then perform post-processing, we resolve all precedence and associativity issues whilst in the process of constructing the tree from the bottom-up. In our example, therefore, we would never actually create the tree shown in figure 4.9. Once it became apparent that we were parsing an infix application and that one of the arguments to its operator was also an infix operator, we would know that we have one of the potential clash situations described in table 4.3 on the previous page, and would attempt to retrieve the precedence belonging to the two operators involved. We retrieve the precedences by using the full operator name of each operator to query the current operator environment, and then fetching

⁹The full operator name is a single string that uses the ' \cap ' symbol to indicate where the operator arguments are placed. For instance, the full operator name of the relational image operator is ' $\cap (\cap)$ '.

the precedence from the returned operator (the full operator name of the current operator is determined from the token stream, and the full operator name of the child operator is determined from its infix application token). In our example, the \cap operator has a precedence of forty, and the \cup operator thirty¹⁰, and we note that, as the child operator's precedence is lower than the precedence of the current operator, a transformation must be performed. In cases where the precedences are equal (or we have the same operator), we fetch the associativities of the operators involved to determine whether transformations are necessary.

Once we have determined that a transformation is necessary we apply the applicable rule from the tree transformation rules shown in figure 4.10 on the following page to create the correct tree. (Note that, the transformation rules are bidirectional as we make no assumptions regarding the initial parsing of the sentence.)

In essence, we can reduce this process to a simple statement: if we encounter one of the eight trees shown in figure 4.10 on the next page, and the precedence of the nested operator is greater than that of the outer operator (or equal, and associativity rules require a transformation), then we transform the tree into its complement.

¹⁰These are the standard precedence values specified in the mathematical toolkit.

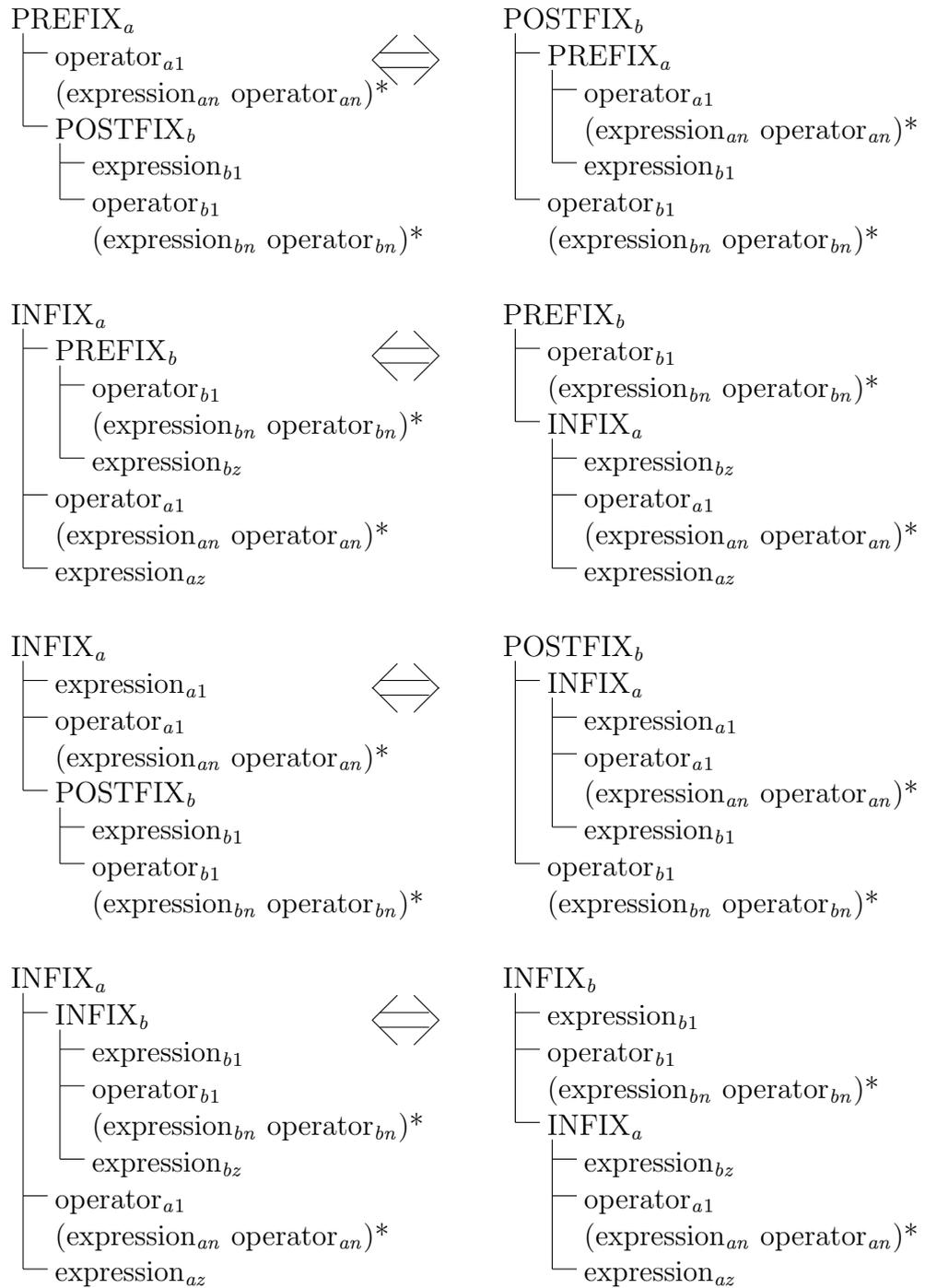


Figure 4.10: Precedence and associativity transformation rules

In our example, we require the fourth transformation rule, and upon applying this rule we are left with the tree shown in figure 4.11, which is the correct parsing of our original Z sentence.

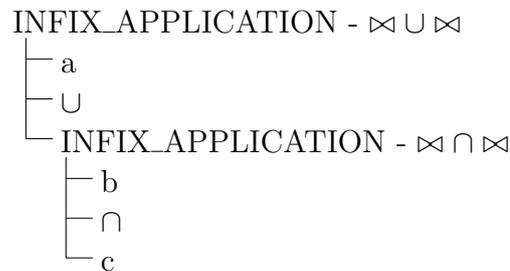


Figure 4.11: Resolved parse tree for $a \cup b \cap c$

When work began on our Z parser, the ISO standard was still in draft format [ISO99]. We completed this work on the resolution of precedence and associativity in user-defined operators before the publication of [ISO02]. Note 3 in section 8.3 of [ISO02] presents a method (first shown in [LdR81]) that can be used to resolve precedence and associativity in user-defined operators. The only difference between the method described there, and our method, is that they choose to assume that all operators are either left or right associated, and then make alterations. We have made no assumptions on the default associativity of operators, and therefore we are required to handle an additional case. We have continued to use our model as we feel that there is no significant drawback in handling all such cases, and it allows our rules to be applied correctly regardless of assumptions made elsewhere.

Example 4.2

We now return to the model we began in example 4.1, and show how the token stream produced in the lexing process is converted to an abstract syntax tree by the parsing process. The token stream produced from lexing is shown below.

```
SECTION KW_SECTION NAME PARENTS NAME END
```

```
ZED LEFTSQ NAME RIGHTSQ END
```

```
SCH LEFTCU NAME RIGHTCU NAME HALFPIPE NAME EQUALS NAME END
```

```
SCH LEFTCU NAME RIGHTCU NAME NLCHAR NAME COLON NAME HALFPIPE NAME
IP NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
END
```

```
SCH LEFTCU NAME RIGHTCU NAME NLCHAR NAME COLON NAME HALFPIPE NAME
IN NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
END
```

We consider first the processing of this stream by the rules that cover the deconstruction of a specification into files and sections, that is rules F.1, F.2 and F.3. The ‘greedy’ option on the paragraph matching within the section rule indicates our preference that paragraphs are associated with sections where possible, and not simply associated with a file creating anonymous sections. The application of these rules to our token stream produces the AST shown in figure 4.12.

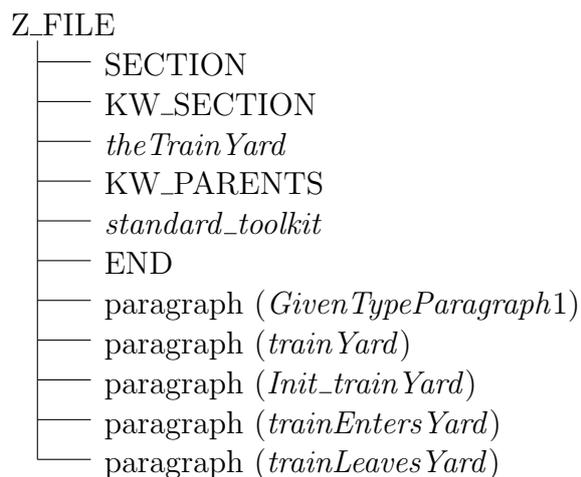


Figure 4.12: Overview of AST for train yard specification

Each paragraph is then processed producing a sub-tree for each. Obviously to examine the creation of each of these sub-trees would be a tedious and redundant task so we examine only the creation of the tree for the *trainEntersYard* paragraph, the token stream for which is shown below.

```
SCH LEFTCU NAME RIGHTCU NAME NLCHAR NAME COLON NAME HALFPIPE NAME
IP NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
END
```

Rule F.4 is applied to the token stream, and we can quickly determine our token stream matches the definition of a schema definition paragraph. Note that the only distinction between a schema definition paragraph and a generic schema definition paragraph is the presence of formals¹¹. A schema definition paragraph is comprised of a name and a schema text; the name has already been matched so we next examine our rules for schema texts (rule F.37). Our token stream at this point is shown below (we've formatted this to enhance clarity).

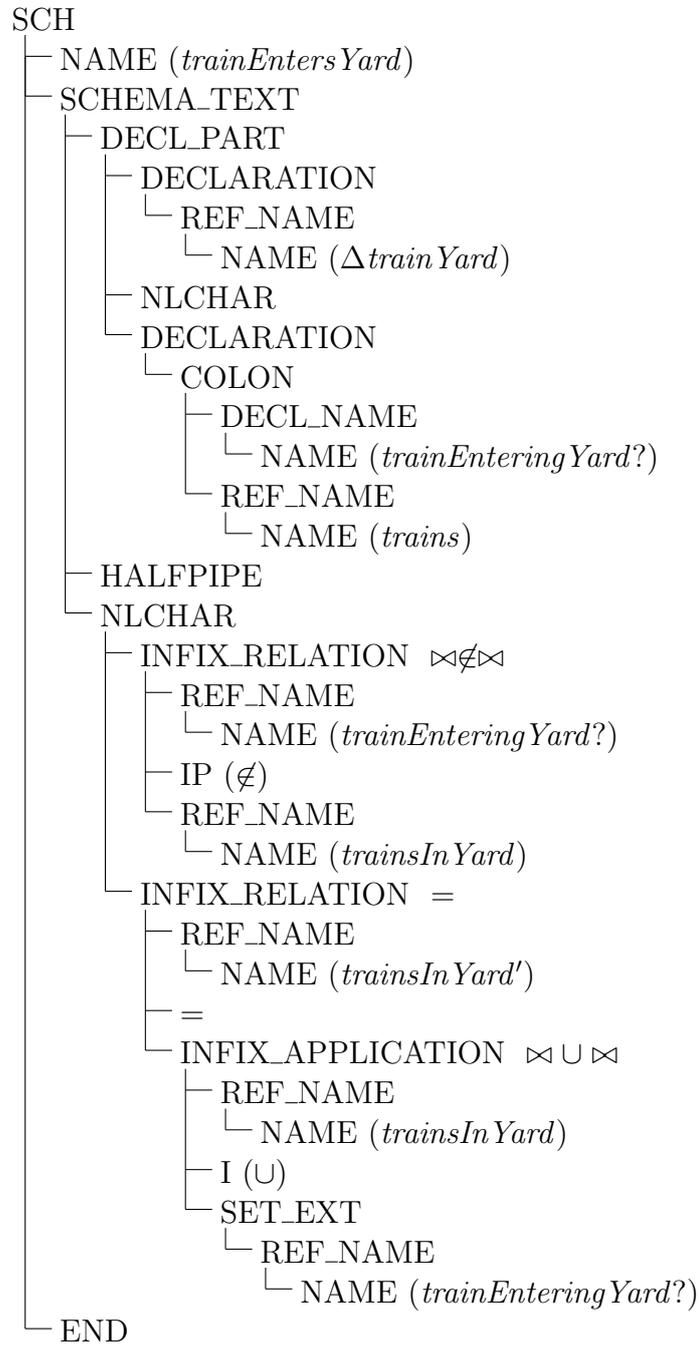
```
NAME NLCHAR NAME COLON NAME
HALFPIPE
NAME IP NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
```

From the rule we can see that a schema text comprises either one of a declaration or a predicate, or both, or possibly neither. In this instance we have both. The declaration part is first split into its component declarations through the use of rule F.38. Each declaration (rule F.39) can then be considered in one of a number of different forms allowing constant declaration, variable declaration or reference declaration. The tokens forming the declaration part of our schema text is shown below.

```
NAME NLCHAR NAME COLON NAME
```

Here we have two declarations split by the NLCHAR token. The first of our declarations is simply an expression – and therefore a reference declaration.

¹¹The generic instantiation parameters for the paragraph.

Figure 4.13: AST for *trainEntersYard*

This expression matches the first sub-rule of rule F.36 and is a simple reference to a previously declared variable. The second declaration is a variable declaration, so we match the first sub rule of rule F.39 (the syntactic predicates on this rule are required to disambiguate the specific syntax of the constant and variable declaration from the reference declaration which can match any expression). A single declaration statement can involve the declaration of many variables, but in this instance just one. The name of our variable is matched with rule F.49 as it is a newly declared name, and our expression – which is also simply a name – by the first sub-rule of rule F.36, as it is a reference to a previously declared name.

We now examine the predicate part of our schema text the token stream for which is shown below.

```
NAME IP NAME NLCHAR NAME EQUALS NAME I LEFTCURLY NAME RIGHTCURLY
```

We first match the predicate rule (F.8) which splits our predicate by NLCHAR and SEMICOLON tokens. This split gives us two further predicates to examine. The first is matched as an infix relation by rule F.64, comprising of two expressions (both of which are references matched by rule F.36) intersected by the IP token.

The second of the two predicates is also matched as an infix relation¹², in this instance, however, while the first of the expressions is again a simple reference, the second expression is matched as an infix application (rule F.71). This infix application comprises two expressions intersected by an I token. The first of these expression is again a simple reference, the second matches the fourth sub-rule of rule F.36 which describes a set extension containing another expression, which again is a simple reference.

The final AST created by this process is shown in figure 4.13 on the preceding page. Note that, tokens such as DECL_PART and REF_NAME are introduced to make the next phase of the parsing process easier, as they eliminate any ambiguity that we have resolved in this phase.

¹²Note that, the equality and set membership operators are special, hard-coded alternatives to the IP token.

4.6.5 Syntax Transformation

The syntax transformation process involves the manipulation of the abstract syntax tree (AST) produced in the parsing process. This procedure entails the elimination of all semantically unnecessary Z syntax, reducing our specification to use the language specified by the annotated grammar presented in chapter 10 of [ISO02]. The representation of our specification will become less readable through this transformation, but the use of a more concise grammar reduces the rules required in the future manipulation of that specification (for instance, when type checking or generating proof obligations). For example, the syntax transformation process eliminates the use of the term ‘false’ and the use of the logical operator, \vee , replacing instances with their logical equivalents (that is, $\text{false} \Leftrightarrow \neg \text{true}$, and $a \vee b \Leftrightarrow \neg(\neg a \wedge \neg b)$).

Chapter 12 of [ISO02] provided us with an initial set of syntactic transformation rules, on which we based this syntax transformation process. The rules presented in [ISO02] are intended to be applied exhaustively until an annotated parse tree results. In order to improve parsing time, we seek to perform all syntactic transformations in a single pass. The rules we use, may therefore, appear to be more complex, but ensure that we need look at a particular sub-tree only once. ANTLR is able to produce parsers that operate not just on a token stream, but also on existing ASTs; our syntactic transformation rules, therefore, take advantage of this fact and we use pattern matching [HO82] to determine when a particular rule is required.

We define a single operation $\mathcal{T}()$ which will apply the correct transformation to the AST passed as its argument, returning the manipulated tree¹³. For instance, the two rules shown in figure 4.14 on the facing page govern the manipulation of ASTs representing equivalence and implication predicates. Our convention for showing these tree transformation rules is to show the AST forming the argument to $\mathcal{T}()$ on the left, and the manipulated AST – that is returned by $\mathcal{T}()$ – on the right.

We have two syntax transformation rules, and we distinguish between

¹³In fact a copy of the transformed AST is returned so that we continue to have a copy of the tree before and after transformation.

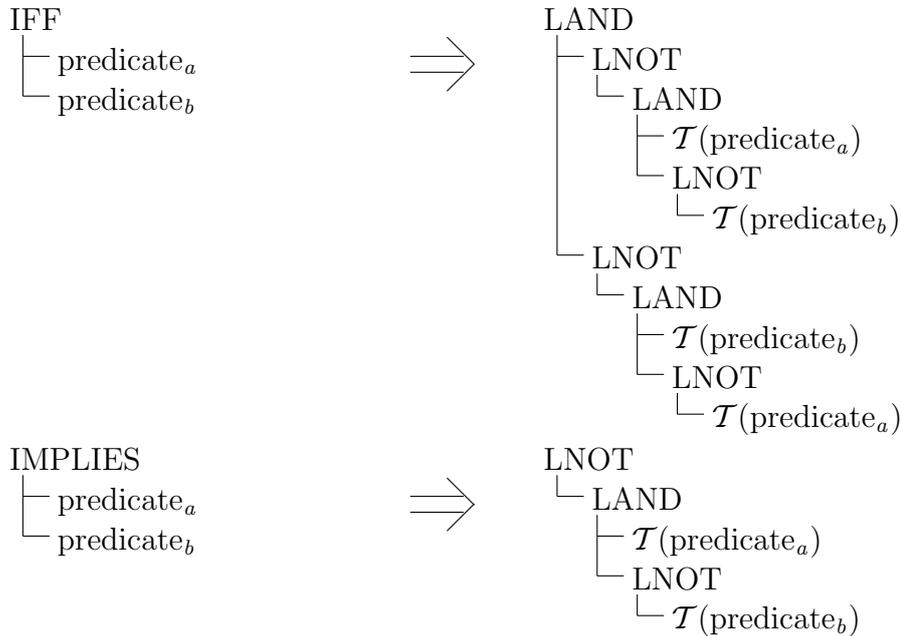


Figure 4.14: Syntactic transformation rules for equivalence and implication predicates

them – and other transformation rules – through the shape and content of the trees (in this case, the token at the head of each tree). ANTLR’s tree walking facilities allow us to specify a tree to match in exactly this way, and also to associate an action that will be performed when that tree is matched.

These example rules enable us to show the difference between our transformation system, and that presented in [ISO02], where the equivalent rules are as follows.

Rule 12.2.4.4 - Equivalence Predicate

$$p_1 \Leftrightarrow p_2 \implies (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$$

Rule 12.2.4.5 - Implication Predicate:

$$p_1 \Rightarrow p_2 \implies \neg p_1 \vee p_2$$

The difference between the rules is immediately apparent. Firstly, our rules refer to the shapes of the trees directly, and are implemented exactly as we specify them. Secondly, and more importantly, the definition of the transformation in our rules is not – at first glance – equivalent to those specified above. When we look closer, however, we can see that this difference is simply the result of our choice to reduce the syntax transformation phase to a single pass of the specification. If we examine the rule for disjunction predicates, we can see that this can be substituted into the rule for implication predicates, which can in turn be substituted into the rule for equivalence predicates.

Rule 12.2.4.5 - Disjunction Predicate:

$$p_1 \vee p_2 \implies \neg (\neg p_1 \wedge \neg p_2)$$

Rule 4.1 - Substitution of disjunction predicate rule into implication predicate rule:

$$p_1 \Rightarrow p_2 \implies \neg (p_1 \wedge \neg p_2)$$

Rule 4.2 - Substitution of implication predicate rule into equivalence predicate rule:

$$p_1 \Leftrightarrow p_2 \implies (\neg (p_1 \wedge \neg p_2)) \wedge (\neg (p_2 \wedge \neg p_1))$$

Once these substitutions have been performed it is obvious that our rules perform the same transformations as those specified in [ISO02]. Appendix G presents a comprehensive description of the syntactic transformation rules

used in place of those described in [ISO02] (note that, there are more rules in our system as we use pattern matching to handle some of the side-conditions, and also it is necessary for us have rules for those situations where no transformation is needed). The remainder of this section details some of the more interesting rules, where we have handled issues raised in [ISO02], and how we have dealt with other problems that have arisen.

Schema and Horizontal Definition Paragraphs

There appears to be some confusion in both [ISO02] and [ISO99] over the syntactic transformation rules for schema and horizontal definition paragraphs. The syntactic transformation rules given for these paragraphs in chapter 12 of [ISO02] are shown below.

Rule 12.2.3.1 - Schema Definition Paragraph:

$$\text{SCH } i \ t \ \text{END} \implies \text{AX } [i == t] \ \text{END}$$

Rule 12.2.3.2 - Generic Schema Definition Paragraph:

$$\text{GENSCH } i \ [i_1, \dots, i_n] \ t \ \text{END} \implies \text{GENAX}[i_1, \dots, i_n] \ [i == t] \ \text{END}$$

Rule 12.2.3.3 - Horizontal Definition Paragraph:

$$\text{ZED } i \ == \ t \ \text{END} \implies \text{AX } [i == t] \ \text{END}$$

Rule 12.2.3.4 - Generic Horizontal Definition Paragraph:

$$\text{ZED } i \ [i_1, \dots, i_n] \ == \ t \ \text{END} \implies \text{GENAX}[i_1, \dots, i_n] \ [i == t] \ \text{END}$$

When we examine annex D of the same document, however, there appear to be some discrepancies. In section D.3.4 the authors are describing the syntactic transformation of the famous birthday book example [Spi92b]. The

step we are concerned with here is the application of rule 12.2.3.1, we show the paragraph before and after transformation below.

```

SCH NAME(BirthdayBook)
[[NAME(known) : P NAME(NAME) ∘ α1] ∧
[NAME(birthday) : NAME(⊗↗↘) [NAME(NAME), NAME(DATE)] ∘ α2]
|
NAME(known) ∈ {NAME(dom) NAME(birthday)}]
END

```

```

AX
[NAME(BirthdayBook) ==
[[NAME(known) : P NAME(NAME) ∘ α1] ∧
[NAME(birthday) : NAME(⊗↗↘) [NAME(NAME), NAME(DATE)] ∘ α2]
|
NAME(known) ∈ {NAME(dom) NAME(birthday)}]]
END

```

This transformation is consistent with the definition provided above. However, this transformation is considered to be the end of transformation on the paragraph, yet it contains an expression that does not appear in the annotated syntax described in chapter 10. This expression would need a rule of the following form.

```

Expression = ...
| expression == expression
| ...
;

```

In fact the ‘==’ token can only be used in a binding extension, and rule 12.2.7.1 states that an expression of this form should have been eliminated.¹⁴

¹⁴We note also that the definition of this rule changed between [ISO99], and [ISO02], but this alteration does not appear to solve the issue, and is in our opinion misplaced.

Rule 12.2.7.1 - Declaration:

$$i == e \implies i : \{e\}$$

$$i_1, \dots, i_n : e \implies [i_1 : e \circ \alpha_1] \wedge \dots \wedge [i_n : e \circ \alpha_1]$$

Applying this rule to the paragraph, however, would give us the following paragraph, which is clearly not what was intended (note particularly, the presence of two schema construction expressions).

```

AX
[[NAME(BirthdayBook) : {
  [[NAME(known) : P NAME(NAME) o alpha_1] ^
  [NAME(birthday) : NAME(xleftrightarrow x) [NAME(NAME), NAME(
DATE))] o alpha_2]
  |
  NAME(known) ^ {NAME(dom) NAME(birthday)}]]]]
END

```

If we then also examine section D.6.2, we can see that the authors are examining the syntactic transformation of the following Z paragraph.

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

Again, the important step is the application of the paragraph transformation rule, in this case 12.2.3.4. The paragraphs before and after the transformation – shown in [ISO02] – are replicated below.

```

ZED
NAME(xleftrightarrow x)[NAME(X), NAME(Y)] == P (NAME(X) ^ NAME(Y))
END

```

```

GENAX [NAME(X), NAME(Y)]
NAME(xleftrightarrow x) == P (NAME(X) ^ NAME(Y))
END

```

This is clearly an incorrect application of 12.2.3.4 as the schema construction brackets around the expression have been omitted. The situation is further

complicated as they are also omitted when applying the declaration rule, 12.2.7.2, (after having also applied the cartesian product rule, 12.2.6.8) giving the following paragraph. (Somewhat confusingly the authors also talk about applying the ‘first Declaration rule in 12.2.7.1’ – it seems apparent to us that if the rules are to be applied exhaustively, that whenever the first rule is applied the second must also be applied.)

```

GENAX [NAME(X), NAME(Y)]
NAME(⊠↔⊠) : { P {NAME(x) : NAME(X);
                  NAME(y) : NAME(Y)
                  • (NAME(x), NAME(y)) } }
END

```

The schema construction brackets around the expression are at this point still missing. However, the application of rule 12.2.7.2 (below) to the paragraph results in the unexplained introduction of these schema construction brackets.

Rule 12.2.7.2 - DeclPart:

$$de_1; \dots; de_n \implies de_1 \wedge \dots \wedge de_n$$

```

GENAX [NAME(X), NAME(Y)]
[NAME(⊠↔⊠) : { P {[NAME(x) : NAME(X) ∘ α1] ∧
                  [NAME(y) : NAME(Y) ∘ α2]
                  • (NAME(x), NAME(y)) } } ∘ α3]
END

```

Given the contradictory nature of the examples we decided that we would choose to stray from the definition provided in [ISO02], and use our own definitions of the rules for the syntactical transformation of schema and horizontal definition paragraphs (we would also choose to retain the definition of the rule for declaration presented in [ISO99]). We feel that the rules presented in figures 4.15 and 4.16 are true to the spirit of the standard, and produce the correct overall results for the examples presented in annex D of that document.

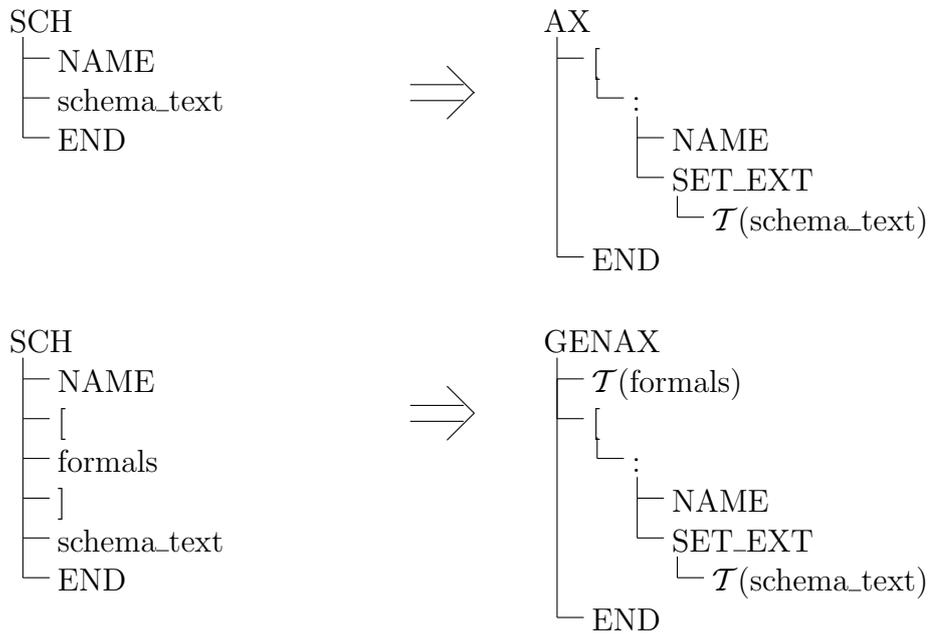


Figure 4.15: Syntactic transformation rules for schema definition paragraphs

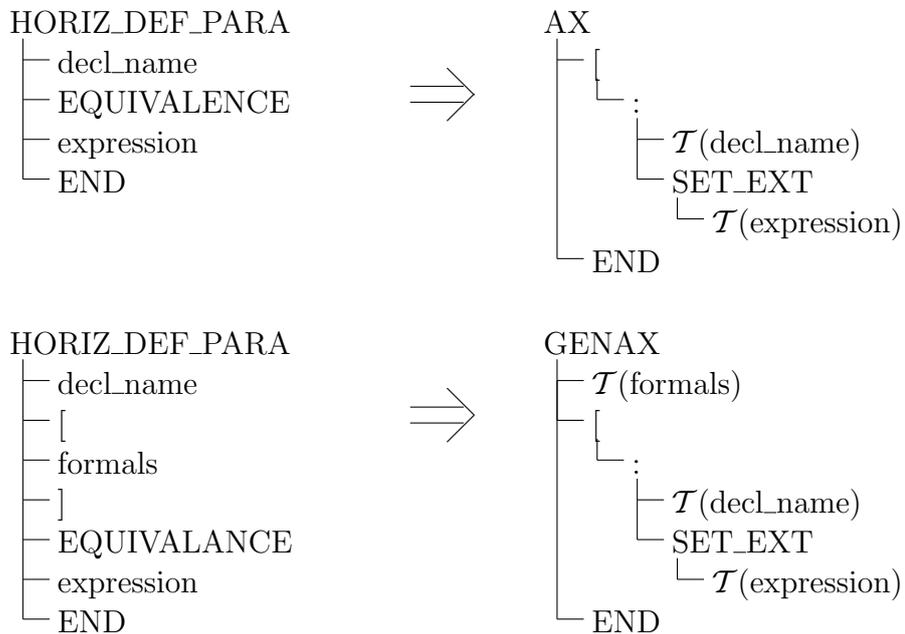


Figure 4.16: Syntactic transformation rules for horizontal definition paragraphs

Generic Operator Definition Paragraphs

The grammar presented in chapter 8 of [ISO02] makes a valid distinction between generic operator definition paragraphs, and generic horizontal definition paragraphs. The syntactic transformation rules presented in chapter 12, however, give no information regarding the handling of generic operator definition paragraphs. We can deduce that there should be a rule for handling generic operator definition paragraphs, however, as these are not included in the annotated grammar presented in chapter 10.

When we again examine section D.6.2, where the authors are demonstrating the syntactic transformation of the following Z paragraph.

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

We can see that the example requires the use of the infix generic name rule (12.2.9.3 below) *before* the use of the generic horizontal definition paragraph rule (12.2.3.4 above).

Rule 12.2.9.3 - InfixGenName:

$$i_1 \text{ in } i_2 \implies \bowtie \text{ in } \bowtie [i_1, i_2]$$

This results in the following progression.

```
ZED
NAME(X) I(↔) NAME(Y) == P (NAME(X) × NAME(Y))
END
```

```
ZED
NAME(↔↔)[NAME(X), NAME(Y)] == P (NAME(X) × NAME(Y))
END
```

```
GENAX [NAME(X), NAME(Y)]
NAME(↔↔) == P (NAME(X) × NAME(Y))
END
```

The rule for the generic operator can be concluded to be the application of a generic name rule, followed by an application of the generic horizontal definition rule. Whilst this method of application is fine for intuitive human manipulation, it does not work particular well when using a mechanized, top-down, tree approach; our system for performing transformations will obtain the tree for the generic operator definition paragraph, but be unable to match it to the paragraph patterns it has available. It is necessary, therefore, for us to introduce a new syntactic transformation rule that will match a generic operator definition paragraph. The rule we have used is shown in figure 4.17.

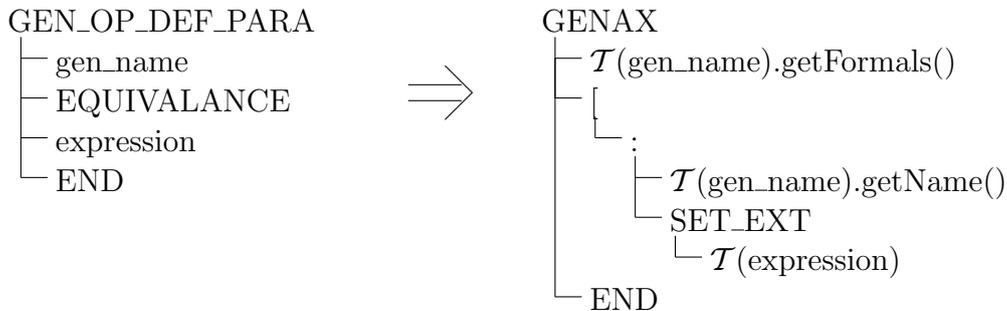


Figure 4.17: Syntactic transformation rule for generic operator definition paragraph

This rule, however, is different to rules we have specified previously, as its structure explicitly depends upon the transformation of its child trees; specifically the manipulation of the generic name. We have shown this in figure 4.17 as operations that can be performed on the transformed tree obtained from the application of $\mathcal{T}()$ to the generic name. This, however, is a convenience; in fact we use ANTLR's ability to return objects from a rule, to return a generic name object that provides two methods: one that returns the tree associated with the generic operator name ('getName()'), and a second that returns the tree associated with the generic operator parameters ('getFormals()').

Characterization

Chapter 9 of [ISO02] provides rules for making explicit characteristic tuples – specified with the concrete syntax – explicit. We felt that the combination of the characterization phase with the syntax transformation phase provided advantages in terms of performance (as only a single syntactic transformation phase was required) without introducing any semantic ambiguity. We therefore perform all manipulations required to make characterizations explicit alongside all other syntactic transformations, and these manipulations are incorporated directly within the syntactic transformation rules.

The first characterization rule defined in [ISO02] is for the lambda function construction expression and is shown below.

Rule 9.3.1 - Lambda Function Construction:

$$\lambda t \bullet e \implies \{t \bullet (\text{chartuple } t, e)\}$$

Our syntactic transformation rule for the lambda function construction expression is shown in figure 4.18.

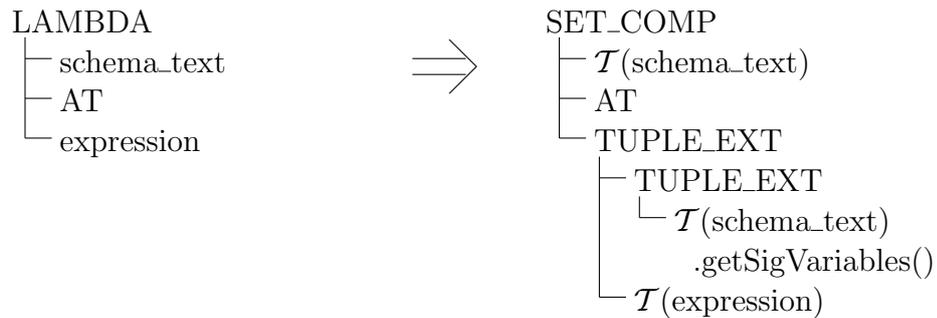


Figure 4.18: Syntactic transformation rule for lambda function construction

Similarly, the characterization rule for characteristic set comprehension expressions can be seen below, and our syntactic transformation rule in figure 4.19 on the next page

Rule 9.3.2 - Characteristic Set Comprehension:

$$\{t\} \implies \{t \bullet \text{chartuple } t\}$$

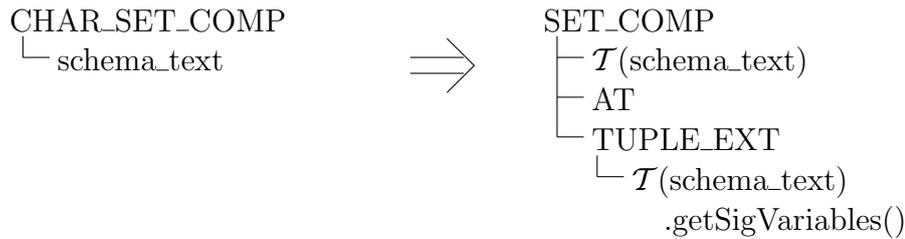


Figure 4.19: Syntactic transformation rule for characteristic set comprehension

Finally, the characterization rule for characteristic definite description expressions can be seen below, and our syntactic transformation rule in 4.20

Rule 9.3.3 - Characteristic Definite Description:

$$(\mu t) \implies \mu t \bullet \text{chartuple } t$$

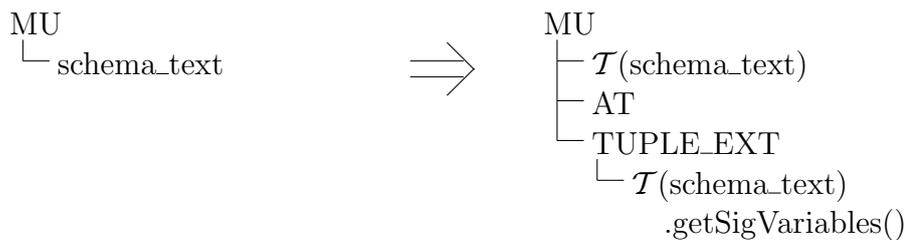


Figure 4.20: Syntactic transformation rule for characteristic definite description

Other than the difference in style, the significant distinction between the two sets of rules is the way in which the characterization tuple is determined. Section 9.2 of [ISO02] defines a function *chartuple* with the following definition.

$$\begin{aligned}
\text{chartuple } t &\implies \text{mktuple}(\text{charac } t) \\
\text{charac } (d_1; \dots; d_n \mid p) &\implies \text{charac } (d_1; \dots; d_n) \\
\text{charac } (d_1; \dots; d_n) &\implies \text{charac } d_1 \wedge \dots \wedge \text{charac } d_n \quad \text{where } n \geq 1 \\
\text{charac } () &\implies \langle \langle \rangle \rangle \\
\text{charac } (i_1, \dots, i_n : e) &\implies \langle i_1, \dots, i_n \rangle \\
\text{charac } (i == e) &\implies \langle i \rangle \\
\text{charac } e &\implies \langle \theta e \rangle \\
\\
\text{mktuple } \langle e \rangle &\implies e \\
\text{mktuple } \langle e_1, \dots, e_n \rangle &\implies (e_1, \dots, e_n) \quad \text{where } n \geq 2
\end{aligned}$$

As we are performing the characterization process in parallel with the syntax transformation phase we can – instead – take advantage of the transformation that takes place in the schema text present in each of these expressions. The application of rules G.68, G.69, G.70 and G.71 to the declaration part of a schema text always results in a number of conjoined variable declaration expressions and schema references. For example, the following declaration will result in the syntactically transformed tree shown in figure 4.21. (We have left **NAME** and **NUMBER** tokens as their parsed value to improve readability.)

$$a == 0; b : \mathbb{N}; \text{mySchema}$$

We can see from the tree that all declarations are transformed into a variable construction expression or remain as a schema reference. With this tree it is relatively simple to write a function (referred to in the rules above as ‘getSigVariables()’) that walks the tree, and creates the characteristic tuple. Obviously, an empty tree gives the empty binding ($\langle \rangle$), and otherwise a tuple is formed from the nodes of the tree; variable construction expression nodes yielding the associated variable, and schema nodes yielding a binding

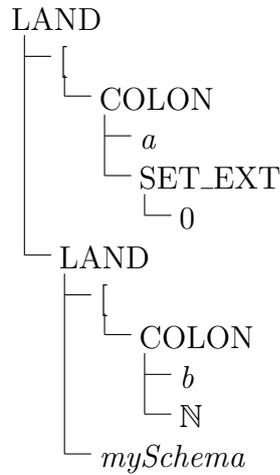


Figure 4.21: Applying the transformation rules to the declaration part of a schema text

construction expression formed from that schema¹⁵. For example, parsing the tree shown in figure 4.21 gives the following tuple.

$$(a, b, \theta mySchema)$$

Cartesian Product Expression

Cartesian product expressions can be transformed in much the same way as the rule specified in [ISO02], the details of which are given below.

Rule 12.2.6.8 - Cartesian Product Expression:

$$e_1 \times \dots \times e_n \implies \{i_1 : e_1; \dots; i_n : e_n \bullet (i_1, \dots, i_n)\}$$

The syntactic transformation rule specified in figure 4.22 on the next page describes a version of this rule, that automatically transforms the included schema text.

Our concern with this syntactic transformation rule is the requirement

¹⁵As with the *chartuple* function, it is not necessary to check a schema reference at this point, as its properties will be examined properly in the type checking phase.

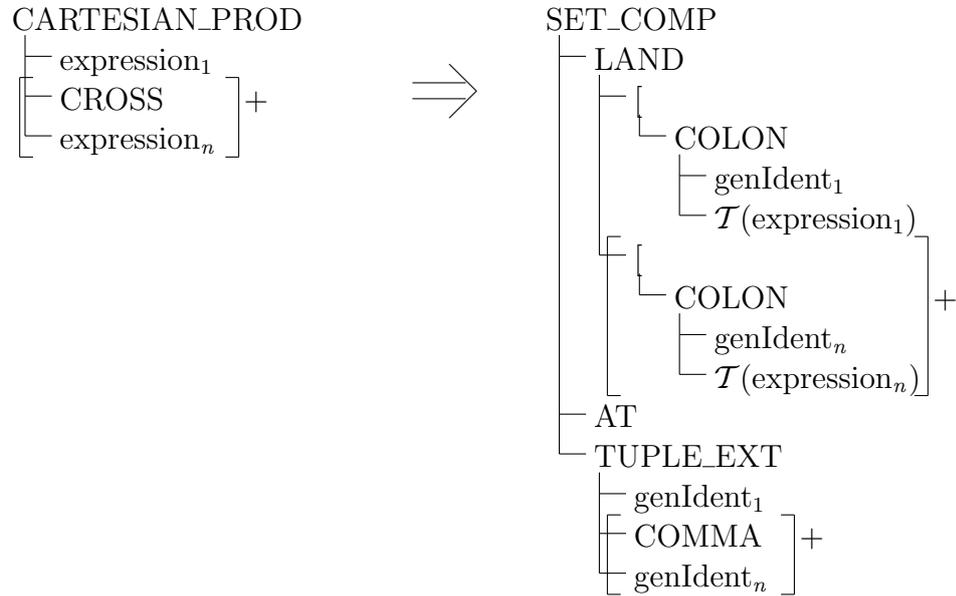


Figure 4.22: Syntactic transformation rule for cartesian products

to use generated identifiers. Clearly, any identifier that could be lexed as a `NAME` token is unsuitable, as it has the potential to clash with other, existing identifiers. On the other hand, if we use an identifier that cannot be lexed as a `NAME` token then the language of the transformed specification does not, strictly, conform to the annotated grammar. A further disadvantage of using an identifier that cannot be lexed is that it will not be possible to use a transformed specification as an input to this parser, or indeed any other parser that obeys the ISO standard.

Despite this, however, we have chosen to use an identifier that cannot be lexed as a `NAME` token, using identifiers of the form ‘*__generatedIdentifier1*’. We have proceeded this way, as time restrictions have prevented us from seeking a more suitable solution. The identifiers are only used internally, however, and the only external impact is the inability to use the specification produced by the syntax transformation process as an input to other tools; this compromise does not affect the semantics nor correctness of a specification.

Example 4.3

We now return, once again, to the example we last visited in example 4.2, and show how the abstract syntax tree (AST) produced in the parsing process is manipulated by the syntactic transformation rules. At the end of that example, we began to concentrate specifically on the AST for the *trainEntersYard* paragraph; in this example, we will again consider only the AST belonging to that paragraph. This AST is shown in figure 4.13 on page 168.

As we examine the AST for our paragraph, we note that it matches the pattern for rule G.1. We apply this rule, therefore, and create the appropriate AST, which is shown in figure 4.23. We will, in this example, build the ASTs in the same way as our tree parser, using tree creation rather than tree manipulation.

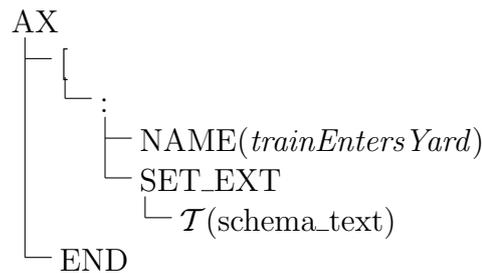
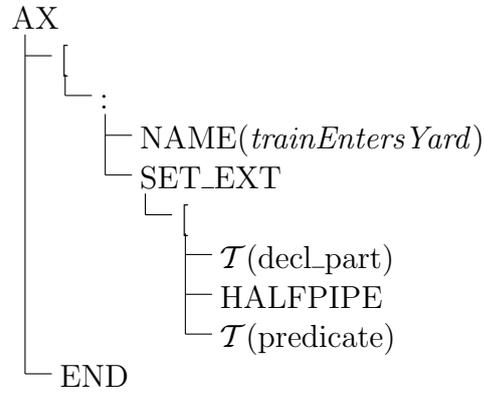
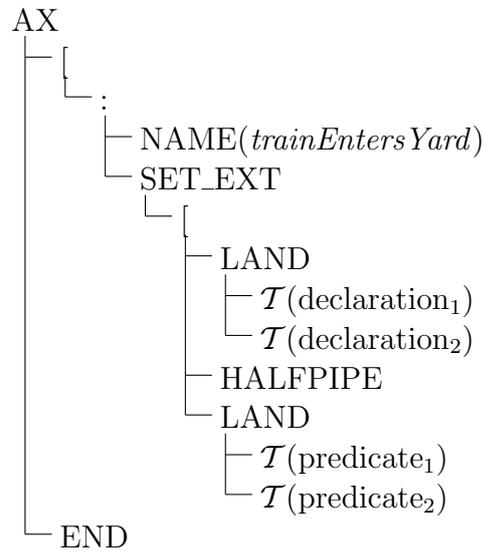


Figure 4.23: Creating the transformed AST for *trainEntersYard* (1)

A side condition of this rule is that we apply our syntactic transformation rules to the part of our tree that matched as a schema text. We, therefore, re-examine our rules, and note that this sub-tree now matches the pattern for rule G.67. We apply this rule, and create the appropriate sub-tree, before attaching this to our current tree resulting in the tree shown in figure 4.24 on the following page.

Having matched this rule we proceed to apply the syntactic transformation rules to the declaration part and predicate from the schema text. Upon examination of the rule we can see that we need to apply rules G.68, and G.10, the application of which results in the AST shown in figure 4.25 on the next page.

Figure 4.24: Creating the transformed AST for *trainEntersYard* (2)Figure 4.25: Creating the transformed AST for *trainEntersYard* (3)

We are now left with four sub-trees requiring transformation, and we select the appropriate rule for each. declaration_1 requires rule G.70, declaration_2 requires rule G.71, predicate_1 requires rule G.28, and predicate_2 requires rule G.29. We append all of the sub-trees created from the application of these rules to our AST resulting in the tree shown in figure 4.26.

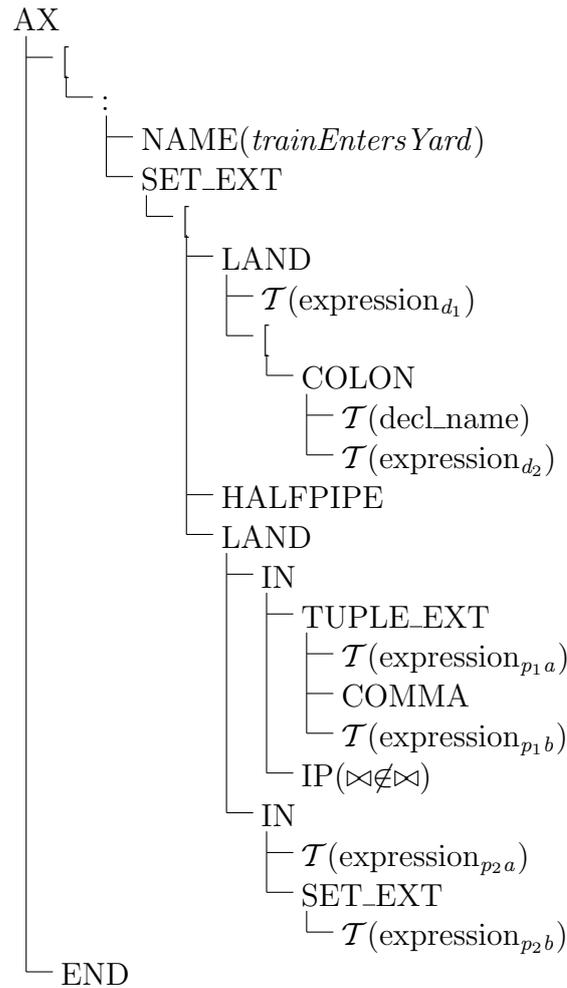


Figure 4.26: Creating the transformed AST for *trainEntersYard* (4)

This time we are left with seven sub-trees that require transformation. Decl_name requires rule G.73. expression_{d_1} , expression_{d_2} , expression_{p_1a} , expression_{p_1b} and expression_{p_2a} all require rule G.46, and expression_{p_2b} requires rule G.61. Once again, we append all of the sub-trees created from

the application of these rules to our AST resulting in the tree shown in figure 4.27.

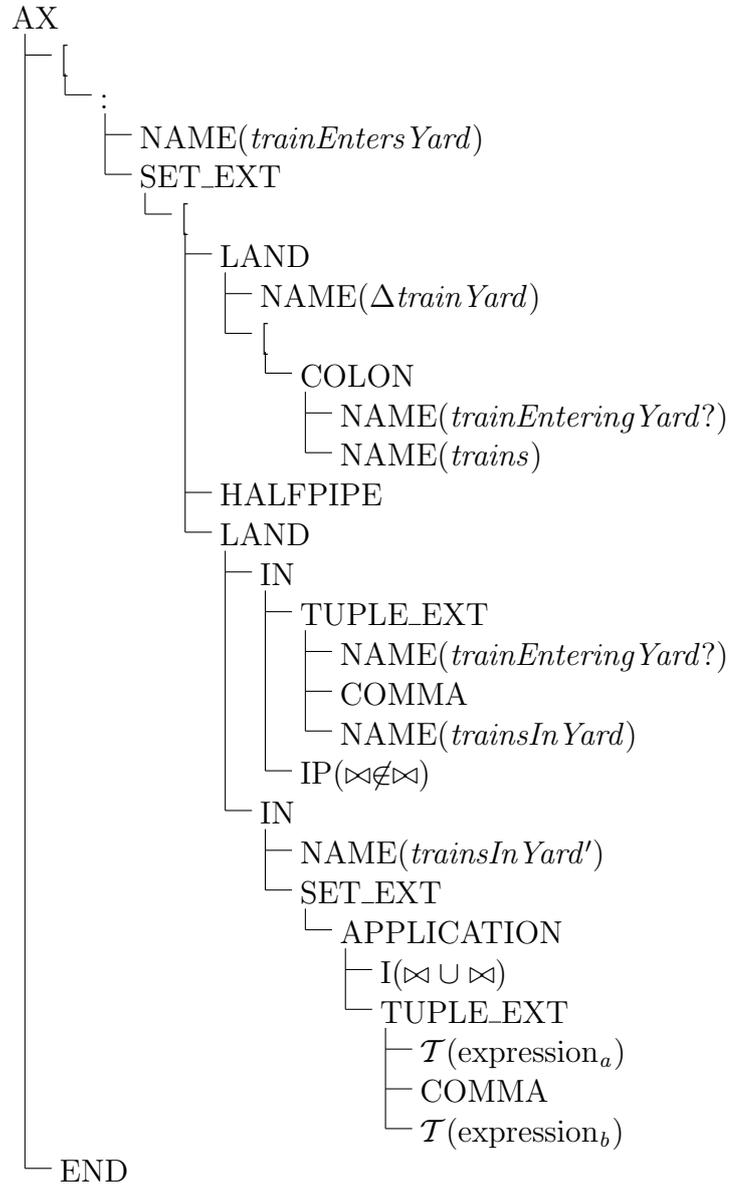


Figure 4.27: Creating the transformed AST for *trainEntersYard* (5)

We now have two untransformed sub-trees remaining; expression_a can be transformed with rule G.46, and expression_b with rule G.54, followed by

rule G.46. These transformations give us the final transformed AST for the *trainEntersYard* paragraph which is shown in figure 4.28.

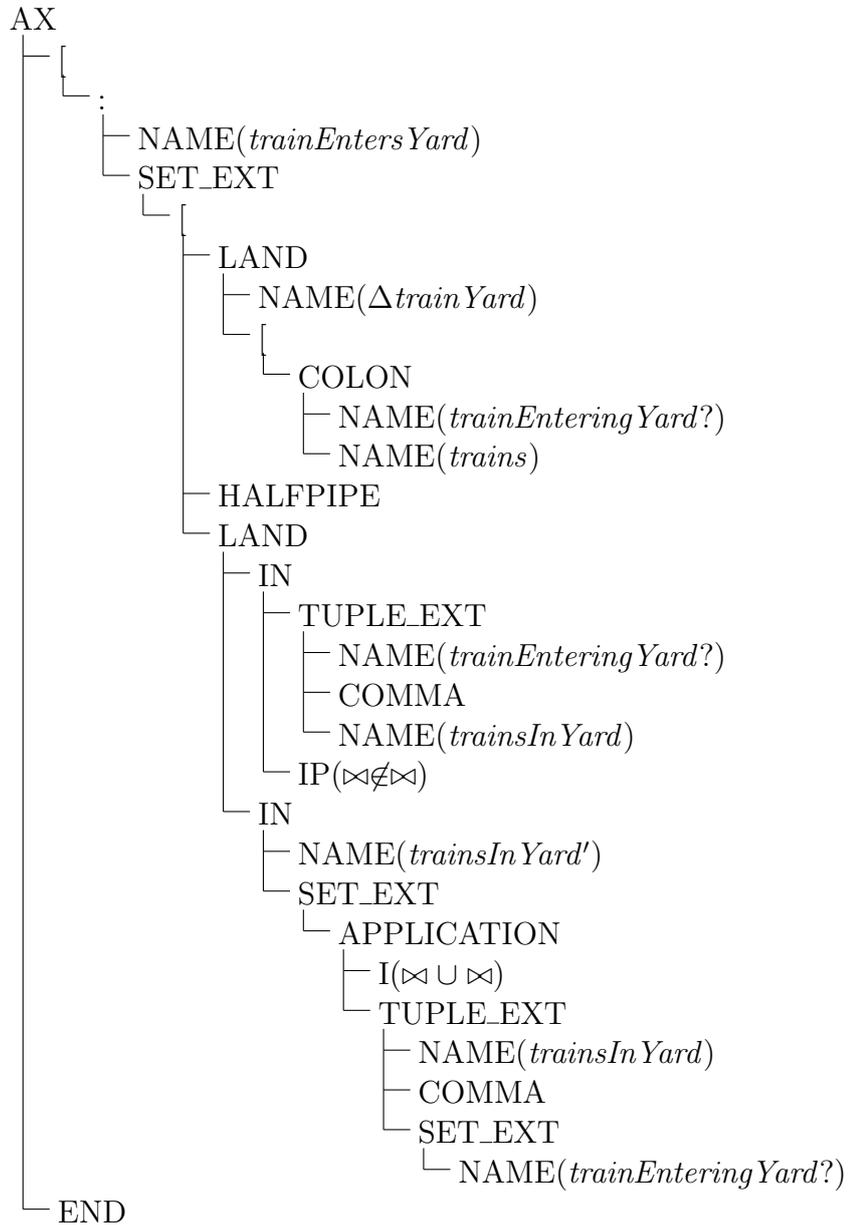


Figure 4.28: Final AST for *trainEntersYard*

4.6.6 Type Checking

The type checking phase involves producing a valid type for every expression and paragraph in a specification. This process involves parsing the abstract syntax tree (AST) produced in the syntax transformation phase, checking that types for expressions and paragraphs can be produced correctly, and producing a final AST that has been annotated with those checked types.

Chapter 13 of [ISO02] again provides us with a starting point for this process. The following section describes how we translate the abstract rules presented there into implementable procedures. In this phase, we follow the rules presented in [ISO02] faithfully. As such, we present only a guide to our type checking framework, and show the methods we have used to translate the rules; we present examples that show the type-checking of a paragraph, an expression and a predicate.

Section 13.2.3 of [ISO02] presents a set of functions that can be used to instantiate generic types. In practice, however, it can be difficult to determine exactly which types are generic, and when it is appropriate to instantiate them (and with what arguments). We provide, therefore, a little more detail about how instantiation – both explicit and implicit – is performed.

Type Objects

Each paragraph and expression in a specification is assigned a type. Whilst it is possible to use textual representations of these types, it is far wiser – in a mechanized environment – to create manipulable objects belonging to a set of classes defining their respective behaviour.

Consider, for example the rule presented in [ISO02] for determining the type of a schema conjunction expression.

Rule 13.2.6.16 - Schema Conjunction Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \Sigma \vdash^{\mathcal{E}} e_2 \circ \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \circ \tau_1) \wedge (e_2 \circ \tau_2) \circ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

Determining the compatibility of two signatures, or indeed calculating their union would be extremely difficult in a text based system. With an object based system the information required of a schema can be incorporated at its construction, and that information retrieved when needed.

Similarly, the scope of generic types can be difficult to control in a text based system. Consider, for example, the following excerpt from the set toolkit (see 4.4.2 and B.5.2)

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f\}$$

We have two implicit generic types in each paragraph – X and Y . When we instantiate the generic types of the relation in our definition of the total function, however, it is difficult – with a text based system – to determine whether the types in our instantiated relation definition are the original X and Y , or the X and Y belonging to our function definition. With an object based system, these difficulties are instantly overcome as each generic type becomes a distinct type, and its nomenclature unimportant.

We assign, therefore, every paragraph and expression belonging to the AST derived from the syntax transformation phase, a $ZType$. The $ZType$ is an abstract class, from which a number of sub-classes are derived giving the genealogy shown in figure 4.29 on the next page. A brief definition, and an example of each of these ‘building-block’ types is given in table 4.4 on page 193.

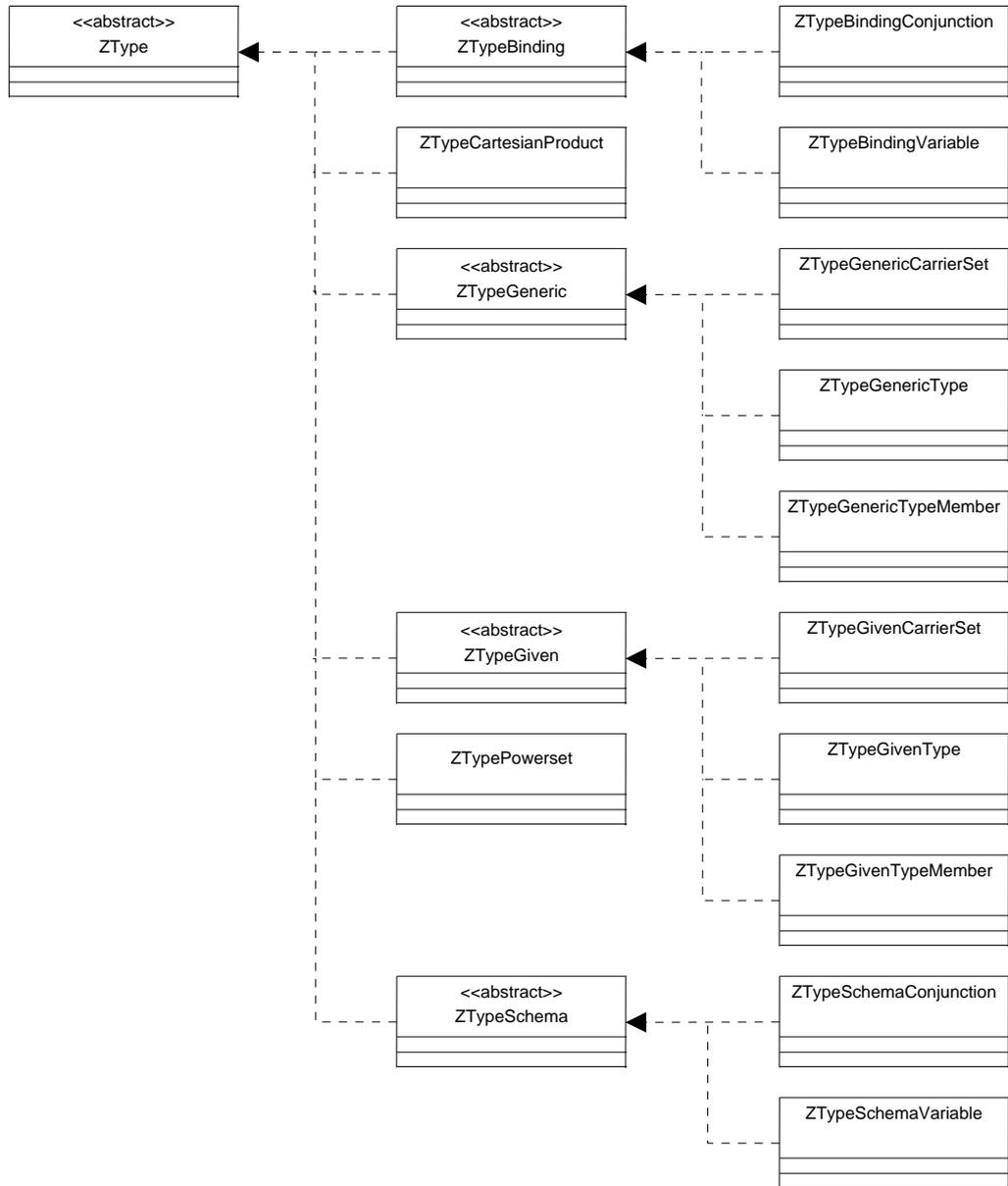


Figure 4.29: Z Type Classes

Table 4.4: Definition of Z Type Classes

Class	Definition	Example
ZTypeBindingConjunction	The combination of two ZTypeBindings	$\langle i : X; j : Y \rangle$
ZTypeBindingVariable	A binding that associates a variable with the ZType of the expression to which it is bound	$\langle i : X \rangle$
ZTypeCartesianProduct	Cartesian product of two ZTypes	$X \times Y$
ZTypeGenericCarrierSet	Carrier set created for a generic type	GENTYPE X
ZTypeGenericType	ZType of a generic type	$\mathbb{P}(\text{GENTYPE } X)$
ZTypeGenericTypeMember	ZType of a variable or expression that is a member of a generic type	GENTYPE X
ZTypeGivenCarrierSet	Carrier set created for a given type	GIVEN X
ZTypeGivenType	ZType of a given type	$\mathbb{P}(\text{GIVEN } X)$
ZTypeGivenTypeMember	ZType of a variable or expression that is a member of a given type	GIVEN X
ZTypePowerset	Powerset of a ZType	$\mathbb{P}(X)$
ZTypeSchemaConjunction	The combination of two ZTypeSchemas	$[i : X; j : Y]$
ZTypeSchemaVariable	A schema that associates a variable with the ZType of the set of which it is a member	$[i : X]$

Type Environments

In order to be able to type the expressions and paragraphs of a specification we need to be able to keep track of variables and the scope within which they can be validly used. We, therefore, create ‘type environments’ that each provide a fixed scope for the declaration of variables. The attributes of the type environment class (`ZTypeEnvironment`) are shown in figure 4.30. We also create classes `ZVariable` and `ZExpression`, which we use to keep track of variables and expressions and their respective types. The `ZPredicate` class is used to keep track of predicates, and contains an attribute that states whether the predicate is internally well-typed.

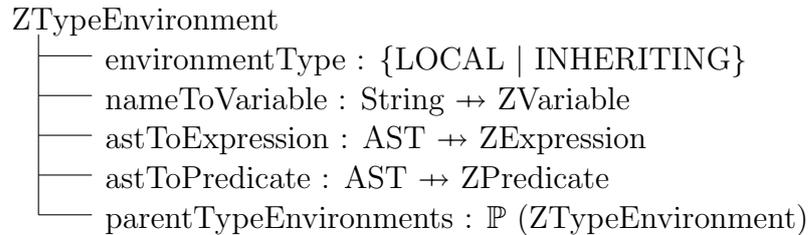


Figure 4.30: Type Environment class

We declare each type environment to be either `LOCAL` or `INHERITING`. These names are not particularly descriptive, but have been chosen due to the lack of anything more appropriate. All type environments can have one or more parent environments from which they inherit variables. Type environments which are declared to be `LOCAL`, however, allow the re-declaration of existing variables. For example, we create a new type environment when parsing a universal quantification predicate, and declare this type environment to be `LOCAL`, which means that we can have bound variables with the same names as those declared in the inherited parent type environments. Alternatively, when we are parsing an axiomatic description paragraph, we create a type environment and declare it to be `INHERITING`; if we try and add a variable to the type environment that has the same name as a variable in a parent section, we generate a typing error. When parsing a section,

paragraph, expression or predicate we choose either to create a new type environment or use the existing type environment; a new type environment will be created wherever new variables can be declared. For example, consider the following specification.

```
section mySection parents standard_toolkit
```

```
| x == {x, y :  $\mathbb{N}$  | x + y = 250}
```

```
| y == x  $\cup$  {x, y :  $\mathbb{N}$  | x = y + 5}
```

For this section we need to create five type environments (which can be seen graphically in figure 4.31); one for the section, one for each paragraph, and one for each set comprehension expression. The section and paragraph type environments are declared as INHERITING, and the type environments belonging to the set comprehension expressions as LOCAL. (Note that x_1 and y_1 are promoted to the section type environment in accordance with the scope rules declared in the fourth paragraph of section 13.3 of [ISO02].)

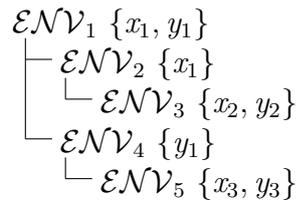


Figure 4.31: Type Environments for *mySection*

The creation of these type environments allows us to easily determine which x and y we are referring to in our specification, as we can query the current type environment at the appropriate point in the parsed AST. Our specification can, therefore, effectively be rewritten as follows.

```
section mySection parents standard_toolkit
```

$$\mid x_1 == \{x_2, y_2 : \mathbb{N} \mid x_2 + y_2 = 250\}$$

$$\mid y_1 == x_1 \cup \{x_3, y_3 : \mathbb{N} \mid x_3 = y_3 + 5\}$$

As well as keeping track of the types of variables, we also use the type environments to relate ASTs to expression and predicate objects. This allows us to easily determine the well-typedness of a sub-tree, and in the case of expressions retrieve an AST's associated type.

The Type Checking Rules

Now that we have a set of types, and an environment in which to perform our type checking, we consider the precise rules required to ensure that a specification is correctly typed, and to generate a type for every expression and paragraph. Again we use ANTLR's ability to generate a tree parser. This parser will transform the tree generated by the syntax transformation phase, producing a tree annotated with the correct types. Unlike the last phase, the majority of the work performed will be in the actions associated with a transaction (rather than in performing the transaction itself). However, it is this process that allows us walk the tree efficiently and effectively, again performing type checking in a single pass of the abstract syntax tree (AST).

We shall demonstrate how the rules presented in chapter 13 of [ISO02] are translated through a series of examples, one for each of the major syntactic blocks: paragraph (example 4.4), expression (example 4.5), and predicate (example 4.6). The behaviour for the remainder of the rules can be deduced from their definition in [ISO02], and the method of translation used for the rules below.

Example 4.4

Consider first, type checking an axiomatic description paragraph. The rule presented for type checking such a paragraph in [ISO02] is shown below.

Rule 13.2.4.2 - Axiomatic Description Paragraph:

$$\frac{\Sigma \vdash^{\mathcal{E}} e \circ \tau}{\Sigma \vdash^{\mathcal{D}} \text{AX } e \circ \tau \text{ END } \circ [\sigma]} \quad (\tau = \mathbb{P}[\sigma])$$

This rule states that the type of the axiomatic paragraph is determined from the type of the expression it contains. Typing this expression is, therefore, the first step in the typing of the paragraph (following the creation of the paragraph's type environment). This means that the typing of our AST is performed firstly on the leaves, and then progresses upwards to the root.

In this instance, the rule states that the type of the expression will always be the powerset of the type of the axiomatic description paragraph.

Once we have completed the typing of the expression, therefore, we use the sub-tree generated to retrieve the 'ZExpression' – and hence the generated type – from the current type environment, and then check that type to determine how to proceed. If no type has been generated for the expression, we know that either that expression contained a typing error, or there were typing errors further down the tree. If the type generated for the expression is an instance of a class other than 'ZTypePowerset', we know that the type generated – and therefore, the expression – is unsuitable for use in an axiomatic description paragraph and create a new typing error. If the type generated for the expression is an instance of the class 'ZTypePowerset', then we can deduce the type of the paragraph by retrieving the type contained within that instance. Our rule also states that the type of the axiomatic description paragraph must be a schema, so we check that the type we have derived is an instance of the 'ZTypeSchema' class, and if not generate a typing error.

Once we have deduced the type, we add that type to the 'ZParagraph' instance associated to this particular axiomatic description paragraph. It is also necessary to check whether we need to promote variables to the section type environment (see the scope rules declared in the fourth paragraph of section 13.3 of [ISO02]). We do this by determining, whether the paragraph's expression has its own type environment, and if so we add the variables

contained within, to the type environment in which the paragraph itself is declared (that is the section's type environment).

Having generated the type, and updated the type environment, the last step is to transform the AST created in the syntax transformation phase, into a typed AST. This is done through the application of transformation¹⁶ rules identical to those of the syntax transformation phase. We define an operation, $\mathcal{Y}()$ which will apply the correct transformation to the AST passed as its argument, and returns the manipulated, typed AST. In practice, it is the call to this operation that triggers all of the above type checking for a particular syntactic section. The transformation rules in this phase are all very similar, and the rule for axiomatic description paragraph is shown in figure 4.32.

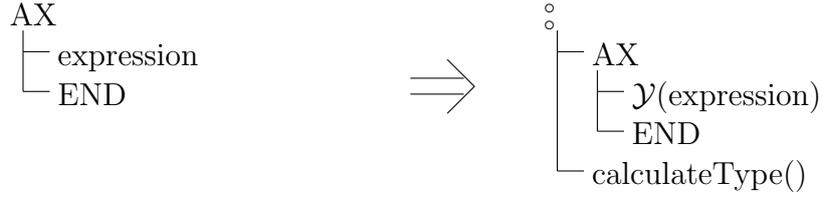


Figure 4.32: Typing transformation rule for axiomatic description paragraph

Example 4.5

We consider next the typing of a schema conjunction expression. The rule presented for type checking such an expression in [ISO02] is shown below.

Rule 13.2.6.16 - Schema Conjunction Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \Sigma \vdash^{\mathcal{E}} e_2 \circ \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \circ \tau_1) \wedge (e_2 \circ \tau_2) \circ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

¹⁶Again, we are actually creating a new AST rather than manipulating the current AST.

Again, the type of the schema conjunction expression is derived from the types of the expressions involved. As there are no variables declared within this type of expression, however, no local type environment is declared, and we continue to use the existing type environment (which could be inherited from a parent paragraph, expression or predicate). The first step, therefore, is to type the two expressions involved in the schema conjunction.

In this instance, the rule states that once we have derived the types of the constituent expressions we need to determine the signature of the two schemas, ensure their compatibility, and perform a union.

The types for the two expressions are retrieved from the current type environment using the appropriate sub-trees. If either of the expressions has an undefined type we know that a typing error has occurred, and we cannot continue typing this expression; thus we generate a typing error. Similarly, if either of the expressions has a type that is not the powerset of a schema type, we know that one of the expressions has an inappropriate type, and again generate a typing error. Performing the union (and checking for compatibility) of the schemas is a relatively simple task, when using the object-based approach. Consider for example the following two schema types.

$$\begin{aligned} & [a : \text{GIVEN } \mathbb{A}; b : \text{GIVEN } \mathbb{A}] \\ & [a : \text{GIVEN } \mathbb{A}; c : \text{GIVEN } \mathbb{A}; d : \text{GIVEN } \mathbb{A}] \end{aligned}$$

These types would each be comprised of the objects shown in figure 4.33 on the following page.

We begin by creating a copy of the first tree which forms the initial value of the conjoined type. We then recursively process the second tree; whenever we reach a node of the ‘ZSchemaVariable’ type, we determine first whether it already belongs in our first tree. If so, then we must determine its compatibility (compatibility means that we skip the node, and incompatibility causes a typing error). If not, then the instance is copied and conjoined to the end of the first tree, through the use of an instance of the ‘ZSchemaType-Conjunction’ class.

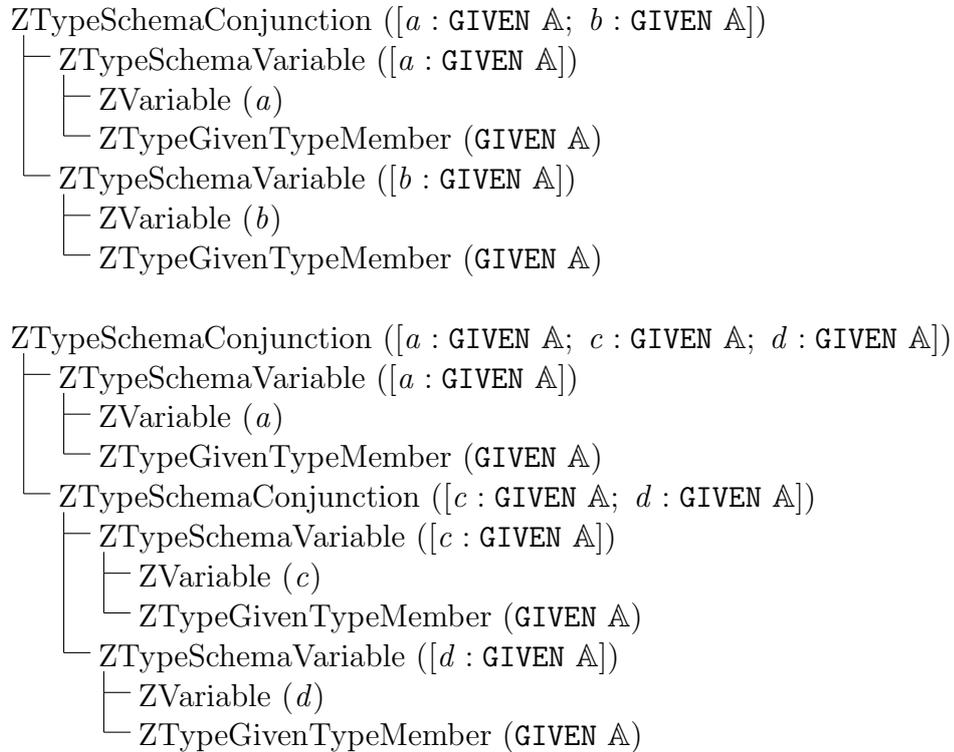


Figure 4.33: Example of two schema type trees prior to conjunction

In our example, the first schema variable we come to is a , which already belongs to the first tree. We, therefore, test that their types are equal, and in this case this is clearly the case (that is, they both are typed as `GIVEN A`). This schema variable is, therefore, skipped. The next schema variable we come to is c , this is not in the first tree so it is appended, and we get the same result when examining the final schema variable, d . The tree of the conjoined type is shown in figure 4.34 on the next page.

Finally, we create an instance of ‘ZExpression’, and associate the generated type with it (obviously incorporating it within an instance of ‘ZType-Powerset’ first). We then perform the tree manipulation required (figure 4.35 on the facing page), and add the transformed AST (and its associated ‘ZExpression’ object) to the current type environment.

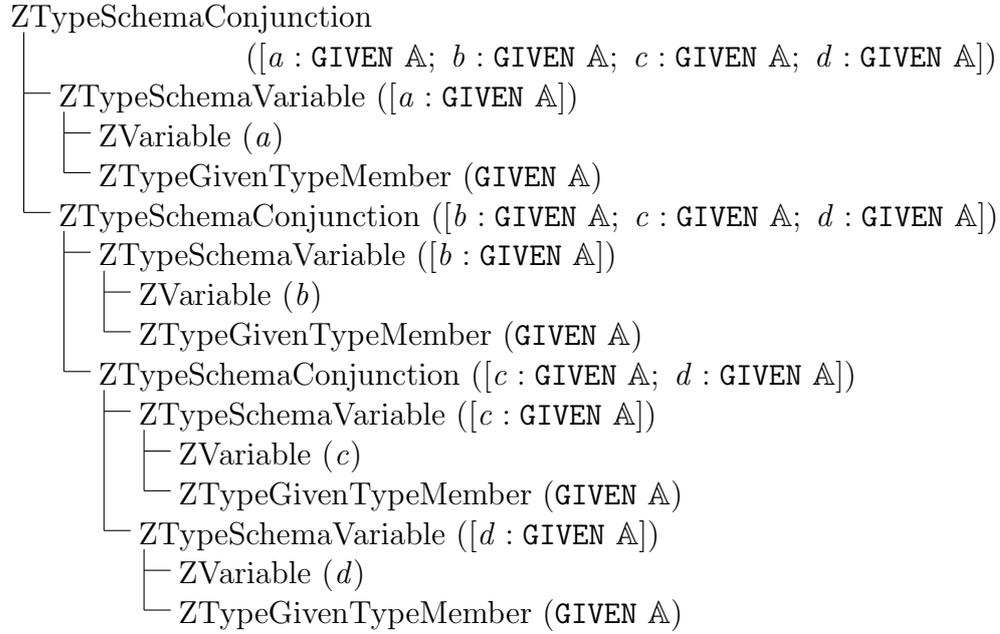


Figure 4.34: Example of schema type tree following conjunction

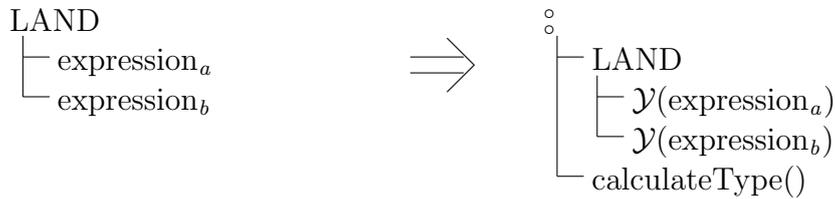


Figure 4.35: Typing transformation rule for schema conjunction expression

Example 4.6

In our final example, we will examine the typing process for a membership predicate. Clearly, the typing process for a predicate is a little different, as we do not generate a type for the predicate itself, merely ensure that the expressions involved are well typed and compatible. The rule for typing a membership predicate is given in [ISO02] as follows.

Rule 13.2.5.1 - Membership Predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \Sigma \vdash^{\mathcal{E}} e_2 \circ \tau_2}{\Sigma \vdash^{\mathcal{P}} (e_1 \circ \tau_1) \in (e_2 \circ \tau_2)} \quad (\tau_2 = \mathbb{P} \tau_1)$$

We are not deriving a type for the predicate, so the important aspect here is ensuring the compatibility of the types of the two expressions. In this instance, we need to ensure that the type of the ‘set’ expression is the powerset of the type of the ‘member’ expression.

There are no variables declared in this type of predicate so we do not need a new type environment, and therefore, our first step is the generation of the types for the two expressions. Again, if either of these expressions cannot be typed correctly, the predicate cannot be considered well-typed and a typing error is generated. Assuming both expressions are typed correctly, we take the type of the ‘set’ expression, and – as with axiomatic description paragraphs – determine its member type. We then compare this directly to the type of the ‘member’ expression. If they are equal we proceed, but if they are not then we generate a typing error.

If the expressions are compatible, it is necessary to add our predicate to a set of ‘well-typed predicates’ in the current type environment. This ensures that any typing errors are propagated up the AST. The transformation rules for predicates are also extremely simple, as they simply require the checking of the constituent parts. The transformation rule for the membership predicate is shown in figure 4.36.

$$\begin{array}{c} \text{IN} \\ \lrcorner \\ \text{expression}_a \\ \lrcorner \\ \text{expression}_b \end{array} \quad \Rightarrow \quad \begin{array}{c} \text{IN} \\ \lrcorner \\ \mathcal{Y}(\text{expression}_a) \\ \lrcorner \\ \mathcal{Y}(\text{expression}_b) \end{array}$$

Figure 4.36: Typing transformation rule for membership predicate

Instantiating Generic Types

All of the above rules apply to paragraphs and expressions with standard types. When using generic operators or generic paragraphs, however, the situation becomes a little more complicated. It is necessary to instantiate generic types with their specific values before proceeding with the remainder of the type check.

We consider first the explicit instantiation of generic types. Here the generic instantiation parameters are presented in the specification itself.

Example 4.7

We begin with a simple specification where we define the generic operator that allows us to specify a relation. We then define a schema *explicitGenericType* where we define a variable, a , that is a relation between natural numbers, and initialize that relation to contain a single pair.

generic 5 rightassoc($_ \leftrightarrow _$)

$X \leftrightarrow Y == \mathbb{P}(X \times Y)$

$\text{explicitGenericType}$
$a : \mathbb{N} \leftrightarrow \mathbb{N}$
$a = \{0 \mapsto 1\}$

After the syntax checking and syntax transformation phases, this specification is altered substantially, and is now in a language that satisfies the annotated grammar. The current specification is shown below. We now proceed to the type checking phase.

generic 5 rightassoc($_ \leftrightarrow _$)

| $[X, Y][\bowtie \leftrightarrow \bowtie : \{\mathbb{P}\{[-g_1 : X] \wedge [-g_2 : Y] \bullet (-g_1, -g_2)\}\}]$

| $[explicitGenericType : \{[a : \bowtie \leftrightarrow \bowtie [\mathbb{N}, \mathbb{N}] \mid a \in \{\{number_literal_0 \mapsto number_literal_1\}\}\}]$

The operator template paragraph is an exception to the rule that every paragraph must have a type, and is not assigned a type. The first axiomatic description paragraph is typed in the normal way, resulting in the following type being assigned to the paragraph.

$[\bowtie \leftrightarrow \bowtie : \mathbb{P}(\mathbb{P}(\text{GENTYPE } X \times \text{GENTYPE } Y))]$

Typing the second axiomatic description paragraph is a little more involved as it requires instantiating this generic type. The first step, is the typing of the variable construction expression $(a : \bowtie \leftrightarrow \bowtie [\mathbb{N}, \mathbb{N}])$, this begins by checking that the two formals can be typed correctly. The type of \mathbb{N} is retrieved from the current type environment, and we therefore have the implicit instantiation parameters $[\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}]$, meaning that we can proceed with the generic instantiation. If any of the formals cannot be typed then clearly we cannot type the expression, and we must generate a typing error.

The next step is to retrieve the type of the generic operator $(\bowtie \leftrightarrow \bowtie)$ from the current type environment. This operator is a variable like any other, and its retrieval proceeds in exactly the same way. The type we retrieve in this instance is $\mathbb{P}(\mathbb{P}(\text{GENTYPE } X \times \text{GENTYPE } Y))$, which has the type tree shown in figure 4.37 on the next page.

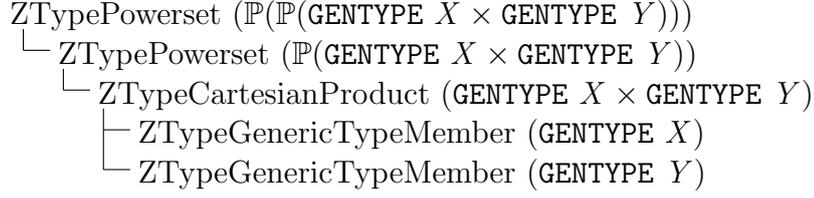


Figure 4.37: Type tree for relation generic operator

We now define an operator $\mathcal{G}()$ that takes the type tree of the generic operator, and the list of types of the formals, and attempt to instantiate the generic type. The rules for this operator are presented in table 4.5 on the following page, given that the arguments to $\mathcal{G}()$ are τ – the type of the current node – and *params* – a sequence of types belonging to the formals.

We define some basic methods for each of our classes that allow us to retrieve the components of a type. For instance, in the ‘ZTypeBindingConjunction’ class we define two methods *getLeft()* and *getRight()* that allow us to retrieve the two binding types involved in the conjunction.

The rule of most interest is that for the ‘ZTypeGenericType’ class. This is where the actual substitution takes place. When creating an instance of ‘ZTypeGenericType’, we record its parameter number, which we can then retrieve at instantiation time to ensure we replace the correct type. If the size of the list of parameters is too short then we will create a typing error.

We can apply the instantiation operator to our example as follows.

$$\begin{aligned}
& \mathcal{G}(\mathbb{P}(\mathbb{P}(\text{GENTYPE } X \times \text{GENTYPE } Y)), [\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}]) \\
& \longrightarrow \mathbb{P}(\mathcal{G}(\mathbb{P}(\text{GENTYPE } X \times \text{GENTYPE } Y), [\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}])) \\
& \longrightarrow \mathbb{P}(\mathbb{P}(\mathcal{G}(\text{GENTYPE } X, [\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}]) \\
& \quad \quad \quad \times \mathcal{G}(\text{GENTYPE } Y, [\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}]))) \\
& \longrightarrow \mathbb{P}(\mathbb{P}(\text{GIVEN } \mathbb{A} \times \mathcal{G}(\text{GENTYPE } Y, [\text{GIVEN } \mathbb{A}, \text{GIVEN } \mathbb{A}]))) \\
& \longrightarrow \mathbb{P}(\mathbb{P}(\text{GIVEN } \mathbb{A} \times \text{GIVEN } \mathbb{A}))
\end{aligned}$$

We, therefore assign the type $\mathbb{P}(\text{GIVEN } \mathbb{A} \times \text{GIVEN } \mathbb{A})$ to *a*, and add the variable to the current type environment. The remainder of the paragraph can then be typed as normal.

Table 4.5: Generic instantiation rules

Class of τ	Instantiated Type
ZTypeBindingConjunction	$\mathcal{G}(\tau.getLeft(), params) \wedge \mathcal{G}(\tau.getRight(), params)$
ZTypeBindingVariable	$[\tau.getVariable() : \mathcal{G}(\tau.getSet(), params)]$
ZTypeCartesianProduct	$\mathcal{G}(\tau.getNthMember(1), params)$ \times \dots \times $\mathcal{G}(\tau.getNthMember(n), params)$
ZTypeGenericType	$params[\tau.getParameterNumber()]$
ZTypeGenericTypeMember	$(\mathcal{G}(\tau.getGenericType())).getMember()$
ZTypeGivenType	τ
ZTypeGivenTypeMember	τ
ZTypePowerset	$\mathbb{P}(\mathcal{G}(\tau.getMember(), params))$
ZTypeSchemaConjunction	$\mathcal{G}(\tau.getLeft(), params) \wedge \mathcal{G}(\tau.getRight(), params)$
ZTypeSchemaVariable	$[\tau.getVariable() : \mathcal{G}(\tau.getSet(), params)]$

We consider now two examples where the instantiation parameters are not explicit in the specification. These examples illustrate where explicit instantiation typically occurs, and how we are able to deduce the instantiation parameters.

Example 4.8

We consider first membership predicates, where generically defined variables and relations can often occur; a particularly widespread example is the emptyset. Normally, when writing a specification, we do not provide a type for

the emptyset explicitly, but as every variable must have a type in Z , we must be able to deduce an applicable type.

The following paragraphs define a generic variable, \emptyset , and show an instance of its use where implicit generic instantiation is required.

$$\emptyset[X] == \{x : X \mid \text{false}\}$$

$\frac{\text{implicitGenericTypeMembership}}{a : \mathbb{P}\mathbb{N}}$
$a = \emptyset$

Once the syntax checking and syntax transformation phases have completed we are left with the following paragraphs, and can begin the type checking process.

$$\mid [X][\emptyset : \{\{[x : X] \mid \neg \text{true}\} \bullet x\}]$$

$$\mid \left[\begin{array}{l} \text{implicitGenericTypeMembership} : \\ \{[a : \mathbb{P}\mathbb{N} \mid a \in \{\emptyset\}]\} \end{array} \right]$$

Once again, the first of our paragraphs can be typed in the normal way, and we assign it the following type.

$$[\emptyset : \mathbb{P}(\text{GENTYPE } X)]$$

Typing the second axiomatic description paragraph seems to proceed as normal until we get to the typing of the membership predicate ($a \in \{\emptyset\}$). The variable a has the type $\mathbb{P}(\text{GIVEN } \mathbb{A})$, but the set is initially typed as $\mathbb{P}(\mathbb{P}(\text{GENTYPE } X))$. If we apply the typing rules for membership predicates presented in example 4.6 we are obviously going to run into trouble. We, therefore, extend our membership predicate rules, to allow for the implicit instantiation of generic types. If when typing a membership predicate our

normal checks fail, we will check whether either member or set expression has a generic type (a generic type is one that has a component that is an instance of ‘ZTypeGeneric’), and if so we will attempt to instantiate that type with implicit parameters.

In this instance it is the ‘set type’ that is generic, and we will attempt to instantiate this type. Unlike the explicit instantiation, however, we also need to examine the ‘member type’ in order to determine what our parameters ought to be. The object trees for these types are shown in figure 4.38.

$$\begin{array}{l} \text{ZTypeGivenType } (\mathbb{P}(\text{GIVEN } \mathbb{A})) \\ \\ \text{ZTypePowerset } (\mathbb{P}(\mathbb{P}(\text{GENTYPE } X))) \\ \quad \lrcorner \text{ZTypeGenericType } (\mathbb{P}(\text{GENTYPE } X)) \end{array}$$

Figure 4.38: Type trees required for implicit instantiation of emptyset

The first step of the instantiation process is to determine the required parameters. We define an operation $\mathcal{DP}()$ that takes two arguments: τ_1 which is the ‘member type’ (in this instance, $\mathbb{P}(\text{GIVEN } \mathbb{A})$, and τ_2 which is the constituent component of the ‘set type’ (in this instance, $\mathbb{P}(\text{GENTYPE } X)$). This operation returns the set of parameters that can be used to instantiate the generic type. The rules for this operation are applied in order, and are presented in table 4.6.

When we add each parameter to our set of parameters we make a note of the parameter’s intended position. We then sort our parameters in order of position so that we have a sequence of types which can be used to instantiate the type. If two parameters have the same position, or there are more, or less, parameters than are required for our generic type, then we generate a typing error as the instantiation cannot proceed. If there are no issues with our parameter list we can continue, and instantiate our type with the $\mathcal{G}()$ operation.

In our example, we have the following sequence of operations.

$$\begin{aligned}
& \mathcal{G}(\mathbb{P}(\mathbb{P}(\text{GENTYPE } X)), \mathcal{DP}(\mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{P}(\text{GENTYPE } X)) \\
& \longrightarrow \mathcal{G}(\mathbb{P}(\mathbb{P}(\text{GENTYPE } X)), [\mathbb{P}(\text{GIVEN } \mathbb{A})]) \\
& \longrightarrow \mathbb{P}(\mathcal{G}(\mathbb{P}(\text{GENTYPE } X), [\mathbb{P}(\text{GIVEN } \mathbb{A})])) \\
& \longrightarrow \mathbb{P}(\mathbb{P}(\text{GENTYPE } \mathbb{A}))
\end{aligned}$$

This instantiated type now matches our original rules for the membership predicate, and we are able to assign types to the necessary expressions appropriately. Note that, once we have determined the generic type it is also necessary to ensure that nodes further down the AST have their generic types instantiated.

Table 4.6: Rules for determining implicit parameters

Condition	Returned Parameters
$\tau_1 : \text{ZTypeGenericType}$	$\{\tau_2\}$
$\tau_2 : \text{ZTypeGenericType}$	$\{\tau_1\}$
$\tau_1 : \text{ZTypeGenericTypeMember}$	$\{\mathbb{P}(\tau_2)\}$
$\tau_2 : \text{ZTypeGenericTypeMember}$	$\{\mathbb{P}(\tau_1)\}$
$\tau_1 : X \wedge \neg (\tau_2 : X)$	\emptyset
$\tau_1 : \text{ZTypeBindingConjunction}$	$\mathcal{DP}(\tau_1.\text{getLeft}(), \tau_2.\text{getLeft}())$ \cup $\mathcal{DP}(\tau_1.\text{getRight}(), \tau_2.\text{getRight}())$
$\tau_1 : \text{ZTypeBindingVariable}$	$\mathcal{DP}(\tau_1.\text{getSet}(), \tau_2.\text{getSet}())$
$\tau_1 : \text{ZTypeCartesianProduct}$	$\mathcal{DP}(\tau_1.\text{getNthMember}(1), \tau_2.\text{getNthMember}(1))$ \cup \dots \cup $\mathcal{DP}(\tau_1.\text{getNthMember}(n), \tau_2.\text{getNthMember}(n))$
$\tau_1 : \text{ZTypePowerset}$	$\mathcal{DP}(\tau_1.\text{getSet}(), \tau_2.\text{getSet}())$
$\tau_1 : \text{ZTypeSchemaConjunction}$	$\mathcal{DP}(\tau_1.\text{getLeft}(), \tau_2.\text{getLeft}())$ \cup $\mathcal{DP}(\tau_1.\text{getRight}(), \tau_2.\text{getRight}())$
$\tau_1 : \text{ZTypeSchemaVariable}$	$\mathcal{DP}(\tau_1.\text{getSet}(), \tau_2.\text{getSet}())$

Example 4.9

Our second example highlights the necessity to handle generic definitions in application expressions, in which function operators are typically used. In this example, we will consider the use of the union operator, which is one of the operators defined within the set toolkit (see 4.4.2 and B.5.2).

The following paragraphs define the union operator, and show an instance of its use in a schema definition paragraph. We also define the total function, which is required for our union operator.

generic 5 rightassoc($_ \rightarrow _$)

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f\}$$

function 30 leftassoc($_ \cup _$)

$\frac{[X]}{\frac{_ \cup _ : (\mathbb{P} X \times \mathbb{P} X) \rightarrow \mathbb{P} X}{\forall a, b : \mathbb{P} X \bullet a \cup b = \{x : X \mid x \in a \vee x \in b\}}}$

$\frac{\textit{implicitGenericTypeApplication}}{b : \mathbb{P} \mathbb{N}}$
$b = \{0\} \cup \{1\}$

As usual, these paragraphs are subjected to the manipulations of the syntax checking, and syntax transformation phases. For brevity, we show only the transformed schema definition paragraph.

$\left[\textit{implicitGenericTypeApplication} : \{[b : \mathbb{P} \mathbb{N} \mid b \in \{\bowtie \cup \bowtie (\{0\}, \{1\})\}]\} \right]$

The first step is to determine the type of the generic axiomatic definition paragraph in which we define the union operator; this is shown below.

$$[\bowtie \cup \bowtie : \mathbb{P}((\mathbb{P}(\text{GENTYPE } X) \times \mathbb{P}(\text{GENTYPE } X)) \times \mathbb{P}(\text{GENTYPE } X))$$

When we attempt to type the paragraph in which we use the union operator, however, we can only proceed with our normal typing method until we reach the application expression $(\bowtie \cup \bowtie (\{0\}, \{1\}))$. The type of the function is $\mathbb{P}((\mathbb{P}(\text{GENTYPE } X) \times \mathbb{P}(\text{GENTYPE } X)) \times \mathbb{P}(\text{GENTYPE } X))$, but the argument has the type $(\mathbb{P}(\text{GIVEN } \mathbb{A}) \times \mathbb{P}(\text{GIVEN } \mathbb{A}))$.

The rule for typing application expressions is given below.

Rule 13.2.6.11 - Application Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \Sigma \vdash^{\mathcal{E}} e_2 \circ \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \circ \tau_1) (e_2 \circ \tau_2) \circ \tau_3} \quad (\tau_1 = \mathbb{P}(\tau_2 \times \tau_3))$$

Clearly, the types we have identified are not compatible under this rule. We, therefore, extend our rules for application expressions in the same way as we did for membership predicates above. If our initial attempt at testing fails, we test if our function has a type that is generic, and if so we will attempt to instantiate it.

However, the method of instantiation in this case is slightly different to that in a membership predicate. If we consider our function operator to have the type $\mathbb{P}(\tau_{1G} \times \tau_{2G})$, the argument expression to have the type τ_{1S} , and the whole application expression to have the type τ_{2S} , then we must first check that we can deduce the instantiation of τ_{1G} from τ_{1S} , then we must ensure that we can perform a valid instantiation of τ_{2G} (that gives τ_{2S}) with the same parameters.

The first step, therefore, is to call the $\mathcal{DP}()$ operation with τ_{1G} and τ_{1S} . If this proceeds correctly, then we attempt to instantiate τ_{2S} with the parameters derived from that operation.

In our example, therefore, we would have the following sequence of operations.

$$\begin{aligned}
& \mathcal{G}(\mathbb{P}(\text{GENTYPE } X), \\
& \quad \mathcal{DP}(\mathbb{P}(\text{GIVEN } \mathbb{A}) \times \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{P}(\text{GENTYPE } X) \times \mathbb{P}(\text{GENTYPE } X))) \\
& \longrightarrow \mathcal{G}(\mathbb{P}(\text{GENTYPE } X), [\mathbb{P}(\text{GIVEN } \mathbb{A})]) \\
& \longrightarrow \mathbb{P}(\text{GIVEN } \mathbb{A})
\end{aligned}$$

It is also necessary to instantiate the generic type of the function operator itself (so that it will have the type $\mathbb{P}(\tau_{1S} \times \tau_{2S})$), and as before, propagate the instantiation down the AST. Once this has been done we have three types that are consistent with rule 13.2.6.11.

Example 4.10

We return now – for a final time – to the train yard example we last visited in example 4.3. We will show how we perform the type checking of the abstract syntax tree (AST) parsed by the syntax transformation phase, and how we create a new AST that is annotated with every paragraph and expression’s type. Once again, we concentrate specifically on the *trainEntersYard* paragraph, the AST with which we begin the type checking phase is shown in figure 4.28 on page 189. Rather than showing how every sub-tree of the type annotated AST is created (which would be rather long winded), we simply present the full tree in figures 4.39, 4.40, and 4.41.

The first syntactic component that we need to type is the axiomatic definition paragraph. We know from rule 13.2.3.2 (which we discussed above), that the type of this paragraph is derived from the type of the expression it contains. We, therefore, create a new type environment, and determine the type of the paragraph’s enclosed expression. Once that expression has been typed we will set the type of our paragraph to be the component type of the powerset type of the expression, and promote any local variables to the section type environment. In this case, we promote *trainEntersYard* so that the schema is available throughout the section (and any inheriting sections).

The expression enclosed within our paragraph is a variable construction expression, the typing of which is governed by rule 13.2.6.13 (below). This rule again states that the typing of our variable construction expression is dependent upon the typing of the set to which our new variable belongs.

Once this set has been typed, we create a new instance of ‘ZVariable’ (with the name *trainEntersYard*) which we assign the appropriate type, and add to the current type environment. We then create an instance of ‘ZType-SchemaVariable’ using our new variable, which can be assigned as the type of our variable construction expression.

Rule 13.2.6.13 - Variable Construction Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e \circ \tau_1}{\Sigma \vdash^{\mathcal{E}} [i : (e \circ \tau_1)] \circ \mathbb{P} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P} \alpha \\ \tau_2 = \mathbb{P}[i : \alpha] \end{array} \right)$$

In this instance, the set in our variable construction expression is specified with a set extension expression. The type of a set extension expression is simply the powerset of the type of its contents, which is expressed formally below.

Rule 13.2.6.3 - Set Extension Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \dots \quad e_n \circ \tau_n}{\Sigma \vdash^{\mathcal{E}} \{(e_1 \circ \tau_1), \dots, (e_n \circ \tau_n)\} \circ \tau} \left(\begin{array}{l} \text{if } n > 0 \text{ then} \\ (\tau_1 = \tau_n) \\ \vdots \\ \tau_{n-1} = \tau_n \\ \tau = \mathbb{P} \tau_1) \\ \text{else } \tau = \mathbb{P} \alpha \end{array} \right)$$

The set extension expression contains only a single member, which is a schema construction expression. The rule for typing schema construction expressions is rule 13.2.6.14. The type of the schema construction expression is the same as that of its component expression (assuming that the component predicate is well-typed). We create a new type environment to which any variables declared within the expression will be promoted for use within the predicate. In this example, we have the variables belonging to the schema Δ *trainYard*, and *trainEnteringYard?* which will be added to our new type

environment in order that they can be used within the predicate.

Rule 13.2.6.14 - Schema Construction Expression:

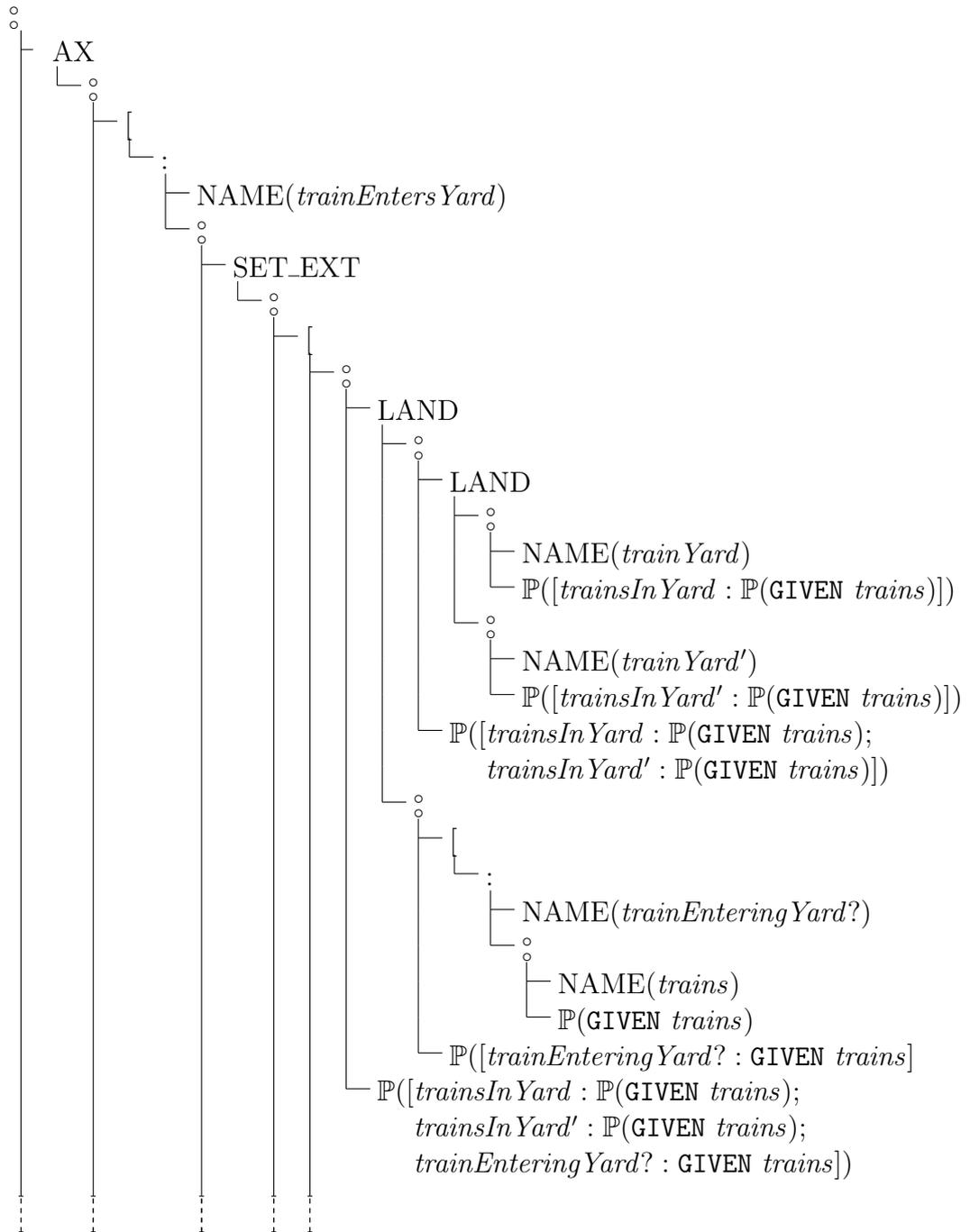
$$\frac{\Sigma \vdash^{\mathcal{E}} e \circ \tau_1 \quad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{E}} [(e \circ \tau_1) \mid p] \circ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \tau_1 \end{array} \right)$$

The expression within our schema construction is a schema conjunction expression, the typing of which we discussed in detail in example 4.5. The expressions involved in the schema conjunction are a reference expression, and a variable construction expression.

The reference expression refers to the schema $\Delta trainYard$, which we automatically translate to $[trainYard; trainYard']$ before searching our type environment to find a type for our reference. When searching for primed references we search for the unprimed variable, and then prime the retrieved type accordingly. In this instance, the variable $trainYard$ is found in our section type environment (a parent of the current type environment), and has the type $[trainsInYard : \mathbb{P}(\text{GIVEN } trains)]$. The type of reference expression can, therefore, be deduced to be $[trainsInYard : \mathbb{P}(\text{GIVEN } trains); trainsInYard' : \mathbb{P}(\text{GIVEN } trains)]$. Note that, when we promote the variables of a schema to type environment, we are able to deduce these directly from the instances of ‘ZTypeSchemaVariable’ that are used to create the schema’s type.

The variable construction expression declares the variable $trainEnteringYard?$, and the type is determined from the reference to the variable $trains$. Again this variable is declared in our section type environment with the type $\mathbb{P}(\text{GIVEN } trains)$. So we can add $trainEnteringYard?$ to the current type environment with the type $\text{GIVEN } trains$, and can assign the type $[trainEnteringYard? : \text{GIVEN } trains]$ to the variable construction expression, and the type $[trainsInYard : \mathbb{P}(\text{GIVEN } trains); trainsInYard' : \mathbb{P}(\text{GIVEN } trains); trainEnteringYard? : \text{GIVEN } trains]$ to the schema conjunction expression.

We now return to the predicate belonging to our schema construction expression, which we must ensure is correctly typed. Our predicate is a

Figure 4.39: Typed AST for *trainEntersYard* (1)

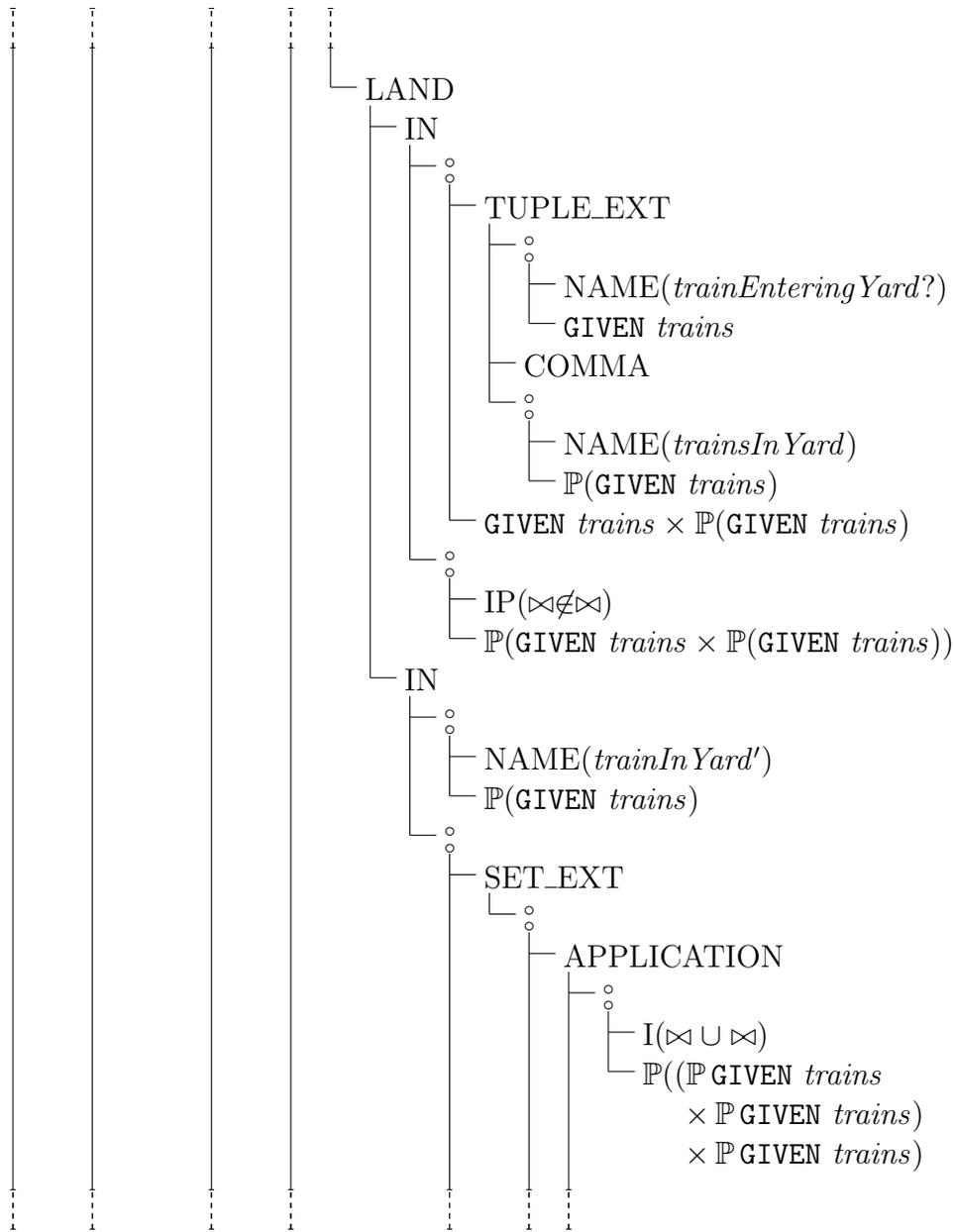
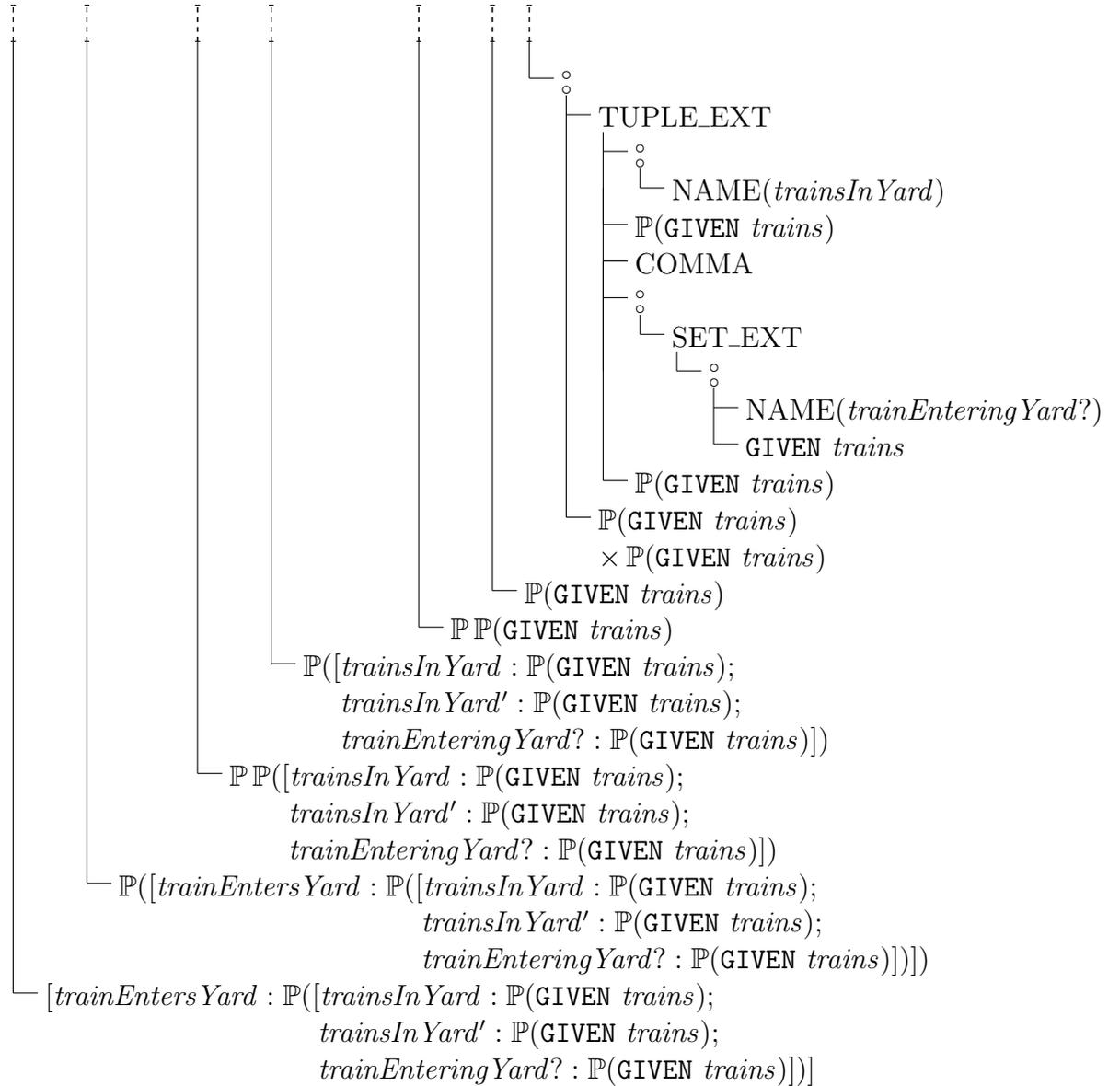


Figure 4.40: Typed AST for *trainEntersYard* (2)

conjunction predicate, and a conjunction predicate is well-typed if the two predicates it conjoins are well-typed.

Figure 4.41: Typed AST for $trainEntersYard$ (3)

Rule 13.2.5.4 - Conjunction Predicate:

$$\frac{\Sigma \vdash^P p_1 \quad \Sigma \vdash^P p_2}{\Sigma \vdash^P p_1 \wedge p_2}$$

Both of the predicates within the conjunction predicate are membership predicates. The first of these requires the implicit generic instantiation of the type for the reference to $\bowtie/\neq/\bowtie$. This instantiation is similar to example 4.8. We retrieve the generic type from the set toolkit's type environment (which is a parent of the our section's type environment), acquiring the type $\mathbb{P}((\text{GENTYPE } X) \times \mathbb{P}(\text{GENTYPE } X))$. We then type the 'member' expression that is a tuple extension expression, which is typed according to rule 13.2.6.6; the type for the expression being the cartesian product of the component types.

Rule 13.2.6.6 - Tuple Extension Expression:

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \circ \tau_1 \quad \dots \quad \Sigma \vdash^{\mathcal{E}} e_n \circ \tau_n}{\Sigma \vdash^{\mathcal{E}} ((e_1 \circ \tau_1), \dots, (e_n \circ \tau_n)) \circ \tau} \quad (\tau = \tau_1 \times \dots \times \tau_n)$$

We retrieve the types for the references in our tuple extension expression from the type environment, and assign the type $\text{GIVEN } \textit{trains} \times \mathbb{P}(\text{GIVEN } \textit{trains})$ to the expression. We then use the rules outlined in tables 4.5 and 4.6 to implicitly instantiate the generic type.

The second of our membership predicates consists of a reference to *trainsInYard'* – which we can type through reference to the current type environment, retrieving the type $\mathbb{P}(\text{GIVEN } \textit{trains})$ – and a set extension expression. The only member of the set extension is an application expression. This application expression requires the implicit instantiation of a generic type very similar to that shown in example 4.9. The only difference being that the instantiation parameters are $\mathbb{P}(\text{GIVEN } \textit{trains})$ rather than $\mathbb{P}(\text{GIVEN } \mathbb{A})$. We can quickly deduce, therefore, that the type of our application expression is $\mathbb{P}(\text{GIVEN } \textit{trains})$. The set extension, therefore, has a type of $\mathbb{P}\mathbb{P}(\text{GIVEN } \textit{trains})$, and we can conclude that our membership predicate – and therefore, our conjunction predicate – is well-typed.

We can now take the type we generated for the expression part of our schema construction expression, and use it to complete the typing of that expression, and propagate that type back up the AST allowing us to complete our type-annotated AST. Once we have a type for our paragraph, and assuming that our other paragraphs are well-typed, we can conclude that our

trainYard section is syntactically correct and well-typed.

4.7 Semantic Translation

Chapters 14 and 15 of [ISO02] deal with the semantics of a Z specification. The first of these chapters discusses instances where different expressions can have the same semantics. It then proceeds to presents rules that allow us to reduce redundancy of notation in our specification, through the transformation of our typed AST for a final time, eliminating those language constructs where a semantic equivalent can be produced from other language elements. The second of these chapters presents a relational definition of each of the remaining language constructs, giving rules from which iterative application will produce a relational model of our specification.

We will not consider the semantic nature of our specifications in this chapter, but shall instead discuss how we produce a semantic model in chapter 6, where we consider the generation of proof obligations for constructs expressed in the Z notation.

4.8 Evaluation: A Comparison of the Frog Parser and CZT

In essence the Frog parser and the CZT had the same goal: to produce a parser capable of creating a type-annotated AST from a Z specification in a way that conforms to the ISO standard. We seek therefore, to evaluate the success of the Frog parser by comparing it with the equivalent tools of the CZT. We also wish to demonstrate how the differences in the focuses of these tools, have produced very different designs, and that these differences have made the development of the Frog parser a worthwhile procedure.

The majority of the differences in the way in which the tools actually parse the Z notation result from the ways in which the designers of the tools have interpreted the ambiguity of [ISO02]. In many ways this interpretation has been influenced by the choice of the grammars used by the parser (more

4.8. EVALUATION: A COMPARISON OF THE FROG PARSER AND CZT221

correctly, the tools used to create those grammars). We have discussed in this chapter that we have used the ANTLR parser generator which generates a predicated LL(k) recursive descent parser. The CZT, however, uses a LALR(1) parser which has been generated by the CUP tool [Hud]. When we consider for example, the necessity to disambiguate between expressions and predicates that we discussed in section 4.6.4, we can see that Frog’s method of using syntactic and semantic predicates is very different to the solution used by the CZT. There all expressions and predicates are parsed as the same language construct, and ‘fixed-up’ at a later point. Whilst these differences can be seen upon examination of the design of each tool, however, they do not affect the result of using either parser, and as each conforms to the ISO standard, identical results should (and, in reality, are) obtained. These interpretation differences can therefore, be seen to be mostly cosmetic.

Apart from these interpretation differences, however, there are also differences in the design of the tools that reflect the different activities for which they are intended to be used. Whilst we have created a parser which will allow us to create and verify models and relationships with the Z notation, the CZT is focused on creating tools where specification in Z is the principal aim. The CZT can certainly be considered (at present) to be a lightweight formal methods tool, and there seems to be little intention to introduce a verification element at any point in the near future. In fact, there currently exists a proposal to attach a Z to B translator to the CZT tool set, with the reasoning behind the proposal being the ability to take advantage of the verification (and particularly refinement) facilities of the B-Toolkit, Atelier-B and RODIN.

This difference between the tools can be seen further in [MFMU05], where Miller describes the way in which a great deal of effort has been extended to ensure that the Z notation parsed by CZT is extendable so that extensions to the parser can be created semi-automatically. This will allow CZT to be used to, relatively easily, provide support to extensions such as object-orientation, real-time features and process algebra. When we created the Frog parser we did not have this goal in mind, and our focus instead was on producing a tool in which we could incorporate the notion of constructs, and

produce semantic models of those constructs (from which we could generate, and discharge, proof obligations). In fact, the very nature of the CZT's extensibility makes the abstract syntax trees it produces much more difficult to use in the generation of semantic models and its heavy use of the visitor pattern makes classes intentionally generic.

We feel that both parsers are able to successfully parse ISO standard Z, and while different approaches have been taken there is not a great deal of difference in their ability to perform this task. Before we consider our work redundant, however, we must consider that while in essence the two tools perform the same task, they are distinguished by the preparation which they perform for the remainder of the tools in the toolkit to which they belong, and the value of the foundations they create (in both instances) cannot be underestimated. In conclusion, we feel that even were we beginning our work on providing mechanized support for retrenchment today we would be required to expend significant effort bending the CZT parser to fit our task, and that the creation of an independent parser may well still have been justified.

4.9 Concluding Remarks

In this chapter we have discussed the creation of a parser for the Z notation. Z is one of the most common, and most flexible formal specification languages used today. A Z specification consists of sections, which in turn, are comprised of paragraphs. A distinguishing feature of Z is its schema calculus. Despite being such a popular specification language there is a distinct lack of open source parsers that conform to the recently published ISO standard. We felt, therefore, that it was necessary to create our own Z parser, which we decided to implement using the ANTLR parser generator. A number of representations can be used for the Z notation, we opted to use the \LaTeX format outlined in [ISO02]. The parsing of a Z specification is achieved by parsing all of the files that belong to that specification, and linking the sections contained within each through their genealogy. We referred to the process of producing a typed AST from a Z specification as syntax checking.

The syntax checking process consists of a number of phases namely: lexing, parsing, syntax transformation and type checking. We have described each of these phases in detail in the preceding sections, principally outlining the differences between our approach and the one documented in [ISO02], particularly where that document has left the specific implementation open-ended. Finally, we have evaluated our parser by comparing it with the independently developed CZT, and have drawn the conclusions that whilst both tools are capable of parsing ISO standard Z, we have shown that there is more than one way to create a standard-conformant parser, and that the differences between the tools reflect the different purposes for which they were intended.

Chapter 5

Modelling in the Z Notation

This chapter describes how we can use the Z notation to specify constructs representing machines and the relationships between them. We start by introducing Frog-CCL (Construct Configuration Language), which we have designed to allow the flexible configuration of constructs, enabling a vast range of machines and relationships to be specified with a common syntax, allowing that whole range of constructs to be used within our tool. We then show how the specification of a construct can be used alongside its configuration to produce proof obligations that can prove the validity of that construct. We then proceed to describe how the configured constructs can be introduced into a Z specification; defining an extension to the Z syntax, showing how that extension is parsed and then describing how that syntax is transformed into a form that satisfies the annotated syntax presented in chapter 10 of the ISO standard for the Z notation [ISO02]. Finally, we provide a sample configuration for a machine, a refinement relationship and a retrenchment relationship. In each case we give an example specification satisfying the configuration and show the proof obligations that will be produced.

5.1 Introduction

We use the notion of a construct to refer to a model that specifies either a machine or a relationship between machines. We define the notion of a machine as being similar to that of an abstract machine in the B-Method, which is described in full in [Abr96]. Put simply, a machine describes the state of a model and the operations that can affect that state. We will not go into much detail here about the nature of a machine, as one of the features of our system will be that this nature is configurable; however we describe the specification for a simple machine in example 5.1 below.

Example 5.1

MACHINE *myNumberMachine*
 TYPE *simpleMachine*
 SECTION *standard_toolkit*
 STATE

$myNumber : \mathbb{N}$

INITIALIZATION

$myNumber = 0$

OPERATION *increment* $\hat{=}$
 BODY

$myNumber' = myNumber + 1$

END *increment*

END *myNumberMachine*

This example describes a machine, *myNumberMachine*, that has a state consisting of a single natural number, *myNumber*.

One of the goals of our system is that a machine must be configurable. As such, it is possible that we will use machines that have different configurations within a single specification. For example, we may have a machine configuration that allows no operations, a machine configuration that requires both initialization and finalization clauses, or two machine configurations that allow machines with the same structure but different proof obligations. In order that we are able to process a machine correctly we need to know which configuration to use. Therefore, we must specify which configuration is being used in the specification of a particular machine. We do this by indicating the name of the configuration in the ‘type’ clause of our machine. For clarity we shall refer to a specific machine configuration as a machine-type. Our example is declared to belong to a machine-type called *simpleMachine*.

The ‘section’ clause of the machine refers to the parent Z section of the machine. A machine is able to use all of the types, schemas and operators of its parent section (and its ancestors). In this case the Z section *standard_toolkit* (a section belonging to the standard Z mathematical toolkit) is specified, allowing the machine to use – for example – all standard Z operators and types.

The ‘state’ clause of the machine allows the declaration of any state variables and any invariant restricting those variables. This declaration is performed with the Z syntax, with a schema text typically being used. Here, we define a single natural number, *myNumber*, with no particular restrictions.

The ‘initialization’ clause of the machine allows us to specify the default values given to the state variables when a new instance of the machine is created. Again, the declaration is given with the Z syntax. In this instance we initialize our variable to zero.

Finally, we declare an operation for our machine named *increment*. Here our operation has no inputs or outputs, and has a body that specifies that after the operation completes the value of *myNumber* will be one greater than when it began.

An example configuration for a machine, and a machine specification that uses that configuration, are presented in section 5.4. Relationship constructs are very similar to machine constructs, and examples of their configurations

and specification are also provided in sections 5.5 and 5.6.

We saw in chapter 4 the initial structure of a Frog specification, we now extend the definition that we presented in figure 4.2 on page 130 to include the notion of a construct. A Frog specification will still consist of a number of Frog files, but we will now allow each file to contain either Z sections or a construct. In order to re-use the existing type checking system we will define a syntax to allow the specification of constructs, but as part of the parsing process, this syntax will be transformed into a format suitable for the Z type checker. As such, each construct will be associated with a Z section, and its contents will be transformed into Z paragraphs (see section 5.3.3). The new structure of a FrogSpecification is given in figure 5.1

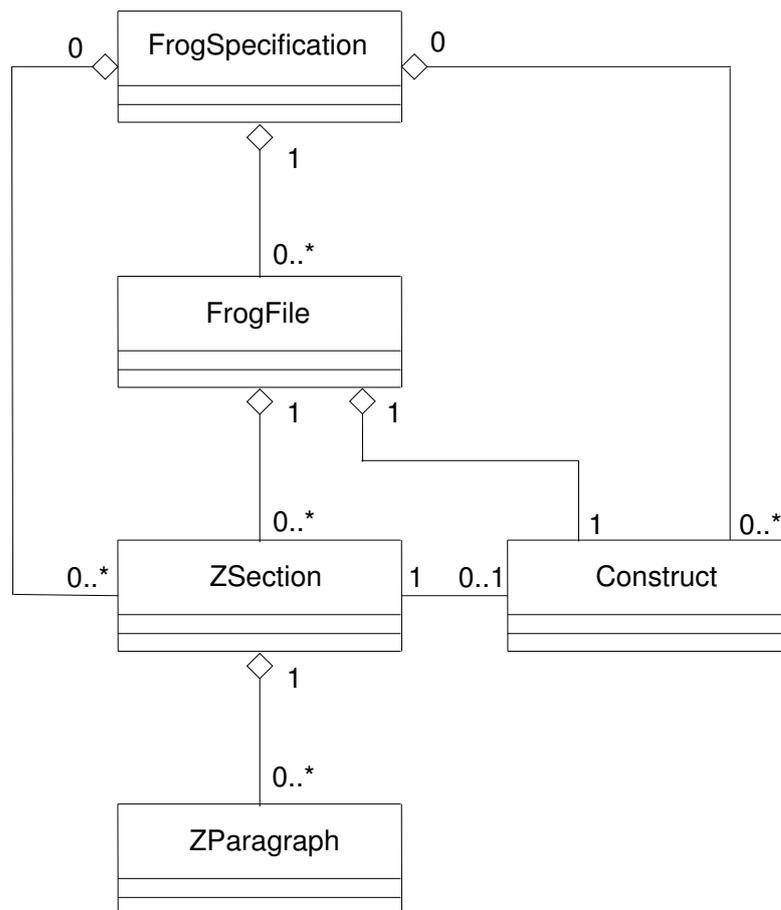


Figure 5.1: Contents of a Frog Specification with Constructs

The new syntax used to specify the constructs is presented in section 5.3.1 and involves an extension to the L^AT_EX Z syntax presented in [ISO02]. The majority of a construct's specification will be done in standard Z, but this extension allows us to wrap our constructs in reusable units, and provide a consistent format for machines and relationships. During the process of converting the construct into a Z section (and its constituent paragraphs), all of the syntax belonging to our extension will be removed and the definition of the resultant section will satisfy the grammar for the annotated syntax presented in chapter 10 of the ISO standard for the Z notation [ISO02].

We have already stated that our intention is to make the notion of a machine and a relationship as flexible as possible. The way in which we will do this is to define a language for specifying the configuration of constructs. This language, which we call Frog-CCL will allow us to specify the clauses that can be used in a particular type of construct, the operational information for a given type of construct and the proof obligations required to validate that construct type. We introduce this language in section 5.2. This language will allow a construct to be configured in such a way that any specification of that construct can be parsed, type-checked and validated (through the generation and discharge of proof obligations) automatically. Clearly, this will prove to be a great advantage when experimenting with the nature of a relationship between machines; it will be possible to configure Frog (or indeed any other tool capable of processing the language) to accept a new relationship without any need for recompilation.

5.2 Frog-CCL

In this section we present a meta-language for describing the configuration of a machine or a relationship. We decided that the first generation of our language should be as simple as possible, whilst providing a great deal of flexibility. We therefore, restricted the options so that the configuration for a construct would involve a declaration of its contents and instructions on how to use those contents to create proof obligations for the construct. There exists the possibility to impose more intricate options in the configuration.

For example, we may wish to configure a machine similar to the implementation machine of the B-Method and enforce a restriction that the machine must be a refinement of another (more abstract machine) and cannot be further refined. We did not feel that this level of detail would be required in the first iteration of Frog, but have attempted to make the definition of Frog-CCL – and its implementation – as extensible as possible. In the first part of this section we shall define the grammar that describes the Frog-CCL language and show how it can be used to configure some simple constructs. In the remainder of the section we show how this configuration can be used to derive a construct’s proof obligations.

5.2.1 Syntax for Frog-CCL

We begin by defining a grammar (again, using the ANTLR, LL(k) notation) to determine whether construct configurations belong to the Frog-CCL language. A construct’s configuration is principally made up of three components. Firstly, we define the clauses that belong to the construct. These clauses will hold the content of the construct and may, for example, represent its state or initialization. Secondly, we define the operation environments belonging to the construct. Each environment will contain related clauses that define behaviour at a sub-construct level (that is, where we may have more than one instance of an operation environment, and the clauses within it, contained in a single instance of a construct), for instance, we may have an operation environment that describes the operations of a machine or the ramifications of a retrenchment. The final part of a construct’s configuration is the generic proof obligations that instruct a tool how to generate the specific proof obligations for an instance of that construct.

The starting token in our grammar represents the definition of a particular construct.

```
construct_def ::= machine_def | relationship_def
```

A construct definition is either a machine definition or a relationship definition.

```

machine_def ::= "DEFINE"
              "MACHINE"
              name
              ( clauses |
                operations_environments |
                proof_obligations
              )*
              "END"

```

A machine definition begins with two starting tokens indicating that we are parsing a machine. The next token specifies the name for the type of machine that we are defining. This is followed by the configuration for the construct's clauses, operation environments and proof obligations. Finally an ending token indicates the end of the machine's configuration.

```

relationship_def ::= "DEFINE"
                   "RELATIONSHIP"
                   name
                   ( clauses |
                     operations_environments |
                     proof_obligations
                   )*
                   "END"

```

The definition for a relationship is basically the same as that for a machine, the only difference being the use of a token to indicate we are parsing a relationship.

```

clauses ::= "CLAUSES" clause (',' clause)*

```

```

clause ::= '(' attribute (',' attribute)* ')'

```

```

attribute ::= "NAME" '=' name
            | "LEVEL" '=' ("MACHINE" | "RELATIONSHIP" | name)

```

```

| "REQUIREMENT" '=' ("MANDATORY" | "OPTIONAL")
| "RELATION" '=' '<' relation (',' relation)* '>'
| "PRE_RELATION" '=' '<' relation (',' relation)* '>'
| "CONTENT" '=' ("SCHEMA_TEXT" | "PREDICATE" )

relation ::= "FROM_MACHINE" '(' name ')
| "TO_MACHINE" '(' name ')
| name

```

Each construct can have a number of clauses, each of which has a number of attributes. A clause may be used, for example, to initialize the state of a machine; the definition of such a clause is given below (figure 5.2).

```

...
( NAME= initialization,
  LEVEL= MACHINE,
  REQUIREMENT= OPTIONAL,
  RELATION= <state>,
  CONTENT= SCHEMA_TEXT
),
...

```

Figure 5.2: Configuration of a machine initialization clause

The first attribute is the name of the clause, this allows the clause to be identified and must be unique within a construct. The name attribute must be specified for every clause. In our example, we give our clause the name 'initialization'.

The next attribute allows us to specify where our clause can be used. We can choose to allow our clause to be used at either the construct level or within a specified operation environment. We use the `MACHINE` or `RELATIONSHIP` tokens to indicate use at the construct level, if we wish the clause to be used at the operation level we must use the name of the required operation environment. The level attribute must be specified for every clause. As state initialization is performed for a construct as a whole, and belongs to a machine we use the `MACHINE` token in our example.

The requirement attribute allows us to indicate whether a clause is mandatory or optional. It is not necessary to specify this attribute for every clause (if the attribute is not specified, the clause is assumed to be optional). As it is possible our machine may not have a state, or one which does not require initialization, we mark our clause as optional.

The relation attribute indicates the variables that are within the scope of the clause, and also the shape of the relation produced when the clause is used in a proof obligation. We can use clauses within the construct as part of the relation, and in relationships we are also able to use the clauses of the source and target machines. If a clause name is used within the relation attribute, and is decorated with an apostrophe (for example, `state'`), we also decorate all of the variables available in that clause when importing ¹. The relation attribute is optional, but must be present if the clause is used in a proof obligation, or we wish to have access to the variables of another clause. In our example, we will need to be able to assign values to the state of the machine so we declare that our relation will consist of the state clause. Similarly, we can define the `PRE_RELATION` attribute that allows us to define the shape of the relation produced from the precondition of the clause. Again, the attribute is optional and need only be defined where the precondition of the clause is used in a proof obligation.

The final attribute lets us declare the intended contents of the clause. We have the option to configure the clause as either a Z schema text or a Z predicate, the difference principally being that a schema text allows the declaration of additional variables to those made available through the relation attribute. The content attribute is mandatory. In this instance, we feel that choosing a schema text allows the use of local variables in a particularly complex initialization (we already have access to the state variables through the relation attribute).

```
operation_environments ::= "OPERATION_ENVIRONMENTS"
```

¹This is typically used to distinguish between the pre-transition and post-transition values of variables. For example, we may have a relation `<state,state'>` which allows the use of both pre-transition and post-transition values of variables, and enables us to distinguish between them.

```

operation_environment
    (',' operation_environment)*

operation_environment ::= '(' operation_env_attribute
    (',' operation_env_attribute)* ')'

operation_env_attribute
    ::= "NAME" '=' name
    | "REQUIREMENT" '=' ("MANDATORY" | "OPTIONAL")

```

Each construct can also have a number of operation environments, each of which again has a number of attributes. An operation environment may be used, for example, to perform an operation (state-transition) on the state of a machine; the definition of such an operation environment is given below (figure 5.3).

```

...
( NAME= operation,
  REQUIREMENT= OPTIONAL
),
...

```

Figure 5.3: Configuration of an operation environment for machine operations

The name attribute allows the operation environment to be identified, and its value must be unique within a construct. It is necessary that a name is given to every operation environment used. In our example, we give the operation environment the name ‘operation’.

The requirement attribute again allows us to specify whether the operation environment is mandatory or optional. In the absence of this attribute, the operation environment is considered optional. In our example we set the attribute to optional as it is possible to have machines without operations.

The content of an operation environment is made up from clauses that belong specifically to that operation environment and provide information at

an operational level (the operation environment which a clause belongs to is specified by its 'LEVEL' attribute).

```

proof_obligations ::= "PROOF_OBLIGATIONS"
                    proof_obligation
                    (',' proof_obligation)*

proof_obligation ::= ("CONSTRUCT_LEVEL" | "OPERATION_LEVEL")
                    ',' '(' univQuant ')'

univQuant ::= ('!' name_list '@')? exQuant

exQuant ::= ('#' name_list '@')? implication

implication ::= disjunction ("=>" disjunction)*

disjunction ::= conjunction ('|' conjunction)*

conjunction ::= negation ('&' negation)*

negation ::= ("NOT")? equality

equality ::= membership ('=' equality)*

membership ::= (atom ':')? atom

atom ::= name
       | '<' name_list '>'
       | "FROM_MACHINE" '(' name ')'
       | "TO_MACHINE" '(' name ')'
       | "PRE" '('
         ( name | "FROM_MACHINE" '(' name ')'
           | "TO_MACHINE" '(' name ')' )
         ')'
       | '(' univQuant ')'

name_list ::= name (',' name)*

```

```

name ::= ('a'..'z' | 'A'..'Z' | '_' )+ ('\'' )?
       | ('a'..'z' | 'A'..'Z' | '_' )+ ('\'' )?
       "." ('a'..'z' | 'A'..'Z' | '_' )+ ('\'' )?

```

Finally, each construct will also have a number of proof obligations. For example, a machine may have a proof obligation to ensure that its initialization is valid. Such a proof obligation can be seen below (figure 5.4).

```

...
(  CONSTRUCT_LEVEL,
   (    # u @ u : state & u : initialization
     )
),
...

```

Figure 5.4: Configuration of proof obligation for machine initialization

The configuration for each proof obligation has two parts. The first specifies the level of the proof obligation; the level can either be construct level or operation level. Construct level proof obligations will be performed once for every construct. Operation level proof obligations will be performed once for every valid operation environment. As initialization occurs at the machine level, the proof obligation in our example is specified to be performed at the construct level.

The second part of the configuration is the proof obligation itself. We provide a basic syntax for creating theorems that can express the proof obligation. This syntax allows the building of proof obligations using universal and existential quantification, implication, conjunction, disjunction, negation, equality, set membership and relations produced from clauses and operation environments (more information on how this syntax is interpreted can be found in section 5.2.2). In our example, we state that there exists a state, 'u', where u is a member of the relation produced from the state clause, and also a member of the relation produced from the initialization clause. That is, there exists a state that is a valid state of the construct and that can also be reached through the initialization of the construct.

Note that, when we specify the proof obligations of a construct, we do so in a syntax closer to that of theorem provers than that of Z. We felt that using this syntax for our configurations would ensure that proof obligations had a consistent notation throughout the tool, reducing the need for translation and eliminating confusion that may occur through the use of different notations in different places.

5.2.2 Generating Proof Obligations

Having shown how it is possible to configure the generic proof obligations for a construct, we now describe how that configuration is used to create the specific proof obligations for an instance of that construct. This process involves a number of steps. Firstly, we produce an instance of each generic construct level proof obligation and one instance – per operation environment – for every operation level proof obligation. When generating proof obligations for relationships, we generate an instance of each generic operation level proof obligation for every matching pair of operations (a matching pair is considered to be one where the names of the operations are identical in both source and target machine). We then use the specification of the construct instance to create relations that replace the clause references in the generic proof obligations. If there are clauses that are used in the generic proof obligation, but are not defined in the specification of the construct instance, we must then eliminate the references to these clauses in our specific proof obligation. We illustrate this process through a series of examples.

Example 5.2

We consider first a simple machine configuration which defines two clauses. The first of these holds the state of the machine and the second the initialization of that state. There is one proof obligation which ensures that there exists a valid initialization of the machine. The Frog-CCL for such a configuration is given below.

```
DEFINE MACHINE machineWithInit
```

```

CLAUSES
  ( NAME= state,
    LEVEL= MACHINE,
    REQUIREMENT= OPTIONAL,
    CONTENT= SCHEMA_TEXT,
    RELATION= <state>
  ),
  ( NAME= initialization,
    LEVEL= MACHINE,
    REQUIREMENT= OPTIONAL,
    CONTENT= SCHEMA_TEXT,
    RELATION= <state>
  )
PROOF_OBLIGATIONS
  ( CONSTRUCT_LEVEL,
    ( # u @ u : state & u : initialization
    )
  )
END

```

We now define a machine, *myNumber*, that obeys this configuration and declares a single state variable.

```

MACHINE myNumber
TYPE machineWithInit
SECTION standard_toolkit
STATE

```

$a : \mathbb{N}$

```

INITIALIZATION

```

$a = 0$

END *myNumber*

We only have one proof obligation in our configuration, it is defined at the construct level and so we create a single generic proof obligations for our specification. This proof obligation is shown below.

$$\vdash? \exists u \bullet u \in \text{state} \wedge u \in \text{initialization}$$

As described above, each clause in the generic proof obligation is now substituted with its equivalent relation. Woodcock and Davies [WD96] describe how we can translate between the language of schemas and relations. The substitution process therefore, involves two steps: creating the schema from the clause, and creating the relation from the resultant schema.

Consider first the generation of the relation for the state clause of our example machine. We examine first the relation attribute of the clause to determine the shape of the relation we wish to produce. In this instance, we require the variables of the state clause to be included in our relation. Hence, the relation we require is of the form described below.

$$\{\text{state_schema} \bullet \theta \text{state_schema}\}$$

Where, the *state_schema* is the schema produced from the state clause. To create the schema that represents the state clause, we first need to examine the relation attribute for the clause. As the clause is self-referencing we do not need to import variables from any other clauses. We then determine that the content of the state clause is itself a schema. The schema for the state clause is as follows.

$$\text{state_schema} == [a : \mathbb{N}]$$

We can then substitute the definition of *state_schema* into our relation definition and resolve the binding construction expression, giving us a final relation as follows.

$$\{a : \mathbb{N} \bullet a\}$$

The final a is, of course, superfluous as the set comprehension is characteristic. We will, therefore, omit the final variable list in future examples. We can now substitute our relational definition of the state clause into our proof obligation, as shown below.

$$\vdash? \exists u \bullet u \in \{a : \mathbb{N}\} \wedge u \in \textit{initialization}$$

Consider now the generation of the relation for the initialization clause of the example machine. Again, we look first at the relation attribute and determine that the relation requires the variables of the state clause. The relation we require is described below.

$$\{\textit{initialization_schema} \bullet \theta \textit{state_schema}\}$$

Again, the *initialization_schema* is the schema produced from the initialization clause and the *state_schema* is the schema we generated previously. To create the schema for the initialization clause we once again reference its relation attribute, which states that we will import the variables of the state clause. Knowing that the content of the initialization clause is a schema we can conjoin the two schemas² to give the *initialization_schema* required, as shown below.

$$\textit{initialization_schema} == \textit{state_schema} \wedge [a = 0] == [a : \mathbb{N} \mid a = 0]$$

We can then substitute our schema definitions into our relation definition and resolve the binding construction, giving a final relation definition as follows.

$$\{a : \mathbb{N} \mid a = 0\}$$

²Clearly, this schema conjunction needs to be performed before we type check our schema. We include it at this point, in this example, to more simply illustrate the instantiation process. The actual point at which this conjunction is performed is described in section 5.3.3.

We can now create the required proof obligation for our specification by replacing the instance of the initialization clause with its relational definition, as shown below.

$$\vdash? \exists u \bullet u \in \{a : \mathbb{N}\} \wedge u \in \{a : \mathbb{N} \mid a = 0\}$$

Example 5.3

Consider now a machine containing a clause where the schema text of the clause declares local variables. Such a machine is described below (of course, there is no need for the local variable in this instance; however, a simple example allows the point to be illustrated equally well).

```
MACHINE myNewNumber
TYPE machineWithInit
SECTION standard_toolkit
STATE
```

$a : \mathbb{N}$

INITIALIZATION

$b : \mathbb{N}$

$a = b; b = 0$

```
END myNewNumber
```

The generation of the relation to represent the initialization clause follows a familiar pattern. As before, the relation we require is as follows.

$$\{initialization_schema \bullet \theta state_schema\}$$

The *state_schema* will be the same as in the previous example and the *initialization_schema* is given below.

$$\begin{aligned} \textit{initialization_schema} &== [\textit{state_schema} \wedge b : \mathbb{N} \mid a = b; b = 0] \\ &== [a : \mathbb{N}; b : \mathbb{N} \mid a = b; b = 0] \end{aligned}$$

We then substitute our schema definitions into our relation definition and resolve the binding construction, giving a final relation definition as follows.

$$\{a : \mathbb{N}, b : \mathbb{N} \mid a = b; b = 0 \bullet a\}$$

The only difference is the necessity to specify the variables required in the final relation. The definition of local variables requires no special consideration. The final proof obligation for the machine is shown below.

$$\vdash? \exists u \bullet u \in \{a : \mathbb{N}\} \wedge u \in \{a : \mathbb{N}, b : \mathbb{N} \mid a = b; b = 0 \bullet a\}$$

Example 5.4

We consider now a more complex machine configuration that defines an operation environment which can include two clauses used to define the inputs to, and body of, an operation. We retain the construct level state clause of the previous example and define an operation level proof obligation. The Frog-CCL for such a configuration is given below.

```

DEFINE MACHINE machineWithOp
  CLAUSES
    (
      NAME= state,
      LEVEL= MACHINE,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state>
    ),
    (
      NAME= inputs,
      LEVEL= operation,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <inputs>
  )

```

```

    ),
    ( NAME= body,
      LEVEL= operation,
      REQUIREMENT= MANDATORY,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state,state',inputs>
    )
  )
OPERATION_ENVIRONMENTS
  ( NAME= operation,
    REQUIREMENT= OPTIONAL
  )
PROOF_OBLIGATIONS
  ( OPERATION_LEVEL,
    ( # u,u',i
      @ u : state & u' : state
      & i : operation.inputs
      & <u,u',i> : operation.body )
    )
  )
)
END

```

We now define a machine that conforms to this configuration. This machine declares a single state variable and an operation that can be used to increment this variable. The specification of this machine is given below.

```

MACHINE myNewestNumber
TYPE machineWithOp
SECTION standard_toolkit
STATE

```

```

  a : N

```

OPERATION *increment* $\hat{=}$
 BODY

$$\frac{}{a' = a + 1}$$

END *increment*

END *myNewestNumber*

Again, as we only have one operation environment in our machine specification we create only a single generic proof obligation which is described below.

$$\vdash? \exists u, u', i \bullet u \in \text{state} \wedge u' \in \text{state} \wedge i \in \text{operation.inputs} \\ \wedge (u, u', i) \in \text{operation.body}$$

Using the process described in the previous examples we generate the relational definitions of the clauses from our proof obligation. These definitions are given below.

$$\text{state} == \{a : \mathbb{N}\} \\ \text{operation.body} == \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\}$$

When we come to replace the *operation.inputs* clause, we notice that no *inputs* clause has been defined in the *increment* operation environment. We therefore need to eliminate the use of this clause in our proof obligation. Having replaced the defined clauses with the relational definitions, our current proof obligation is as follows.

$$\vdash? \exists u, u', i \bullet u \in \{a : \mathbb{N}\} \wedge u' \in \{a : \mathbb{N}\} \wedge i \in \text{operation.inputs} \\ \wedge (u, u', i) \in \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\}$$

The first stage in the elimination of the unused clause is the replacement of references to that clause – in the proof obligation – with the special term *undefined*. This term simply serves as a place marker to aid the removal of

references to the unused term and will never remain in a proof obligation beyond the clause elimination stage. This replacement has two steps: the first is to replace direct references to this clause (in this case *operation.inputs*) with the undefined term. In our example this results in the following proof obligation.

$$\begin{aligned} \vdash? \exists u, u', i \bullet u \in \{a : \mathbb{N}\} \wedge u' \in \{a : \mathbb{N}\} \wedge i \in \text{undefined} \\ \wedge (u, u', i) \in \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\} \end{aligned}$$

We then examine every membership predicate in our proof obligation. If the second argument to the membership operator is a clause, we then re-examine that clause's relation attribute. If the relation for the clause contains a reference to the undefined clause, we determine the position in the sequence produced from the relation of that undefined clause. We then use that position to replace the corresponding term in the first argument of the membership operator with the *undefined* term. If the replaced term is a bound variable, then all occurrences of that bound variable (within its scope) will also be replaced with the *undefined* term. In our example this results in the following proof obligation.

$$\begin{aligned} \vdash? \exists u, u', i \bullet u \in \{a : \mathbb{N}\} \wedge u' \in \{a : \mathbb{N}\} \wedge \text{undefined} \in \text{undefined} \\ \wedge (u, u', \text{undefined}) \in \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\} \end{aligned}$$

Now that we have removed all references to undefined clauses, our next step is to eliminate all instances of the *undefined* term. This is done by repeatedly applying the rules defined in table 5.1 on the next page until there are no remaining uses of the *undefined* term within our proof obligation.

The first application of these rules to the proof obligation of our example, results in the following proof obligation.

$$\begin{aligned} \vdash? \exists u, u', i \bullet u \in \{a : \mathbb{N}\} \wedge u' \in \{a : \mathbb{N}\} \wedge \text{undefined} \\ \wedge (u, u') \in \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\} \end{aligned}$$

A reference to the *undefined* term remains in our proof obligation. It is therefore necessary to apply our rules to the proof obligation once again.

Table 5.1: Logical rules for handling undefined clauses

Original term	Resolved term
$x \in \text{undefined}$	undefined
$P \wedge \text{undefined}$	P
$P \vee \text{undefined}$	P
$\forall x \bullet \text{undefined}$	undefined
$\exists x \bullet \text{undefined}$	undefined
$\forall n_1, \dots, n_{i-1}, \text{undefined}, n_{i+1}, \dots, n_j \bullet P$	$\forall n_1, \dots, n_{i-1}, n_{i+1}, \dots, n_j \bullet P$
$\exists n_1, \dots, n_{i-1}, \text{undefined}, n_{i+1}, \dots, n_j \bullet P$	$\exists n_1, \dots, n_{i-1}, n_{i+1}, \dots, n_j \bullet P$
$P \Rightarrow \text{undefined}$	P
$\text{undefined} \Rightarrow P$	P
$x = \text{undefined}$	true
$(n_1, \dots, n_{i-1}, \text{undefined}, n_{i+1}, \dots, n_j)$	$(n_1, \dots, n_{i-1}, n_{i+1}, \dots, n_j)$

This process produces the following proof obligation.

$$\begin{aligned} \vdash? \exists u, u', i \bullet u \in \{a : \mathbb{N}\} \wedge u' \in \{a : \mathbb{N}\} \\ \wedge (u, u') \in \{a : \mathbb{N}; a' : \mathbb{N} \mid a' = a + 1\} \end{aligned}$$

It can be now seen that we have successfully eliminated the references to the operation.inputs clause, and that our final proof obligation is the one required in our configuration.

5.3 Incorporating Constructs within a Frog Specification

In this section we first describe the extension to the Z syntax required in order to incorporate constructs within a Frog specification. We then consider the parsing of this extension, before finally considering the way in which this extended syntax can be transformed to provide a suitable input to the existing Z type checker.

5.3.1 Extending the Z Syntax

In order to allow our constructs to be parsed within our existing Z framework we extend the \LaTeX syntax presented in [ISO02] to allow the specification of machines and relationships. An annotated EBNF grammar for this extension is given below.

```
file ::= ( section | paragraph )+
      | modelling_spec
```

```
modelling_spec ::= machine | relationship
```

A Frog file contains either Z sections and paragraphs or a modelling specification. A file containing a modelling specification contains a single construct, either a machine or a relationship. A Frog file containing a construct will be considered to belong to a specification in the same way as a file containing sections or paragraphs.

```
machine ::= "\begin{machine}"
          '{' NAME '}'
          '{' NAME '}'
          ( '{' NAME (',' NAME)* '}' )?
          ( clause
            | operation_env
          )+
          "\end{machine}"
```

A machine specification consists of a starting token followed by the declaration of the machine's name, the machine's type and then, optionally, the specification of a number of Z sections that become the parents of the machine. One or more clauses or operation environments are then specified until an ending token is declared.

```
relationship ::= "\begin{relationship}"
               '{' NAME '}'
               '{' NAME '}'
               '{' NAME ( "\AS" NAME )? '}'
               '{' NAME ( "\AS" NAME )? '}'
               ( clause
                 | operation_env
               )+
               "\end{relationship}"
```

A relationship specification consists of a starting token followed by the declaration of four names giving the name of the relationship, the type of the relationship and the names of source and target machine respectively (we can optionally define an alias for our source and target machines to make referring to their variables simpler). One or more clauses or operation environments are then specified until an ending token is declared.

```
clause ::= "\clause"
          '{' NAME '}'
          '{' ( predicate | schema_text ) '}'
```

A clause simply consists of the clause token followed by the declaration of the clauses type, and then its definition which takes the form of either a predicate or a schema text (both of which are defined in the existing Z grammar).

```
operation_env ::= "\begin{openv}"
                '{' NAME '}'
                '{' NAME '}'
                ( clause )+
                "\end{openv}"
```

The specification of an operation environment consists of a starting token followed by the declaration of two names giving, firstly, the name for the operation environment, and secondly, its type. These names are followed by the definition of one or more clauses. Finally an ending token is declared.

The actual parsing of this grammar requires the use of the Frog-CCL definition for the construct specified.

In addition to the extension we have presented for incorporating Frog constructs within a *Z* specification, we now introduce a new symbol that allows us to disambiguate between conflicting variables when specifying relationships. As a relationship construct typically incorporates the variables belonging to machines it is necessary that we are able to distinguish between those using the same identifiers. The ‘ \gg ’ symbol is used to relate a variable with the machine to which it belongs.

Example 5.5

In this example we illustrate the use of the ‘ \gg ’ notation. Consider first two machines that each define a single variable, *a*. We may define such machines as follows.

```
MACHINE myFirstMachine
TYPE machine
SECTION standard_toolkit
STATE
```

a : \mathbb{N}

```
END myFirstMachine
```

```
MACHINE mySecondMachine
TYPE machine
SECTION standard_toolkit
STATE
```

a : \mathbb{N}

END *mySecondMachine*

Consider next the necessity to specify a relationship between those machines. We may wish to specify that the a of one machine should always be three less than that of the other. Without some form of explicit disambiguation we would be unable to indicate which a should be greater. When using the ‘ \gg ’ notation, this relationship can be expressed easily, and unambiguously, as follows.

```
RELATIONSHIP myRelationship
TYPE relationship
FROM myFirstMachine
TO mySecondMachine
MYCLAUSE
```

$$\mathit{myFirstMachine} \gg a = \mathit{mySecondMachine} \gg a + 3$$

END *myRelationship*

5.3.2 Parsing the Extended Grammar

When parsing a construct we need to ensure that as well as being grammatically correct, the construct obeys the rules created in the configuration of its type. These rules determine how a construct is parsed – as well as how the parsed data is interpreted – so it is necessary to process them dynamically.

When a construct is parsed therefore, the first thing that must be determined is the type of the construct. For example we may have a machine or a retrenchment relationship. We know whether a construct is a machine or relationship through the syntax used; the type of machine or relationship is determined by taking the name declared in the construct’s specification and searching for a match in the available construct configurations. If no match is found, the construct will be rejected, otherwise, the rules from the

matched configuration are used for the parsing.

Assuming a valid configuration has been found, a Z section is then created and associated with the construct. (The name of the Z section created will be the name of the construct with either ‘_machine_’ or ‘_relationship_’ prefixed.³) If a machine’s specification includes a parent section it will be declared as the parent of the newly created section; if no parent section is specified then the standard toolkit will be used as a parent. The section created for a relationship will have two parent sections, the sections of the source and target machines (if these sections and hence their corresponding constructs are missing, then the relationship cannot be parsed successfully). The declaration of these parent sections allows their variables to be referred to within the clauses of a construct.

We consider now the parsing of the construct’s clauses. Each clause again has a type and the configuration for the clause is retrieved from the construct’s configuration using the name declared. The clause’s configuration determines the level of the clause and its content. The level of the clause indicates whether it can be used at the top (construct) level or only within a specific operation environment. Should the clause be used at a level contradicting the level specified in the configuration it will not be possible to parse the construct successfully. The clause’s content is also indicated and declares whether the clause should contain a predicate or a schema text. While the grammar above gives the impression that the parser could attempt to match both a schema text and a predicate, in practice the use of ANTLR’s semantic predicates means that the parser will only ever try and match the type of content specified in the clause’s configuration.

Finally, we consider the parsing of the operation environments. In a machine an operation environment is typically used to describe the behaviour of an operation, and in a relationship is used to describe the link between the operations of the source and target machines. Again, each operation environment has a type, and the configuration for the operation is determined from the construct’s configuration using the name declared. It is necessary

³Whilst the use of an initial underscore character is technically illegal in the Z notation, we will use it internally within our system to prevent unintentional name clashes.

to ensure that any clauses that are declared to belong to an operation environment, and that have been declared mandatory, are present when parsing the operation environment.

Before a construct is said to have been parsed successfully, it is also necessary to ensure that all mandatory clauses and operation environments are present in the specification (that is those clauses and operation environments declared as mandatory in the construct configuration).

5.3.3 Syntax Transformation for the Extended Grammar

Once our construct has been parsed successfully it is necessary to transform it into a format suitable for type checking and the generation of proof obligations. This stage involves a process similar to that described in 4.6.5, which transforms our construct definition into a specification that matches the grammar for the annotated Z syntax described in chapter 10 of [ISO02].

As described above, every construct is translated into an equivalent Z section and every clause into an equivalent Z paragraph. We illustrate the transformation process through a simple example.

Example 5.6

The first step is to configure a simple machine. We define a single construct level clause and an operation environment which may contain an operation level clause. The configuration for this machine can be seen below.

```

DEFINE MACHINE transformMachine
  CLAUSES
    (
      NAME= constructLevelClauseSchema,
      LEVEL= MACHINE,
      CONTENT= SCHEMA_TEXT,
      RELATION= <constructLevelClauseSchema>
    ),
    (
      NAME= constructLevelClausePredicate,

```

```

        LEVEL= MACHINE,
        CONTENT= PREDICATE,
        RELATION= <constructLevelClauseSchema>
    ),
    ( NAME= operationLevelClause,
      LEVEL= operationEnvType,
      CONTENT= SCHEMA_TEXT,
      RELATION= <constructLevelClauseSchema,
                constructLevelClauseSchema'>
    )
OPERATION_ENVIRONMENTS
    ( NAME= operationEnvType,
      REQUIREMENT= OPTIONAL
    )
END

```

In this configuration we are defining three clauses: *constructLevelClauseSchema* that can be used at the construct level and will contain a schema text, *constructLevelClausePredicate* that can be used at the construct level and will contain a predicate and *operationLevelClause* that is used at the operation level and will contain a schema text. In this instance, the names of the clauses are intended to represent their use. Clearly, in a more practical example the names should reflect the intended content rather than the nature of the configuration.

The relation attribute of each clause details the shape of the relation produced from the clause, but also the other clauses to whose variables this clause has access. To show how the different configurations of this attribute are used in a transformation we include three different types of reference. The clause *constructLevelClauseSchema* has a relation attribute that is self-referencing. The clause *constructLevelClausePredicate* references the *constructLevelClauseSchema* clause and so will have access to the variables defined within that clause. The clause *operationLevelClause* references

the *constructLevelClauseSchema* clause and also its primed version; this ensures that the clause will have access to two sets of the variables defined within the referenced clause and could, for example, use these to represent pre-transitional and post-transitional states.

We now specify a simple machine, *myMachine* that conforms to this configuration (that is the machine-type, *transformMachine*).

```

MACHINE myMachine
TYPE transformMachine
SECTION parentSection
CONSTRUCTLEVELCLAUSESHEMA

    schemaForConstructLevelClauseSchema

CONSTRUCTLEVELCLAUSEPREDICATE

    constantForConstructLevelClausePredicate

OPERATIONENVTYPE operationEnvName  $\hat{=}$ 
OPERATIONLEVELCLAUSE

    schemaForOperationLevelClauseSchema

END operationEnvName

END myMachine

```

We assume that the schemas, *schemaForConstructLevelClauseSchema* and *schemaForOperationLevelClauseSchema*, that we have referenced in the specification of *myMachine* are valid and defined within the parent section. Similarly, we assume that we have declared a constant, *constantForConstructLevelClausePredicate*, in the machine's parent section that represents a valid predicate.

We now examine how this machine specification is transformed into the Z annotated syntax. The first stage involves the creation of a Z section. This section will contain all of the information specific to a construct, and is considered to be semantically equivalent to the construct definition. The

section created for our example is shown below.

```
section _machine_myMachine parents parentSection
```

The name of the new section is the name of the construct with ‘_machine_’ or ‘_relationship_’ prefixed to avoid name clashes with existing sections. The parent section of our new section is the parent declared in the construct’s definition, in this case *parentSection*.

We now examine the clauses of our example. The first of these is configured to be a construct level clause that contains a schema text (*constructLevelClauseSchema*). The paragraph representing this clause is shown below.

```
_myMachine_constructLevelClauseSchema  
schemaForConstructLevelClauseSchema
```

The schema definition paragraph for a construct level clause is given a name equal to the clause name prefixed with an underscore and the name of the machine. The configuration for the specified clause is then examined to determine how the transformation proceeds. The relation attribute of the clause is examined first to determine which other clauses are referenced. In our example, the only clause in the relation is a self-reference (which we ignore for purposes of variable scope), and therefore, we have no variables to import. We then examine the content attribute of the clause. In this instance, the clause is a schema text. This schema text (*schemaForConstructLevelClauseSchema*) has its syntax transformed in the same way as all other *Z* schema texts. The body of our new paragraph therefore becomes simply the schema text produced by the transformation of the clause’s definition.

The second clause is again a construct level clause, but this time contains a predicate. The paragraph that should be generated to represent this clause is shown below.

$_myMachine_constructLevelClausePredicate$
$_myMachine_constructLevelClauseSchema$
$constantForConstructLevelClausePredicate$

Again, the paragraph name is derived from the clause name, and the clause's configuration is examined. The next step is to examine the relation attribute, in this instance we have a reference to our previous clause. It is necessary therefore, to include a reference to the $_myMachine_constructLevelClauseSchema$ schema in our paragraph (as schema names can be derived solely from the clause name, the order of declaration of the clauses is unimportant). We examine next the content attribute of our configuration, in this clause the content is set to be a predicate. The definition of our predicate ($constructLevelPredicateClauseDefinition$) is transformed in the normal manner, using the syntax transformation rules outlined in [ISO02]. As the body of a schema definition paragraph must be a schema text, we integrate our transformed predicate definition with the referenced schema forming the declaration into a schema text expression (a predicate without a referenced schema is possible, but without any declared variables its use would be limited).

The final clause is defined within an operation environment. The paragraph that is generated to represent this clause is shown below.

$_myMachine_operationEnvName_operationEnvLevelClause$
$_myMachine_constructLevelClause$
$\wedge _myMachine_constructLevelClause'$
$\wedge schemaForOperationLevelClauseSchema$

The only difference between transforming a construct level and operation level clause is in the name of the paragraph produced. An operation level clause will have both the name of the machine, and the name of the operation environment to which it belongs, prefixed to its name in order

to produce the name of its Z paragraph. We then examine the relation attribute of the clause used. In this instance, we have a reference to the pre-transition and post-transition state of our first clause (*constructLevelClause*). It is necessary therefore, to include a reference to both the primed and unprimed *_myMachine_constructLevelClauseSchema* schema in our paragraph. The content of our clause is configured to be a schema text, and following the standard transformation of the schema text, we can incorporate it directly into our paragraph. We have more than one schema in the body of our paragraph, so we must use schema conjunction to join these producing the one schema text that the schema definition paragraph requires.

We have shown how the principal combinations of configurations can be used to transform a construct into the Z annotated syntax specified in [ISO02]. There are of course many different ways in which a construct could be configured, but as all of these could be transformed with the rules specified above, we do not describe these further.

5.4 Configuring and Specifying a Machine

The first thing that must be done is to configure a machine. We define a number of machines using Frog-CCL in the examples below.

We require a simple machine that allows us to specify a state, specify how that state is initialized and specify the transitions (operations) that can be performed on that state. We also require that it is possible to ensure the internal consistency of our machine. We will take this opportunity to exhibit the flexibility of Frog-CCL to show how this could be achieved using a number of alternate configurations.

Example 5.7

The configuration for the first of our alternative approaches is shown below.

```
DEFINE MACHINE machineA
  CLAUSES
    ( NAME= state,
```

```

    LEVEL= MACHINE,
    REQUIREMENT= OPTIONAL,
    CONTENT= SCHEMA_TEXT,
    RELATION= <state>
),
( NAME= initialization,
  LEVEL= MACHINE,
  REQUIREMENT= OPTIONAL,
  CONTENT= SCHEMA_TEXT,
  RELATION= <state>
),
( NAME= inputs,
  LEVEL= operation,
  REQUIREMENT= OPTIONAL,
  CONTENT= SCHEMA_TEXT,
  RELATION= <inputs>
),
( NAME= outputs,
  LEVEL= operation,
  REQUIREMENT= OPTIONAL,
  CONTENT= SCHEMA_TEXT,
  RELATION= <outputs>
),
( NAME= body,
  LEVEL= operation,
  REQUIREMENT= MANDATORY,
  CONTENT= SCHEMA_TEXT,
  RELATION= <state,state',inputs,outputs>,
  PRE_RELATION= <state,inputs>
)
OPERATION_ENVIRONMENTS
( NAME= operation,
  REQUIREMENT= OPTIONAL

```

```

    )
  PROOF_OBLIGATIONS
    ( CONSTRUCT_LEVEL,
      ( # u @ u : state & u : initialization
        )
    ),
    ( OPERATION_LEVEL,
      ( # u,u',i,o
        @ u : state & i : operation.inputs
        & u' : state & o : operation.outputs
        & <u,u',i,o> : operation.body
      )
    )
  )
END

```

We define our machine as being of machine-type *machineA*, in order to distinguish it from the machine-types we will present in the subsequent examples. In practice, the machine-type would be allocated a name that would better reflect the reasoning behind its configuration (and may typically be made based on the particular application).

We define two clauses at the construct level a state clause and an initialization clauses, both clauses are optional as it is not necessary for every machine to have a state. Both construct level clauses are also declared to contain schema texts. In the case of the state clause this allows us to specify our state variables in the declaration part and any restrictions (invariant) on those variables in the predicate part. With the initialization clause we use a schema text as it allows us to declare additional variables that may be useful in the specification of particularly complex initializations. The relation for both state and initialization clause contains a reference to the state clause. It should be noted that it is possible for a clause (in this instance the state clause) to refer to itself in its relation attribute.

Whilst using a self-reference, such as that described above, is permissible in Frog-CCL, tools that parse Frog-CCL need to be aware that when

considering a relation attribute that contains such a reference (directly or indirectly), an attempt should not be made to include a reference to its own schema when transforming into Z syntax. However, when generating a proof obligation for that clause such tools need to produce a relation that contains the clause's own variables.

Our machine contains three further clauses, all of which have been declared at the operation level – specifically at the level of the operation environment named ‘operation’. This operation environment is used to encapsulate the specification of a state transition, and consists of three clauses defining the operation inputs, outputs and its body. The inputs and outputs clauses are both defined as optional and as containing a schema text (which allows a specifier to declare the input and output variables plus any restrictions); the relations for both clauses are simply self-references. The body clause of the operation is more interesting as it is here where the transition itself is specified. The body clause is mandatory as an operation would not make sense without it. Again the body clause is specified to contain a schema text, allowing the declaration of local variables where necessary. The relation clause indicates that the pre-transitional and post-transitional values of the state variables, the input variables, and the output variables are made available for use in the operation's body clause. The ‘pre’ relation clause shows the variables that are used in the declaration of the clause's precondition; in this instance we have – as one would expect –the pre-transitional state variables and the input variables⁴.

Our machine is configured to have two proof obligations, one at the construct level and one at the operation level. The first of these is the initialization proof obligation and states that there must exist a valid state (u), that is also a member of the relation produced from the initialization clause. That is, there exists at least one state in which the machine can be correctly initialized.

⁴This relation is required as Frog makes no assumptions that the standard Z strokes are used to indicate operation inputs or outputs or to indicate post-transitional values of variables.

The second of these is the operation proof obligation and as it is at operation level will be performed for every valid operation environment. This operation proof obligation states that there exists a valid pre-transitional state (u), a valid post-transitional state (u'), a sequence of valid inputs (i) and a sequence of valid outputs (o), such that the sequence (u, u', i, o) is a member of the relation produced from the operation's body clause. That is, there exists at least one pre-transitional state and set of inputs, and one post-transitional state and set of outputs, that can be related through a transition of the operation.

We consider now the specification of a machine using this configuration, and return to the example that we first discussed in example 2.1 on page 47. In that example we considered the modelling of a train yard. The machine below specifies the behaviour described in our previous example (for conciseness we will look solely at the *trainEntersYard* operation).

MACHINE *trainYard*
 TYPE *machineA*
 SECTION *theTrainYard*
 STATE

$trainsInYard : \mathbb{P} \textit{trains}$

INITIALIZATION

$trainsInYard = \emptyset$

OPERATION *trainEntersYard* $\hat{=}$
 INPUTS

$trainEnteringYard? : \textit{trains}$

BODY

$trainEnteringYard? \notin trainsInYard$
 $trainsInYard' = trainsInYard \cup \{trainEnteringYard?\}$

END *trainEntersYard*

END *trainYard*

There are some noticeable differences to the way in which our model is presented. Firstly, it is not possible for us to declare the *trains* type directly in the machine, and we therefore need to set a parent section for a machine (which we specify in the ‘section’ clause). In this instance we use the *theTrainYard* section we defined in example 2.1. This may be seen as a failure of our construct based system, but we feel that the declaration of given types and operators should be performed outside constructs, as this will lead to greater modularity, and hence, reusability. The second difference is the lack of need to refer to the state of model when specifying our initialization clause and our operation. For example, the previous initialization of our model is shown below; it can be seen that it was necessary to refer to the *trainYard* schema in our definition. In our machine however, this is no longer necessary as the configuration of the relation attribute for the clause ensures that the variables of the state are already available to us.

<i>Init_trainYard</i>
<i>trainYard</i>
<i>trainsInYard</i> = \emptyset

Our example machine can be specified in the extended Z \LaTeX notation as shown below ⁵.

```
\begin{machine}{trainYard}{machine}{theTrainYard}
\clause{state}{
  trainsInYard : \power trains
}
\clause{initialization}{
```

⁵In fact, when processing the Z \LaTeX definition shown above (and using Frog’s \LaTeX macros), the machine will be formatted exactly as shown.

```

    \where
    trainsInYard = \emptyset
}

\begin{openv}{operation}{trainEntersYard}
\clause{inputs}{
    trainEnteringYard? : trains
}
\clause{body}{
    \where
    trainEnteringYard? \notin trainsInYard \\\
    trainsInYard' = trainsInYard \cup
                    \{ trainEnteringYard? \}
}
\end{openv}

\end{machine}

```

Consider now the generation of proof obligations for our machine. Firstly, we examine the generation of the initialization proof obligation. We begin with the proof obligation specified in our configuration (shown below).

$$\vdash? \exists u \bullet u \in \textit{state} \wedge u \in \textit{initialization}$$

We now apply the process described in 5.2.2 to replace the clause names with their respective relations. This gives us the following.

$$\begin{aligned} \vdash? \exists u \bullet u \in \{ \textit{trainsInYard} : \mathbb{P} \textit{trains} \mid \textit{true} \} \\ \wedge u \in \{ \textit{trainsInYard} : \mathbb{P} \textit{trains} \mid \textit{trainsInYard} = \emptyset \} \end{aligned}$$

This can clearly be seen to be the correct initialization proof obligation, and also through simple observation can be seen to be true.

We consider now the generation of the operation proof obligations. As we only have one operation in our example we will create only a single proof obligation and again we begin with the proof obligation specified in our

configuration.

$$\begin{aligned} \vdash? \exists u, u', i, o \bullet & u \in \text{state} \wedge u' \in \text{state} \\ & \wedge i \in \text{operation.inputs} \wedge o \in \text{operation.outputs} \\ & \wedge (u, u', i, o) \in \text{operation.body} \end{aligned}$$

As no output clause has been specified it can be omitted from our proof obligation (the methods used to determine which clauses are present, and therefore form part of our proof obligation, have been discussed above in example 5.4). Again, we can generate the appropriate relations from the configuration of our clauses and our specification, giving us a proof obligation as follows.

$$\begin{aligned} \vdash? \exists u, i, u' \\ \bullet & u \in \{\text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{true}\} \\ & \wedge u' \in \{\text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{true}\} \\ & \wedge i \in \{\text{trainEnteringYard?} : \text{trains} \mid \text{true}\} \\ & \wedge (u, u', i) \in \{\text{trainsInYard}, \text{trainsInYard}' : \mathbb{P} \text{trains}; \\ & \quad \text{trainEnteringYard?} : \text{trains} \\ & \quad \mid \text{trainEnteringYard?} \notin \text{trainsInYard} \\ & \quad \wedge \text{trainsInYard}' = \text{trainsInYard} \\ & \quad \cup \{\text{trainEnteringYard?}\}\} \end{aligned}$$

We will not describe how this proof obligation can be discharged as, again, the proof can be performed with relative ease and we merely wished to demonstrate how the generation of machine proof obligations could be automated from a machine configuration and specification.

However, when we come to examine our proof obligation, we realize it has a fatal flaw. Whilst it guarantees that there exists one state that satisfies the precondition and leads to a valid transition, it does not verify that, whenever the precondition is met, a defined transaction occurs.

Example 5.8

The proof obligation in our previous example did not establish the internal consistency of a machine, the problem being that we did not distinguish

between the part of our operation that expressed the precondition and the part that expressed its behaviour. A traditional method of defining the set of pre-transitional states and inputs that satisfy an operation's preconditions is to use Z's 'pre' operator. This operator will produce one schema from another where the generated schema describes the set of states for which the operation – described by the other schema – is defined. Frog-CCL allows us to use this operator directly in our construct's proof obligations. For example, we can replace the second proof obligation of the configuration in the last example with the following definition (giving a configuration with machine-type, *machineB*).

```
( OPERATION_LEVEL,
  ( # u,i
    @ u : state & i : operation.inputs
      & <u,i> : PRE(operation.body)
    )
  )
)
```

This proof obligation states that for each operation there is a pre-transitional state (u) and sequence of valid inputs (i) such that their pair (u, i) is a member of the relation generated from the schema that is produced by applying Z's 'pre' operator to the schema associated with the 'body' clause of the operation. That is, there exists at least one state and set of inputs that satisfies the precondition of the operation. The way we have derived the precondition guarantees that all such state, input pairs that satisfy the precondition will lead to valid transitions.

We have shown Woodcock and Davies's method [WD96] for determining the precondition using 'pre' in section 2.3. This method is automatable and can be performed mechanically by a tool.

We now return to the specification for the train yard. As the structure of the machine is configured to be the same as in the previous example, we will use exactly the same specification. Therefore, we can progress immediately to the generation of the operation proof obligations (the initialization proof obligation will be identical). We only have one operation and therefore have

only a single proof obligation. The creation of the proof obligation begins with the generic obligation detailed in our configuration.

$$\vdash? \exists u, i \bullet u \in \text{state} \wedge i \in \text{operation.inputs} \\ \wedge (u, i) \in \text{pre operation.body}$$

We then need to calculate the precondition for the schema produced from the operation's 'body' clause. This produces the following.

$\text{pre_trainYard_operation_trainEntersYard}$
$\text{trainsInYard} : \mathbb{P} \text{trains}$
$\text{trainEnteringYard?} : \text{trains}$
$\text{trainEnteringYard?} \notin \text{trainsInYard}$

We can instantiate the proof obligation by generating the appropriate relations from the configuration of our clauses, our specification, and the precondition schema above (note that the shape of the relation produced from the precondition schema is derived from the pre-relation attribute specified in the 'body' clause's configuration), giving us a proof obligation as follows.

$$\vdash? \exists u, i \\ \bullet u \in \{\text{trainsInYard} : \mathbb{P} \text{trains} \mid \text{true}\} \\ \wedge i \in \{\text{trainEnteringYard?} : \text{trains} \mid \text{true}\} \\ \wedge (u, i) \in \{\text{trainsInYard} : \mathbb{P} \text{trains}; \\ \text{trainEnteringYard?} : \text{trains} \\ \mid \text{trainEnteringYard?} \notin \text{trainsInYard}\}$$

Again this proof obligation can be easily discharged and we will not present the details here. Obviously, this proof obligation is different to that specified in the previous example, but they both provide the same verification that there exists a state and set of inputs from which a transition of the operation can be performed. Here however, whilst we do not explicitly refer to the post-transitional state or the outputs of the operation in this instance, the calculated precondition ensures that all of the pre-transitional states and

inputs allowed, are those for which the operation's behaviour is defined.

Example 5.9

Some may argue, however, that as when we configure a construct we know its purpose, we can move away from the generic schemas of *Z* and define clauses that more exactly meet our needs when defining a machine. Such a configuration (with machine-type *machineC*) may be seen below.

```
DEFINE MACHINE machineC
  CLAUSES
    ( NAME= state,
      LEVEL= MACHINE,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state>
    ),
    ( NAME= initialization,
      LEVEL= MACHINE,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state>
    ),
    ( NAME= inputs,
      LEVEL= operation,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <inputs>
    ),
    ( NAME= outputs,
      LEVEL= operation,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <outputs>
```

```

    ),
    ( NAME= pre,
      LEVEL= operation,
      REQUIREMENT= OPTIONAL,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state,inputs>
    ),
    ( NAME= post,
      LEVEL= operation,
      REQUIREMENT= MANDATORY,
      CONTENT= SCHEMA_TEXT,
      RELATION= <state,state',inputs,outputs>
    )
OPERATION_ENVIRONMENTS
    ( NAME= operation,
      REQUIREMENT= OPTIONAL
    )
PROOF_OBLIGATIONS
    ( CONSTRUCT_LEVEL,
      ( # u @ u : state & u : initialization
      )
    ),
    ( OPERATION_LEVEL,
      ( # u,i
        @ u : state & i : operation.inputs
        & <u,i> : operation.pre
      )
    ),
    ( OPERATION_LEVEL,
      ( ! u,i
        @ u : state & i : operation.inputs
        & <u,i> : operation.pre
        =>
      )
    )

```

```

        ( # u',o
          @ u' : state & o : operation.outputs
            & <u,u',i,o> : operation.post
        )
    )
)
END

```

This configuration forces a user to split their operation specification into the parts that deal with the precondition of the operation, and those that deal with the postcondition. This approach is familiar to users of other construct-based terminologies (for example, B and VDM).

We replace the ‘body’ clause with an optional precondition clause – where the user must specify any precondition to their operation – and a mandatory postcondition clause where the user must indicate the post-transitional values of any state variables and outputs. The proof obligations for operations reflect these changes, and we now have two operation level proof obligations. The first of these is basically the same as that presented in the previous example except that the precondition is specified by the user rather than derived. The second states that for all pre-transitional states (u) and sets of inputs (i) whose pair belongs to the relation generated from the ‘pre’ clause, there exists a post-transitional state (u') and set of outputs o , such that (u, u', i, o) belongs to the relation generated from the ‘post’ clause. That is, if a state and set of inputs satisfy the operation’s precondition, then it must be possible to take a step of the operation that results in a valid state and produces correct outputs.

We have a pair of proof obligations therefore, that allow us to claim that there exists a state and set of inputs that meets the precondition and that whenever the precondition is met there is valid post-transitional state and set of outputs produced.

Using this configuration we would need a slightly different machine specification which we present below.

MACHINE *trainYard*
 TYPE *machineC*
 SECTION *theTrainYard*
 STATE

$trainsInYard : \mathbb{P} \textit{trains}$

INITIALIZATION

$trainsInYard = \emptyset$

OPERATION *trainEntersYard* $\hat{=}$

INPUTS

$trainEnteringYard? : \textit{trains}$

PRE

$trainEnteringYard? \notin trainsInYard$

POST

$trainsInYard' = trainsInYard \cup \{trainEnteringYard?\}$

END *trainEntersYard*

END *trainYard*

We have made no change to the initialization proof obligation and the instantiation remains identical to the previous examples. Similarly, the first operation level proof obligation results in an almost identically instantiated obligation to that created in our last example. The second operation proof obligation however, requires reexamination. Once again, we generate a single proof obligation for our *trainEntersYard* operation. The generic definition for which is shown below.

$$\begin{aligned}
& \vdash? \forall u, i \bullet u \in \text{state} \wedge i \in \text{operation.inputs} \wedge (u, i) \in \text{operation.pre} \\
& \quad \Rightarrow \exists u', o \bullet u' \in \text{state} \wedge o \in \text{operation.outputs} \\
& \quad \quad \wedge (u, u', i, o) \in \text{operation.post}
\end{aligned}$$

The ‘outputs’ clause is not used, so we can eliminate it using the aforementioned method. The remainder of the obligation can be instantiated using the relations derived from our specification to give the following.

$$\begin{aligned}
& \vdash? \forall u, i \\
& \quad \bullet u \in \{\text{trainsInYard} \in \mathbb{P} \text{trains} \mid \text{true}\} \\
& \quad \quad \wedge i \in \{\text{trainEnteringYard?} \in \text{trains} \mid \text{true}\} \\
& \quad \quad \wedge (u, i) \in \{\text{trainsInYard} \in \mathbb{P} \text{trains} \\
& \quad \quad \quad \wedge \text{trainEnteringYard?} \in \text{trains} \\
& \quad \quad \quad \mid \text{trainEnteringYard?} \notin \text{trainsInYard}\} \\
& \quad \Rightarrow \exists u' \\
& \quad \quad \bullet u' \in \{\text{trainsInYard} \in \mathbb{P} \text{trains} \mid \text{true}\} \\
& \quad \quad \quad \wedge (u, u', i) \in \{\text{trainsInYard} \in \mathbb{P} \text{trains} \\
& \quad \quad \quad \quad \wedge \text{trainsInYard}' \in \mathbb{P} \text{trains} \\
& \quad \quad \quad \quad \wedge \text{trainEnteringYard?} \in \text{trains} \\
& \quad \quad \quad \quad \mid \text{trainsInYard}' = \text{trainsInYard} \\
& \quad \quad \quad \quad \cup \{\text{trainEnteringYard?}\}\}
\end{aligned}$$

This proof obligation is again different to those of our previous examples, but – with its partner – still essentially shows the same machine correctness as when using the ‘pre’ operator. The choice between which machine configuration to use depends on the aesthetics that make the specification most suitable to its application.

In this instance, where we are dealing principally with the relationships between machines, we feel that the separation of precondition and postcondition clarifies the expected behaviour (and allows us to be certain of the precondition actually used and to strengthen it where required) and so we will continue to use the approach presented in this example. Therefore, we will use the current machine-type as the default throughout the remainder of this thesis and will refer to it as ‘machine’ rather than ‘machineC’.

A complete example showing how a specification of a machine with this configuration is used to generate proof obligations is described in appendix A.

This example also illustrates how schemas with multiple values are handled by our flattening approach to producing proof obligations.

5.5 Configuring and Specifying a Refinement

Having specified the configuration for some simple machines, it is now possible to configure relationships between machines. In example 5.10 below, we configure a relationship that allows us to express a refinement between two machines.

Example 5.10

In this example we will define the configuration for a refinement relationship, this relationship will allow us to specify the retrieve relation between the two machines and to detail the proof obligations required to validate the relationship. In this instance we will require three proof obligations. The first showing that the initialization of both machines results in a pair of states that belong to the retrieve relation. The second will show – for each operation in the involved machines – that where the retrieve relation holds and the precondition is met for the operation of the more abstract machine, there is an equivalent state that meets the precondition of the operation belonging to the more concrete machine. The final proof obligation will demonstrate – again, for each operation in the involved machines – that where the retrieve relation holds before a state transition in the more concrete machine, then there is an equivalent transition in the more abstract machine that re-establishes the retrieve relation. We present the Frog-CCL configuration for such a relationship below.

```
DEFINE RELATIONSHIP refinement
```

```
  CLAUSES
```

```
    ( NAME= retrieve,
      LEVEL= RELATIONSHIP,
      REQUIREMENT= MANDATORY,
      CONTENT= PREDICATE,
```

```

        RELATION= <FROM_MACHINE(state),TO_MACHINE(state)>
    )
PROOF_OBLIGATIONS
    (   CONSTRUCT_LEVEL,
        (   ! v @ v : TO_MACHINE(initialization)
            => ( # u @ u : FROM_MACHINE(initialization)
                & <u,v> : retrieve )
        )
    ),
    (   OPERATION_LEVEL,
        (   ! u, v, i
            @ <u,i> : FROM_MACHINE(operation.pre)
                & <u,v> : retrieve
            =>
                <v,i> : TO_MACHINE(operation.pre)
        )
    ),
    (   OPERATION_LEVEL,
        (   ! u, v, v', i, o
            @ <v,v',i,o> : TO_MACHINE(operation.post)
                & <u,v> : retrieve
                & <u,i> : FROM_MACHINE(operation.pre)
            =>
                ( # u'
                    @ <u,u',i,o> : FROM_MACHINE(operation.post)
                        & <u',v'> : retrieve
                )
        )
    )
)
END

```

We define our relationship to be of relationship-type ‘refinement’; it should be noted, however, that we do not – in our configuration – indicate the types

of the machines that will be involved in the refinement. This is intentional as it ensures maximum flexibility in the relation of machines. For example, a relationship could correctly be used between machines of machine-type *machineA*, *machineB* or a mix of the two. Obviously the validity of relating two machines is application-dependent, the only enforced restriction being the syntactic compatibility of the machines with the relationship, and with each other. For example, a clause used in the proof obligation of a relationship that is derived from a source machine must be present in that source machine. The semantic compatibility of the machines is left to the specifier of the relationship to ensure⁶.

Our configuration declares a single clause, the retrieve clause which is used to specify the relationship between the states of the two machines. As the retrieve clause is the only clause in the relationship, the refinement would be nonsensical without it and therefore the clause is mandatory. The relation attribute makes available the state variables of both the source machine and the target machine. As mentioned above there is no restriction on the type of machine used with the relationship. However, the configuration of any machine used with this relationship must contain the clauses specified in the refinement's configuration, if these clauses are missing then errors will be created at parse time (for missing clauses that are used in the relation attribute), or when proof obligations are generated (for missing clauses that are used in the proof obligations). The retrieve clause is configured to contain a predicate, as it is unlikely the declaration of additional variables would be required.

There are no operation environments declared in the refinement relationship, but it should be noted that the lack of operation environments in the relationship does not prevent us from reasoning about the operations of the machines involved in an instance of the relationship.

⁶Again, we could have chosen to make the configuration tighter, and in the future perhaps we will give the option to do so. At present, however, we have opted for the configuration that gives us most flexibility.

We declare three proof obligations in our configuration for the refinement relationship. The first is declared to operate at the construct level and involves ensuring that for every initialization of the more concrete machine (v), there exists an initialization of the more abstract machine (u), such that the pair of initialized states (u, v) is a member of the retrieve relation. That is, if it is possible to initialize our concrete machine, then we must also be able to initialize the abstract machine in an equivalent state.

The second proof obligation is defined for every pair of matched operations belonging to the two machines. This proof obligation will ensure that if there exists a state of the abstract machine (u) and set of inputs (i), such that the precondition of the abstract machine's operation is met, and for that state there is an equivalent state of the concrete machine (v), such that the pair of states (u, v) belongs to the retrieve relation; then the state of the concrete machine (v) and the set of inputs (i) will meet the precondition of the concrete machine's operation. That is, for all states and sets of inputs where the precondition of the operation of the more abstract machine holds and the retrieve relation holds, then the precondition of the more concrete machine must also be satisfied.

The final proof obligation is again declared at the operation level, and there will, therefore, be a proof obligation for every pair of operations in the two machines. This operation level proof obligation will ensure that for every pre-transition state (u) of the more abstract machine, pre-transition (v) and post-transition state (v') of the more concrete machine, set of inputs (i) and outputs (o), such that the pair of pre-transition states (u, v) belongs to the retrieve relation, the pre-transition state of the more abstract machine and inputs (u, i) satisfy the precondition of the more abstract machine's operation, and there exists a transition of the more concrete machine's operation that begins in state (v) takes a set of inputs (i) to result in a state (v') producing a set of outputs (o); there exists a post-transition state (u') of the more abstract machine that can be reached through a transition of the more abstract machine's operation from the state (u) given inputs (i) that produces the set of outputs (o), and that together with the post-transition state of the more concrete machine (v') re-establishes the retrieves relation.

That is, for all states and sets of inputs where it is possible to make a transition in the concrete machine, where the retrieve relation holds and the precondition of the abstract machine holds, there must exist a transition of the operation belonging to the more abstract machine that results in a state that re-establishes the retrieve relation and which produces identical outputs.

Consider now, the specification of a refinement using this configuration. Again we return to the example first discussed in example 2.1, where we considered the modelling of a train yard. In that example, we created two models, one of which was more concrete and could be considered a refinement of the other. Originally our machines had two operations, but for conciseness we will now look solely at the *trainEntersYard* operation. We created a machine that was equivalent to the more abstract model in example 5.9 above. We now specify a machine that is equivalent to the more concrete model.

MACHINE *trainYardR*
 TYPE *machine*
 SECTION *theTrainYard*
 STATE

trainsInYard : iseq *trains*

INITIALIZATION

trainsInYard = $\langle \rangle$

OPERATION *trainEntersYard* $\hat{=}$
 INPUTS

trainEnteringYard? : *trains*

PRE

trainEnteringYard? \notin ran *trainsInYard*

POST

$$\overline{\text{trainsInYard}' = \text{trainsInYard} \wedge \langle \text{trainEnteringYard?} \rangle}$$

END *trainEntersYard*

END *trainYardR*

The differences between this and the original specification are principally the same as those discussed for the more abstract model in the previous section. The only other difference of note is that we have referred to the operation of both machines with the same name (*trainEntersYard*) this allows us to be sure we are referring to the same operation in each machine, and is necessary for the automation of operation pairing. When specifying the two machines in *Z* it was not possible to use the same name for the two operations, as two schemas cannot be given the same name in a single *Z* section. We now consider the specification of our refinement relationship. The relationship below describes the behaviour we require.

RELATIONSHIP *trainYard_to_trainYardR*
 TYPE *refinement*
 FROM *trainYard* AS *t*
 TO *trainYardR* AS *tR*
 RETRIEVE

$$t \gg \text{trainsInYard} = \text{ran } tR \gg \text{trainsInYard}$$

END *trainYard_to_trainYardR*

The way in which the relationship is presented is clearly very different to our previous example. However, the meaning of the specification is identical.

The advantages of the construct based approach are immediately clear. The type of relationship involved is immediately apparent, and it can quickly

be determined which other constructs are involved in the relationship (in this instance the machines *trainYard* and *trainYardR*).

The retrieve relation specified is also slightly different (for comparison, the previous definition is given below).

$G_{trainYard}$
$trainYard$
$trainYardR$
<hr style="width: 50%; margin-left: 0;"/> $trainsInYard = \text{ran } trainsInYardR$

In example 2.1 we reasoned about the state schemas of our two models, in our refinement relationship specification, our retrieve clause has access to the state clause so can reason about the state variables directly.

Our example relationship can be expressed in the extended Z \LaTeX notation as follows.

```

\begin{relationship}{trainYard\_to\_trainYardR}
  {refinement}{trainYard \AS t}{trainYardR \AS tR}
  \clause{retrieve}{
    t\>> trainsInYard = \ran tR\>> trainsInYard
  }
\end{relationship}

```

Finally, we consider the proof obligations for our refinement relationship. Firstly, we examine the generation of the initialization proof obligation. We begin with the proof obligation specified in our configuration (shown below).

$$\begin{aligned}
& \vdash? \forall v \bullet v \in \text{TO_MACHINE}(\textit{initialization}) \\
& \Rightarrow (\exists u \bullet u \in \text{FROM_MACHINE}(\textit{initialization}) \\
& \quad \wedge (u, v) \in \textit{retrieve})
\end{aligned}$$

We then use our specifications to substitute the clauses with their corresponding relations. This results in the following theorem.

$$\begin{aligned}
& \vdash? \forall v \\
& \bullet v \in \{trainsInYard \in \text{iseq } trains \mid trainsInYard = \langle \rangle\} \\
& \Rightarrow (\exists u \bullet u \in \{trainsInYard \in \mathbb{P} \text{ trains} \mid trainsInYard = \emptyset\} \\
& \quad \wedge (u, v) \in \{t \gg trainsInYard \in \mathbb{P} \text{ trains}, \\
& \quad \quad tR \gg trainsInYard \in \text{iseq } trains \mid \\
& \quad \quad t \gg trainInYard = \text{ran } tR \gg trainInYardR\})
\end{aligned}$$

This proof obligation is similar to that presented in example 2.1 on page 47, the only difference (other than the notational difference) is that in the previous example, the bound variables in the universal and existential quantification clauses are typed. In that example, this would have ensured that u and v were members of the state relation of each machine. We do not require this restriction when configuring our relationship, as we would like to assume that both machines are consistent in their own right, and allow the proof obligation generated to focus on the proof that the relationship is a valid refinement. Of course, the flexibility of the configuration allows the shape of the proof obligations to suit an individual's specific needs. As we considered the discharge of this proof obligation in the previous example, we will not re-examine it.

We progress now to the basic proof obligation for the termination condition (which we have derived from our configuration). This is shown below.

$$\begin{aligned}
& \vdash? \forall u, v, i \\
& \bullet (u, i) \in \text{FROM_MACHINE}(operation.pre) \\
& \quad \wedge (u, v) \in \text{retrieve} \\
& \quad \Rightarrow (v, i) \in \text{TO_MACHINE}(operation.pre)
\end{aligned}$$

We use the specifications of our machines, and relationships to instantiate the proof obligation as follows.

$$\begin{aligned}
& \vdash? \forall u, v, i \\
& \bullet (u, i) \in \{ \text{trainsInYard} : \mathbb{P} \text{trains}; \text{trainEnteringYard?} : \text{trains} \\
& \quad | \text{trainEnteringYard?} \notin \text{trainsInYard} \} \\
& \wedge (u, v) \in \{ t \gg \text{trainsInYard} : \mathbb{P} \text{trains}; \\
& \quad \text{tR} \gg \text{trainsInYard} : \text{iseq } \text{trains} \\
& \quad | t \gg \text{trainsInYard} = \text{ran } \text{tR} \gg \text{trainsInYardR} \} \\
& \Rightarrow (v, i) \in \{ \text{trainsInYard} : \mathbb{P} \text{trains}; \text{trainEntersYard?} : \text{trains} \\
& \quad | \text{trainEntersYard?} \notin \text{ran } \text{trainsInYard} \}
\end{aligned}$$

As before, this proof obligation is equivalent to the one presented in example 2.1 (again we omit the typing of the bound variables here), and we will not consider it further.

We consider now the final proof obligation which ensures that every state transition in the more abstract machine has an equivalent in the more concrete machine. The basic proof obligation from our configuration is presented below.

$$\begin{aligned}
& \vdash? \forall u, v, v', i, o \\
& \bullet (v, v', i, o) \in \text{TO_MACHINE}(\text{operation.post}) \\
& \quad \wedge (u, v) \in \text{retrieve} \\
& \quad \wedge (u, i) \in \text{FROM_MACHINE}(\text{operation.pre}) \\
& \Rightarrow (\exists u' \\
& \quad \bullet (u, u', i, o) \in \text{FROM_MACHINE}(\text{operation.post}) \\
& \quad \wedge (u', v') \in \text{retrieve})
\end{aligned}$$

We then use the specifications of our machines and substitute the clauses with the appropriate relations. This results in the following theorem.

The first will show that wherever the more concrete machine can be initialized, there is an equivalent initialization of the more abstract machine. The second will show – for each operation in the involved machines – that for every pair of pre-transition states that belong to the retrieve and within relations where a state transition is possible in the more concrete machine, there is an equivalent state transition in the more abstract machine that will give a pair of post-transition states that belong either to the retrieve relation or both the concedes and output relations.

```
DEFINE RELATIONSHIP retrenchment
```

```
  CLAUSES
```

```
    ( NAME= retrieve,
      LEVEL= RELATIONSHIP,
      REQUIREMENT= OPTIONAL,
      CONTENT= PREDICATE,
      RELATION= <FROM_MACHINE(state),TO_MACHINE(state)>
    ),
    ( NAME= within,
      LEVEL= ramifications,
      REQUIREMENT= OPTIONAL,
      CONTENT= PREDICATE,
      RELATION= <FROM_MACHINE(state),TO_MACHINE(state),
                FROM_MACHINE(inputs),TO_MACHINE(inputs)>
    ),
    ( NAME= concedes,
      LEVEL= ramifications,
      REQUIREMENT= OPTIONAL,
      CONTENT= PREDICATE,
      RELATION= <FROM_MACHINE(state),TO_MACHINE(state),
                FROM_MACHINE(state'),TO_MACHINE(state'),
                FROM_MACHINE(inputs),TO_MACHINE(inputs),
                FROM_MACHINE(outputs),TO_MACHINE(outputs)>
    ),
```

```

(  NAME= output,
  LEVEL= ramifications,
  REQUIREMENT= OPTIONAL,
  CONTENT= PREDICATE,
  RELATION= <FROM_MACHINE(state),TO_MACHINE(state),
            FROM_MACHINE(state'),TO_MACHINE(state'),
            FROM_MACHINE(inputs),TO_MACHINE(inputs),
            FROM_MACHINE(outputs),TO_MACHINE(outputs)>
)
OPERATION_ENVIRONMENTS
(  NAME= ramifications,
  REQUIREMENT= OPTIONAL
)
PROOF_OBLIGATIONS
(  CONSTRUCT_LEVEL,
  (    ! v @ v : TO_MACHINE(initialization)
    => ( # u @ u : FROM_MACHINE(initialization)
        & <u,v> : retrieve )
  )
),
(  OPERATION_LEVEL,
  (    ! u, v, i, j
    @ <u,i> : FROM_MACHINE(operation.pre)
    & <u,v> : retrieve
    & <u,v,i,j> : ramifications.within
    =>
    <v,j> : TO_MACHINE(operation.pre)
  )
),
(  OPERATION_LEVEL,
  (    ! u, v, v', i, j, p
    @ <v,v',j,p> : TO_MACHINE(operation.post)
    & <u,v> : ramifications.retrieve
  )
)

```

```

& <u,v,i,j> : ramifications.within
& <u,i> : FROM_MACHINE(operation.pre)
=> ( # u',o @ <u,u',i,o> :
      FROM_MACHINE(operation.post)
      & (( <u',v'> : retrieve
          & <u,v,u',v',i,j,o,p> :
              ramifications.output )
          | <u,v,u',v',i,j,o,p> :
              ramifications.concedes )
      )
)
)
END

```

We define our relationship to be of type ‘retrenchment’ and again it should be noted that there is no restriction on the machine-types of the machines involved in the relationship.

Our configuration defines a single clause at the construct level, this is the retrieve clause and it is configured in much the same way as for the refinement relationship (see example 5.10 above) – the only difference being that the retrieve clause is optional in a retrenchment relationship. The remainder of the clauses are all configured to be used within the ramifications operations environment. The ramifications operation environment is where we will configure the clauses that – for each operation – will provide the augmentation to the retrieve relation. This operation environment is configured as optional. In practice, however, a retrenchment without any ramifications may be better modelled as a refinement. There are three clauses that can be used within the ramifications operation environment: the within, concedes and output clauses. All three clauses are optional and contain a predicate. Only the pre-transition variables are available in the within clause, that is the state variables of more abstract and more concrete machine and the inputs to the current operation. The concedes and output clauses use both the pre-transition and post-transition state of the two machines and have available

the state variables of both machines as well as the inputs and outputs of the operation concerned.

There are three proof obligations configured for our retrenchment relationship. The first is declared at the construct level, and ensures that for every possible initialization in the more concrete machine, there is an equivalent initialization in the more abstract machine. The configured proof obligation states that for every initialization of the more concrete machine (v), there exists an initialization of the more abstract machine (u), such that the pair of initialized states (u, v) is a member of the retrieve relation. That is, whenever it is possible to initialize the concrete machine, there is an equivalent initialization of the abstract machine that establishes the retrieve relation.

The second of the proof obligations is configured at the operation level and will apply to every pair of matched operations. This proof obligation states that if there is a state of the more abstract machine (u) and set of inputs to the more abstract machine's operation (i), that satisfies the precondition of that operation and there is also an equivalent state of the concrete machine (v) and set of inputs to the concrete machine's operation (j), such that the pair of states (u, v) belongs to the retrieve relation, and the quadruple formed from both states and sets of inputs (u, v, i, j) belongs to the within relation; then that state of the concrete machine (v), together with the set of inputs for the concrete machine's operation (j) will satisfy that operation's precondition. That is, if the precondition of an operation of the abstract machine is satisfied such that there is an equivalent state and set of inputs in the concrete machine for which the retrieve and within relations hold, then the precondition of the concrete machine must also hold.

The final proof obligation is declared at the operation level. A theorem will be produced, therefore, for every pair of operations in the two machines. This operation level proof obligation states that for every pre-transition state (v) and post-transition state (v') of the operation belonging to the more concrete machine – given a set of valid inputs (j), and producing outputs (p) – where there is a pre-transition state of the more abstract machine (u) and set of valid inputs to the operation of the more abstract machine (i) such that the pair of pre-transition states (u, v) belongs to the retrieve relation

and the sequence (u, v, i, j) belongs to the within relation and the pair of the pre-transition state of the abstract machine and the inputs to its operation (u, i) belong to the precondition relation of the abstract machine's operation; there exists a post-transition state of the more abstract machine (u') that can be reached – through the application of the operation – from the pre-transition state, u , when given the set of inputs, i , that produces a set of outputs (o), and that either the pair of post-transition states (u', v') of the two machines belongs to the retrieve relation, and the sequence $(u, v, u', v', i, j, o, p)$ belongs to the output relation, or the sequence $(u, v, u', v', i, j, o, p)$ belongs to the concedes relation. That is, if it is possible to make a transition in the concrete machine's operation such that there is an equivalent state and set of inputs in the abstract machine for which the retrieve and within relations hold and the precondition of the abstract machine's operations is satisfied, then there is a transition of the abstract machine's operation that will establish either the retrieve and output relations or the concedes relation.

We now re-examine the example introduced in example 2.2, where we considered a model of the stars in the sky. We had a more abstract model that was able to add new stars without restrictions and a more concrete model where an upper limit was placed on the number of stars we could store in our sky.

The first step is to redefine the schemas we presented previously as machines. The definition for the first of these machines (the more abstract *starsInTheSky*) is shown below.

MACHINE *starsInTheSky*
 TYPE *machine*
 SECTION *theStarsInTheSky*
 STATE

starsInSky : \mathbb{P} *stars*

INITIALIZATION

starsInSky = \emptyset

OPERATION *discoverStar* $\hat{=}$
 INPUT

newStar? : *stars*

PRE

newStar? \notin *starsInSky*

POST

starsInSky' = *starsInSky* \cup {*newStar?*}

END *discoverStar*

END *starsInTheSky*

This definition is equivalent to that presented in the earlier example, a set of stars is declared and is initialized to the emptyset and a single operation is defined to allow the discovery of new stars.

The definition for the second machine (the more concrete *starsInTheSkyC*) is shown below.

MACHINE *starsInTheSkyC*
 TYPE *machine*
 SECTION *theStarsInTheSky*
 STATE

starsInSky : \mathbb{P} *stars*

INITIALIZATION

starsInSky = \emptyset

OPERATION *discoverStar* $\hat{=}$
 INPUT

newStar? : *stars*

OUTPUT

message! : *MESSAGES*

PRE

newStar? \notin *starsInSky*

POST

#starsInSky < *upperlimit*
 \Rightarrow *starsInSky'* = *starsInSky* \cup {*newStar?*}
 \wedge *message!* = *starAdded*

#starsInSky \geq *upperlimit*
 \Rightarrow *starsInSky'* = *starsInSky*
 \wedge *message!* = *starArrayFull*

END *discoverStar*

END *starsInTheSkyC*

Again, this definition is equivalent to the former example, and is very similar to *starsInTheSky* with the principal difference being the introduction of a check in the *discoverStar* operation.

We consider now the specification of our retrenchment relationship. The relationship below describes the necessary behaviour.

RELATIONSHIP *starsInTheSky_to_starsInTheSkyC*
 TYPE *retrenchment*
 FROM *starsInTheSky* AS *s*
 TO *starsInTheSkyC* AS *sC*

RETRIEVE

$$s \gg \text{starsInSky} = sC \gg \text{starsInSky}$$

RAMIFICATIONS $\text{discoverStar} \hat{=}$
WITHIN

$$s \gg \text{newStar?} = sC \gg \text{newStar?}$$

OUTPUT

$$sC \gg \text{message!} = \text{starAdded}$$

CONCEDES

$$\begin{aligned} \#sC \gg \text{starsInSky} &\geq \text{upperlimit} \\ s \gg \text{starsInSky}' &= sC \gg \text{starsInSky}' \cup \{sC \gg \text{newStar?}\} \\ sC \gg \text{message!} &= \text{starArrayFull} \end{aligned}$$

END discoverStar END $\text{starsInTheSky_to_starsInTheSkyC}$

The way in which we have expressed our retrenchment is again different to the schema notation that we used in our earlier example, but again it expresses the same meaning. As with the clauses in the refinement relationship, there is no need to import variables as this is done implicitly through the configuration of the construct. The ramifications for each operation are also brought together into a single operation environment, rather than being expressed across several schema.

This relationship can be expressed in our extended Z \LaTeX notation as follows.

```
\begin{relationship}{starsInTheSky\_to\_starsInTheSkyC}
  {retrenchment}{starsInTheSky \AS s}{starsInTheSkyC \AS sC}
\clause{retrieve}{
  s \>> starsInSky = sC \>> starsInSky
}
```

```

\begin{openv}{ramifications}{discoverStar}
\clause{within}{
  s \>> newStar? = sC \>> newStar?
}
\clause{output}{
  sC \>> message! = starAdded
}
\clause{concedes}{
\# sC \>> starsInSky \geq upperlimit \
s \>> starsInSky' = sC \>> starsInSky'
\cup \{ sC \>> newStar? \} \
sC \>> message! = starArrayFull
}
\end{openv}
\end{relationship}

```

Finally, we consider the generation of the proof obligations for our retrenchment relationship. Firstly, we examine the initialization proof obligation. We begin with the proof obligation specified in our construct's configuration.

$$\begin{aligned}
& \vdash? \forall v \bullet v \in \text{TO_MACHINE}(\textit{initialization}) \\
& \Rightarrow (\exists u \bullet u \in \text{FROM_MACHINE}(\textit{initialization}) \\
& \quad \wedge (u, v) \in \textit{retrieve})
\end{aligned}$$

We then use our specifications to substitute the clauses with their corresponding relations. This results in the following theorem.

$$\begin{aligned}
& \vdash? \forall u \\
& \bullet u \in \{ \textit{starsInSky} \in \mathbb{P} \textit{stars} \mid \textit{starsInSky} = \emptyset \} \\
& \Rightarrow (\exists v \bullet v \in \{ \textit{starsInSky} \in \mathbb{P} \textit{sky} \mid \textit{starsInSky} = \emptyset \} \\
& \quad \wedge (u, v) \in \{ s \gg \textit{starsInSky} \in \mathbb{P} \textit{stars}; \\
& \quad \quad \quad sC \gg \textit{starsInSky} \in \mathbb{P} \textit{stars} \\
& \quad \quad \quad \mid s \gg \textit{starsInSky} = sC \gg \textit{starsInSky} \})
\end{aligned}$$

As with the proof obligations for the refinement relationship, the obligations generated are trivial and similar to those specified in our previous example,

so will not discuss them further.

We now consider the second proof obligation, which is the first to examine the relationship between our machine's operations.

$$\begin{aligned}
& \vdash? \forall u, v, i, j \\
& \bullet (u, i) \in \text{FROM_MACHINE}(\text{operation.pre}) \\
& \quad \wedge (u, v) \in \text{retrieve} \\
& \quad \wedge (u, v, i, j) \in \text{within} \\
& \Rightarrow (v, j) \in \text{TO_MACHINE}(\text{operation.pre})
\end{aligned}$$

Once again, we use the specifications of our machines and relationships to instantiate the proof obligation as follows.

$$\begin{aligned}
& \vdash? \forall u, v, i, j \\
& \bullet (u, i) \in \{ \text{starsInSky} : \mathbb{P} \text{ stars}; \text{newStar?} : \text{stars} \\
& \quad \mid \text{newStar?} \notin \text{starsInSky} \} \\
& \quad \wedge (u, v) \in \{ s \gg \text{starsInSky} : \mathbb{P} \text{ stars}; sC \gg \text{starsInSky} : \mathbb{P} \text{ stars} \\
& \quad \mid s \gg \text{starsInSky} = sC \gg \text{starsInSky}C \} \\
& \quad \wedge (u, v, i, j) \in \{ s \gg \text{starsInSky} : \mathbb{P} \text{ stars}; sC \gg \text{starsInSky} : \mathbb{P} \text{ stars}; \\
& \quad \quad s \gg \text{newStar?} : \text{stars}; sC \gg \text{newStar?} : \text{stars} \\
& \quad \quad \mid s \gg \text{newStar?} = sC \gg \text{newStar?} \} \\
& \Rightarrow (v, j) \in \{ \text{starsInSky} : \mathbb{P} \text{ stars}; \text{newStar?} : \text{stars} \\
& \quad \mid \text{newStar?} \notin \text{starsInSky} \}
\end{aligned}$$

This proof obligation is trivial and again very similar to those specified in our refinement example, so we will not discuss it further.

We proceed now to consider our final retrenchment proof obligation to show the applicability of our relationship.

$$\begin{aligned}
& \vdash? \forall u, v, v', i, j, p \\
& \bullet (v, v', j, p) \in \text{TO_MACHINE}(\text{operation.post}) \\
& \quad \wedge (u, v) \in \text{retrieve} \\
& \quad \wedge (u, v, i, j) \in \text{within} \\
& \quad \wedge (u, i) \in \text{FROM_MACHINE}(\text{operation.pre}) \\
& \Rightarrow (\exists u', o \\
& \quad \bullet (u, u', i, o) \in \text{FROM_MACHINE}(\text{operation.post}) \\
& \quad \quad \wedge (((u', v') \in \text{retrieve} \wedge (u, v, u', v', i, j, o, p) \in \text{output}) \\
& \quad \quad \vee (u, v, u', v', i, j, o, p) \in \text{concedes})
\end{aligned}$$

The clauses are then substituted with their relations in accordance with our specification.

$$\begin{aligned}
& \vdash? \forall u, v, v', i, j, p \\
& \bullet (u, i) \in \{starsInSky : \mathbb{P} stars; newStar? : stars \\
& \quad | newStar? \notin starsInSky\} \\
& \wedge (u, v) \in \{s \gg starsInSky, sC \gg starsInSky : \mathbb{P} stars; \\
& \quad | s \gg starsInSky = sC \gg starsInSky\} \\
& \wedge (v, v', j, p) \in \\
& \quad \{starsInSky, starsInSky' : \mathbb{P} stars; newStar? : stars \\
& \quad | (\#starsInSky < upperlimit \\
& \quad \Rightarrow starsInSky' = starsInSky \cup \{newStar?\} \\
& \quad \quad \wedge message! = starAdded) \\
& \quad \wedge (\#starsInSky \geq upperlimit \\
& \quad \Rightarrow starsInSky' = starsInSky \\
& \quad \quad \wedge message! = starArrayFull)\} \\
& \wedge (u, v, i, j) \in \{s \gg starsInSky : \mathbb{P} stars; sC \gg starsInSky : \mathbb{P} stars; \\
& \quad s \gg newStar? : stars; sC \gg newStar? : stars \\
& \quad | s \gg newStar? = sC \gg newStar?\} \\
& \Rightarrow \exists u' \\
& \bullet (u, u', i) \in \{starsInSky, starsInSky' : \mathbb{P} stars; newStar? : stars \\
& \quad | starsInSky' = starsInSky \cup \{newStar?\}\} \\
& \wedge (((u', v') \in \{s \gg starsInSky, sC \gg starsInSky : \mathbb{P} stars \\
& \quad | s \gg starsInSky = sC \gg starsInSky\} \\
& \wedge (u, u', v, v', i, j, p) \in \\
& \quad \{s \gg starsInSky, s \gg starsInSky' : \mathbb{P} stars; \\
& \quad sC \gg starsInSky, sC \gg starsInSky' : \mathbb{P} stars; \\
& \quad s \gg newStar? : stars; sC \gg newStar? : stars; \\
& \quad sC \gg message! : MESSAGES \\
& \quad | message! = starArrayFull\}) \\
& \vee (u, u', v, v', i, j, p) \in \\
& \quad \{s \gg starsInSky, s \gg starsInSky' : \mathbb{P} stars; \\
& \quad sC \gg starsInSky, sC \gg starsInSky' : \mathbb{P} stars; \\
& \quad s \gg newStar? : stars; sC \gg newStar? : stars; \\
& \quad sc \gg message! : MESSAGES \\
& \quad | \#sC \gg starsInSky \geq upperlimit \\
& \quad \wedge s \gg starsInSky' = sC \gg starsInSky' \cup \{sC \gg newStar?\} \\
& \quad \wedge sC \gg message! = starArrayFull\})
\end{aligned}$$

As with the refinement proof obligations, this proof obligation is equivalent to the one produced in chapter 2 (again, we don't type the bound variables in this instance), and so we will not discuss it further.

5.7 Machine Inclusion

Users of the B-Method will know that an advantage of its abstract machine, is that small components can be combined to create specifications for large systems. This has the advantage that each individual component is relatively understandable and reusable. In the B-Method the INCLUDES clause (see section 7.2.3 of [Abr96] for more details) is used to incorporate one machine within another. In the system we have designed, parent sections can be used to give the same effect. This is illustrated below in example 5.12.

Example 5.12

In this example we will define two machines, the second of which will inherit behaviour from the first. The first definition will describe a machine that allocates tickets, keeping track of which tickets have been allocated. The second definition also describes a machine that allocates tickets, but this time the tickets are associated with the person who will use them⁷.

We begin the definition of our first machine by defining an enclosing section where we can define our given type.

```
section ticketing parents standard_toolkit
```

We define a section called *ticketing* which inherits from the standard toolkit, providing access to all the usual types and operators.

```
[tickets]
```

We then declare a given types paragraph, where we specify a single given

⁷Note that, in this world a person is assumed to be able to use only a single ticket.

type *tickets*, which is the set of all available tickets. We now begin the definition of our machine.

MACHINE *ticketing*
 TYPE *machine*
 SECTION *ticketing*
 STATE

$allocatedTickets : \mathbb{P} tickets$

INITIALIZATION

$allocatedTickets = \emptyset$

OPERATION *allocateTicket* $\hat{=}$
 OUTPUTS

$allocatedTicket! : tickets$

PRE

$\exists allocatableTicket \bullet allocatableTicket \in tickets$
 $\wedge allocatableTicket \notin allocatedTickets$

POST

$allocatedTicket! \notin allocatedTickets$
 $allocatedTickets' = allocatedTickets \cup \{allocatedTicket!\}$

END *allocateTicket*

END *ticketing*

Notice first that we have given the same name to our machine as to its previously specified parent section. This does not cause a name clash as the name of the Z Section created from our construct is not given the same name as

the construct itself, but an internal name generated from the original construct name (see section 5.3.3). We have defined two construct level clauses declaring a set to record which tickets have been allocated and initializing that set to be the emptyset. We have also defined an operation which selects a ticket that has not yet been allocated and then outputs that ticket and adds it to the allocated set.

We consider now the definition of our second machine. Again we begin by defining an enclosing section.

section *assignTickets* parents *standard_toolkit*

We define a section called *assignTickets* which inherits from the standard toolkit, providing access to all the usual types and operators.

[*people*]

We then declare a given types paragraph, where we specify a single given type *people*, which is the set of all people who can be assigned a ticket. We now begin the definition of our machine.

MACHINE *assignTickets*
 TYPE *machine*
 SECTION *assignTickets*, *MACHINE.ticketing*
 STATE

assignedTickets : *people* \leftrightarrow *tickets*
MACHINE.ticketing.state

allocatedTickets = ran *assignedTickets*

INITIALIZATION

MACHINE.ticketing.initialization

assignedTickets = \emptyset

OPERATION *allocateAndAssignTicket* $\hat{=}$
 INPUTS

personUsingTicket? : *people*

OUTPUTS

allocatedTicket! : *tickets*

PRE

MACHINE.ticketing.allocateTicket.pre

POST

MACHINE.ticketing.allocateTicket.post

$$\begin{aligned} \text{assignedTickets}' = \text{assignedTickets} \cup \\ \{ \text{personUsingTicket?} \mapsto \text{allocatedTicket!} \} \end{aligned}$$

END *allocateAndAssignTicket*

END *assignTickets*

This machine (*assignTickets*) declares two parent sections: the first being our enclosing section and the second is a reference to our previous machine. As the *ticketing* machine and its enclosing, parent section have the same name we must use a ‘*MACHINE.*’ prefix to indicate that we want to have the construct’s section as a parent rather than the section that forms the parent of that construct. The general rule when examining parent sections is that available section names will be matched first, if no matching section names are found then machine names will then be examined. If the ‘*MACHINE.*’ prefix is used, then only machine names will be searched. If, for example, we had named the enclosing section of the machine *ticketing*, as *ticketingSection*, we would be able to refer to the construct simply as *ticketing* when using it as a parent section.

The first clause of our machine is the state clause where we declare a

partial function between *people* and *tickets* which we use to keep track of which people have been assigned which tickets. We also declare a reference to the state clause of the ticketing machine (here it is necessary to refer to the clause as *MACHINE.ticketing.state*; firstly, we need to disambiguate between the construct and section named *ticketing* as described above, and secondly, we need to disambiguate between the state of the current machine and that of the incorporated *ticketing* machine)⁸. We can use a predicate in our state clause to describe the relationship between our newly declared state variables and those imported from another machine. In our example, we state that it is not possible for a person to have been assigned a ticket that has not yet been allocated.

The second clause of our machine initializes the variables of our machine. We initialize the variables of our imported machine by declaring its initialization clause and set the starting value of our function used to track the assignation of tickets to be the emptyset.

Finally, we define an operation that allows us – when given a person – to allocate a ticket and assign it to that person, outputting the allocated ticket. Note that it is necessary to declare the *allocatedTicket!* variable in the outputs clause of the operation. Despite the fact that the variable is declared as an output in the *allocateTicket* operation, we need the outputs clause to be complete in order to generate proof obligations correctly. We also feel that this prevents the obfuscating of inputs and outputs to an operation. In the body of the operation we declare a reference to the *allocateTicket* operation environment; on this occasion we do not need to dereference this with the machine name, the clause is unique within the current scope and does not clash with any defined schema or clause. The operation defined in the *ticketing* machine is extended, the behaviour to allocate – and output – a valid ticket is inherited, but now we also record the person who has been allocated that ticket.

Of course, this simple example is trivial and the use of only a single

⁸It should be noted that this reference is not really a reference to the clause of the other machine, but to the schema produced from that clause in the syntax transformation phase (see section 5.3.3).

machine would not have produced an overly complex specification. We hope, however, that this example has illustrated the point that simple machines, can be combined and extended to form larger and more complex machines.

5.8 Concluding Remarks

In this chapter we have introduced the Frog construct configuration language, known as Frog-CCL. A construct is a machine or a relationship (between machines) and can be incorporated into a Frog specification – being roughly equivalent to a *Z* section. The content and behaviour of a construct is configurable through its definition in Frog-CCL. This definition also includes the generic proof obligations which, when instantiated, will allow us to show the consistency of a specific instance of that construct. The Frog parser was extended to parse constructs that have been defined in this way, accepting an extended version of the \LaTeX syntax and transforming it to satisfy the annotated syntax specified in chapter 10 of [ISO02]. We have presented a sample configuration for a machine, a refinement relationship and a retrenchment relationship. Where these constructs are specified in the remainder of this thesis, it can be presumed that these are the configurations used unless specified otherwise.

Chapter 6

Generating Proof Obligations

This chapter describes the generation of proof obligations for modelling constructs, and the ways in which a user can attempt to discharge those obligations. We begin by presenting an overview of the process by which we achieve this; highlighting the distinction between the user's view of proof obligation generation, and the more complex perspective required when using a mechanized approach. We then proceed to show how a semantic model of a construct – that can be used for reasoning about that construct – is created from its specification. We define a language in which our semantic model can be expressed, and an automatable operation that can be used to translate a specification into that language. We then proceed to discuss how we use the semantic model created to derive proof obligations for a specific construct from the generic proof obligations presented in that construct's Frog-CCL configuration. Following this, we describe a number of available theorem provers and the possible advantages and disadvantages that could arise from their use. We then concentrate on showing how the theorem prover-independent proof obligations – whose creation will have just been described – can be translated into a format suitable for a specific theorem prover (Isabelle/ZF), and how a user is able to use such a theorem prover to discharge those obligations.

6.1 Introduction

The driving force behind retrenchment is the intention to extend the scope of systems whose specification and design can be captured formally. Retrenchment's advantage over refinement is that it is able to present a formal relationship between any pair of models in a clear, concise manner, whereas there are circumstances when using refinement leads to an obfuscated relationship (or the lack of any relationship whatsoever). While being able to specify a relationship between models certainly has its uses, these are limited without the ability to prove that the relationship is valid. Indeed, with retrenchment relationships, it could be suggested that the necessity of proof is even greater, as it is only with a rigorous approach that all potential 'gaps' between two models' functionality can be identified.

In fact, it can be argued that the necessity of proof is always great, and that while specifying a system using any of formal method's techniques can help to highlight errors, the real advantage is being able to prove that the system is consistent through the generation and discharge of proof obligations. We feel, therefore, that in any tool that claims to support formal development, provision must be made for the validation of models (and their relationships), and as such we incorporate into our tool a system that allows us to generate proof obligations. We also provide a facility that feeds those obligations to external theorem provers so that users are able to attempt the proof of their obligations with existing tools.

In chapter 4 we showed how a specification could be created in the Z notation, and in chapter 5, we introduced the concept of constructs that could be used to encapsulate models and the relationships between those models. In our tool, we will concentrate our efforts on creating proof obligations for constructs¹. These proof obligations will be created through an examination of the construct's specification and its configuration. Once proof obligations have been created, they will be translated into the syntax of an external theorem prover which could then be used to attempt the discharge of some

¹We have considered the possibility of creating lemmas that are based on conjectures within a Z specification, but this is beyond the scope of the current research.

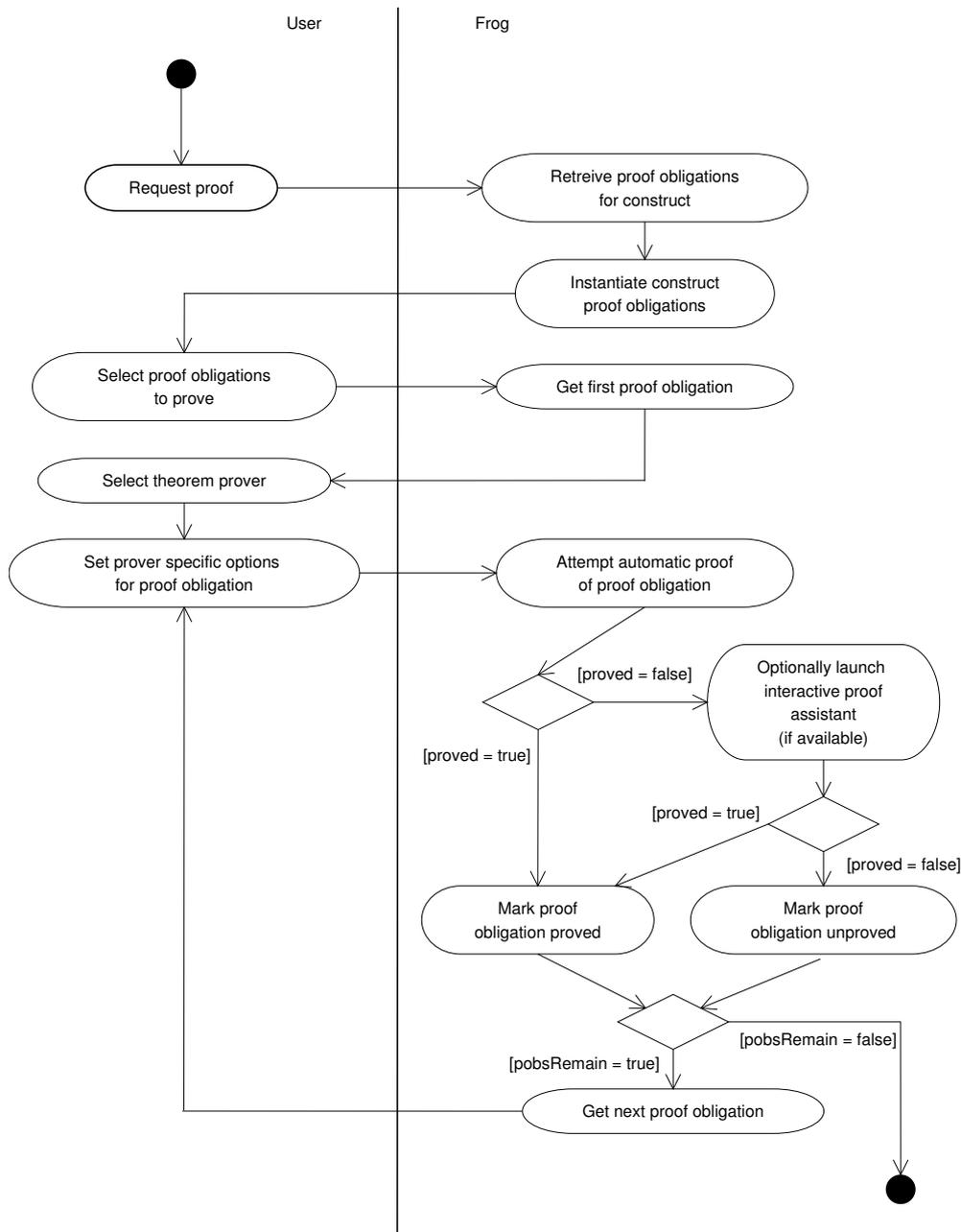


Figure 6.1: Proving a Construct

or all of those obligations. An overview of this process is given in figure 6.1.

In section 5.2.2 we showed how the Frog-CCL configuration of a construct can be used to generate the proof obligations for an instance of that construct.

However, the proof obligations we created in that section were created in the Z notation. In order that we can use an established theorem prover to discharge the proof obligations that we generate, we need to produce those proof obligations in a syntax that such theorem provers can understand. This first requires the creation of a representation of our construct from which we can derive that construct's meaning; we refer to this model as the semantic model of the construct. It is the semantic model of a construct that is used – together with its configuration – to build the proof obligations required to verify it. The generation of the semantic model is a complex process, and can lead to a disparity in the perceived difficulty in the creation of proof obligations. From the user's perspective, the generation of a set of proof obligations from a construct's specification can seem a relatively simple, and intuitive task. When we consider the mechanization of this process we first need to consider the incorporation of every reference and inherited operator; for instance there is no guarantee that a theorem prover's definition of the union operator will tally with that of our definition and so we must include this definition (and every definition on which it depends) in every proof obligation that uses that operator. The language which we use to describe the semantic model of a construct, and the process by which it can be derived from a Z specification (that is the specification that contains the syntax transformed Z section associated with the construct) are described fully in section 6.2.

Once we have the semantic model we are able to instantiate the generic proof obligations presented in the construct's configuration. When generating the proof obligations for a construct in section 5.2.2 we presented a methodology through which this could be achieved. This description however, outlined a method that could be followed by a human to produce an obligation in the Z notation, but did not describe the generation process at a level of detail required for the automation of the procedure. We rectify this by presenting a more sophisticated description in section 6.3.

Once proof obligations have been created, it makes sense that a user can attempt to discharge them. In section 6.4 we describe a number of the theorem provers that we considered when looking for an initial proof assistant

with which our users may be able to do this. Section 6.5 describes how the proof obligations we have generated can be transformed into a format which can be read by Isabelle/ZF, and shows how the proof process can proceed. While many of these theorem provers are often referred to as ‘automated’, we have realized that the majority of the responsibility in discharging the proof remains with the user, and that the theorem provers can perhaps be better considered as interactive proof assistants. Similarly therefore, once our tool has generated proof obligations – and whilst our tool may provide some rudimentary assistance – the proof burden lies primarily with the tool’s users.

6.2 Producing a Semantic Model

Whilst the Z notation is extremely useful in the creation of specifications, it has not been used as an input language to any widely available theorem prover (with the possible exception of those which are directly attached to existing Z tools – see section 6.4.9). Typically, theorem provers use a syntax based on first or higher order logic. It is necessary therefore, to create a model of our specification which can be easily translated into the formats required by a variety of theorem provers; we call this the semantic model of the specification. This section begins with the definition of an intermediary language that we use within the semantic model, and proceeds to discuss how we translate elements of Z notation into this language. We then proceed to describe the creation of a semantic model for a construct. The semantic model for a construct is considered to be the semantic binding environment of its associated Z section. Each section, paragraph and expression belonging to a Z specification generates a set of semantic bindings that together form that syntactic block’s semantic binding environment. The set of semantic bindings that constitute a construct’s semantic model will eventually form the building blocks used in the instantiation of the construct’s generic proof obligations.

6.2.1 The ZF Language

The language that we have chosen to use within our semantic model is based on the language of Zermelo-Fraenkel set theory and as such we will refer to it as the ZF language. The obvious motivator for using Zermelo-Fraenkel set theory is that Z was originally based on this particular flavour of set theory and hence the close ties between the two should make translation a relatively trouble free process, whilst still reducing our specification to a language than can be understood by theorem provers. Our ZF language shall be theorem prover-independent, but transformation to it will hopefully perform the majority of translation required in obtaining a proof obligation in any suitable theorem prover's specific notation (the ideal being that we generate a proof obligation that requires simple syntactic substitution for use in a particular theorem prover).

Note that, the translation of the Z specification into the ZF language does not in itself create the semantic model of the specification. The ZF language is merely used to describe the component parts of each semantic binding. A set of such semantic bindings will then constitute the specification's semantic model.

ZF Language Classes

We shall define our ZF language through the specification of a number of ZF language classes (all of which shall be considered subclasses of a superclass ZFObject). We use this object-oriented description here as it makes it easier to refer to the components of our language in the remainder of this chapter (of course, these classes also form the basis of our implementation – see section 7.4).

Table 6.1: ZF Language Classes

Class	Definition
ZFCarrierSet	An extension of the ZFName class that allows us to maintain the (notional) distinction between variables and carrier sets. For example, <i>ZFCarrierSet001</i> .
ZFCartesianProduct	Cartesian product. For example, $i_1 \times i_2$
ZFEmptyset	The emptyset. For example, \emptyset .
ZFName	Provides a distinct name to every variable, also removes any special formatting so that all variable names can be given to automated theorem provers in ASCII format. For example, <i>ZFName001</i> .
ZFNameFunction	An extension of the ZFName class that allow us to provide an abstraction for Z function applications. Automated theorem provers do not typically allow the same notion of a function application so we introduce a variable that defines the set resulting from the application, this variable is then used to refer to that application. For example, <i>ZFName001</i> .
ZFNameGeneric	An extension of the ZFName class that allow us to distinguish between a variable, and a generic placeholder. For example, <i>ZFName001</i> .
ZFNameInstantiation	An extension of the ZFName class that allows to refer to the variable created when a generic placeholder is instantiated, and that enables to maintain a record of the instantiation parameters. For example, <i>ZFName001</i> .

Class	Definition
ZFObject	The abstract superclass of all ZF language classes.
ZFOrdered	An ordered tuple of ZFObjects. For example, $\langle i_1, i_2 \rangle$.
ZFPowerset	Powerset. For example, $\mathbb{P} i$
ZFPredicate	The abstract superclass of all ZF language predicate classes.
ZFPredicateConjunction	Conjunction of two or more ZFPredicate instances. For example, $p_1 \wedge p_2$.
ZFPredicateDisjunction	Disjunction of two or more ZFPredicate instances. For example, $p_1 \vee p_2$.
ZFPredicateEquality	Equality between two ZFPredicate instances. For example, $p_1 = p_2$.
ZFPredicateExist	Existential quantifier, has a list of bound variables and a predicate. For example, $\exists x, y \bullet p$.
ZFPredicateExist1	Unique existential quantifier, has a list of bound variables and a predicate. For example, $\exists_1 x, y \bullet p$.
ZFPredicateUniv	Universal quantifier, has a list of bound variables, a constraint, and a predicate. For example, $\forall x, y \bullet c \Rightarrow p$.
ZFPredicateImplication	Implication between two ZFPredicate instances. For example, $p_1 \Rightarrow p_2$.
ZFPredicateMembership	Membership test between two ZFObject instances. For example, $i_1 \in i_2$.
ZFPredicateNegation	Negation of ZFPredicate instance. For example, $\neg p$.
ZFPredicateParentheses	Parenthesizing of a ZFPredicate instance. For example, (p) .

Class	Definition
ZFPredicateTruth	A value that indicates truth. For example, <i>true</i> .
ZFSetExtension	A set created from the semantic bindings (see 6.2.2) of other ZFObjects. For example, $\{(z_1, h_1, p_1), (z_2, h_2, p_2)\}$
ZFSetSeparation	Set abstraction, has a list of bound variables, a type (which consists of a ZFObject) for those bound variables, and a predicate. For example, $\{x, y : t \bullet p\}$

The classes we present in table 6.1 on page 305 give us a language that can represent all of the necessary Z notation (after suitable interpretation) and should be directly supportable (again, after suitable interpretation) in any theorem prover that supports Zermelo-Fraenkel set theory. Each class falls into one of three categories. The first of these are the elements of first order logic (those classes names are prefixed with ZFPredicate), the majority of which are logical operators except ZFPredicateTruth which provides us with a constant whose value is always *true* and ZFPredicateParentheses which simply allows us to structure our predicates. The second category is formed by the elements of set theory and includes the classes ZFCartesianProduct, ZFEmptyset, ZFOrdered, ZFPowerset, ZFSetExtension and ZFSetSeparation. The final category has a number of members, but each of these is used to represent a single variable, and the distinction between the classes is only maintained to facilitate in the determination of additional properties. The classes that belong to this category are ZFCarrierSet and those classes whose name is prefixed with ZFName.

Translating to the ZF Language

Translating a Z specification to the ZF language is typically performed as part of the creation of the semantic model of that specification (see section 6.2.2).

This allows us to limit the translations that we are required to perform as it is only necessary to interpret the Z fragments required in the generation of the semantic bindings.

The majority of these translations can be performed without difficulty as either we reduce the complexity whilst creating the semantic binding, or the syntax is roughly equivalent (for instance, there is not much work to be done translating a Z conjunction into a ZF language conjunction). There are however, a number of areas in which the translation between the Z notation and that of the ZF language is not as straightforward as it may seem.

The first of these is where we have a similar syntactic block in both syntaxes, but the representation is slightly different. Consider, for example, the Z notation's set comprehension expression, which one would expect to be equivalent to the ZF language's set separation class as both of these notations refer to the concept of set construction. The Z notation for set comprehension is as follows.

$$\{ \textit{declaration} \mid \textit{predicate} \bullet \textit{expression} \}$$

For example, we may use the following statement to create a set of natural numbers, each of which is one greater than a prime.

$$\{ x : \mathbb{N}; y : \mathbb{N} \mid y \in \textit{primes} \wedge x = y + 1 \bullet x \}$$

In the ZF language, however, there is some overlap between the declaration and expression parts of the Z notation's set comprehension expression, where the notation is as follows².

$$\{ \textit{declaration} \bullet \textit{predicate} \}$$

Here, we use the declaration to not only assign types to the variable we use, but also to indicate the shape of the elements that will become members of the set. For instance, if we were to attempt the creation of the same set as

²In [Hay92] – a comparison between Z and VDM – Hayes describes the advantages of using Z's notation for set comprehension rather than the Zermelo-Fraenkel method (which was at the time the only option in VDM).

above, and used – what seems like – the obvious equivalent, we would write the following.

$$\{x, y : \mathbb{N} \bullet y \in \text{primes} \wedge x = y + 1\}$$

The trouble here, of course, is that instead of creating a set of natural numbers, we are now creating a set of pairs. Therefore, when we are translating a set comprehension expression, we need to make some changes. Using human reasoning, we can determine that an equivalent set separation expression in the ZF language is as follows.

$$\begin{aligned} &\{x : \text{GIVEN } \mathbb{A} \\ &\bullet (\exists y \bullet x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y \in \text{primes} \wedge x = y + 1)\} \end{aligned}$$

When we come to translate a set comprehension expression, it is necessary for us to be able to determine the nature of the expression where we select the elements of the set. Unfortunately, Z allows us a free use of the full range of expressions in this selection, and as such it is impossible for us to (automatically) present a declaration that matches the Z specification, and conforms syntactically to the ZF language. Hence, it is necessary for us to introduce a variable which can store the result of the selection expression. In our example, this would be introduced as follows.

$$\begin{aligned} &\{z : \text{GIVEN } \mathbb{A} \\ &\bullet (\exists x, y \bullet x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y \in \text{primes} \wedge x = y + 1 \wedge z = x)\} \end{aligned}$$

More generally, we can state that given a Z set comprehension expression of the following form:

$$\{i_1 : t_1; \dots; i_n : t_n \mid p \bullet e\}$$

the translated set separation of the ZF language would be as follows (where the expression e has type τ).

$$\{x : \tau \bullet (\exists i_1 \dots i_n \bullet i_1 \in t_1 \wedge \dots \wedge i_n \in t_n \wedge p \wedge x = e)\}$$

There are a number of other syntactic blocks in which such disparities occur³. Fortunately, however, they can be resolved using similar methods to those used in this case, and we will not discuss them further here.

If the translation of our Z notation is so trouble free, you may ask why we need to create a semantic model at all. Surely, it would be easiest just to translate our specification and use this interpreted version as the input to the generation of proof obligations? The answer to this lies in the need to incorporate all required information in the proof obligation passed to the theorem prover. Let us consider again the machine we presented in example 5.2, and which we present again below.

```
MACHINE myNumber
TYPE machineWithInit
SECTION standard_toolkit
STATE
```

$$a : \mathbb{N}$$

```
INITIALIZATION
```

$$a = 0$$

```
END myNumber
```

Originally, we presented the following proof obligation for this machine.

$$\vdash? \exists u \bullet u \in \{a : \mathbb{N}\} \wedge u \in \{a : \mathbb{N} \mid a = 0\}$$

This makes perfect sense to the human reader, but how would a theorem prover be able to determine that zero is a member of the set we have defined as \mathbb{N} ?⁴ What we really need is a proof obligation along the lines of the

³Notice, for example, the change to the universal quantifier in the set separation expression.

⁴Obviously this is a side-effect of our choice to use a deep embedding.

following⁵.

$$\begin{aligned} \vdash? \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \wedge \mathbb{N} \in \mathbb{P}\mathbb{A} \wedge 0 \in \mathbb{N} \wedge 1 \in \mathbb{N} \\ \Rightarrow \exists u \bullet u \in \{a : \mathbb{N}\} \wedge u \in \{a : \mathbb{N} \mid a = 0\} \end{aligned}$$

One method of passing all this information would be to translate our specification, and all the parent sections required. Once we had this translation we could make it suitable for our theorem prover by defining a series of constants and functions that represent the specification's (Z) syntactic blocks, and using those constants and function to build a lemma that represents our proof obligation.

Our initial attempt at generating proof obligations for our construct used this method. What we found, however, was that not only was the resulting theory file huge (as it would – in all practical circumstances – need to contain the definitions for everything within the mathematical toolkit), the use of constants and functions still made the theory unreadable to all but a mechanical eye. If a proof was unable to be discharged with an automatic tactic, then discerning which tactic should be applied was all but impossible. Furthermore, we discovered that many theorem provers require constants and functions to be unwound explicitly. We were left with two options: automatically calculate an unwinding strategy for the constants used within the proof obligation, or leave the constants to be unwound interactively. The first option resulted in immense lemmas and tactic strategies, that confused not only the reader, but the theorem prover itself (ensuring that the theorem prover took extremely long times to prove even the most simple obligations)⁶. The second option was constrained by the existing legibility option and made the proof process have a level of complexity equal to (or even greater than) a pen-and-paper proof.

⁵Note that, in this example we have included the fact that both zero and one are members of the natural numbers. This is a consequence of Z 's schema orientation. The constants zero and one are both defined within the same schema in Z 's parent section, if we wish to import the properties of one of these constants we implicitly import the properties of the other.

⁶Note that, this option is essentially equivalent to not using constants or functions at all, but simply provided a degree of legibility for the reader.

To overcome these problems, we began again and proposed an alternate strategy where in the construction of the semantic model we associated each variable with a ‘mini-proof obligation’ that comprised: a set of hypotheses that outline the assumptions we must make when using the variable and a set of goals that we will need to establish if we are using the variable correctly. We called this triple a semantic binding, and describe its structure more fully in the next section. When we now generated the proof obligations for a construct we were able to import only those details belonging to the variables that we used, hence only those which were actually required in the construct’s proof. This led to greater legibility not only for the human trying to prove the obligation, but also for the theorem prover.

6.2.2 Semantic Bindings

We create a semantic model of the Z specification in the ZF language, through the generation of semantic bindings. We define a semantic binding as being the association of a set of semantic assumptions and constraints to a variable. In effect each semantic binding is a ‘mini-proof obligation’ associated with a variable and these are pieced together to build the proof obligation for the construct.

Semantic Binding Environments

Every section, paragraph and expression generates a semantic binding environment. A semantic binding environment is a collection of the semantic bindings that give that syntactic block its meaning and comprises the bindings belonging to each of its declared variables. Note however, that a paragraph or expression only ever has meaning in the context of its section and as such the semantic binding environments of these blocks should not be used independently. Each semantic binding comprises the bound object, a set of hypotheses and a set of goals. All of these components will be translated to the ZF language described in section 6.2.1. The bound object is typically a ZF representation of a Z variable. We use other ZF objects in bindings

when generating the semantic bindings of certain expressions, but these bindings have little meaning other than as a stepping-stone in the generation of bindings for the blocks that contain those expressions.

Consider, for example, the following section which has a single schema definition paragraph where we declare a variable that is a natural number, and initialize that variable to zero.

section *mySection* parents *standard_toolkit*

<i>mySchema</i>	
$a : \mathbb{N}$	
$a = 0$	

Following translation the schema definition paragraph is represented by the following axiomatic description paragraph.

| [*mySchema* : {[$a : \mathbb{N}$ | $a \in \{\textit{number_literal_0}\}$]}]

If we now consider the semantic binding of the schema construction expression (within the context of the semantic binding environment of the *standard_toolkit*), [$a : \mathbb{N}$ | $a : \{\textit{number_literal_0}\}$], we can deduce the following binding (here we use a textual representation rather than the ZF representation for simplicity).

Bound Object:	a
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A}), \mathbb{N} \in \mathbb{P} \ \mathbb{A}, \textit{number_literal_0} \in \mathbb{N},$ $\textit{number_literal_1} \in \mathbb{N}$
Goals:	$a \in \mathbb{N}, a \in \{\textit{number_literal_0}\}$

There is a single variable declared and hence a single semantic binding. The

hypotheses are those facts about the surrounding environment that must be true for our binding to make sense and are derived from the references made in the expression; for instance, when we reference \mathbb{N} we must import the conditions that give it meaning, on this occasion that the natural numbers are a subset of all numbers, and that the set of all numbers is itself a subset of an arbitrary given type. The goals are the clauses that give our binding its own meaning, and represent the constraints on our bound variable.

Once we have determined the semantic binding environment for our schema construction expression, we can deduce the following semantic binding environment for the axiomatic description paragraph.

Bound Object:	<i>mySchema</i>
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A}, \textit{number_literal_0} \in \mathbb{N},$ $\textit{number_literal_1} \in \mathbb{N}$
Goals:	$\textit{mySchema} \in \{\langle a \rangle\}$

Again there is only a single binding as we only have one variable at the paragraph level (*mySchema*). What is interesting to note is that the goal for the binding refers directly to the binding of *a*. That is, *mySchema* is a member of the set of possible bindings of *a*. We will use the terminology $\langle a \rangle$ to refer directly to the semantic binding for *a* (shown above).

The semantic binding environment for the section, *mySection*, is the singleton set containing the semantic binding of the axiomatic description paragraph.

Once we have defined a semantic binding environment for a section we are able to proceed with the generation of proof obligations. We can illustrate this briefly by again considering the following machine.

```
MACHINE myNumber
TYPE machineWithInit
SECTION standard_toolkit
```

STATE

 $a : \mathbb{N}$

INITIALIZATION

 $a = 0$
END *myNumber*

When we instantiate the proof obligation we create relational definitions for our two clauses ('STATE' and 'INITIALIZATION'). However, instead of using the above specification we use the semantic bindings of their associated paragraphs. From these bindings we are able to derive the bindings for the variables that the construct's configuration states form the relational definition. (In this instance a). If we followed the process detailed in the remainder of this section we will retrieve the following bindings (where the first is for the 'STATE' clause and the second for the 'INITIALIZATION' clause).

Bound Object:	a
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A}), \mathbb{N} \in \mathbb{P} \ \mathbb{A}, number_literal_0 \in \mathbb{N},$ $number_literal_1 \in \mathbb{N}$
Goals:	$a \in \mathbb{N}$

Bound Object:	a
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A}), \mathbb{N} \in \mathbb{P} \ \mathbb{A}, number_literal_0 \in \mathbb{N},$ $number_literal_1 \in \mathbb{N}$
Goals:	$a \in \mathbb{N}, a \in \{number_literal_0\}$

If we used these bindings to instantiate our initialization proof obligation we would create the following (the specific process by which we do this is described in section 6.3).

$$\begin{aligned}
& \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \\
& \mathbb{N} \in \mathbb{P} \mathbb{A} \\
& \textit{number_literal_0} \in \mathbb{N} \\
& \textit{number_literal_1} \in \mathbb{N} \\
& \vdash? \\
& \exists u \bullet u \in \{a : \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N}\} \\
& \quad \wedge u \in \{a : \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge a \in \{\textit{number_literal_0}\}\}
\end{aligned}$$

The remainder of this section details the rules required to mechanize the generation of semantic binding environments for a specification.

Functions

Before defining the rules that we use to create the semantic binding environment for a Z specification, we first present a number of functions that we will use in the definition of those rules. Table 6.2 provides an overview of all the functions that we use in the creation of semantic bindings and their environments.

Table 6.2: Summary of functions used in the creation of semantic bindings and their environments

Function	Arguments	Result
ZF()	A Z variable or predicate.	The argument translated into its ZF language equivalent.
Q()	A semantic binding environment and a Z variable.	The semantic binding of the Z variable (assuming it is contained) within the semantic binding environment.

Function	Arguments	Result
$R^{sb}()$	A source sequence of ZF objects, a target sequence of ZF objects and a semantic binding.	The semantic binding where all instances of ZF objects in the source sequence that form any part of the semantic binding have been replaced by the equivalently placed ZF object in the target sequence.
$R^{set}()$	A source sequence of ZF objects, a target sequence of ZF objects and a set of ZF objects.	The set of ZF objects where all instances of ZF objects in the source sequence that form part of or any of the set's ZF objects have been replaced by the equivalently placed ZF object in the target sequence.
$R^{zf}()$	A source sequence of ZF objects, a target sequence of ZF objects and a ZF object.	The ZF object where all instances of ZF objects in the source sequence that form part of or the entirety of the ZF object have been replaced by the equivalently placed ZF object in the target sequence.
$\tau()$	A Z variable or expression.	The ZF representation of the argument's type.
$S^s()$	A Z section.	The semantic binding environment generated from the section.
$S^d()$	A Z paragraph and a parent semantic binding environment.	The semantic binding environment generated from the paragraph in the context of the parent semantic binding environment.

Function	Arguments	Result
$S^p()$	A Z predicate and a parent semantic binding environment.	A tuple containing the ZF language equivalent of the Z predicate and a list of references to Z variables used in the predicate.
$S^e()$	A Z expression and a parent semantic binding environment.	The semantic binding environment generated from the expression in the context of the parent semantic binding environment.
$ref()$	A Z expression.	A list of Z variables referenced by the expression.
$G()$	A Z variable.	A sequence of the ZF objects that form the parameters to the generic paragraph in which the variable is declared
$G^\tau()$	A Z variable.	A sequence of the ZF carrier sets that form the types to the parameters of the generic paragraph in which the variable is declared

Rule 6.1 - Translating to the ZF Language:

We define a function, $ZF()$, that allows us to translate terms into the ZF language described in section 6.2.1. Typically, these translations simply involve the instantiation of the appropriate class; when dealing with variables the function will retrieve an existing translation where possible. For example, we may have the following Z fragments.

a

$$\forall x : \mathbb{N} \mid x \in a \bullet x \in b$$

Calling $\text{ZF}(a)$ for the first time would create an instance of ZFName , which for example we could refer to as ZFName001 .

Calling $\text{ZF}(\forall x : \mathbb{N} \mid x \in a \bullet x \in b)$ would lead to the creation of a number of ZF language objects. Firstly, we translate the bound variables and create an instance of ZFName to refer to x . We would then translate the two predicates, known as the constraint and the predicate, the constraint being the predicate part of the quantification that is within the schema text and the predicate being the remaining predicate part. Note that, the typing of the bound variable is moved from the declaration part of the schema text to the constraint in line with the guidelines presented in section 6.2.1. Translating the constraint therefore, entails creating a $\text{ZFPredicateConjunction}$ object, and two $\text{ZFPredicateMembership}$ objects ($x \in \mathbb{N}$ and $x \in a$; the existing ZFNames for x , a and \mathbb{N} would also be retrieved). Translating the predicate involves the creation of a single $\text{ZFPredicateMembership}$ object ($x \in b$). Once the component parts have been translated they are used to create an instance of ZFPredicateUniv . This instance would have an object structure as shown in figure 6.2 on the following page.

Rule 6.2 - Querying the Semantic Binding Environment:

We define a function, $\mathbf{Q}()$, that allows us to query a semantic binding environment so that we can determine the semantic binding (that is the triple formed from its ZF object, hypotheses and goals – (z, h, p)) for a particular Z variable.

$$(z, h, p) \in \Sigma \wedge z = \text{ZF}(i) \longrightarrow \mathbf{Q}(\Sigma, i) = (z, h, p)$$

For example, the following set of semantic bindings could be a valid semantic binding environment, Σ_{abc} .

$$\Sigma_{abc} = \{(\text{ZFName}_1, h_1, p_1), (\text{ZFName}_2, h_2, p_2), (\text{ZFName}_3, h_3, p_3)\}$$

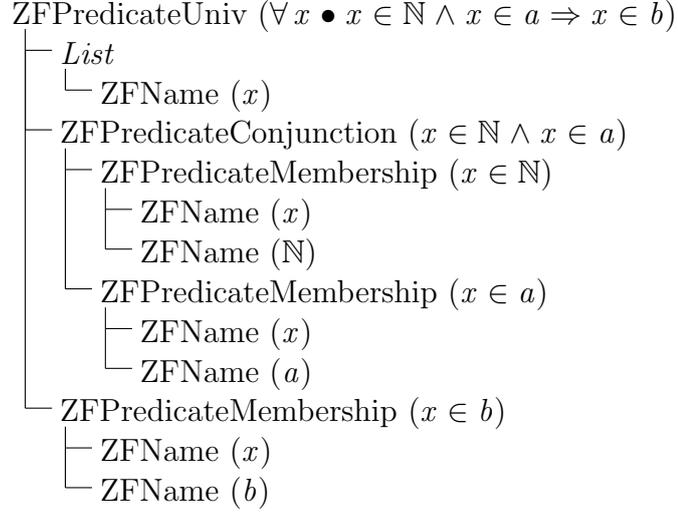


Figure 6.2: Example of ZFObject tree for universal quantification

If we then queried this semantic binding environment with the Z variable, b – where the ZF object used to represent b is $ZFName2$ – we would retrieve the semantic binding associated with that variable. This can be seen more clearly as follows.

$$Q(\Sigma_{abc}, b) = (ZFName_2, h_2, p_2)$$

Rule 6.3 - ZF Object Replacement Rules:

We define three functions that allow us to perform substitution (or renaming) of ZF objects. Each of these functions takes a sequence of source ZFObjects and a sequence of target ZF objects where each source ZF object is to be replaced with the equivalently-placed target ZF object. In addition $R^{sb}()$ takes a semantic binding and will perform substitutions on that semantic binding's ZF object, hypotheses and goals. Instead of a semantic binding, $R^{set}()$ takes a set of ZF objects and attempts to form a substitution on every member of that set. Similarly, $R^{zf}()$ takes a single ZF object and will perform a substitution on either any matching part of the object or the entire object as appropriate. We assume that the arguments to the function are not contradictory (that is, no attempt is made to recursively replace objects

with each other), and will make the earliest substitutions possible.

The rules for the $R^{sb}()$ function, are simply rewriting rules, and this function is simply an abbreviation. When $R^{sb}()$ is applied to a semantic binding we apply $R^{zf}()$ to its ZF object and $R^{set}()$ to its sets of hypotheses and goals (using the same source and target sequences as arguments).

$$R^{sb}(s, t, (z, h, p), \Sigma) = (R^{zf}(s, t, z, \Sigma), R^{set}(s, t, h, \Sigma), R^{set}(s, t, p, \Sigma))$$

Similarly, the rules for the $R^{set}()$ function allow us to perform the required substitutions for all members of a set of ZF Objects. This involves simply the application of the $R^{zf}()$ function to each of the set's members (again, the same source and target sequences are used as arguments).

$$R^{set}(s, t, \emptyset, \Sigma) = \emptyset$$

$$R^{set}(s, t, \{z\} \cup zobjects, \Sigma) = \{R^{zf}(s, t, z, \Sigma)\} \cup R^{set}(s, t, zobjects, \Sigma)$$

The rules for the $R^{zf}()$ function, give instructions to handle substitution for every element of the ZF language. Note that, as ZFName is the superclass of a number of other classes we can use a single rule to handle all ZFObject instances that belong to that class (type).

If the object on which the substitution is to be performed is the emptyset then no substitution can take place, either on the entire object or any of its constituent parts (of which, of course, there are none). The result of the application is the emptyset.

$$R^{zf}(s, t, ZF(\emptyset), \Sigma) = ZF(\emptyset)$$

If the object on which the substitution is to be performed is one of the objects which is to be substituted, then the entire object is replaced. The ZF object that is used as a replacement has the same index in the target sequence as the original ZF object possessed in the source sequence. The result of the

application is the replaced ZF object.

$$\langle x, z \rangle \in s \wedge \langle x, y \rangle \in t \longrightarrow \mathbf{R}^{zf}(s, t, z, \Sigma) = y$$

If the object on which the substitution is to be performed is not one of the objects which is to be substituted (that is $z \notin \text{ran}(s)$), then we will attempt to perform a substitution on the component parts of that object. If the object is a ZFName then it has no component parts so no substitution can be performed. The results of the application is the original ZF object.

$$z \notin \text{ran}(s) \longrightarrow \mathbf{R}^{zf}(s, t, z : \text{ZFName}, \Sigma) = z$$

If the object on which the substitution is to be performed is a sequence of objects then we attempt to perform the substitution on each of the objects contained within that sequence. The result of the application is the sequence formed from applying $\mathbf{R}^{zf}()$ to each of the objects in the original sequence.

$$\begin{aligned} & \text{ZF}(x_1, \dots, x_n) \notin \text{ran}(s) \\ & \longrightarrow \mathbf{R}^{zf}(s, t, \text{ZF}((x_1, \dots, x_n)), \Sigma) \\ & \quad = \text{ZF}((\mathbf{R}^{zf}(s, t, x_1, \Sigma), \dots, \mathbf{R}^{zf}(s, t, x_n, \Sigma))) \end{aligned}$$

If the object on which the substitution is to be performed is a powerset then we attempt to perform the substitution on the component object. The result of the application is the powerset with the component formed from applying $\mathbf{R}^{zf}()$ to the original component object.

$$\begin{aligned} & \text{ZF}(\mathbb{P}(x)) \notin \text{ran}(s) \\ & \longrightarrow \mathbf{R}^{zf}(s, t, \text{ZF}(\mathbb{P}(x)), \Sigma) = \text{ZF}(\mathbb{P}(\mathbf{R}^{zf}(s, t, x, \Sigma))) \end{aligned}$$

If the object on which the substitution is to be performed is a conjunction of objects then we attempt to perform the substitution on each of the objects contained within that conjunction. The result of the application is the conjunction formed from applying $\mathbf{R}^{zf}()$ to each of the objects in the original

conjunction.

$$\begin{aligned} & \text{ZF}(x_1 \wedge \dots \wedge x_n) \notin \text{ran}(s) \\ & \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(x_1 \wedge \dots \wedge x_n), \Sigma) \\ & \quad = \text{ZF}(\text{R}^{zf}(s, t, x_1, \Sigma) \wedge \dots \wedge \text{R}^{zf}(s, t, x_n, \Sigma)) \end{aligned}$$

If the object on which the substitution is to be performed is a disjunction of objects then we attempt to perform the substitution on each of the objects contained within that disjunction. The result of the application is the disjunction formed from applying $\text{R}^{zf}()$ to each of the objects in the original disjunction.

$$\begin{aligned} & \text{ZF}(x_1 \vee \dots \vee x_n) \notin \text{ran}(s) \\ & \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(x_1 \vee \dots \vee x_n), \Sigma) \\ & \quad = \text{ZF}(\text{R}^{zf}(s, t, x_1, \Sigma) \vee \dots \vee \text{R}^{zf}(s, t, x_n, \Sigma)) \end{aligned}$$

If the object on which the substitution is to be performed is an equality comparison between objects then we attempt to perform the substitution on the two objects involved. The result of the application is the equality formed from applying $\text{R}^{zf}()$ to each of the original component objects.

$$\begin{aligned} & \text{ZF}(x_1 = x_2) \notin \text{ran}(s) \\ & \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(x_1 = x_2), \Sigma) \\ & \quad = \text{ZF}(\text{R}^{zf}(s, t, x_1, \Sigma) = \text{R}^{zf}(s, t, x_2, \Sigma)) \end{aligned}$$

If the object on which the substitution is to be performed is an existential quantification then we attempt to perform the substitution on the bound variable and the constraining predicate. The result of the application is the existential quantification formed from applying $\text{R}^{zf}()$ to each of the original bound variables and the original constraining predicate.

$$\begin{aligned} & \text{ZF}(\exists x \bullet p) \notin \text{ran}(s) \\ & \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(\exists x \bullet p), \Sigma) \\ & \quad = \text{ZF}(\exists \text{R}^{zf}(s, t, x, \Sigma) \bullet \text{R}^{zf}(s, t, p, \Sigma)) \end{aligned}$$

$$\begin{aligned}
& \text{ZF}(\exists_1 x \bullet p) \notin \text{ran}(s) \\
& \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(\exists_1 x \bullet p), \Sigma) \\
& \quad = \text{ZF}(\exists_1 \text{R}^{zf}(s, t, x, \Sigma) \bullet \text{R}^{zf}(s, t, p, \Sigma))
\end{aligned}$$

If the object on which the substitution is to be performed is an universal quantification then we attempt to perform the substitution on the bound variable, the constraining predicate and the predicate. The result of the application is the universal quantification formed from applying $\text{R}^{zf}()$ to each of the original bound variables, the original constraining predicate and the original predicate.

$$\begin{aligned}
& \text{ZF}(\forall x \bullet c \Rightarrow p) \notin \text{ran}(s) \\
& \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(\forall x \bullet c \Rightarrow p), \Sigma) \\
& \quad = \text{ZF}(\forall \text{R}^{zf}(s, t, x, \Sigma) \\
& \quad \quad \bullet \text{R}^{zf}(s, t, c, \Sigma) \Rightarrow \text{R}^{zf}(s, t, p, \Sigma))
\end{aligned}$$

If the object on which the substitution is to be performed is an implication between objects then we attempt to perform the substitution on the two objects involved. The result of the application is the implication formed from applying $\text{R}^{zf}()$ to each of the original component objects.

$$\begin{aligned}
& \text{ZF}(x_1 \Rightarrow x_2) \notin \text{ran}(s) \\
& \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(x_1 \Rightarrow x_2), \Sigma) \\
& \quad = \text{ZF}(\text{R}^{zf}(s, t, x_1, \Sigma) \Rightarrow \text{R}^{zf}(s, t, x_2, \Sigma))
\end{aligned}$$

If the object on which the substitution is to be performed is a membership test between objects then we attempt to perform the substitution on the two objects involved. The result of the application is the membership predicate formed from applying $\text{R}^{zf}()$ to each of the original component objects.

$$\begin{aligned}
& \text{ZF}(x_1 \in x_2) \notin \text{ran}(s) \\
& \longrightarrow \text{R}^{zf}(s, t, \text{ZF}(x_1 \in x_2), \Sigma) \\
& \quad = \text{ZF}(\text{R}^{zf}(s, t, x_1, \Sigma) \in \text{R}^{zf}(s, t, x_2, \Sigma))
\end{aligned}$$

If the object on which the substitution is to be performed is a negation then we attempt to perform the substitution on the component object. The result

of the application is the negation with the component formed from applying $R^{zf}()$ to the original component object.

$$\begin{aligned} & \text{ZF}(\neg(x)) \notin \text{ran}(s) \\ & \longrightarrow R^{zf}(s, t, \text{ZF}(\neg(x)), \Sigma) = \text{ZF}(\neg(R^{zf}(s, t, x, \Sigma))) \end{aligned}$$

If the object on which the substitution is to be performed is a parenthesized object then we attempt to perform the substitution on the component object. The result of the application is the parenthesization of the component formed from applying $R^{zf}()$ to the original component object.

$$\begin{aligned} & \text{ZF}((x)) \notin \text{ran}(s) \\ & \longrightarrow R^{zf}(s, t, \text{ZF}((x)), \Sigma) = \text{ZF}((R^{zf}(s, t, x, \Sigma))) \end{aligned}$$

If the object on which the substitution is to be performed is the truth constant then no substitution can take place, either on the entire object or any of its constituent parts (which, of course, there are none). The result of the application is the truth constant.

$$R^{zf}(s, t, \text{ZF}(\text{true}), \Sigma) = \text{ZF}(\text{true})$$

If the object on which the substitution is to be performed is a set extension then we attempt to perform the substitution on the semantic bindings contained within. The result of the application is the set extension formed from applying $R^{sb}()$ to each of the original semantic bindings.

$$\begin{aligned} & \text{ZF}(\{(z_1, h_1, p_1), \dots, (z_n, h_n, p_n)\}) \notin \text{ran}(s) \\ & \longrightarrow R^{zf}(s, t, \text{ZF}(\{(z_1, h_1, p_1), \dots, (z_n, h_n, p_n)\}), \Sigma) \\ & \quad = \text{ZF}(\{R^{sb}(s, t, (z_1, h_1, p_1), \Sigma), \dots, R^{sb}(s, t, (z_n, h_n, p_n), \Sigma)\}) \end{aligned}$$

If the object on which the substitution is to be performed is a set separation then we attempt to perform the substitution on the bound variables, their type and the constraining predicate. The result of the application is the set separation formed from applying $R^{sb}()$ to each of the original bound variables,

their type and the original constraint.

$$\begin{aligned}
& \text{ZF}(\{x : \tau \bullet p\}) \notin \text{ran}(s) \\
& \longrightarrow \text{R}^{\text{zf}}(s, t, \text{ZF}(\{x : \tau \bullet p\}), \Sigma) \\
& \quad = \text{ZF}(\{\text{R}^{\text{zf}}(s, t, x, \Sigma) : \text{R}^{\text{zf}}(s, t, \tau, \Sigma) \\
& \quad \quad \bullet \text{R}^{\text{zf}}(s, t, p, \Sigma)\})
\end{aligned}$$

Rule 6.4 - Retrieving a Variable or Expression's Type:

We declare an abstract function, $\tau()$, that allows us to determine the ZF representation of the type of an expression or variable. We won't present a definition for this function as it simply retrieves the type that we associated with each variable and expression in the type checking phase of the parsing process and applies the $\text{ZF}()$ function we have described previously.

Sections

We now progress to the rules for the translation of a Z specification. We consider first the translation of sections. The semantic binding environment of each section provides the starting point for the generation of proof obligations. We define the function $\text{S}^{\text{s}}()$ which when applied to a Z section will result in the semantic binding environment for that section. The rules that define the behaviour of $\text{S}^{\text{s}}()$ are given below.

Rule 6.5 - Prelude Section:

$$\begin{aligned}
& \text{S}^{\text{d}}(d_1, \emptyset) = \sigma_1 \\
& \wedge \text{S}^{\text{d}}(d_2, \sigma_1) = \sigma_2 \\
& \wedge \dots \\
& \wedge \text{S}^{\text{d}}(d_n, \sigma_1 \oplus \sigma_2 \oplus \dots \oplus \sigma_{n-1}) = \sigma_n \\
& \longrightarrow \text{S}^{\text{s}}(\text{section prelude parents END } d_1 \ d_2 \ \dots \ d_n) = \sigma_1 \oplus \sigma_2 \oplus \dots \oplus \sigma_n
\end{aligned}$$

The semantic binding environment of the prelude section is equal to the overridden union of the semantic binding environments $(\sigma_1 \dots \sigma_n)$ of its constituent paragraphs $(d_1 \dots d_n)$. We calculate the semantic binding environment for each paragraph by applying $\mathbf{S}^p()$ to the paragraph's definition and the contextual semantic binding environment. We discussed in section 4.5 how our parser relies on the ordering of Z paragraphs. Here again we rely on this ordering, each paragraph is processed in the contextual semantic binding environment produced from the overridden union of the paragraphs prior to it.

Rule 6.6 - Inheriting Section:

$$\begin{aligned}
& \mathbf{S}^s(\text{section prelude } \dots) = \Sigma_0 \\
& \wedge \mathbf{S}^s(\text{section } p_1 \dots) = \Sigma_1 \\
& \wedge \mathbf{S}^s(\text{section } p_2 \dots) = \Sigma_2 \\
& \wedge \dots \\
& \wedge \mathbf{S}^s(\text{section } p_m \dots) = \Sigma_m \\
& \wedge \mathbf{S}^d(d_1, \Sigma_0 \oplus \Sigma_1 \oplus \Sigma_2 \oplus \dots \oplus \Sigma_m) = \sigma_1 \\
& \wedge \mathbf{S}^d(d_2, \Sigma_0 \oplus \Sigma_1 \oplus \Sigma_2 \oplus \dots \oplus \Sigma_m \oplus \sigma_1) = \sigma_2 \\
& \wedge \dots \\
& \wedge \mathbf{S}^d(d_n, \Sigma_0 \oplus \Sigma_1 \oplus \Sigma_2 \oplus \dots \oplus \Sigma_m \oplus \sigma_1 \oplus \sigma_2 \oplus \dots \oplus \sigma_{n-1}) = \sigma_n \\
& \longrightarrow \mathbf{S}^s(\text{section } i \text{ parents } p_1, p_2, \dots, p_m \text{ END } d_1 d_2 \dots d_n) \\
& \quad = \sigma_1 \oplus \sigma_2 \oplus \dots \oplus \sigma_n
\end{aligned}$$

The semantic binding environment of an inheriting section is also equal to the overridden union of the semantic binding environment of its constituent paragraphs. In this instance however, we will determine the semantic binding environments of the paragraphs in the context of the overridden union of the semantic binding environments $(\Sigma_1 \dots \Sigma_m)$ produced by applying $\mathbf{S}^s()$ to the section's parent sections $(p_1 \dots p_m)$, plus the semantic binding environments $(\sigma_1 \dots \sigma_n)$ produced by applying $\mathbf{S}^d()$ to any previous paragraphs $(d_1 \dots d_n)$.

Paragraphs

We now consider the creation of semantic binding environments for paragraphs. We define the function $\mathbf{S}^d()$ the application of which to a paragraph

and a contextual semantic binding environment, will result in the semantic binding environment for that paragraph.

Rule 6.7 - Given Type Paragraph:

$$\begin{aligned} & \mathbf{S}^d([i_1, \dots, i_n], \Sigma) \\ &= \{(\mathbf{ZF}(i_1), \emptyset, \{\mathbf{ZF}(i_1 \in \mathbb{P}(j_1))\}), \dots, (\mathbf{ZF}(i_n), \emptyset, \{\mathbf{ZF}(i_n \in \mathbb{P}(j_n))\})\} \end{aligned}$$

When processing a given type paragraph we create an arbitrary carrier set (instance of $\mathbf{ZFCarrierSet} - j_1, \dots, j_n$) for each given type (i_1, \dots, i_n) . We then add a semantic binding to the environment for each given type, where the binding name is the \mathbf{ZFName} instance created from the given type's $\mathbf{ZVariable}$ instance. There are no hypotheses for these bindings and the goal is that each given type is a subset of its assigned carrier set.

Rule 6.8 - Axiomatic Description Paragraph:

$$\mathbf{S}^e(e, \Sigma) = \sigma \longrightarrow \mathbf{S}^d(\mathbf{AX} \ e \ \mathbf{END}, \Sigma) = \sigma$$

The semantic binding environment of an axiomatic description paragraph is equivalent to the semantic binding environment of the expression it contains. We retrieve the semantic binding environment of the expression by applying $\mathbf{S}^e()$ to the Z expression and the contextual semantic binding environment.

Rule 6.9 - Generic Axiomatic Description Paragraph:

$$\begin{aligned} & \sigma_p = \{(\mathbf{ZF}(i_1), \emptyset, \{\mathbf{ZF}(i_1 \in \mathbb{P}(j_1))\}), \dots, (\mathbf{ZF}(i_n), \emptyset, \{\mathbf{ZF}(i_n \in \mathbb{P}(j_n))\})\} \\ & \wedge \mathbf{S}^e(e, \Sigma \oplus \sigma_p) = \sigma_e, \\ & \longrightarrow \mathbf{S}^d(\mathbf{GENAX} \ [i_1, \dots, i_n] \ e \ \mathbf{END}, \Sigma) = \sigma_e \end{aligned}$$

With a generic axiomatic description paragraph the semantic binding environment is also equivalent to the semantic binding environment of the expression it contains. In this instance, however, the contextual semantic binding environment – in which the expression’s semantic binding environment is determined – is augmented by the bindings of the declared generic types (i_1, \dots, i_n) . For each generic type we create an arbitrary carrier set (j_1, \dots, j_n) and create a binding with no hypotheses and a goal that states that the generic type is a subset of that carrier set.

Rule 6.10 - Conjecture Paragraph:

$$S^d(\vdash? \ p \ \text{END}, \Sigma) = \emptyset$$

Rule 6.11 - Generic Conjecture Paragraph:

$$S^d([i_1, \dots, i_n] \vdash? \ p \ \text{END}, \Sigma) = \emptyset$$

Conjecture paragraphs contain only theories about the specification and do not have any bindings. The semantic binding environments for both conjecture paragraphs and generic conjecture paragraphs are equivalent to the emptyset.

Predicates

Now we consider how predicates are handled during the generation of semantic binding environments. Whilst we do not create semantic bindings for predicates themselves, they can contain expressions which have references that must be included in the semantic binding environment of the expression in which the predicate is used. The processing of predicates also allows them to be translated into the ZF language, which means that they can be added directly to the hypotheses and goals of other bindings. We define a function

$S^p()$ whose application to a predicate and a contextual semantic binding environment results in a tuple containing the ZF language equivalent of the predicate and a set of references that are required in the construction of that predicate (that is a list of variable which are referred to in the predicate's definition). We declare an abstract function, $\text{ref}()$ that will return a set of references used in an expression. We won't define this function here as it simply parses an expression looking for all reference instances and adds them to a set.

Rule 6.12 - Membership Predicate:

$$\begin{aligned} S^e(e_1, \Sigma) = \{(z_1, h_1, p_1)\} \wedge S^e(e_2, \Sigma) = \{(z_2, h_2, p_2)\} \\ \longrightarrow S^p(e_1 \in e_2, \Sigma) = (\text{ZF}(e_1 \in e_2 \wedge p_1 \wedge p_2), \text{ref}(e_1) \cup \text{ref}(e_2)) \end{aligned}$$

The membership predicate is converted into its ZF language equivalent, and the hypotheses from member and set expressions are merged to give the predicate's hypotheses. Any outstanding goals from the referenced expressions must now be included as part of the predicate. For instance, if we have the following membership predicate.

$$a \in \{0\}$$

The bindings for a and 0 will each have an outstanding goal when we calculate the ZF language equivalent of our membership predicate. These will be $a \in \mathbb{N}$ and $0 \in \mathbb{N}$ respectively. When we translate our membership predicate we must incorporate these goals. Therefore, the ZF language equivalent of the above predicate will be the following.

$$a \in \mathbb{N} \wedge 0 \in \mathbb{N} \wedge a \in \{0\}$$

In order that the appropriate hypotheses will also be included the application of $S^p()$ produces a list of the referenced variables. The list of references for the above membership predicate would simply be $\{a, 0\}$. When this predicate is incorporated within a semantic binding the hypotheses associated with these

variables will be imported.

Rule 6.13 - Truth Predicate:

$$\mathbf{S}^{\mathbf{P}}(\text{true}) = (\mathbf{ZF}(\text{true}), \emptyset)$$

The truth predicate is simply converted to the ZF language equivalent, there are no references required for its use.

Rule 6.14 - Negation Predicate:

$$\begin{aligned} \mathbf{S}^{\mathbf{P}}(p, \Sigma) &= (z, r) \\ \longrightarrow \mathbf{S}^{\mathbf{P}}(\neg p, \Sigma) &= (\mathbf{ZF}(\neg z), r) \end{aligned}$$

The references of the negation predicate are simply the references of the predicate being negated.

Rule 6.15 - Conjunction Predicate:

$$\begin{aligned} \mathbf{S}^{\mathbf{P}}(p_1, \Sigma) = (z_1, r_1) \wedge \mathbf{S}^{\mathbf{P}}(p_2, \Sigma) = (z_2, r_2) \\ \longrightarrow \mathbf{S}^{\mathbf{P}}(p_1 \wedge p_2, \Sigma) = (\mathbf{ZF}(z_1 \wedge z_2), r_1 \cup r_2) \end{aligned}$$

The references of the conjunction predicate are the union of the references of the predicates involved in the conjunction.

Rule 6.16 - Universal Quantification Predicate:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= (\{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}, \perp) \\ \wedge \mathbf{S}^{\mathbf{P}}(p, \Sigma \oplus \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}) &= (z, r) \\ \longrightarrow \mathbf{S}^{\mathbf{P}}(\forall e \bullet p, \Sigma) \\ &= (\mathbf{ZF}(\forall y_1, \dots, y_n \bullet o_1 \wedge \dots \wedge o_n \Rightarrow z), \mathbf{ref}(e) \cup r) \end{aligned}$$

With a universal quantification predicate, the ZF language representation of the predicate is derived from the semantic binding environment of the expression in combination with the constituent predicate. The bound objects in the

expression's environment $(y_1 \dots y_n)$ become the bound variables of the quantification and the conjunction of the goals of those bound objects $(o_1 \dots o_n)$ become its constraining predicate. The constituent predicate is translated in a contextual semantic binding environment overridden with that of the expression and completes the representation of the universal quantification predicate in the ZF language

The references of the universal quantification predicate are the union of the references required by all of the bound objects in the expression's semantic binding environment and the references of the constituent predicate.

Rule 6.17 - Unique Existential Quantification Predicate:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= (\{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}, \perp) \\ \wedge \mathbf{S}^p(p, \Sigma \oplus \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}) &= (z, r) \\ \longrightarrow \mathbf{S}^p(\exists_1 e \bullet p, \Sigma) & \\ &= (\mathbf{ZF}(\exists_1 y_1, \dots, y_n \bullet o_1 \wedge \dots \wedge o_n \wedge z), \mathbf{ref}(e) \cup r) \end{aligned}$$

With the unique existential quantification predicate, the ZF language representation of the predicate is derived from the semantic binding environment of the expression in combination with the constituent predicate. The bound objects in the expression's environment $(y_1 \dots y_n)$ become the bound variables of the quantification and the conjunction of the goals of those bound objects $(o_1 \dots o_n)$, and the translated constituent predicate, becomes the predicate. Again the constituent predicate is translated in a semantic binding environment overridden by that of the expression.

The references of the universal quantification predicate are the union of the references required by all of the bound objects in the expression's semantic binding environment, and the references of the constituent predicate.

Expressions

Next we consider the generation of semantic binding environments for expressions. We define the function $\mathbf{S}^e()$ which when applied to an expression and a contextual semantic binding environment, results in the semantic binding

environment of that expression.

Rule 6.18 - Reference Expression:

$$Q(\Sigma, i) = (z, h, p) \longrightarrow S^e(i, \Sigma) = \{(z, h \cup p, \emptyset)\}$$

With a reference expression, the semantic binding of the referenced variable is retrieved from the contextual semantic binding environment. The semantic binding of the referenced variable is added to the semantic binding environment. As we are using the variable as a reference however, the goals are now combined with the hypotheses, and the goals of the new binding are empty.

Rule 6.19 - Generic Instantiation Expression:

$$\begin{aligned} Q(\Sigma, i) &= (z, h, p) \\ \wedge S^e(e_1, \Sigma) &= \{(z_{1_1}, h_{1_1}, p_{1_1}), \dots, (z_{1_n}, h_{1_n}, p_{1_n})\} \\ \wedge \dots \\ \wedge S^e(e_n, \Sigma) &= \{(z_{n_1}, h_{n_1}, p_{n_1}), \dots, (z_{n_n}, h_{n_n}, p_{n_n})\} \\ \longrightarrow S^e(i[e_1, \dots, e_n], \Sigma) \\ &= \{ZF(z[z_{1_1}, \dots, z_{1_n}, \dots, z_{n_1}, \dots, z_{n_n}]), \\ &\quad R^{set}(\langle z \rangle \wedge G(i) \wedge G^\tau(i), \\ &\quad \langle ZF(z[z_{1_1}, \dots, z_{1_n}, \dots, z_{n_1}, \dots, z_{n_n}]), z_{1_1}, \dots, z_{1_n}, \dots, z_{n_1}, \dots, z_{n_n}, \\ &\quad ZF(\tau(z_{1_1})), \dots, ZF(\tau(z_{1_n})), \dots, ZF(\tau(z_{n_1})), \dots, ZF(\tau(z_{n_n})) \rangle), \\ &\quad h \cup p \cup h_{1_1} \cup \dots \cup h_{1_n} \cup \dots \cup h_{n_1} \cup \dots \cup h_{n_n}), \\ &\quad \emptyset\} \end{aligned}$$

When calculating the semantic binding environment for a generic instantiation expression we translate the expression into a standard reference expression through the instantiation of the generic objects. Note that, this semantic instantiation is distinct from the instantiation of generic type that we performed in the type checking phase of the parsing process.

The generation of the semantic binding for the generic instantiation expression is more complex than most other expressions. It is determined from

the semantic binding of the referenced variable (i) (retrieved from the contextual semantic binding environment) and those contained in the semantic binding environments of each of the expressions that form the instantiation parameters ($e_1 \dots e_n$). The ZF object created for the binding is distinct from the generic ZF object. That is, we create a new ZF object for every specific instantiation of a generic variable; even if we use the same parameters to instantiate a generic variable, the resultant ZF objects will not be identical. The hypotheses for the binding are formed from the union of the hypotheses and goals of the referenced generic variable and the hypotheses of each of the bindings belonging to the semantic binding environments of the instantiation parameter's expressions. In order for these hypotheses to make sense however, it is necessary to replace all references to the generic variable with references to the instantiated variable. Similarly, we need to replace any references to each generic type with a reference to the ZF object of the appropriate parameter. Note that, in some cases – for instance when set separation expressions are used – it is also necessary to replace the types used within a hypothesis, that is replacing the carrier set associated with a generic type with the type of the instantiation parameter. To assist with this, we define two abstract functions, $G()$ and $G^\tau()$ each of which produces a sequence when applied to a Z variable. The application of the first will lead to a sequence containing the ZF objects representing the generic parameters of the paragraph in which the variable is defined. The second leads to a sequence containing the ZF carrier sets that form the types of those same generic parameters.

We will illustrate the creation of the semantic binding environment for a generic instantiation expression through a simple example. Consider the declaration of the variable *myGenericVar* within a generic axiomatic description paragraph.

$$\boxed{\begin{array}{l} [X] \\ \hline \hline \text{myGenericVar} : \mathbb{P} X \end{array}}$$

Then consider the instantiation of that variable within the declaration of another variable in a separate (non-generic) paragraph.

$$| \quad myVar : myGenericVar[\mathbb{N}]$$

Now consider the calculation of the semantic binding environment for the generic instantiation expression. The first step is to retrieve the semantic binding for the variable to be instantiated from the contextual semantic binding environment.

$$\begin{aligned} & Q(\Sigma, myGenericVar) \\ &= \{(\mathbf{ZF}(myGenericVar), \emptyset, \{\mathbf{ZF}(myGenericVar \in \mathbb{P} X)\})\} \end{aligned}$$

We can then generate the semantic binding environment of the single expression used as an instantiation parameter.

$$S^e(\mathbb{N}, \Sigma) = \{(\mathbf{ZF}(\mathbb{N}), \{\mathbf{ZF}(\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\}, \emptyset)\}$$

We now proceed to use this information to generate the semantic binding environment for the generic instantiation expression (note that, as we don't have any reference to types in our hypotheses we do not consider the replacement of carrier sets with instantiated sets).

$$G(myGenericVar) = \langle \mathbf{ZF}(X) \rangle$$

$$\begin{aligned} & S^e(myGenericVar[\mathbb{N}], \Sigma) \\ &= \{(\mathbf{ZF}(myGenericVar[\mathbb{N}]), \\ & \quad R^{set}(\langle \mathbf{ZF}(myGenericVar), \mathbf{ZF}(X) \rangle, \\ & \quad \langle \mathbf{ZF}(myGenericVar[\mathbb{N}]), \mathbf{ZF}(\mathbb{N}) \rangle, \\ & \quad \{\mathbf{ZF}(myGenericVar \in \mathbb{P} X)\} \cup \{\mathbf{ZF}(\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\} \\ & \quad \emptyset)\} \\ &= \{(\mathbf{ZF}(myGenericVar[\mathbb{N}]), \\ & \quad \{(\mathbf{ZF}(myGenericVar[\mathbb{N}] \in \mathbb{P} \mathbb{N}, \mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\}, \\ & \quad \emptyset)\} \end{aligned}$$

Firstly, we determine the generic parameters of our referenced variable, and then substitute the data we have gathered into the definition of the generic

instantiation expression's semantic binding environment. We then use the replace rules (see rule 6.3) to substitute all references to the generic types, and we are left with a semantic binding environment containing a single binding that conveys the meaning of the instantiated variable.

Rule 6.20 - Set Extension Expression:

$$\begin{aligned}
& \mathbf{S}^e(e_1, \Sigma) = \{(z_1, h_1, p_1)\} \\
& \wedge \dots \\
& \wedge \mathbf{S}^e(e_n, \Sigma) = \{(z_n, h_n, p_n)\} \\
& \longrightarrow \mathbf{S}^e(\{e_1, \dots, e_n\}, \Sigma) \\
& \quad = \{(\mathbf{ZF}(\{(z_1, h_1, p_1), \dots, (z_n, h_n, p_n)\}), \\
& \quad \quad h_1 \cup \dots \cup h_n, p_1 \cup \dots \cup p_n)\}
\end{aligned}$$

The semantic binding environment generated for a set extension expression illustrates the point that the semantic binding environments of expressions have no meaning in isolation and are only given meaning within the context of a section. The semantic binding environment for a set extension expression will have a single semantic binding, but that binding contains a ZF object that represents a set of other semantic bindings; that is, the semantic binding for a set extension expression is the set of bindings produced from each of the expressions it contains (note that, each of these expressions may have a semantic binding environment containing only a single binding). We have chosen this method of representation as it allows us to retrieve the top level of bindings for a schema, which proves to be extremely useful as we can use these bindings when instantiating proof obligations. For example, if we have a schema definition paragraph as follows.

$$\boxed{\begin{array}{l} \textit{mySchemaPara} \\ a : \mathbb{N}; b : \mathbb{N}; c : \mathbb{N} \end{array}}$$

We will create a semantic binding environment for the paragraph that contains a single semantic binding as shown below.

Bound Object:	<i>mySchemaPara</i>
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A}$
Goals:	$mySchemaPara \in \{\langle a, b, c \rangle\},$ $a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N}$

We have noted previously that, the scope rules declared in the fourth paragraph of section 13.3 of [ISO02] state that, declarations at paragraph level are actually in scope for the whole of the section (and any descendent sections). In order to maintain the structure of the semantic binding environment however, we feel that referencing such variables through the paragraph's semantic binding environment provides a better solution than promoting each of the bindings to the section level. The advantages of being able to maintain a link between a variable and the paragraph in which it was declared are clear, and there is no compromise to the semantic model obtained as it can only ever be visualized internally.

Rule 6.21 - Set Comprehension Expression:

$$\begin{aligned}
& \mathbf{S}^e(e_1, \Sigma) = \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\} \\
& \wedge \mathbf{S}^e(e_2, \Sigma \oplus \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}) = \{(z, h, p)\} \\
& \wedge \tau(e_1) = \mathbb{P}[i_1 : t_1, \dots, i_n : t_n] \\
& \wedge \mathbf{Q}(\Sigma, i_1) = (w_1, e_1, l_1) \\
& \wedge \dots \\
& \wedge \mathbf{Q}(\Sigma, i_n) = (w_n, e_n, l_n) \\
& \wedge \mathbf{Q}(\Sigma, t_1) = (x_1, f_1, m_1) \\
& \wedge \dots \\
& \wedge \mathbf{Q}(\Sigma, t_n) = (x_n, f_n, m_n) \\
& \longrightarrow \mathbf{S}^e(\{e_1 \bullet e_2\}, \Sigma) \\
& = \{(\mathbf{ZF}(\{v : \tau(e_2) \\
& \quad \bullet \forall w_1 \in x_n \wedge \dots \wedge w_n \in x_n \Rightarrow v = z \wedge o_1 \wedge \dots \wedge o_n\}, \\
& \quad g_1 \cup \dots \cup g_n \cup h, \emptyset)\}
\end{aligned}$$

With a set comprehension expression the semantic binding environment contains a single semantic binding that is determined from the semantic binding environments of its component expressions which we shall refer to as the schema expression (e_1) and the characterization expression (e_2). A ZFSetSeparation object is used in the binding and this requires a list of bound variables, a type and a constraint (all of which are also ZF objects). We have discussed the difficulties in translating between the set comprehension expression and a ZFSetSeparation object in section 6.2.1. We follow the procedure there to translate the Z definition and create the ZF object required for the semantic binding. The bound variable (v) is a new variable created to store the result of the characterization expression. The type is the ZF object derived from the type of the characterization expression⁷ ($\tau(e_2)$). The constraint is a universal quantification over the declared variables ($w_1 \dots w_n$), where the constraint is the typing of those variables. The predicate is formed from the conjunction of the goals of all the bindings in the semantic binding environment of the schema expression ($o_1 \dots o_n$) and the premise that the bound variable is equal to the result of the characterization expression ($v = z$). The hypotheses for the new semantic binding are the union of the hypotheses of the semantic bindings contained in the schema expression's semantic binding environment, with the hypotheses of the semantic binding from the characterization expression. There are no goals for the semantic binding of a set comprehension expression.

Rule 6.22 - Powerset Expression:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= \{(z, h, p)\} \\ \longrightarrow \mathbf{S}^e(\mathbb{P} e, \Sigma) &= \{(\mathbf{ZF}(\mathbb{P}(z)), h, p)\} \end{aligned}$$

The semantic binding environment of a powerset expression contains a single

⁷Note that, the translation of Z types to ZF objects may cause errors if an unexpected type should occur; for example, it would be invalid to use a schema type within a set separation.

semantic binding derived from the single semantic binding contained in the semantic binding environment of its component expression. The hypotheses and the goals remain the same, but the binding name is the powerset of the binding name of the component expression.

Rule 6.23 - Tuple Extension Expression:

$$\begin{aligned}
\mathbf{S}^e(e_1, \Sigma) &= \{(z_1, h_1, p_1)\} \\
&\wedge \dots \\
&\wedge \mathbf{S}^e(e_n, \Sigma) = \{(z_n, h_n, p_n)\} \\
&\longrightarrow \mathbf{S}^e(\langle e_1, \dots, e_n \rangle, \Sigma) \\
&= \{(\mathbf{ZF}(\{(z_1, \dots, z_n)\}), h_1 \cup \dots \cup h_n, p_1 \cup \dots \cup p_n)\}
\end{aligned}$$

The semantic binding environment of a tuple extension expression contains a single semantic binding derived from the semantic bindings contained in the semantic binding environments of its component expressions (each one of which must contain only a single semantic binding). The ZF language object used in the resultant semantic binding is an instance of ZFOrdered, which retains the identity of the individual objects and maintains their order. The hypotheses and goals of the tuple extension expression's semantic binding are formed from the union of the respective elements of the semantic bindings obtained from its component expressions.

Rule 6.24 - Binding Extension Expression:

$$\begin{aligned}
\mathbf{S}^e(e_1, \Sigma) &= \{(z_1, h_1, p_1)\} \\
&\wedge \dots \\
&\wedge \mathbf{S}^e(e_n, \Sigma) = \{(z_n, h_n, p_n)\} \\
&\longrightarrow \mathbf{S}^e(\langle i_1 == e_1, \dots, i_n == e_n \rangle, \Sigma) \\
&= \{(\mathbf{ZF}(i_1), h_1 \cup p_1, \{\mathbf{ZF}(i_1 \in z_1)\}), \dots, \\
&\quad (\mathbf{ZF}(i_n), h_n \cup p_n, \{\mathbf{ZF}(i_n \in z_n)\})\}
\end{aligned}$$

With a binding extension expression we have a semantic binding environment

for each binding's expression where each of those semantic binding environments contains a single semantic binding. We create a semantic binding in the binding extension expression's semantic binding environment for every bound variable where the hypotheses are formed from the union of the hypotheses and goals of the semantic binding belonging to the appropriate expression. The goal for each semantic binding is that the newly bound variable belongs to the expression to which it is bound.

Rule 6.25 - Variable Construction Expression:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= \{(z, h, p)\} \\ \longrightarrow \mathbf{S}^e([i : e], \Sigma) &= \{(\mathbf{ZF}(i), h \cup p, \{\mathbf{ZF}(i \in z)\})\} \end{aligned}$$

The variable construction expression is semantically equivalent to a binding extension expression that contains only a single binding. That is, we create a semantic binding environment that contains a single semantic binding, where the hypotheses are formed from the union of the hypotheses and goals of the component expression, and a goal is created to state that the newly bound variable belongs to that component expression.

Rule 6.26 - Schema Construction Expression:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\} \\ \wedge \mathbf{S}^p(p, \Sigma \oplus \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}) &= (z, r) \\ \wedge \{s_1, \dots, s_n\} &= r \setminus \{y_1, \dots, y_n\} \\ \wedge \mathbf{Q}(\Sigma, s_1) &= (z_1, h_1, p_1) \\ \wedge \dots & \\ \wedge \mathbf{Q}(\Sigma, s_n) &= (z_n, h_n, p_n) \\ \longrightarrow \mathbf{S}^e([e \mid p], \Sigma) &= \{(z_1, g_1 \cup h_1 \cup \dots \cup h_n \cup p_1 \cup \dots \cup p_n, o_1 \cup z), \\ &\dots, \\ &(z_n, g_n \cup h_1 \cup \dots \cup h_n \cup p_1 \cup \dots \cup p_n, o_n \cup z)\} \end{aligned}$$

With a schema construction expression, the semantic bindings generated are derived from the semantic bindings of its component expression (e) and the semantic translation of its component predicate (p – which is determined in a contextual semantic binding environment overridden by that of the component expression). Each of the bindings retrieved from the semantic binding environment of the component expression is added to the semantic binding environment of the schema construction expression, but with their hypotheses augmented by the hypotheses and goals of any (non-local) references required by the component predicate and their goal constrained by its ZF language predicate.

Rule 6.27 - Schema Negation Expression:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= \{(z, h, \{p_1, \dots, p_n\})\} \\ \longrightarrow \mathbf{S}^e(\neg e, \Sigma) &= \{(z, h, \{\mathbf{ZF}(\neg p_1), \dots, \mathbf{ZF}(\neg p_n)\})\} \end{aligned}$$

The semantic binding environment of a schema negation expression contains a single semantic binding derived from the single semantic binding contained in the semantic binding environment of its component expression. The ZF object bound and the hypotheses remain the same, but the goals are negated.

Rule 6.28 - Schema Conjunction Expression:

$$\begin{aligned} \mathbf{S}^e(e_1, \Sigma) &= \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\} \\ \wedge \mathbf{S}^e(e_2, \Sigma) &= \{(z_1, h_1, p_1), \dots, (z_n, h_n, p_n)\} \\ \longrightarrow \mathbf{S}^e(e_1 \wedge e_2, \Sigma) &= \{(y_1, g_1 \cup h_1 \cup \dots \cup h_n, o_1 \cup p_1 \cup \dots \cup p_n), \dots, \\ &\quad (y_n, g_n \cup h_1 \cup \dots \cup h_n, o_n \cup p_1 \cup \dots \cup p_n), \\ &\quad (z_1, g_1 \cup \dots \cup g_n \cup h_1, o_1 \cup \dots \cup o_n \cup p_1), \dots, \\ &\quad (z_n, g_1 \cup \dots \cup g_n \cup h_n, o_1 \cup \dots \cup o_n \cup p_n)\} \end{aligned}$$

With a schema conjunction expression, the semantic bindings generated are

derived from the selective merging of the properties belonging to the semantic bindings contained in the two component expressions. All of the semantic bindings belonging to the semantic binding environment of the first component expression are added to the semantic binding environment of the schema conjunction expression, but their hypotheses are augmented by the union of the hypotheses of the semantic bindings belonging to the semantic binding environment of the second component expression and their goals augmented by the union of the goals of those semantic bindings. Similarly, all the semantic bindings belonging to the semantic binding environment of the second component expression are added to the semantic binding environment of the schema conjunction expression – unless they have already been added as part of the first expression – with their hypotheses and goals supplemented with those derived from the first component expression.

At first it may appear strange that we conjoin all of the hypotheses and goals of one expression with each of the bindings in the other. The reason for this becomes obvious if we examine the following schema definition paragraphs and their conjunction.

The schema, $mySchema_1$, declares the variable a and specifies some constraints on that variable whose definition depends on the other declared variable, b .

$mySchema_1$ $a : \mathbb{N}; b : \mathbb{N}$
$a = b * b$

The schema, $mySchema_2$, also declares a variable a and specifies some constraints on that variable which, in this instance, depend upon another declared variable, c .

$mySchema_2$ $a : \mathbb{N}; c : \mathbb{N}$
$a = 2 * c$

The schema, $mySchema_3$, is formed from the schema conjunction of $mySchema_1$ and $mySchema_2$. The variable a has constraints defined in both of the schemas and these also rely on the variables defined within the respective schemas.

$$mySchema_3 == mySchema_1 \wedge mySchema_2$$

If we consider the binding we require for a in the conjoined paragraph, it soon becomes clear that we need the hypotheses of the first schema (that is, $b \in \mathbb{N}$) as b is used in our constraint. However, we also need the hypotheses of the second schema (that is, $c \in \mathbb{N}$) as c is also used in our constraint. Similarly, in order for a to be defined as required we also need the goals from both bindings. This leaves us requiring a semantic binding for a – in $mySchema_3$ – as follows⁸.

Bound Object:	a
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A}), \mathbb{N} \in \mathbb{P} \ \mathbb{A}, number_literal_0 \in \mathbb{N},$ $number_literal_1 \in \mathbb{N}, \bowtie * \bowtie \in \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}),$ $\bowtie + \bowtie \in \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}), a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N}$
Goals:	$a \in \{\bowtie * \bowtie(b, b)\},$ $a \in \{\bowtie * \bowtie(\bowtie + \bowtie(number_literal_1, number_literal_1), c)\}$

We are unable to easily predict which goals will require which hypotheses of the opposing schema, nor can we predict which goals of an opposing schema will impact a bound variable. Hence the simplest solution is to perform the universal merge that we have outlined in our rule above.

⁸Dealing with the applications in a semantic binding is more complex than presented here, we will present this in more detail in rule 6.33.

Rule 6.29 - Schema Universal Quantification Expression:

$$\begin{aligned}
\mathbf{S}^e(e, \Sigma) &= \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\} \\
&\wedge \mathbf{S}^p(p, \Sigma \oplus \{(y_1, g_1, o_1), \dots, (y_n, g_n, o_n)\}) = (z, r) \\
&\wedge \{s_1, \dots, s_n\} = r \setminus \{y_1, \dots, y_n\} \\
&\wedge \mathbf{Q}(\Sigma, s_1) = (z_1, h_1, p_1) \\
&\wedge \dots \\
&\wedge \mathbf{Q}(\Sigma, s_n) = (z_n, h_n, p_n) \\
&\longrightarrow \mathbf{S}^e(\forall e \bullet p, \Sigma) \\
&= \{(y_1, g_1 \cup h_1 \cup \dots \cup h_n \cup p_1 \cup \dots \cup p_n, o_1 \cup z), \\
&\quad \dots, \\
&\quad (y_n, g_n \cup h_1 \cup \dots \cup h_n \cup p_1 \cup \dots \cup p_n, o_n \cup z)\}
\end{aligned}$$

The semantic binding environment of a schema universal quantification expression contains semantic bindings that are derived from the semantic bindings of its component expression and the semantic translation of its component predicate (which is determined in a contextual semantic binding environment overridden by that of the component expression). Each of the bindings retrieved from the semantic binding environment of the component expression is added to the semantic binding environment of the schema universal quantification expression, but with their hypotheses augmented by the hypotheses and goals of any (non-local) references required by the component predicate and their goal constrained by its predicate.

Rule 6.30 - Schema Renaming Expression:

$$\begin{aligned}
\mathbf{S}^e(e, \Sigma) &= \{(z_1, h_1, p_1), \dots, (z_n, h_n, p_n)\} \\
&\wedge \mathbf{Q}(\Sigma, i_1) = (y_1, g_1, o_1) \\
&\wedge \dots \\
&\wedge \mathbf{Q}(\Sigma, i_n) = (y_n, g_n, o_n) \\
&\longrightarrow \mathbf{S}^e(e[j_1/i_1, \dots, j_n/i_n], \Sigma) \\
&= \{\mathbf{R}^{sb}(\langle y_1, \dots, y_n \rangle, \langle \mathbf{ZF}(j_1), \dots, \mathbf{ZF}(j_n) \rangle), (z_1, h_1, p_1), \Sigma, \dots, \\
&\quad \mathbf{R}^{sb}(\langle y_1, \dots, y_n \rangle, \langle \mathbf{ZF}(j_1), \dots, \mathbf{ZF}(j_n) \rangle), (z_n, h_n, p_n), \Sigma)\}
\end{aligned}$$

The semantic binding environment of a schema renaming expression inherits all of the semantic bindings contained in the semantic binding environment of its component expression. Each of those semantic bindings, however, is subject to the application of the ZF object replacement function (where the object to be replaced is retrieved from the contextual semantic binding environment).

Rule 6.31 - Binding Construction Expression:

$$\begin{aligned} \tau(\theta e^{dec}) &= \langle i_1 : t_1, \dots, i_n : t_n \rangle \\ \longrightarrow \mathbf{S}^e(\theta e^{dec}, \Sigma) &= \mathbf{S}^e(\langle i_1 == i_1^{dec}, \dots, i_n == i_n^{dec} \rangle, \Sigma) \end{aligned}$$

The semantic binding environment for a binding construction expression is equivalent to that for a binding extension expression, where the variables in the signature of the binding construction expression are bound to themselves with the optional decoration appended.

Rule 6.32 - Binding Selection Expression:

$$\mathbf{S}^e(e, \Sigma) = \sigma \longrightarrow \mathbf{S}^e(e.i, \Sigma) = \{\mathbf{Q}(\sigma, i)\}$$

With a binding selection expression, the semantic binding environment contains a single semantic binding. This binding is retrieved from the semantic binding environment of the component expression by querying that environment with the bound variable which has been selected.

Rule 6.33 - Application Expression:

$$\begin{aligned} \mathbf{S}^e(e_1, \Sigma) &= \{(z_1, h_1, p_1)\} \wedge \mathbf{S}^e(e_2, \Sigma) = \{(z_2, h_2, p_2)\} \\ \longrightarrow \mathbf{S}^e(e_1 e_2, \Sigma) &= \{(\mathbf{ZF}(j), \{\mathbf{ZF}(j \in \tau(e_1 e_2))\} \cup h_1 \cup h_2, \\ &\quad \{\mathbf{ZF}(\exists f \bullet (\forall x \bullet \langle z_2, y \rangle \in z_1 \Leftrightarrow x \in f) \wedge j \in f)\} \cup p_1 \cup p_2)\} \end{aligned}$$

For certain types of expression we are unable to pre-determine the specific semantic definition of that expression and must allow for all possible semantics. For instance, when we try to determine the semantic binding environment of an application expression we immediately run into problems. Consider the following application expression – and its syntactically transformed equivalent – involving the union operator.

$$a \cup b \quad \longrightarrow \quad \bowtie \cup \bowtie (a, b)$$

When we create the single semantic binding that would belong to this expression's semantic binding environment, we wish to refer to the result of the application. We cannot refer to this result directly as firstly we have no variable which represents this value, and secondly, there is a set of possible results.

In order to get around this we create a variable in which we can store the set of possible results and add a suitable constraint that – along with goals of the two component expressions – is added to the set of goals for our application expression's semantic binding. For instance, in the above example we may create the variable *results*, and add the following constraint.

$$\forall x \bullet \langle \langle a, b \rangle, x \rangle \in \bowtie \cup \bowtie \Leftrightarrow x \in \text{results}$$

We then create a second variable (in this instance, *var*), which we use as the bound variable in the application expression's semantic binding and add a clause to the goals that the variable must belong to the set of valid results for our application expression, that is to the set *results*. In our example, our final binding is as follows. (Where *hyp*₁, *hyp*₂, *goal*₁ and *goal*₂ are the relevant hypotheses, and goals of the component expressions respectively, and *e* is the application expression.)

Bound Object:	var
Hypotheses:	$\{var \in \tau(e)\} \cup hyp_1 \cup hyp_2$
Goals:	$\{\exists results$ $\bullet (\forall x \bullet \langle \langle a, b \rangle, x \rangle \in \bowtie \cup \bowtie \Leftrightarrow x \in results)$ $\wedge var \in results\}$ $\cup goal_1 \cup goal_2$

Rule 6.34 - Tuple Selection Expression:

$$\begin{aligned}
S^e(e, \Sigma) &= \{(z, h, p)\} \\
&\wedge \tau(e) = t_1 \times \dots \times t_n \\
&\longrightarrow S^e(e.num, \Sigma) \\
&= \{(ZF(j), \{ZF(j \in \tau(e.num))\} \cup h, \\
&\quad \{ZF(\exists f \bullet (\forall x_1, \dots, x_n \bullet \langle x_1, \dots, x_n \rangle \in \{z\} \Leftrightarrow x_{num} \in f) \\
&\quad \wedge j \in f)\} \\
&\quad \cup p)\}
\end{aligned}$$

Like the application expression, the tuple selection expression requires the use of intermediate variables in the creation of its semantic binding. In this case we are unable to predict the nature of the expression that creates the tuple we will select from. If this component expression were a tuple extension we would be able to simply select the appropriate binding. If however, it were a reference expression we would be unable to pre-determine the semantic bindings that would form the tuple. Consider the following tuple selection expression (syntactic transformation does not affect the syntax of this particular expression).

R.1

It is impossible for us to determine the nature of the reference, R , from our current context. We create therefore, an intermediary set in which we can

store all the possible values of the appropriate selection from this tuple. In order to do this we first need to determine the number of elements in the tuple. We achieve this by examining the type, and can deduce the number of elements from the number of types involved in the cartesian product that must form a tuple's top-level type. Knowing this, we can define a set that contains all the possible values for the selected element of the tuple. In this instance – assuming that the type of R is $\mathbb{N} \times \mathbb{N}$, and the new set is *results* – we would create the following constraint which together with the goals of the component expression's semantic binding is added to the goals of the tuple selection expression's semantic binding.

$$\forall x_1, x_2 \bullet \langle x_1, x_2 \rangle \in R \Leftrightarrow x_1 \in results$$

We then create a second variable (in this instance, *var*), which we use as the bound variable in the tuple selection expression's semantic binding and add a clause to the goals that the variable must belong to the set of valid results for our tuple selection expression, that is to the set *results*. In our example therefore, our final binding is as follows. (Where *hyp* and *goal* are the hypotheses and goals of the component expression, and e is the tuple selection expression.)

Bound Object:	var
Hypotheses:	$\{var \in \tau(e)\} \cup hyp$
Goals:	$\{\exists results$ $\bullet (\forall x_1, x_2 \bullet \langle x_1, x_2 \rangle \in R \Leftrightarrow x_1 \in results)$ $\quad \wedge var \in results\}$ $\cup goal$

Rule 6.35 - Definite Description Expression:

$$\begin{aligned} \mathbf{S}^e(e_1, \Sigma) &= \{(z_1, h_1, p_1)\} \wedge \mathbf{S}^e(e_2, \Sigma) = \{(z_2, h_2, p_2)\} \\ &\longrightarrow \mathbf{S}^e(\mu e_1 \bullet e_2, \Sigma) \\ &= \{(\mathbf{ZF}(z_2), h_1 \cup h_2 \cup p_1 \cup p_2, \{\mathbf{ZF}(z_2 \in z_1)\})\} \end{aligned}$$

The semantic binding environment of a definite description expression contains a single semantic binding derived from the single semantic binding contained in the semantic binding environment of its component expressions, which we shall refer to as the schema expression (e_1), and the characterization expression (e_2). The bound object is the bound object of the characterization expression's semantic binding. The hypotheses are formed from the union of the hypotheses and goals belonging to both of the component expression's semantic bindings. In the goals we create a constraint that states that the characterization expression is a member of the schema expression.

Rule 6.36 - Schema Hiding Expression:

$$\begin{aligned} \mathbf{S}^e(e, \Sigma) &= \{(y_1, g_1, o_{1_1}, \dots, o_{1_n}), \dots, (y_n, g_n, o_{n_1}, \dots, o_{n_m})\} \\ &\wedge \mathbf{Q}(\Sigma, i_1) = (z_1, h_1, p_1) \\ &\wedge \dots \\ &\wedge \mathbf{Q}(\Sigma, i_n) = (z_n, h_n, p_n) \\ &\longrightarrow \mathbf{S}^e(e \setminus (i_1, \dots, i_n), \Sigma) \\ &= \{(y_1, g_1, \mathbf{ZF}(\exists z_1, \dots, z_n \bullet o_{1_1} \wedge \dots \wedge o_{1_n})), \dots, \\ &\quad (y_n, g_n, \mathbf{ZF}(\exists z_1, \dots, z_n \bullet o_{n_1} \wedge \dots \wedge o_{n_m}))\} \triangleleft \{z_1, \dots, z_n\} \end{aligned}$$

With a schema hiding expression, the semantic binding environment is formed by taking all of the semantic bindings that do not involve the binding of one of the hidden objects (retrieved from the contextual semantic binding environment) that are also contained within the semantic binding environment of the component expression. The goals of each of the remaining semantic bindings are altered so that they are now subject to an existential quantification

over the hidden objects.

Rule 6.37 - Schema Precondition Expression:

$$\begin{aligned}
\mathbf{S}^e(e, \Sigma) &= \{(y_1, g_1, o_{1_1}, \dots, o_{1_n}), \dots, (y_n, g_n, o_{n_1}, \dots, o_{n_m})\} \\
&\wedge \tau(e) = \mathbb{P}[u_1 : s_1, \dots, u_n : s_n] \\
&\wedge \tau(\text{pre } e) = \mathbb{P}[v_1 : t_1, \dots, v_n : t_n] \\
&\wedge \{w_1, \dots, w_n\} = \{u_1, \dots, u_n\} \setminus \{v_1, \dots, v_n\} \\
&\wedge \mathbf{Q}(\Sigma, w_1) = (z_1, h_1, p_1) \\
&\wedge \dots \\
&\wedge \mathbf{Q}(\Sigma, w_n) = (z_n, h_n, p_n) \\
&\longrightarrow \mathbf{S}^e(\text{pre } e, \Sigma) \\
&= \{(y_1, g_1, \mathbf{ZF}(\exists z_1, \dots, z_n \bullet o_{1_1} \wedge \dots \wedge o_{1_n})), \dots, \\
&\quad (y_n, g_n, \mathbf{ZF}(\exists z_1, \dots, z_n \bullet o_{n_1} \wedge \dots \wedge o_{n_m}))\} \triangleleft \{z_1, \dots, z_n\}
\end{aligned}$$

The semantic binding environment for a schema precondition expression is derived from the semantic binding environment of the component schema, and the pre-determined type of that schema and the schema precondition expression itself. The difference in the signatures of these two types is determined to resolve which variables are local to the schema (typically, inputs and outputs); the semantic bindings for all non-local variables in the schema's semantic binding environment are then added to that of the schema precondition expression. The goals of each of these semantic bindings are altered so that they are now subject to an existential quantification over the local variables.

Rule 6.38 - Schema Composition Expression:

$$\begin{aligned}
& \mathbf{S}^e(e_1, \Sigma) = \{(y_1, g_1, o_{1_1}, \dots, o_{1_n}), \dots, (y_n, g_n, o_{n_1}, \dots, o_{n_n})\} \\
& \wedge \mathbf{S}^e(e_2, \Sigma) = \{(z_1, h_1, p_{1_1}, \dots, p_{1_n}), \dots, (z_n, h_n, p_{n_1}, \dots, p_{n_n})\} \\
& \wedge \tau(e_1) = \mathbb{P}[u_1 : r_1, \dots, u_n : r_n] \\
& \wedge \tau(e_2) = \mathbb{P}[v_1 : s_1, \dots, v_n : s_n] \\
& \wedge \tau(e_1 \circ e_2) = \mathbb{P}[w_1 : t_1, \dots, w_n : t_n] \\
& \wedge I = \{u_1, \dots, u_n\} \setminus \{w_1, \dots, w_n\} \\
& \wedge J = \{v_1, \dots, v_n\} \setminus \{w_1, \dots, w_n\} \\
& \wedge P = \{\langle i_1, j_1 \rangle, \dots, \langle i_n, j_n \rangle\} \\
& \wedge \forall x \bullet x' \in I \wedge x \in J \Rightarrow \langle x', x \rangle \in P \\
& \wedge \mathbf{Q}(\Sigma, i_1) = (k_1, d_1, a_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, i_n) = (k_n, d_n, a_n) \\
& \wedge \mathbf{Q}(\Sigma, j_1) = (l_1, e_1, b_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, j_n) = (l_n, e_n, b_n) \\
& \wedge \mathbf{Q}(\Sigma, w_1) = (m_1, f_1, c_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, w_n) = (m_n, f_n, c_n) \\
& \wedge bv_1 = k_1 \wedge bv_1 = l_1 \\
& \wedge \dots \\
& \wedge bv_n = k_n \wedge bv_n = l_n \\
& \longrightarrow \mathbf{S}^e(e_1 \circ e_2, \Sigma) \\
& = \{(y_1, g_1, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n}))), \\
& \quad \dots, \\
& (y_n, g_n, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n}))), \\
& (z_1, h_1, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n}))), \\
& \quad \dots, \\
& (z_n, h_n, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n})))\} \\
& \triangleleft \{m_1, \dots, m_n\}
\end{aligned}$$

A schema composition expression allows us to refer to the schema that represents the operation of performing the operations represented by two other schema expressions in sequence. That is, the operation represented by the first schema moves the state to an intermediate state which forms the starting state for the operation represented by the second schema. The semantic binding environment for a schema composition expression is derived from the semantic binding environment of the component schemas and the predetermined types of both those schemas and the schema composition expression itself. The signature of the schema composition expression is used to determine which semantic bindings of the component expression's semantic binding environment are included in its own semantic binding environment. The signatures of the two component schema expressions are used to determine which variables form the intermediate state – with the variables of one schema being matched with the appropriate variables of the other – an existential quantification over which is added to the goals of each semantic binding. Note that, these goals have previously been conjoined with the sum of the goals belonging to the semantic bindings of the component schema expression to which it does not itself belong.

Whilst the aforementioned rule for determining the semantic binding environment appears, at first glance, to be relatively complex it can be soon seen that the majority of the work involves matching the output variables of the first expression with the input variables of the second. These variables exist simply to hold the post-transition state of the first expression and the pre-transition state of the second expression, hence when composed this intermediate state can be removed. This process first involves producing sets of the variables contained in the two expressions' signatures (u_1, \dots, u_n and v_1, \dots, v_n) as well as a set of those contained in the signature of the composition (w_1, \dots, w_n). The variables contained in the composed expression's signature are then removed from the first two sets to produce two more sets (I and J). A set of pairs (P) is then produced from the members of these sets where the post-transition state of a variable in the first expression's signature is mapped to the pre-transition state of that variable in the

second expression's signature. The semantic bindings for each of these variables are then retrieved $(k_1, \dots, k_n$ and $l_1, \dots, l_n)$. As the semantic binding of the post-transition state of the variable in the first expression's signature should be identical to that of the pre-transition state of the variable in the second expression's signature we can then assert that the semantic bindings are equal to each other and to an arbitrary variable that can be used in the semantic binding of the schema composition (bv_1, \dots, bv_n) . These arbitrary variables are used to replace both of the alternate bindings $(k_1, \dots, k_n$ and $l_1, \dots, l_n)$ in the semantic binding environment of the schema composition thus eliminating the intermediate state.

Rule 6.39 - Schema Piping Expression:

$$\begin{aligned}
& \mathbf{S}^e(e_1, \Sigma) = \{(y_1, g_1, o_{1_1}, \dots, o_{1_n}), \dots, (y_n, g_n, o_{n_1}, \dots, o_{n_n})\} \\
& \wedge \mathbf{S}^e(e_2, \Sigma) = \{(z_1, h_1, p_{1_1}, \dots, p_{1_n}), \dots, (z_n, h_n, p_{n_1}, \dots, p_{n_n})\} \\
& \wedge \tau(e_1) = \mathbb{P}[u_1 : r_1, \dots, u_n : r_n] \\
& \wedge \tau(e_2) = \mathbb{P}[v_1 : s_1, \dots, v_n : s_n] \\
& \wedge \tau(e_1 \gg e_2) = \mathbb{P}[w_1 : t_1, \dots, w_n : t_n] \\
& \wedge I = \{u_1, \dots, u_n\} \setminus \{w_1, \dots, w_n\} \\
& \wedge J = \{v_1, \dots, v_n\} \setminus \{w_1, \dots, w_n\} \\
& \wedge P = \{\langle i_1, j_1 \rangle, \dots, \langle i_n, j_n \rangle\} \\
& \wedge \forall x \bullet x! \in I \wedge x? \in J \Rightarrow \langle x!, x? \rangle \in P \\
& \wedge \mathbf{Q}(\Sigma, i_1) = (k_1, d_1, a_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, i_n) = (k_n, d_n, a_n) \\
& \wedge \mathbf{Q}(\Sigma, j_1) = (l_1, e_1, b_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, j_n) = (l_n, e_n, b_n) \\
& \wedge \mathbf{Q}(\Sigma, w_1) = (m_1, f_1, c_1) \wedge \dots \wedge \mathbf{Q}(\Sigma, w_n) = (m_n, f_n, c_n) \\
& \wedge bv_1 = k_1 \wedge bv_1 = l_1 \\
& \wedge \dots \\
& \wedge bv_n = k_n \wedge bv_n = l_n \\
& \longrightarrow \mathbf{S}^e(e_1 \gg e_2, \Sigma) \\
& = \{(y_1, g_1, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n}))), \\
& \quad \dots, \\
& (y_n, g_n, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n}))), \\
& (z_1, h_1, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{1_1} \wedge \dots \wedge p_{1_n}))), \\
& \quad \dots, \\
& (z_n, h_n, \mathbf{R}^{zf}(\langle k_1, \dots, k_n, l_1, \dots, l_n \rangle, \\
& \quad \langle bv_1, \dots, bv_n, bv_1, \dots, bv_n \rangle, \\
& \quad \mathbf{ZF}(\exists k_1, \dots, k_n, l_1, \dots, l_n \\
& \quad \bullet o_{1_1} \wedge \dots \wedge o_{1_n} \wedge o_{n_1} \wedge \dots \wedge o_{n_n} \wedge p_{n_1} \wedge \dots \wedge p_{n_n})))\} \\
& \triangleleft \{m_1, \dots, m_n\}
\end{aligned}$$

A schema piping expression allows us to refer to the schema that represents the operation of performing the operations represented by two other schema expressions with the outputs of the first schema providing the inputs to the second. The semantic binding environment for a schema piping expression is determined in much the same way as that for a schema composition expression, the only difference being the method of matching the appropriate variables of the two component schema expressions.

Rule 6.40 - Schema Decoration Expression:

$$\begin{aligned} \tau(e) &= \mathbb{P}[v_1 : t_1, \dots, v_n : t_n] \\ \longrightarrow \mathbf{S}^e(e^{dec}, \Sigma) &= \mathbf{S}^e(e[v_1^{dec}/v_1, \dots, v_n^{dec}/v_n], \Sigma) \end{aligned}$$

The semantic binding environment for a schema decoration expression is equivalent to that for a schema renaming expression, where the schema's variables are replaced by the decorated equivalents of those expressions.

References to Schemas

Unfortunately, the rules presented above do not present the whole story. There are two circumstances where special conditions apply; both of these exceptions concern rule 6.18 which deals with the reference expression.

The first of these exceptions is where a reference is made to a variable that represents a schema. Clearly, the definition provided in rule 6.18 is only intended to retrieve a single semantic binding. Before we apply this rule to a reference expression, we check whether this the expression refers to a schema (this can easily be determined from the reference expression's type).

Rule 6.41 - Reference to Schema Expression:

$$\mathbf{Q}(\Sigma, i) = (z, h, \mathbf{ZF}(z \in \{\langle b_1, \dots, b_n \rangle\})) \longrightarrow \mathbf{S}^e(i, \Sigma) = \{b_1, \dots, b_n\}$$

With an expression that references a schema, the semantic binding of the schema variable is retrieved from the contextual semantic binding environment. Instead of adding the semantic binding of the referenced variable to the semantic binding environment, we examine the goal of that semantic binding which – as we know how all schemas are constructed – will always involve a membership predicate that indicates the referenced variable is a member of the set of semantic bindings of its signature variables. We examine therefore, this set of semantic bindings and can add the semantic bindings we find to the reference expression’s semantic binding environment. For example, we considered the following paragraph previously in this section.

$\begin{array}{l} \textit{mySchemaPara} \\ a : \mathbb{N}; b : \mathbb{N}; c : \mathbb{N} \end{array}$
--

This paragraph had the following semantic binding.

Bound Object:	$\textit{mySchemaPara}$
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A}$
Goals:	$\textit{mySchemaPara} \in \{\langle a, b, c \rangle\},$ $a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N}$

Consider then, that we have a second schema definition paragraph, that references this schema, such as the one below.

$\begin{array}{l} \textit{mySchemaPara_Init} \\ \textit{mySchemaPara} \\ a = 0 \\ b = 0 \\ c = 0 \end{array}$
--

When we create the semantic binding environment for the reference expression *mySchemaPara* we do not mean to add the semantic binding for that paragraph, but instead to import the semantic bindings of its variables. If we apply rule 6.41 then the reference expression will generate the following semantic binding environment.

$$\begin{aligned} & \{(\text{ZF}(a), \{\text{ZF}(\text{arithmos} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\}, \\ & \quad \{\text{ZF}(a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N})\}), \\ & (\text{ZF}(b), \{\text{ZF}(\text{arithmos} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\}, \\ & \quad \{\text{ZF}(a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N})\}), \\ & (\text{ZF}(c), \{\text{ZF}(\text{arithmos} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A})\}, \\ & \quad \{\text{ZF}(a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N})\}) \} \end{aligned}$$

The semantic binding environment for the *mySchemaPara_Init* thus becomes the set containing the single semantic binding shown below.

Bound Object:	<i>mySchemaPara_Init</i>
Hypotheses:	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}), \mathbb{N} \in \mathbb{P} \mathbb{A}$
Goals:	$\text{mySchemaPara_Init} \in \{\langle a, b, c \rangle\},$ $a \in \mathbb{N}, b \in \mathbb{N}, c \in \mathbb{N},$ $a = 0, b = 0, c = 0$

Implicit Instantiation of Generic Operators

The second exception concerns references to variables where the variable has been declared with generic parameters, but none are explicitly provided (a situation which would be handled by rule 6.19). One could be forgiven for thinking that we had already covered the implicit instantiation of generics in section 4.6.6, unfortunately however, there is a distinction between the instantiation of types and the instantiation of variables.

It is therefore, necessary to extend rule 6.18 again. After we have determined whether or not a referenced variable is a schema, we will determine

whether that variable requires generic instantiation (again, this can be easily determined from the variable's type).

Rule 6.42 - Reference to Variable with Generic Parameters Expression:

$$\begin{aligned}
& e = i \wedge \{a_1, \dots, a_n\} = \mathcal{DP}(\tau(e), \tau(i)) \\
& \wedge \mathbf{Q}(\Sigma, a_1) = (y_1, g_1, o_1) \\
& \wedge \dots \\
& \wedge \mathbf{Q}(\Sigma, a_n) = (y_n, g_n, o_n) \\
& \wedge \mathbf{Q}(\Sigma, i) = (z, h, p) \\
& \longrightarrow \mathbf{S}^e(e, \Sigma) \\
& \quad = \{(\mathbf{ZF}(z[y_1, \dots, y_n]), \\
& \quad \quad \mathbf{R}^{set}(\langle z \rangle \wedge \mathbf{G}(i) \wedge \mathbf{G}^\tau(i), \\
& \quad \quad \langle \mathbf{ZF}(z[y_1, \dots, y_n], y_1, \dots, y_n, \tau(y_1), \dots, \tau(y_n)) \rangle, \\
& \quad \quad h \cup p \cup g_1 \cup \dots \cup g_n \cup o_1 \cup \dots \cup o_n), \\
& \quad \quad \emptyset)\}
\end{aligned}$$

The parameters required for the implicit instantiated are determined with the operation, $\mathcal{DP}()$, which we defined in table 4.6 on page 210. Note that, in the above rule, we need to make a distinction between the expression, e , and the reference within that expression, i , as we use the contrasting types of the two terms as arguments to the operation that determines our implicit instantiation parameters. Once the implicit instantiation parameters have been calculated, we retrieved their semantic bindings from the contextual semantic binding environment and proceed in much the same way as for an explicit generic instantiation (see rule 6.19).

Suppose for example we have the following axiomatic description paragraph.

$$| \quad A = \{0\} \cup \{1\}$$

Which will be translated to produce the following in the syntax transformation process.

$$| \quad A : \{\bowtie \cup \bowtie (\{0\}, \{1\})\}$$

When we come to calculate the semantic binding for the reference, $\bowtie \cup \bowtie$, we must implicitly instantiate it. The types for the expression and reference are calculated, and the implicit instantiation parameters determined.

$$\begin{aligned}\tau(e) &= \mathbb{P}(\text{GIVEN } \mathbb{A}) \\ \tau(i) &= \mathbb{P}(\text{GENTYPE } X)\end{aligned}$$

$$\mathcal{DP}(\tau_e, \tau(i)) = \{\text{GIVEN } \mathbb{A}\}$$

The semantic bindings for the instantiation parameter and the referenced variable are retrieved and are used to create the hypotheses for the newly instantiated ZF object, with references to generic objects being replaced by their equivalent, instantiated version (in the same way as an explicit generic instantiation). This leaves the semantic binding environment for the expression as follows.

$$\begin{aligned}
& S^e(\bowtie \cup \bowtie) \\
& = \{(\mathbf{ZF}(\bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ A})])), \\
& \quad \mathbf{ZF}(\{ \\
& \quad \quad \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ A})] \\
& \quad \quad \in \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}], \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}], \\
& \quad \forall a\ b \bullet a \in \mathbb{P}(\mathbf{GIVEN\ A}) \wedge b \in \mathbb{P}(\mathbf{GIVEN\ A}) \\
& \quad \quad \Rightarrow \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ A})] \\
& \quad \quad \quad \in \{\{x : \mathbb{P}(\mathbf{GIVEN\ A}) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbf{GIVEN\ A}) \wedge \neg(\neg(x \in a) \wedge \neg(x \in b))\}\}, \\
& \quad \quad \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}], \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}] \\
& \quad \quad \in \{\{f : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \bullet f \in \bowtie \leftrightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}], \\
& \quad \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}] \\
& \quad \quad \quad \wedge \forall x \bullet x \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}\} \\
& \quad \quad \quad \quad \Rightarrow \exists_1 y \bullet y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge \langle x, y \rangle \in f\}\} \\
& \quad \quad \quad \bowtie \leftrightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}], \\
& \quad \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}] \\
& \quad \quad \in \{\mathbb{P}\{s, t : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \bullet s \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\} \\
& \quad \quad \quad \quad \wedge t \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ A}))\}\}\}, \\
& \quad \quad \emptyset\}
\end{aligned}$$

Example 6.1

We now return to our recurrent example of modelling the behaviour of a train yard, we last examined this specification in example 5.9 where we presented the following machine.

MACHINE *trainYard*
TYPE *machine*
SECTION *theTrainYard*

STATE

 $trainsInYard : \mathbb{P} \text{ trains}$

INITIALIZATION

 $trainsInYard = \emptyset$ OPERATION $trainEntersYard \hat{=}$
INPUTS $trainEnteringYard? : \text{trains}$

PRE

 $trainEnteringYard? \notin trainsInYard$

POST

 $trainsInYard' = trainsInYard \cup \{trainEnteringYard?\}$ END $trainEntersYard$ END $trainYard$

Now we will show how the semantic binding environment is generated for the post condition of the $trainEntersYard$ operation. The first step is to show the Z paragraph produced for this operation in the syntactic transformation phase. This paragraph is shown below.

$$\left[\begin{array}{l} _trainYard_trainEntersYard_post : \{ \\ _trainYard_state \wedge _trainYard_state' \\ \wedge _trainYard_trainEntersYard_inputs \\ | _trainsInYard' \in \{\bowtie \cup \bowtie (_trainsInYard, \{trainEnteringYard?\})\} \} \end{array} \right]$$

We will assume that the remainder of the machine's section has had its

semantic bindings generated successfully, and that the contextual semantic binding environment is as we expect. We then look at our paragraph and determine that rule 6.8 is the best starting point, but this simply states that the semantic binding environment for the paragraph is the same as for the enclosed expression. We continue to examine the rule for the various component expressions and realize that in this instance it is best to illustrate the creation of the semantic binding environment from the bottom-up.

We will begin therefore with the three reference expressions $_trainYard_state$, $_trainYard_state'$ and $_trainYard_trainEntersYard_inputs$. As these rules are all references to schemas, we apply rule 6.41 and query the contextual semantic binding environment obtaining the following results.

$$\begin{aligned} & Q(\Sigma, _trainYard_state) \\ & = \{(ZF(trainsInYard), \\ & \quad \{ZF(trains \in \mathbb{P}(\text{GIVEN } trains)\}, \{trainsInYard \in \mathbb{P} trains\})\} \end{aligned}$$

$$\begin{aligned} & Q(\Sigma, _trainYard_state') \\ & = \{(ZF(trainsInYard'), \\ & \quad \{ZF(trains \in \mathbb{P}(\text{GIVEN } trains)\}, \{trainsInYard' \in \mathbb{P} trains\})\} \end{aligned}$$

$$\begin{aligned} & Q(\Sigma, _trainYard_trainEntersYard_inputs) \\ & = \{(ZF(trainEnteringYard?), \\ & \quad \{ZF(trains \in \mathbb{P}(\text{GIVEN } trains)\}, \{trainEnteringYard? \in trains\})\} \end{aligned}$$

These reference expressions are linked with two schema conjunction expressions (one nested within the other). We can calculate the semantic binding environment for the outer of those schema conjunction expressions by applying rule 6.28 twice to give the following result.

$$\begin{aligned}
& S^e(_trainYard_state \wedge _trainYard_state' \\
& \quad \wedge _trainYard_trainEntersYard_inputs, \Sigma) \\
& = \{(\mathbf{ZF}(trainsInYard), \\
& \quad \{\mathbf{ZF}(trains \in \mathbb{P}(\mathbf{GIVEN} trains))\}, \{\mathbf{ZF}(trainsInYard \in \mathbb{P} trains, \\
& \quad trainsInYard' \in \mathbb{P} trains, trainEnteringYard? \in trains)\}) \\
& \quad (\mathbf{ZF}(trainsInYard'), \\
& \quad \{\mathbf{ZF}(trains \in \mathbb{P}(\mathbf{GIVEN} trains))\}, \{\mathbf{ZF}(trainsInYard \in \mathbb{P} trains, \\
& \quad trainsInYard' \in \mathbb{P} trains, trainEnteringYard? \in trains)\}) \\
& \quad (\mathbf{ZF}(trainEnteringYard?), \\
& \quad \{\mathbf{ZF}(trains \in \mathbb{P}(\mathbf{GIVEN} trains))\}, \{\mathbf{ZF}(trainsInYard \in \mathbb{P} trains, \\
& \quad trainsInYard' \in \mathbb{P} trains, trainEnteringYard? \in trains)\})\}
\end{aligned}$$

This schema conjunction now forms the expression part of a schema construction, so according to rule 6.26 we now use the semantic binding environment we have calculated in conjunction with the contextual semantic binding environment to calculate the predicate part of the schema construction expression.

The predicate part of the schema construction expression comprises a membership predicate. This predicate consists of a reference expression, and an application expression enclosed within a set extension expression. The reference expression requires the processing of the result obtained from querying the contextual semantic binding environment – using rule 6.18 – giving the following result.

$$\begin{aligned}
& S^e(trainsInYard', \Sigma) \\
& = \{(\mathbf{ZF}(trainsInYard'), \\
& \quad \{\mathbf{ZF}(trains \in \mathbb{P}(\mathbf{GIVEN} trains), trainsInYard \in \mathbb{P} trains, \\
& \quad trainsInYard' \in \mathbb{P} trains, trainEnteringYard? \in trains\}, \emptyset)\}
\end{aligned}$$

The semantic binding environment for the application expression is deduced by first querying the contextual semantic binding environment and implicitly instantiating the reference to $\bowtie \cup \bowtie$ (a process which we examined when explaining rule 6.42).

$$\begin{aligned}
& S^e(\bowtie \cup \bowtie, \Sigma) \\
& = \{(\mathbf{ZF}(\bowtie \cup \bowtie [\mathbf{GIVEN\ trains}]), \\
& \quad \mathbf{ZF}(\{(trains \in \mathbb{P}(\mathbf{GIVEN\ trains}), \\
& \quad \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ trains})] \\
& \quad \in \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}}, \\
& \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}}, \\
& \quad \forall a\ b \bullet a \in \mathbb{P}(\mathbf{GIVEN\ trains}) \wedge b \in \mathbb{P}(\mathbf{GIVEN\ trains}) \\
& \quad \Rightarrow \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ trains})] \\
& \quad \in \{\{x : \mathbb{P}(\mathbf{GIVEN\ trains}) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbf{GIVEN\ trains}) \wedge \neg(\neg(x \in a) \wedge \neg(x \in b))\}\}, \\
& \quad \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}}, \\
& \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}] \\
& \quad \in \{\{f : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))) \\
& \quad \quad \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \bullet f \in \bowtie \leftrightarrow \bowtie \\
& \quad \quad \quad [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}}, \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}] \\
& \quad \wedge \forall x \\
& \quad \quad \bullet x \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\} \\
& \quad \quad \Rightarrow \exists_1 y \bullet y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge \langle x, y \rangle \in f\}\} \\
& \quad \bowtie \leftrightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}}, \\
& \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}] \\
& \quad \in \{\mathbb{P}\{s, t : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))) \\
& \quad \quad \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \bullet s \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\} \\
& \quad \quad \quad \wedge t \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}\}\}, \\
& \quad \emptyset)\}
\end{aligned}$$

Once we have this reference we can apply rules 6.20 and 6.33 appropriately, to give the semantic binding for the entire application expression which is shown below.

$$\begin{aligned}
& S^e(\{\bowtie \cup \bowtie (trainsInYard, \{trainEnteringYard?\}), \Sigma\}) \\
& = \{ZF(\{\downarrow var \downarrow\}), \\
& \quad ZF(\{(trains \in \mathbb{P}(\mathbf{GIVEN\ trains}), trainsInYard \in trains, \\
& \quad \quad trainsInYard' \in trains, trainEnteringYard? \in trains, \\
& \quad \quad \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ trains})] \\
& \quad \quad \in \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}, \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}], \\
& \quad \quad \forall a\ b \bullet a \in \mathbb{P}(\mathbf{GIVEN\ trains}) \wedge b \in \mathbb{P}(\mathbf{GIVEN\ trains}) \\
& \quad \quad \Rightarrow \bowtie \cup \bowtie [\mathbb{P}(\mathbf{GIVEN\ trains})] \\
& \quad \quad \in \{\{x : \mathbb{P}(\mathbf{GIVEN\ trains}) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbf{GIVEN\ trains}) \wedge \neg(\neg(x \in a) \wedge \neg(x \in b))\}\}, \\
& \quad \quad \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}, \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}], \\
& \quad \quad \in \{\{f : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))) \\
& \quad \quad \quad \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet f \in \bowtie \leftrightarrow \bowtie \\
& \quad \quad \quad \quad [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}, \\
& \quad \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}]\} \\
& \quad \quad \wedge \forall x \\
& \quad \quad \quad \bullet x \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\} \\
& \quad \quad \quad \Rightarrow \exists_1 y \bullet y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge \langle x, y \rangle \in f\} \\
& \quad \quad \bowtie \leftrightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}, \\
& \quad \quad \quad \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}], \\
& \quad \quad \in \{\mathbb{P}\{s, t : (\mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))) \\
& \quad \quad \quad \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \bullet s \in \{x, y : \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \times \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \\
& \quad \quad \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains})) \wedge y \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\} \\
& \quad \quad \quad \quad \wedge t \in \mathbb{P}(\mathbb{P}(\mathbf{GIVEN\ trains}))\}\}, \\
& \quad \quad ZF(\{\forall x \bullet \langle \langle trainsInYard, \{trainEnteringYard?\} \rangle, x \rangle \\
& \quad \quad \quad \in \bowtie \cup \bowtie [\mathbf{GIVEN\ trains}] \Leftrightarrow x \in results\}, var \in results\})\}
\end{aligned}$$

We can now put this information together to create the semantic information for the membership predicate in accordance with rule 6.12.

$$\begin{aligned}
& \mathbf{S}^p(\text{trainsInYard}' \in \{\bowtie \cup \bowtie (\text{trainsInYard}, \{\text{trainEnteringYard}'\})\}, \Sigma) \\
&= (\mathbf{ZF}(\text{trainsInYard}' \in \{\langle \! \langle \text{var} \! \rangle \! \rangle\} \wedge \text{var} \in \text{results} \wedge \\
&\quad \forall x \bullet \langle \langle \text{trainsInYard}, \{\text{trainEnteringYard}'\} \rangle, x \rangle \\
&\quad \in \bowtie \cup \bowtie [\mathbf{GIVEN} \text{trains}] \Leftrightarrow x \in \text{results}), \\
&\quad \{\mathbf{ZF}(\text{trainsInYard}', \text{trainsInYard}, \text{trainEnteringYard}', \\
&\quad \bowtie \cup \bowtie [\mathbf{GIVEN} \text{trains}])\})
\end{aligned}$$

We can now complete the semantic binding environment for our schema construction expression using rule 6.26. (We declare two constants, h and p to help abbreviate our large list of hypotheses and goals.)

$$\begin{aligned}
h_{post} = & \text{ZF}(\{(trains \in \mathbb{P}(\text{GIVEN } trains)), \\
& \bowtie \cup \bowtie [\mathbb{P}(\text{GIVEN } trains)] \\
& \in \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}], \\
& \quad \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}], \\
& \forall a \ b \bullet a \in \mathbb{P}(\text{GIVEN } trains) \wedge b \in \mathbb{P}(\text{GIVEN } trains) \\
& \Rightarrow \bowtie \cup \bowtie [\mathbb{P}(\text{GIVEN } trains)] \\
& \quad \in \{\{x : \mathbb{P}(\text{GIVEN } trains) \\
& \quad \bullet x \in \mathbb{P}(\text{GIVEN } trains) \wedge \neg(\neg(x \in a) \wedge \neg(x \in b))\}\}, \\
& \bowtie \rightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}], \\
& \quad \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}] \\
& \in \{\{f : (\mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))) \\
& \quad \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \bullet f \in \bowtie \leftrightarrow \bowtie \\
& \quad \quad [\{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}], \\
& \quad \quad \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}] \\
& \wedge \forall x \\
& \quad \bullet x \in \{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\} \\
& \quad \Rightarrow \exists_1 y \bullet y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge \langle x, y \rangle \in f\} \\
& \bowtie \leftrightarrow \bowtie [\{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}], \\
& \quad \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}] \\
& \in \{\mathbb{P}\{s, t : (\mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))) \\
& \quad \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \bullet s \in \{x, y : \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \times \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \\
& \quad \quad \bullet x \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains)) \wedge y \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\} \\
& \quad \wedge t \in \mathbb{P}(\mathbb{P}(\text{GIVEN } trains))\}\}\}
\end{aligned}$$

Each semantic binding in a schema construction expression's semantic binding environment includes not just the hypotheses from the component expression, but also hypotheses resulting from the use of the reference variables in the component predicate. In this instance, the hypotheses for each of our bindings are identical (as they each only make reference to *trains*).

$$\begin{aligned}
p_{post} = & \{ZF(trainsInYard \in \mathbb{P} \text{trains}, trainsInYard' \in \mathbb{P} \text{trains}, \\
& \text{trainEnteringYard?} \in \text{trains}, \\
& (\text{trainsInYard}' \in \{\langle \text{var} \rangle\} \wedge \text{var} \in \text{results} \wedge \\
& \forall x \bullet \langle \langle \text{trainsInYard}, \{\text{trainEnteringYard?}\} \rangle, x \rangle \\
& \in \bowtie \cup \bowtie [\text{GIVEN } \text{trains}] \Leftrightarrow x \in \text{results})\}
\end{aligned}$$

As with the hypotheses, the goals are identical for each of the semantic bindings in the schema construction's semantic binding environment and are formed from the union of the goals resulting from the component expression and the predicate of the component predicate.

$$\begin{aligned}
& S^e([\text{trainYard_state} \wedge \text{trainYard_state}' \\
& \quad \wedge \text{trainYard_trainEntersYard_inputs} \\
& \quad | \text{trainsInYard}' \\
& \quad \in \{\bowtie \cup \bowtie (\text{trainsInYard}, \{\text{trainEnteringYard?}\}), \Sigma\}) \\
= & \{(ZF(\text{trainsInYard}), h, p), (ZF(\text{trainsInYard}'), h, p), \\
& (ZF(\text{trainEnteringYard?}), h, p)\}
\end{aligned}$$

The completion of the schema's construction allows us to see the meaning of our semantic model. Our semantic binding environment contains three variables, which given a set of hypotheses (references outside of the paragraph) are constrained by a set of predicates (the goals). If we re-examine our original schema, we can see the correlation between this meaning, and our original intent.

We now apply rule 6.20, to form the semantic binding environment for the set extension expression. As we have explained previously, this set is formed from the semantic bindings contained in the schema constructions semantic binding environment.

$$\begin{aligned}
& S^e(\{\{[\text{trainYard_state} \wedge \text{trainYard_state}' \\
& \quad \wedge \text{trainYard_trainEntersYard_inputs} \\
& \quad | (\text{trainEnteringYard?}, \text{trainsInYard}) \in \bowtie \not\in \bowtie \\
& \quad \wedge \text{trainsInYard}' \\
& \quad \in \{\bowtie \cup \bowtie (\text{trainsInYard}, \{\text{trainEnteringYard?}\})\}], \Sigma\}) \\
= & \{(ZF(\{\langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle\}), h, p)\}
\end{aligned}$$

This set extension expression forms the component expression in a variable

construction expression, and we can now apply rule 6.25 to determine that variable construction expression's semantic binding environment.

$$\begin{aligned}
& S^e([\textit{trainYard_trainEntersYard_post} : \{ \\
& \quad [\textit{trainYard_state} \wedge \textit{trainYard_state}' \\
& \quad \wedge \textit{trainYard_trainEntersYard_inputs} \\
& \quad | (\textit{trainEnteringYard?}, \textit{trainsInYard}) \in \bowtie \not\in \bowtie \\
& \quad \wedge \textit{trainsInYard}' \\
& \quad \in \{\bowtie \cup \bowtie (\textit{trainsInYard}, \{\textit{trainEnteringYard?}\})\}], \Sigma) \\
& = \{(\text{ZF}(\textit{trainYard_trainEntersYard_post}), h \cup p, \\
& \quad \{\text{ZF}(\textit{trainYard_trainEntersYard_post} \\
& \quad \in \{\downarrow \textit{trainsInYard}, \textit{trainsInYard}', \textit{trainEnteringYard?} \downarrow\})\})\}
\end{aligned}$$

This now completes the generation of the semantic binding environment for our paragraph as rule 6.8 states that the semantic binding environment for such a paragraph is identical to that of its component expression.

We will now show how a section's semantic binding environment is used to instantiate the generic proof obligations that have been specified in each construct's configuration. In example 6.3, we will show how the semantic binding environment we have created for this paragraph will be used to generate a machine operation's proof obligation.

6.3 Instantiation of Generic Proof Obligations

In section 5.2.2 we showed how a generic proof obligation could be instantiated using a construct's Frog-CCL configuration and specification. Of course, this method of instantiation produces a proof obligation that uses the Z notation. In order that we can pass our proof obligations to theorem provers we must describe them using the syntax of those theorem provers. As we may wish to interface with more than one theorem prover we initially create our proof obligations in the theorem-prover independent ZF language that we described in section 6.2.1. This section describes the process that we use to generate those theorem-prover independent proof obligations using the configuration of a construct and its semantic model.

When parsing the configuration for each construct type, we create a tree

that represents each of the specified proof obligations. For instance, the proof obligation that we associated with a machine's initialization in example 5.4, can be represented by the following tree.

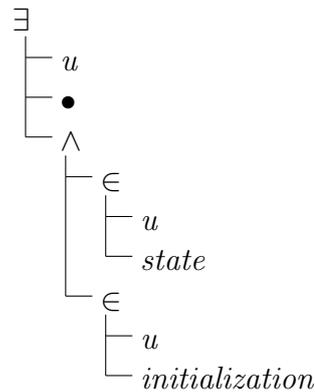


Figure 6.3: Tree for Initialization Proof Obligation of Machine Configuration

When we come to instantiate these proof obligations for a specific construct, our starting point is the proof obligation trees that are associated with that construct's type. In practice, we use ANTLR to create a tree walker that parses the tree in the same way as when performing syntax transformation in Z (see section 4.6.5). In this instance we don't construct another tree, but instead use the actions associated with each rule to build a proof obligation in our ZF language. We will parse this tree once for construct level proof obligations, and for operation level proof obligations we will parse one time for every valid operation environment in a machine, and every pair of matching operation environments in a relationship (the proof obligation is generated within the context of the appropriate operation environments).

The process for generating a proof obligation involves examining each of the nodes in turn (from root to leaves) and performing some action dependent on the nature of that node. The next part of this section details the actions that need to be performed for each node type. We then present a simple example showing how proof obligations are created from the semantic model produced in example 6.1. The proof obligation generated at each stage consists of a set of hypotheses, and a statement which must be proved on the

assumption of those hypotheses.

Universal Quantification

The first step when processing a universal quantification node is to create ZFName objects for the bound variables and add these variables to the scope. Each of these bound variable has a flag that will record whether they are actually used in this particular instantiation.

Once we have done this, we proceed to instantiate the proof obligation for the child node of the universal quantification. If this proof obligation is invalid or undefined, then so is the proof obligation of the universal quantification. Otherwise, we determine which of our bound variables has been actually used and then use these along with the hypotheses and statement of the child node's proof obligation to create an instance of ZFPredicateUniv (note that, we never generate a constraint for our universal quantifications at this time).

Existential Quantification

The process for instantiating the proof obligation of an existential quantification node, is almost identical to that for a universal quantification node, with the obvious exception being that we create a ZFPredicateExists node to form the statement of our proof obligation.

Implication

Firstly, we instantiate the proof obligations for the two children of the implication. If either is found to be invalid then the proof obligation for our implication is also invalid. If one of our two children is found to be undefined then the proof obligation for the implication is equal to the proof obligation created from the defined child. If both children are undefined, however, then the implication's proof obligation is also undefined. If we find that the proof obligations for both children are identical then the proof obligation for the implication is equal to either.

In the remaining, default instance we merge the sets of hypotheses from the child proof obligations, and create a new instance of a `ZFPredicateImplication` from the combination of their statements.⁹ This new set of hypotheses and the conjoined statement form the proof obligation for the implication.

Conjunction

The process for calculating the proof obligation of a conjunction node is very similar to that for an implication node. The first difference is that when we compare the proof obligations of the two child nodes, we also check whether one has been incorporated in the other, and in that instance we simply set the proof obligation of the conjunction to be equal to that of the larger of these. For example, we may have the following two statements.

$$P \wedge Q \wedge R \qquad P \wedge Q$$

The proof obligation for the conjunction becomes the larger of the child conjunctions (that is $P \wedge Q \wedge R$) rather than the conjunction of the two terms. This helps in reducing the size of our proof obligations.

The other difference is that in the default case the proof obligation's statement is created from an instance of `ZFPredicateConjunction`.

Disjunction

The proof obligation for a disjunction node is calculated in a very similar way to a conjunction. The only difference is in the default case where the proof obligation's statement is produced by creating an instance of `ZFPredicateDisjunction` from the combination of the child obligation's statements.

Negation

Firstly, we instantiate the proof obligation for the child node of the negation. If this node is invalid or undefined then so is the negation. Otherwise, the set

⁹In all cases where we create a statement from two or more other statements, we use instances of `ZFPredicateParentheses` to ensure that we maintain the correct precedence and associativity.

of hypotheses for the negation's proof obligation is equal to that of its child's and the statement in a new instance of `ZFPredicateNegation` that contains the child proof obligation's statement.

Membership

A membership node has two 'atoms' as children rather than proof obligations. These atoms refer either to bound variables or the sets derived from a construct's clauses. There are three types of atom: a variable, a tuple or a set. A variable refers to the `ZFName` of a single bound variable, a tuple a list of such references and the set gives an instance of `ZFSetSeparation`, a set of hypotheses needed when creating the set, and an indication of the shape of that set's members.

When calculating the proof obligation, we can say that if the second atom (representing the set) is invalid or undefined, then the proof obligation for the membership node is also. Further to this, the way in which the proof obligation is determined, depends upon the nature of the first atom (that is, the element we are testing for membership of the set represented by the second atom). If this atom is a variable, and is not invalid, then the proof obligation is created from the set of hypotheses belonging to the set atom, and an instance of `ZFPredicateMembership` created from the `ZFName` of the variable and the `ZFSetSeparation` of the set.

If the first atom is a tuple atom then we must first determine whether all of the elements to which its members refer have been defined. We discussed the need to eliminate unused clauses in example 5.4, presenting a set of rules in table 5.1. This is achieved by first retrieving the indication of the shape of the set atom's members, this is represented as a list of all the members of the default shape that indicates which have been defined and which have not. For example, we may have a membership tree as follows.

$$\begin{array}{l} \in \\ \left[\begin{array}{l} \langle u, v, i, j \rangle \\ \textit{within} \end{array} \right. \end{array}$$

In a particular instance, calculating the within clause may reveal that there are no inputs to the operation of either of its involved machines. In which case, we would receive the following indication from the set atom.

$$\langle \text{DEFINED}, \text{DEFINED}, \text{UNDEFINED}, \text{UNDEFINED} \rangle$$

We would then eliminate the elements of our tuple that correspond to the undefined elements in this list (as we mentioned previously we record the elimination of these bound variables so that we only use the necessary variables in our parent quantification expression), and we would be left with a tuple containing two elements rather than four (that is $\langle u, v \rangle$). The proof obligation for our membership node could then be created from the set of hypotheses belonging to the atom, and an instance of `ZFPredicateMembership` created from the `ZFOrdered` object that represents the tuple and the `ZFSetSeparation` of the set.

Equality

Like membership, the children of an equality node must be atoms. The atoms on either side must be of the same type. If either is invalid or undefined then so is the proof obligation of the equality node. If the atoms are variables or tuples, then the proof obligation contains an empty set of hypotheses and the statement is formed from an instance of `ZFPredicateEquality` constructed with the appropriate `ZFName` or `ZFOrdered` objects. If the atoms are sets, then the set of hypotheses is the union of their hypotheses, and the statement again an instance of `ZFPredicateEquality` formed from the `ZFSetSeparation` objects.

References

Reference nodes are processed to produce the atoms referred to by membership and equality clauses, and may be of the following forms.

- *NAME*
- $\langle \text{NAME}, \dots, \text{NAME} \rangle$

- *NAME.NAME*
- *FROM_MACHINE(ref)*
- *TO_MACHINE(ref)*
- *PRE(ref)*

Each of these references is handled slightly differently, but falls into one of three categories: references to bound variables, references to clauses, and references to weakest preconditions.

The first two examples can refer to either a bound variable or a list of bound variables respectively. In the first instance we simply create a variable atom object which refers to the bound variable and its ZFName. Where we have a list of bound variables we create a tuple atom which contains a ZFOrdered object that we create to store the ordered list of variables.

A single name can also refer to a construct level clause in the current construct. Similarly, the dot notation is used to refer to operation level clauses, and the from and to notations allow us – when in a relationship – to refer to the clauses of source and target machines respectively.

When we refer to a clause, we first determine whether that clause is valid in the specified construct, if not we set the atom to be invalid. We then determine whether the clause has been defined in the construct (or operation environment depending on the level), and if not we set the atom to be undefined. We then retrieve the relational structure of the clause from its configuration. For instance, if we were in a retrenchment relationship and made a reference to *ramifications.within*, then we would retrieve the following relational structure.

$$\langle FROM_MACHINE(state), TO_MACHINE(state), \\ FROM_MACHINE(inputs), TO_MACHINE(inputs) \rangle$$

We then determine whether each of these is valid and defined in the appropriate construct (and operation environment within, where necessary) and make a note of which clauses are defined and which are not (any invalid

clauses will lead to the whole atom being declared invalid). We then extract the schema variables for each of the defined clauses (priming where necessary) and together with their types we can create a typed expression for our clause's relational definition. That is, we have a relational definition as follows.

$$\{expression \bullet predicate\}$$

The expression part of this definition is derived solely from the schema variables of the referenced clauses in the relation attribute of the clause's configuration. For example, suppose we have the following two machines.

```
MACHINE machineOne
TYPE machine
SECTION standard_toolkit
STATE
```

$$a : \mathbb{N}$$

```
OPERATION increment  $\hat{=}$ 
POST
```

$$\frac{}{a' = a + 1}$$

```
END increment
```

```
END machineOne
```

```
MACHINE machineTwo
TYPE machine
SECTION standard_toolkit
STATE
```

$$b : \mathbb{N}$$

OPERATION *increment* $\hat{=}$
 POST

$$\frac{}{b' = b + 1}$$

END *increment*

END *machineTwo*

Suppose we have specified that the within clause belonging to the ramifications for the *increment* operation in a retrenchment between the two machines (which we have aliased as $m1$ and $m2$) is $m1 \gg a = m2 \gg b$. Then we would retrieve the schema variables and types to give the following beginning to our relational definition of the within clause ¹⁰.

$$\{\langle m1 \gg a, m1 \gg b \rangle : \text{GIVEN } \mathbb{A} \times \text{GIVEN } \mathbb{A} \bullet \text{predicate}\}$$

We then proceed to determine the predicate, which we do by examining the semantic model of the construct. The first step is to retrieve the semantic binding for the clause, which we do by retrieving the semantic binding of the clause's paragraph's name from the construct's semantic binding environment. We can then retrieve the semantic bindings of all the variables used in the clause using the method described in rule 6.20. The predicate part of our relational definition is then formed from the conjunction of these semantic binding's goals. (The union of their hypotheses will also form the hypotheses for the set atom we create for the reference node.) In our example, this will give us the following relational definition of the within clause.

$$\{\langle m1 \gg a, m2 \gg b \rangle : \text{GIVEN } \mathbb{A} \times \text{GIVEN } \mathbb{A} \\ \bullet m1 \gg a \in \mathbb{N} \wedge m2 \gg b \in \mathbb{N} \wedge m1 \gg a = m2 \gg b\}$$

¹⁰Note, that where a single clause introduces more than one variable, they are grouped within the declaration to form a typed tuple.

We use this definition to create an instance of `ZFSetSeparation`, which together with the set of hypotheses and an indication of the shape of the set's members (that is which of the clause's relation elements have been defined and which have not) is used to create the set atom for the reference node.

The final type of reference is to the precondition of a clause's schema. The schemas that represent the weakest preconditions for each schema belonging to a construct are created using the method, alluded to earlier and defined in [WD96]. When we refer to the preconditions of clauses we use the same method as more direct references to other clauses. The principal difference is that as there is no configuration for these precondition schemas, there is no relational attribute from which we can determine whether all clauses have been used. Therefore, we use the 'pre-relation' attribute of the clause to which we refer.

Example 6.2

In this section, we have defined the rules that allow us to instantiate a generic proof obligation for a construct using the semantic model of that construct. We will illustrate this process by revisiting the train yard example, and instantiating a proof for one of its operations.

We begin by examining the tree produced from the generic proof obligation, which is shown in figure 6.4. As in previous examples, whilst the process actually works from root to leaves, it is easier to describe from the bottom-up.

We will begin, therefore, with the first part of our implication ($u \in state \wedge i \in operation.inputs \wedge \langle u, i \rangle \in operation.pre$). The first of the memberships involves a reference to a bound variable (which produces a variable atom, $ZF(u)$) and a reference to a construct-level clause. The relation attribute for the state clause is self referencing, so we produce the following beginning to the construction of our relational view.

$$\{trainsInYard : \mathbb{P}(\text{GIVEN } trains) \bullet predicate\}$$

The semantic binding for the state paragraph in this instance is then derived

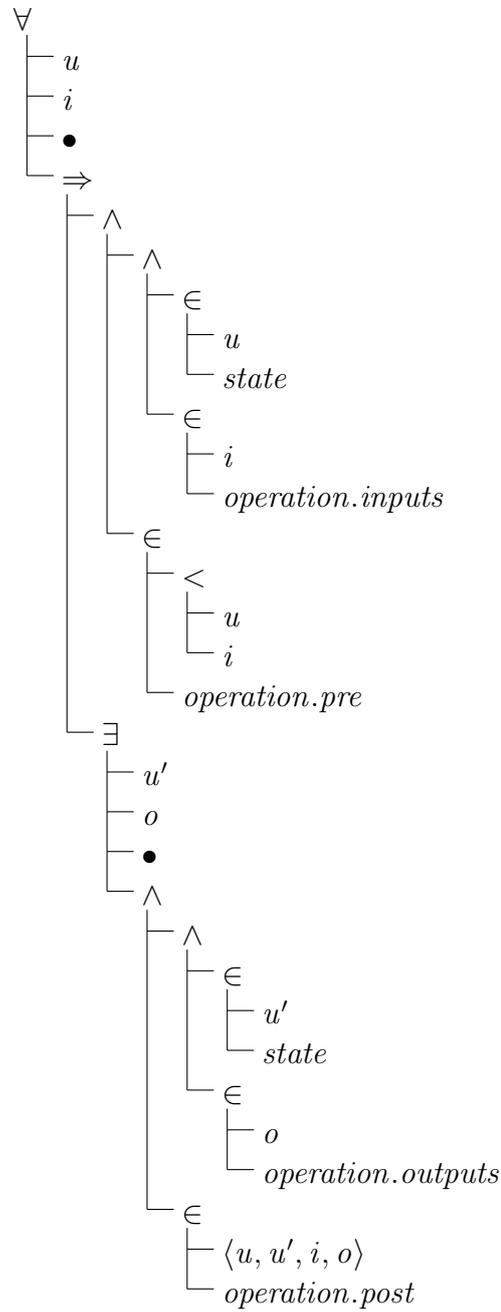


Figure 6.4: Tree for Operation Proof Obligation of Machine Configuration

from the semantic model of the construct (Σ) and is shown below.

$$\begin{aligned} & \mathbf{Q}(\Sigma, _trainYard_state) \\ & == (\mathbf{ZF}(_trainYard_state), \{\mathbf{ZF}(trains \in \mathbf{GIVEN} \ trains)\}, \\ & \quad \{\mathbf{ZF}(_trainYard_state \in \{\langle \ trainsInYard \ \rangle\}) \}) \end{aligned}$$

This allows us to retrieve the semantic binding for each of the schema variables from the set of bindings associated with the paragraph (in this instance, just *trainsInYard*). This semantic binding is shown below.

$$\begin{aligned} & (\mathbf{ZF}(trainsInYard), \{\mathbf{ZF}(trains \in \mathbf{GIVEN} \ trains)\}, \\ & \quad \{\mathbf{ZF}(trainsInYard \in \mathbb{P} \ trains)\}) \end{aligned}$$

This allows us to complete the construction of our relational view, giving the following set separation expression.

$$\{trainsInYard : \mathbb{P}(\mathbf{GIVEN} \ trains) \bullet trainsInYard \in \mathbb{P} \ trains\}$$

From this, we are able to create a set atom (the hypotheses of *trainsInYard* yard are passed on, and all referenced clauses have been used) and with that the proof obligation for the first of our memberships.

$$\begin{aligned} & \mathbf{ZF}(trains \in \mathbf{GIVEN} \ trains) \\ & \vdash? \\ & \mathbf{ZF}(u \in \{trainsInYard : \mathbb{P}(\mathbf{GIVEN} \ trains) \bullet trainsInYard \in \mathbb{P} \ trains\}) \end{aligned}$$

The proof obligation for the second membership can be determined in exactly the same way, and we can conjoin those proof obligations to give the following proof obligation.

$$\begin{aligned} & \mathbf{ZF}(trains \in \mathbf{GIVEN} \ trains) \\ & \vdash? \\ & \mathbf{ZF}(u \in \{trainsInYard : \mathbb{P}(\mathbf{GIVEN} \ trains) \bullet trainsInYard \in \mathbb{P} \ trains\} \\ & \quad \wedge i \in \{trainEnteringYard? : \mathbf{GIVEN} \ trains \\ & \quad \quad \bullet trainEnteringYard? \in \ trains\}) \end{aligned}$$

When we consider the third membership, things become a little more complex as we make a reference to the *operation.pre* clause which in turn references the relation ‘not-in’ (\notin). The semantic binding for the clause’s paragraph is

shown below (again we will use some constants to avoid needless repetition).

$$\begin{aligned}
h_{pre} = & \text{ZF}(\{(trains \in \mathbb{P}(\text{GIVEN } trains), \\
& \bowtie \notin \bowtie [\text{GIVEN } trains] \in \{ \bowtie \leftrightarrow \bowtie [\text{GIVEN } trains, \mathbb{P}(\text{GIVEN } trains)] \}, \\
& \forall x a \bullet x \in \text{GIVEN } trains \wedge a \in \mathbb{P}(\text{GIVEN } trains) \Rightarrow \\
& \quad \neg (\langle x, a \rangle \in \bowtie \notin \bowtie [\text{GIVEN } trains] \wedge \neg \neg (x \in a)) \\
& \quad \wedge \neg (\neg (\langle x, a \rangle \in \bowtie \notin \bowtie [\text{GIVEN } trains]) \wedge \neg (x \in a))\}, \\
& \bowtie \leftrightarrow \bowtie [\text{GIVEN } trains, \mathbb{P}(\text{GIVEN } trains)] \\
& \in \{ \mathbb{P}\{x, y : \text{GIVEN } trains \times \mathbb{P}(\text{GIVEN } trains) \\
& \quad \bullet x \in \text{GIVEN } trains \wedge y \in \mathbb{P}(\text{GIVEN } trains)\} \},
\end{aligned}$$

$$\begin{aligned}
p_{pre} = & \{ \text{ZF}(trainsInYard \in \mathbb{P} \text{ trains}, trainEnteringYard? \in \text{trains} \\
& (trainEnteringYard?, trainsInYard) \in \bowtie \notin \bowtie [\text{GIVEN } trains] \}
\end{aligned}$$

$$\begin{aligned}
& \text{Q}(\Sigma, _trainYard_trainEntersYard_pre) \\
= & \{ (\text{ZF}(_trainYard_trainEntersYard_pre), h_{pre} \cup p_{pre}, \\
& \{ \text{ZF}(_trainYard_trainEntersYard_pre \\
& \quad \in \{ \langle _trainsInYard, _trainEnteringYard? \rangle \} \} \} \}
\end{aligned}$$

We use this semantic binding to instantiate this part of the proof obligation. Again, the first step is to retrieve the relational shape from the clause's configuration, which gives us the following.

$$\langle state, operation.inputs \rangle$$

From this we can produce the following beginning to our clause's relational definition.

$$\begin{aligned}
& \{ trainsInYard : \mathbb{P}(\text{GIVEN } trains); trainEnteringYard? : \text{GIVEN } trains \\
& \bullet predicate \}
\end{aligned}$$

To instantiate our predicate we examine the semantic model of the clause's paragraph that we presented above. From this we are able to determine the semantic bindings for the variables required within our relational definition.

$$\begin{aligned} & (\text{ZF}(\text{trainsInYard}), h_{pre}, p_{pre}) \\ & (\text{ZF}(\text{trainEnteringYard?}), h_{pre}, p_{pre}) \end{aligned}$$

From here we are able to complete the relational definition for our clause and instantiate the proof obligation for the third membership. This gives us the following proof obligation. Note that, we define a function `list` that takes a set of hypotheses and converts it into a list for use within our proof obligation.

$$\begin{aligned} & \text{list}(h_{pre}) \\ & \vdash? \\ & \text{ZF}(\langle u, i \rangle \in \{(\text{trainsInYard}, \text{trainEnteringYard?}) \\ & \quad : \mathbb{P}(\text{GIVEN trains}) \times \text{GIVEN trains} \\ & \quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\ & \quad \wedge \text{trainEnteringYard?} : \text{trains} \\ & \quad \wedge ((\text{trainEnteringYard?}, \text{trainsInYard}) \\ & \quad \in \bowtie \not\in \bowtie [\text{GIVEN trains}])\}) \end{aligned}$$

We can now combine this with our previous results to present the instantiated proof obligation for the first part of our implication. This gives us the following:

$$\begin{aligned} & \text{list}(h_{pre}) \\ & \vdash? \\ & \text{ZF}(u \in \{\text{trainsInYard} : \mathbb{P}(\text{GIVEN trains}) \bullet \text{trainsInYard} \in \mathbb{P} \text{trains}\} \\ & \wedge i \in \{\text{trainEnteringYard?} : \text{GIVEN trains} \\ & \quad \bullet \text{trainEnteringYard?} \in \text{trains}\} \\ & \wedge \langle u, i \rangle \in \{(\text{trainsInYard}, \text{trainEnteringYard?}) \\ & \quad : \mathbb{P}(\text{GIVEN trains}) \times \text{GIVEN trains} \\ & \quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\ & \quad \wedge \text{trainEnteringYard?} : \text{trains} \\ & \quad \wedge ((\text{trainEnteringYard?}, \text{trainsInYard}) \\ & \quad \in \bowtie \not\in \bowtie [\text{GIVEN trains}])\}) \end{aligned}$$

We can now consider the second part of our implication ($\exists u', o \bullet (u' \in \text{state} \wedge o \in \text{operation.outputs}) \wedge \langle u, u', i, o \rangle \in \text{operation.post}$). Here we have an existential quantification over three conjoined terms. The first of the

conjoined terms is the primed version of the state membership we discussed above and produces an equivalent proof obligation.

$$\begin{aligned} & \text{ZF}(\text{trains} \in \text{GIVEN trains}) \\ & \vdash? \\ & \text{ZF}(u' \in \{\text{trainsInYard} : \mathbb{P}(\text{GIVEN trains}) \bullet \text{trainsInYard} \in \mathbb{P} \text{trains}\}) \end{aligned}$$

When we consider the second membership we find that the *operation.outputs* clause has not been defined for the current operation environment of our construct. The atom produced from this clause is therefore undefined, and the proof obligation for the membership is also undefined. As our rules above state, when we have a conjunction between a defined term and an undefined term, the proof obligation for the conjunction is equivalent to that of the defined term and is therefore, the one stated above.

The last of the conjoined terms uses the *operation.body* clause so our first step in deriving its relational definition is to retrieve the clause's relation attribute, which is shown below.

$$\langle \text{state}, \text{state}', \text{operation.inputs}, \text{operation.outputs} \rangle$$

This allows us to create the declaration part of our relational definition (noting that no outputs clause has defined).

$$\begin{aligned} & \{ \langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle \\ & : \mathbb{P}(\text{GIVEN trains}) \times \mathbb{P}(\text{GIVEN trains}) \times \text{GIVEN trains} \bullet \text{predicate} \} \end{aligned}$$

The next stage is to create the predicate. This requires the use of the semantic model produced in example 6.1. We present this below for convenience (note that we shall continue to use the constants h_{post} and p_{post} that we defined in that example as abbreviations for the hypotheses and goals defined).

$$\begin{aligned} & \text{Q}(\Sigma, _trainYard_trainEntersYard_body) \\ & == (\text{ZF}(\text{trainEntersYard}), h_{post} \cup p_{post}, \\ & \quad \{ \text{ZF}(\text{trainEntersYard} \\ & \quad \quad \in \{ \langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle \} \} \}) \end{aligned}$$

From this we can derive the following semantic bindings for the schema variables of our clause.

$$\begin{aligned} &(\text{ZF}(\text{trainsInYard}), h_{\text{post}}, p_{\text{post}}) \\ &(\text{ZF}(\text{trainsInYard}'), h_{\text{post}}, p_{\text{post}}) \\ &(\text{ZF}(\text{trainEnteringYard?}), h_{\text{post}}, p_{\text{post}}) \end{aligned}$$

We are now in a position to instantiate the proof obligation for the membership. We note that the outputs clause has not been defined, so we remove the bound variable, o , from our tuple atom. We also record that this bound variable has not been used, so it will not be introduced by our existential quantifier.

$$\begin{aligned} &\text{list}(h_{\text{post}}) \\ &\vdash? \\ &\text{ZF}(\langle u, u', i \rangle \in \{\langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle \\ &\quad : \mathbb{P}(\text{GIVEN } \text{trains}) \times \mathbb{P}(\text{GIVEN } \text{trains}) \\ &\quad \times \text{GIVEN } \text{trains} \\ &\quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\ &\quad \wedge \text{trainsInYard}' : \mathbb{P} \text{trains} \\ &\quad \wedge \text{trainEnteringYard?} : \text{trains} \\ &\quad \wedge \text{trainsInYard}' \in \{\text{var}\} \\ &\quad \wedge \text{var} \in \text{results} \\ &\quad \wedge \forall x \bullet \langle \langle \text{trainsInYard}, \{\text{trainEnteringYard?}\} \rangle, x \rangle \\ &\quad \quad \in \bowtie \cup \bowtie [\text{GIVEN } \text{trains}] \Leftrightarrow x \in \text{results})) \end{aligned}$$

Using these proof obligations we are able to combine the conjoined terms and instantiate the proof obligation for the second part of our implication, giving the following obligation. Remembering, of course, to eliminate the bound variable, o .

$$\begin{aligned}
& \text{list}(h_{post}) \\
& \vdash? \\
& \text{ZF}(\exists u' \bullet u' \in \{\text{trainsInYard} : \mathbb{P}(\text{GIVEN trains}) \bullet \text{trainsInYard} \in \mathbb{P} \text{trains}\}) \\
& \quad \wedge \langle u, u', i \rangle \in \{\langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle \\
& \quad \quad \quad : \mathbb{P}(\text{GIVEN trains}) \times \mathbb{P}(\text{GIVEN trains}) \\
& \quad \quad \quad \times \text{GIVEN trains} \\
& \quad \quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\
& \quad \quad \wedge \text{trainsInYard}' : \mathbb{P} \text{trains} \\
& \quad \quad \wedge \text{trainEnteringYard?} : \text{trains} \\
& \quad \quad \wedge \text{trainsInYard}' \in \{\text{var}\} \\
& \quad \quad \wedge \text{var} \in \text{results} \\
& \quad \quad \wedge \forall x \bullet \langle \langle \text{trainsInYard}, \{\text{trainEnteringYard?}\} \rangle, x \rangle \\
& \quad \quad \quad \in \bowtie \cup \bowtie [\text{GIVEN trains}] \Leftrightarrow x \in \text{results})\}
\end{aligned}$$

This can then be combined with the proof obligation for the first part of our implication, and used to create the proof obligation for our root, universal quantifier node. This will give a final proof obligation for our operation as follows.

$$\begin{aligned}
& \text{list}(h_{pre} \cup h_{post}) \\
& \vdash? \\
& \text{ZF}(\forall u, i \bullet u \in \{\text{trainsInYard} : \mathbb{P}(\text{GIVEN trains}) \\
& \quad \bullet \text{trainsInYard} \in \mathbb{P} \text{trains}\} \\
& \wedge i \in \{\text{trainEnteringYard?} : \text{GIVEN trains} \\
& \quad \bullet \text{trainEnteringYard?} \in \text{trains}\} \\
& \wedge \langle u, i \rangle \in \{\langle \text{trainsInYard}, \text{trainEnteringYard?} \rangle \\
& \quad : \mathbb{P}(\text{GIVEN trains}) \times \text{GIVEN trains} \\
& \quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\
& \quad \wedge \text{trainEnteringYard?} : \text{trains} \\
& \quad \wedge ((\text{trainEnteringYard?}, \text{trainsInYard}) \\
& \quad \in \bowtie \notin \bowtie [\text{GIVEN trains}]) \\
& \Rightarrow \\
& \exists u' \bullet u' \in \{\text{trainsInYard} : \mathbb{P}(\text{GIVEN trains}) \\
& \quad \bullet \text{trainsInYard} \in \mathbb{P} \text{trains}\} \\
& \wedge \langle u, u', i \rangle \in \{\langle \text{trainsInYard}, \text{trainsInYard}', \text{trainEnteringYard?} \rangle \\
& \quad : \mathbb{P}(\text{GIVEN trains}) \times \mathbb{P}(\text{GIVEN trains}) \\
& \quad \times \text{GIVEN trains} \\
& \quad \bullet \text{trainsInYard} : \mathbb{P} \text{trains} \\
& \quad \wedge \text{trainsInYard}' : \mathbb{P} \text{trains} \\
& \quad \wedge \text{trainEnteringYard?} : \text{trains} \\
& \quad \wedge \text{trainsInYard}' \in \{\text{var}\} \\
& \quad \wedge \text{var} \in \text{results} \\
& \quad \wedge \forall x \bullet \langle \langle \text{trainsInYard}, \{\text{trainEnteringYard?}\} \rangle, x \rangle \\
& \quad \in \bowtie \cup \bowtie [\text{GIVEN trains}] \Leftrightarrow x \in \text{results})\}
\end{aligned}$$

6.4 Theorem Provers

A mechanical system that can reason at a human level has long been the subject of extensive research. Indeed, the aim has existed for as long as modern computer systems, with the first theorem proving programs appearing alongside the first prototype computers in the 1950's. The first theorem prover of significance was the Logic Theory Machine proposed by Newell, Shaw and Simon [NSS57], who developed a tool to assist proofs in a small subset of propositional calculus and introduced the concept of discharging a proof by working from goal to hypotheses (backwards inference). From this point, logicians became interested in automated theorem proving and believed that

first-order logics provided the syntax and semantics for describing both everyday discourse and more complex pure mathematics. As the years have passed, automated theorem proving has become a thriving research area, and it has been established that no system will allow a computer to completely reproduce human reasoning. However, the results of this research have led to systems that can significantly reduce the human involvement in discharging a proof.

Theorem provers can be split into general categories: genuinely ‘automated’ theorem provers and interactive proof assistants. For first-order logic, identifying theorems is recursively enumerable. That is, given a theorem prover with infinite resources, a proof will always be found for any theorem where such a proof exists. Automated theorem provers, are those that utilize this exhaustive approach (the principal goal being the efficient implementation of this task with finite resources). The main disadvantage of this approach is that invalid statements (that is those for which no proof exists) cannot always be recognized. In these cases, a first-order theorem prover may fail to terminate and will continue to search for a proof *ad infinitum*. Despite these theoretical limits, practical automated theorem provers have been developed that are able to tackle increasingly complex problems with relatively limited resources.

Interactive theorem provers allow a human user to give hints to the system. Depending on the degree of automation, the prover can essentially be reduced to a proof checker – with the user providing the proof in a formal way – or the majority of the proof can be performed automatically – with the user simply outlining an abstract approach. Interactive theorem provers are used for a variety of tasks, but some may argue that their use in the field of mathematical theorem proving is becoming more limited as fully automatic systems have been increasingly able to prove more difficult theorems in this area – including some whose proof has long eluded human mathematicians (although some mathematicians claim that the poor readability of the proofs generated by automated theorem provers limits their value). The principal disadvantage of interactive theorem provers is that they require a proficient user. Whereas an automated theorem prover can simply involve the creation

of a theorem and the start of the prover, an interactive theorem prover requires a user to comprehend the entirety of the proof process. We could argue however, that automated theorem provers simply displace this requirement and that they require adaption to *efficiently* handle proofs of any particular ‘type’ – whereupon they will be able to handle many theorems of that ‘type’. Interactive proof assistants are much more effective in discharging one-off proofs, where the human prover is able to assist the theorem prover in handling the intuitive aspects (without any modification to the theorem prover itself) and typically allow a user to create strategies that will automatically discharge proofs of the same ‘type’. A further advantage of interactive theorem provers is that not all problems can be expressed in first order logic, and as interactive theorem provers are not restricted to fully automatable logics, they are able to make use of higher order logics, proving theorems that are beyond the scope of fully automated provers.

6.4.1 Otter

Otter [Kal01, McC03] is one of the oldest and most popular automated theorem provers available. It comes packaged with the Mace tool [McC01] that will search for counter-examples to unproved theorems.

Otter is a first-order resolution theorem prover. This means it proves a theorem by negating the statement to be proved and then attempts to prove this negated statement with the existing set of axioms. The discovery of a contradiction in this proof shows that the negated goal is inconsistent with those axioms, and thus the original theorem must be consistent.

We have stated previously that we chose to use ZF set theory to describe our proof obligations. However, ZF set theory is unsuitable for a first-order resolution theorem prover such as Otter as it can have no finite axiom system in first order logic. No particular axiom system is included with Otter, but traditionally von Neumann-Bernays-Gödel (NBG) axioms are used [BLM⁺87, Qua92].

Boyer et al showed in [BLM⁺87] that Gödel’s finite axiomizations for set theory can be used to prove theorems involving set theory within Otter’s

first-order logic. However, as Belifante notes in [Bel96].

One of the obvious drawbacks in using Gödel's formalism is that the usual class formation $\{x \mid p(x)\}$ is not available; instead, one must reformulate such expressions in terms of Gödel's primitives, ...

Whilst it would be possible to translate our proof obligations to a suitable NBG syntax, our reliance on the relational definitions of schemas – and hence the usual set-builder notation – would make the translation far more difficult.

6.4.2 Vampire

Vampire [RV01, RV02] is an award winning, first-order resolution theorem prover. These awards are the result of Vampire's ability to resolve a proof very quickly. Vampire claims to be extremely simple to use whilst still providing the power required for the most demanding user. Vampire is able to accept a range of syntaxes, including that of the Otter theorem prover. Vampire also generates detailed, first-order proofs that (it claims) are easily checkable by human mathematicians.

As with Otter, the main disadvantage to our use of Vampire is its restriction to first-order logic with equality. Whilst it is possible to expand set-theoretic language constructs into first-order predicate logic before passing to Vampire, any proofs generated in this manner would be essentially unreadable to the average user (without some sort of reverse translation).

One of Vampire's distinguishing features however, is its 'limited resource strategy' that allows it to find proofs quickly when either time or memory is a limited resource. This feature has led to suggestions that Vampire can be used alongside interactive theorem provers; where at each step Vampire will attempt to discharge any remaining sub-goals of the current proof.

6.4.3 SPASS

SPASS [Wei99, Wei01] is another example of a first-order resolution theorem prover. Like Otter and Vampire, it is unable to provide the necessary support

for ZF set theory and was unlikely to be of immediate use in discharging our proof obligations.

There exists a number of other automated theorem provers that exhibit the same characteristics as Otter, Vampire and SPASS. As these generally have similar advantages and disadvantages to those that we have discussed, and we have covered some of the most well known, we will not consider any further theorem provers in this vein.

6.4.4 PVS

PVS (Prototype Verification System) [ORS92, SCR96] is a specification language integrated with support tools and an interactive theorem prover. Unlike some interactive theorem provers, PVS's strives for a high level of automation. It is particularly aimed at the verification of large scale industrial specifications, where proofs are large but shallow (that is, the theorems are not particularly difficult to prove, but are difficult to handle due to their size).

PVS has its own specification language based on classical, typed higher-order logic. The focus on the verification of specification means that the syntax for the language is significantly different to that of other theorem provers. This sort of syntax lends itself to the shallow embedding of a specification language such as that we discussed in 3.2.2.

We have discussed above, that we made the decision to create our proof obligations with the ZF set theory. This decision was made on the basis that the set theory is the standard foundation for both Z and mathematics more generally. However, there is much debate over whether theorems concerning specifications are best expressed in set theory or higher order logic [Gor96]. The developers of PVS outline in section 1.2 of [OSRSC99] their belief that strong type checking is a vital part of specification (itself a controversial subject [LP99]) and that this led to their choice of higher order logic over set theory.

6.4.5 KIV

KIV (Karlsruhe Interactive Verifier) [BRSS99, BRS⁺00] is a tool for formal systems development. It can be employed for the development of safety critical systems from formal requirements specifications down to executable code, including the verification of safety requirements and the correctness of implementations. The heart of KIV is the interactive theorem prover which follows the Edinburgh LCF approach¹¹ [MJG79], and is used to provide specific proof support for all validation and verification tasks. KIV supports both the functional and the state-based approach to describe hierarchically structured systems. The functional approach uses higher-order algebraic specifications. For the state-based approach KIV offers three approaches: abstract state machines, parallel programs and state charts.

Again, the higher-order approach is not suitable for the proof obligations that Frog will generate, and whilst the state-based approach may be suitable for a shallow embedding, this is not the route that we had chosen to pursue.

6.4.6 Coq/LogiCal

Coq [BC04, Coq05] was an interactive proof assistant, the development of which was terminated in 2000. LogiCal is Coq's successor (although somewhat confusingly the Coq name is still used in many places including the program's documentation) and is a proof assistant which handles mathematical assertions of the 'calculus of inductive constructions' [Wer94] (a derivative of the calculus of constructions). Coq is not actually a theorem prover but includes automatic theorem proving tactics.

Using Coq would have presented similar problems to those exhibited in the use of PVS. Coq has its own specification syntax (Gallina) – in which a shallow embedding is feasible – and does not base its specifications on set theory, but creates them within its own strongly typed framework. In essence,

¹¹With the Edinburgh LCF approach, the theorem prover is implemented as a library in some programming language. This library implements a set of verified functions. New theorems can only be created by using the functions in the library. If the functions are correctly implemented, all theorems proven in the system must be valid. Hence, a large system can be built on top of a small trusted kernel.

Coq is a complete package for specification and verification. Attempting to separate these tasks by performing specification externally would simply lead to an embedding. Doing this would tie us to Coq, and this is not the approach that we envisaged.

6.4.7 HOL

The various HOL (Higher Order Logic) systems [GM93, NPW02] are a family of interactive theorem proving systems sharing similar logics and implementation strategies. All of these systems follow the Edinburgh LCF approach. As may be obvious from the name, the HOL system uses higher-order logic. Its distinguishing feature is the high degree of programmability that comes from using the meta language ML¹² [MTM97] as its syntax.

HOL is one of the oldest and most popular interactive theorem provers, and has numerous descendants, including ProofPower¹³ [Art91, KA96] and Isabelle (below).

As with PVS and KIV, the limiting factor here is the use of higher order logic rather than set theory. There are a selection of other interactive theorem provers that use higher order logic, but as using these will always lead to this limitation, we will not discuss them at this juncture.

¹²The language ML has now achieved status as a programming language in its own right, although it was originally designed as the proof management language for LCF.

¹³It has recently come to our attention that the ownership of the ProofPower theorem prover has transferred from ICL to Lemma1. A result of this change is that the source code of the ProofPower theorem prover – and associated tools – has become publicly available (subject to the GPL). The ProofPower has good, inbuilt support for Z specification and verification when used with its PPZed package. Unfortunately, we did not become aware of this situation until quite late in our development process and we did not have time to research whether these tools could be used to our advantage. However, investigating an integration between Frog and ProofPower is certainly an area that we would like to pursue in the future – see section 6.4.9.

6.4.8 Isabelle/ZF

Isabelle¹⁴ [Pau94, NPW02] is a generic, interactive theorem prover based on Standard ML. Isabelle is a generic theorem prover as it allows users to define their own logics using a meta-logic based on a weak type theory. Isabelle has quite similar properties to its predecessor HOL: a programming meta-language, tactics and the generation of a fully-expanded proof. However, there are also significant differences and the two should not be confused.

Three object logics are commonly used (although many more exist): higher-order logic, first-order logic and ZF set theory (on top of first-order logic). The combination of Isabelle and the object logic of ZF set theory is known as Isabelle/ZF [Noe93, Pau04].

Whilst Isabelle/HOL has extensive libraries and many tools and applications have been developed for use with it, Isabelle/ZF has a less comprehensive library and does not have the same external tool support. However, Isabelle/ZF provides more advanced constructions for sets than simply-typed Isabelle/HOL. In fact, there is an one-to-one relationship between our ZF language and that used by Isabelle/ZF (see section 6.5.1).

Work is now in progress to produce more automation in Isabelle/ZF proofs through integration with the Vampire automated theorem prover. At the time of writing, a prototype has been implemented [MP04, MQPss] that is essentially complete and is able to demonstrate the key points. This tool is not yet available publicly, but the completion of such a tool would be of great benefit in the discharge of the proof obligations generated by our tool – particularly where these are large and shallow. A major advantage of this tool is that the proofs are translated back into the Isabelle/ZF syntax, and as such are far more readable than those generated were Vampire to be used in isolation.

¹⁴Also referred to as Isabelle/HOL as the most commonly used object logic is an implementation of higher order logic.

6.4.9 Z-Specific Theorem Provers

There exist a number of theorem provers that have been developed to provide direct support for the Z notation. In section 3.1.5 we introduced a number of toolkits that provided support for deductive reasoning over Z specifications; particularly CADiZ (see section 4.2.1) and Z/EVES (see section 4.2.4). We noted in that section, that as Z does not have a set of standard proof obligations, these theorem provers are not restricted to discharging theorems of a certain shape, but are capable of tackling any given theorem – including one derived from a construct’s configuration. One may be forgiven for thinking, therefore, that using one of these tools – or ProofPower (discussed above) – to discharge a construct’s proof obligations would be the simplest option, and that one of the tools should therefore be our theorem prover of choice.

It is important at this stage to remember our reasons for using the Z notation. The principle motive was the ability to reuse a notation that was well specified and widely understood. This prevented us from needing to create a notation, and the user from having to learn one. When choosing this notation, however, we always considered the future possibility of allowing users a selection of notations (perhaps even creating an extension point from which users would be able to define their own syntax). In fact, we noted in section 3.4 that as one of the prime goals for Frog was flexibility, the design of the toolkit would not be dependent on a particular notation. In order to achieve this decoupling it was necessary to represent the proof obligations in a flexible and independent manner. Of course, it is possible to argue that as we are currently supporting a single notation, the easiest and most powerful approach would be to use theorem provers that take advantage of that notation’s strengths (for instance, the schematic approach of Z). We felt, however, that it was important to demonstrate Frog’s flexibility and show how a generic theorem prover could be used when attempting to discharge its proof obligations.

Furthermore, the ability to support retrenchment was a significant motivation in Frog’s creation and retrenchment is typically used in situations where powerful theorem proving is required, for example, when expressing

the relationship between a machine using integers and another using real numbers with given tolerances. The ability to interface with potent and established theorem provers such as those mentioned in the previous sections was considered to be a major requirement of retrenchment's users.

It should be noted that the power of using Z theorem provers for discharging proof obligations generated against a Z specification cannot be ignored. Frog's architecture was always intended to be modular and the principal modules are discussed fully in section 7.2. In the future, we envisage having multiple parsing factories capable of processing a variety of notations. Similarly, we may have multiple proof obligation generation factories, for instance, one that creates theorems using the flattening approach described in sections 6.2 and 6.3, a second that takes full advantage of the power of schemas within Z , and perhaps a third that uses a shallow embedding approach. A way in which we may configure this modular approach to ensure compatibility between modules is also discussed in section 7.2.

6.4.10 Summary

We have detailed previously that our aim is to provide support for as many theorem provers as possible and allow users to use the tool of their preference. In order to get our tool up and running, it was necessary to select one theorem prover in which we could demonstrate the ability of our tool to generate proof obligations in a valid format.

The Z -specific theorem provers that we discussed in the previous section were discounted for the reasons that we gave in that section. The remaining theorem provers that we discussed could be split into three categories: the automated theorem provers, the interactive theorem provers based on higher order logic (or unique systems) and the interactive theorem provers based on set theory.

We eliminated the automated theorem provers as we would be unable to create readable proofs for our set theoretic proof obligations. Similarly, we rejected the interactive theorem provers based on higher order logic and were left with the interactive theorem provers based on set theory. As there

was only one theorem prover in this category – Isabelle/ZF – we decided to initially use this theorem prover to discharge our proof obligations.

6.4.11 Evaluating our Choice

As we will see in the following sections, while Isabelle/ZF has provided natural support for our proof obligations, we have found that the discharge of obligations with this tool would prove to be a somewhat manual task for our tool’s users. This theorem prover is a minor member of the Isabelle family and we have been unable to find existing tactics that are able to discharge our proof obligations easily. Paulson notes in [Pau92] that:

Isabelle does not find proofs automatically. Proofs require a skilled user, who must decide which lemmas to prove and which tools to apply.

Whilst this is fine for ‘skilled users’ of our tool, it does detract somewhat from the benefits of our tool, where we would like to be able to verify our proof obligations (semi-)automatically leaving the user free to concentrate on building their specification.

In retrospect, we may well have been better choosing one of the higher-order interactive theorem provers (as we will certainly need to develop support for them in the future). We perhaps overestimated at the time of choosing Isabelle’s support for the ZF set theory and also the advantage it would give us in discharging our proofs. It may well have proved less effort to generate valid proof obligations for the (better supported) higher-order logic provers, but we feel that we could not have predicted the trouble that we would have with Isabelle/ZF and at the time it seemed an obvious selection. If nothing else we have certainly learnt from the experience, and we feel this would reduce the time required to extend our tool to generate proof obligations in the higher-order logic. Having said all this, if Meng’s integration with the Vampire theorem prover delivers the automation it promises, we could find that the difficulties we have found with actually discharging proofs in Isabelle could be eliminated and we would be left with a very powerful proof tool.

6.5 Discharging Proof Obligations

In the previous sections we have outlined a process for generating a construct's proof obligations and have selected a theorem prover with which we will allow users to discharge those proof obligations. We must now describe how proof obligations in our ZF language can be used as an input to the Isabelle/ZF theorem prover, and how that theorem prover can be used to attempt the verification of those obligations. This section begins by presenting the process through which we translate our ZF language proof obligations into the syntax used by Isabelle/ZF. We then proceed to describe how a user could attempt to discharge proof obligations in this syntax using the tactics and rules of Isabelle/ZF. We highlight the issues preventing easy discharge of the proof obligations and indicate the stumbling blocks that limit the extent to which Frog can provide automated assistance. We illustrate these processes with a continuation of our running example.

6.5.1 Translation

Once we have generated our proof obligations in our generic ZF language, we allow a user to attempt their discharge using a theorem prover. The first stage of this process, is the translation of each proof obligation into a format suitable for that theorem prover.

The theorem prover that we have initially selected to utilize in the discharge of our proof obligations is Isabelle/ZF. Thus, we must translate the syntax of our proof obligations to Isabelle/ZF's syntax, and create files that can be used as an input to its batch processing tool ('isatool').

Some examples of the differences in syntax between Isabelle/ZF and Z can be seen in table 6.3. Principally, there is a one-to-one correspondence between the two syntaxes, so the translation process is relatively simple. We take advantage of the character dictionary we defined in section 4.6.3 to perform this task. This character dictionary allows a user to configure the specific syntax used when generating a proof obligation for a particular theorem prover. For example, in Isabelle/ZF the union operator can be described by '\/' or '\<union>'.

Table 6.3: Examples of differences between Isabelle/ZF syntax and Z syntax

Symbol	Rendered	Isabelle/ZF	Z
Union	\cup	<code>\/</code>	<code>\cup</code>
Existential Quantifier	\exists	<code>\<exists></code>	<code>\exists</code>
Conjunction	\wedge	<code>&</code>	<code>\land</code>
Powerset	\mathbb{P}	<code>Pow</code>	<code>\power</code>
Implication	\Rightarrow	<code>--></code>	<code>\implies</code>
Emptyset	\emptyset	<code>0</code>	<code>\emptyset</code>
Bullet	\bullet	<code>.</code>	<code>@</code>

Other than this translation between symbols, the only other task involved in changing the syntax of the proof obligations concerns the translation of identifiers. In Isabelle/ZF we are unable to use underscores within identifiers, nor can we use any of the Z strokes (`?`, `!`, `'`). Other theorem provers have similar restrictions, so rather than trying to fulfil individual requirements, we reduce all identifiers to unique alphanumeric strings (for example, `ZFName1`, `ZFName2`, `...`).

In order to pass our proof obligations to Isabelle/ZF, we need to place them within Isabelle/ZF theory files. The general format for an Isabelle/ZF theory file can be seen below.

```
theory theory_name = parent_theory :  
  
lemma  
  “ [| hypotheses ==> goal |] ”  
  
  apply(tactics)  
  done  
  
end
```

The theory name is a unique identifier that we assign to the proof obligation. The parent theory will always be the ‘Main’ theory of Isabelle’s ZF libraries. The hypotheses will clearly be a semi-colon separated list of the translated hypotheses belonging to the proof obligation. Similarly, the goal is the translated statement of the proof obligation. We will discuss how the tactics field is completed in the following section.

Example 6.3

We consider again the ZF language proof obligation that we created in example 6.2. We will now present the Isabelle/ZF theory file that corresponds to this proof obligation. For the sake of brevity we will not include all of the hypotheses, nor the complete goal, but present enough of each to allow the reader to compare with the ZF language equivalent presented in the previous example.

```

theory theory_name = parent_theory :

lemma
  “ [|
    ZFCarrierSet206 \<in> Pow ( Inf ) ;
    ZFName205 \<in> Pow ( ZFCarrierSet206 ) ;
    ZFName214 \<in> ZFName81 ;
    ( \<forall> ZFName82 ZFName83 . ( ( ZFName82 \<in> ZFCarrierSet206 & ZFName83 \<in> Pow ( ZFCarrierSet206 ) ) --> \<not> ( \<ZFName82 , ZFName83 \<in> ZFName214 & \<not> ( \<not> ( ZFName82 \<in> ZFName83 ) ) ) & \<not> ( \<not> ( ZFName82 \<in> ZFName83 ) & \<not> ( \<ZFName82 , ZFName83 \<in> ZFName214 ) ) ) ) ) ;
    ZFName81 \<in> { Pow ( { \<ZFName61 , ZFName62 \> : ( Pow ( ZFCarrierSet206 ) * Pow ( Pow ( ZFCarrierSet206 ) ) ) . ZFName61 \<in> Pow ( ZFCarrierSet206 ) & ZFName62 \<in> Pow ( Pow ( ZFCarrierSet206 ) ) ) } ) } ;
    ...
  |]
  ==>
  ( \<forall> ZFName229 ZFName230 . (
    ZFName229 \<in> { ZFName208 : Pow ( ZFCarrierSet206 ) .
ZFName208 \<in> Pow ( ZFName205 ) }
    & ZFName230 \<in> { ZFName212 : ZFCarrierSet206 . ZFName212
\<in> ZFName205 }
    ...
  )

apply(tactics)
done

end

```

6.5.2 Proving in Isabelle/ZF

Theorem proving in Isabelle/ZF involves the construction of proof trees. These proof trees are derived rules that are created from the composition of other rules. Isabelle/ZF uses both forward and backwards inference. Forward inference involves matching the hypotheses to antecedents of inference rules and replacing the hypotheses with the matching consequents. Those consequents are then matched to the antecedents of further rules and replaced. This process continues until the result of a replacement matches the goal. Backward inference begins with the goal, and matches that goal to the consequents of rules, whereupon the current goal is replaced by the antecedents of that rule, and a new match is sought. This process continues until the current goal matches the hypotheses.

Rules are either created by a user or inherited from other theorems. Common rules are collected together into Isabelle's standard libraries. Different libraries exist for the different logics available. Hence, Isabelle/HOL's libraries are distinct from those of Isabelle/ZF. The library used is indicated in the first line of a theory file where the theory is declared to belong to a parent theory. The constants and rules of that parent theory (and its ancestors) will then be available for use throughout that theory file.

Isabelle/ZF has a number of special rules known as tactics. These tactics typically involve some form of repetitive rule application intended to exhaustively apply rules in a given fashion until either the rules are exhausted or no unproved sub-goals remain. The tactics of Isabelle/ZF are described in table 6.4 on the following page (derived from [MP04], p.3).

Whilst translating our theorem-prover independent proof obligation into an Isabelle/ZF theory file, we will prompt the user to ask which rules and tactics they wish to apply to the proof obligation. We shall provide them with the option to use one of the automatic tactics or to use a combination of existing and derived rules created by the user. When the rules and tactics have been specified they will be substituted into the theory file (replacing the line `apply(tactics)` in the example theory file above).

Table 6.4: Tactics of Isabelle/ZF

Tactic	Description
simp	The simplifier, which performs conditional rewriting augmented by other code, including a decision procedure for linear arithmetic.
blast	A sort of generic tableaux theorem prover. It performs forward and backwards chaining using any lemmas supplied by the user
auto	A naive combination of the previous two tactics. It interleaves rewriting and chaining. However, this treatment of equality is primitive compared with that provided by a good resolution prover.

We now consider how Isabelle/ZF can be used to discharge proof obligations by returning to our example involving the modelling of a train yard.

Example 6.4

In the following examples we use some new symbols that we will introduce here. We use the reserved ‘♠’ decoration to describe the carrier set of a particular given type. The carrier set of a type is the set containing the elements of that type whose members are determined by its constraints. For example, if we declared a given type *oneOrTwo* as follows:

$$[oneOrTwo]$$

We could then apply the following constraint.

$$| \quad oneOrTwo = \{x : \mathbb{N} \mid x = 1 \vee x = 2\}$$

The carrier set for *oneOrTwo*, that is ♠*oneOrTwo*, would then be equal to the set $\{1, 2\}$.

In addition to this symbol we also introduce the ‘∞’ notation which will

refer to Isabelle/ZF's infinite set.

The first of our machine proof obligations involves showing that there exists a state into which our machine can be initialized. Taking our generic proof obligation from the machine configuration and instantiating it using the specification of the train yard machine (both of which are defined in example 5.9 on page 267) results in the following proof obligation being created.

$$\begin{aligned}
& \spadesuit \text{trains} \in \mathbb{P}(\infty) \\
& \text{trains} \in \mathbb{P}(\spadesuit \text{trains}) \\
& \emptyset[\mathbb{P}(\spadesuit \text{trains})] \\
& \in \{ \{ _VAR29 : \mathbb{P}(\spadesuit \text{trains}) \\
& \quad \bullet \forall x \bullet (x \in \mathbb{P}(\spadesuit \text{trains}) \wedge \neg \text{true} \wedge _VAR29 = x) \} \} \\
& \vdash? \\
& \exists _u \bullet _u \in \{ \text{trainsInYard} : \mathbb{P}(\spadesuit \text{trains}) \bullet \text{trainsInYard} \in \mathbb{P}(\text{trains}) \} \\
& \quad \wedge _u \in \{ \text{trainsInYard} : \mathbb{P}(\spadesuit \text{trains}) \\
& \quad \bullet \text{trainsInYard} \in \mathbb{P}(\text{trains}) \\
& \quad \wedge \text{trainsInYard} \in \{ \emptyset[\mathbb{P}(\spadesuit \text{trains})] \} \}
\end{aligned}$$

This proof obligation will then be translated into the following Isabelle/ZF lemma (contained within a larger theory file)¹⁵.

```

lemma "
[]
ZFCarrierSet263 \<in> Pow ( Inf ) ;
ZFName262 \<in> Pow ( ZFCarrierSet263 ) ;
ZFName267
\<in> { { ZFName94 : Pow ( ZFCarrierSet263 )
        . ( \<forall> ZFName95 . ( ZFName95 \<in> Pow ( ZFCarrierSet263 )
            & \<not> ( (0=0) )
            & ZFName94 = ZFName95 ) ) } }
[]
==>
( \<exists> ZFName284
  . ( ZFName284 \<in> { ZFName265 : Pow ( ZFCarrierSet263 )
                    . ZFName265 \<in> Pow ( ZFName262 ) }

```

¹⁵Note that, we use the term '0 = 0' to indicate truth as Isabelle/ZF has no truth constant.

```
& ZFName284 \<in> { ZFName265 : Pow ( ZFCarrierSet263 )
  . ZFName265 \<in> Pow (ZFName262 )
  & ZFName265 \<in> { ZFName267 } } ) ) "
```

If we submit this lemma to the Isabelle/ZF theorem prover and instruct the prover to apply the ‘auto’ tactic, we receive the following output.

```
ML> use_thy "trainYard_pob_0";
Loading theory "trainYard_pob_0"
### Interactive-only command "pr" (line 244 of
"/home/si/uni/software/frog/.isabelle/trainYard/trainYard_pob_0/
trainYard_pob_0.thy")
proof (prove): step 1

fixed variables: ZFCarrierSet263, ZFName262, ZFName267

goal (lemma):
No subgoals!
lemma
  [| ?ZFCarrierSet263.0 : Pow(Inf); ?ZFName262.0 : Pow(?ZFCarrierSet263.0);
   ?ZFName267.0 :
   {{ZFName94: Pow(?ZFCarrierSet263.0) .
    ALL ZFName95.
      ZFName95 : Pow(?ZFCarrierSet263.0) &
      0 ~= 0 & ZFName94 = ZFName95}} |]
==> EX ZFName284.
  ZFName284 :
  {ZFName265: Pow(?ZFCarrierSet263.0) .
   ZFName265 : Pow(?ZFName262.0)} &
  ZFName284 :
  {ZFName265: Pow(?ZFCarrierSet263.0) .
   ZFName265 : Pow(?ZFName262.0) & ZFName265 : {?ZFName267.0}}
val it = () : unit
```

No sub-goals remain so our proof obligation has been successfully discharged. Of course, this is a trivial proof and we should not raise our hopes that Isabelle/ZF’s tactics will be able to tackle all of our proof obligations. This can be illustrated by progressing to the next proof obligation.

Example 6.5

The next proof obligation for our train yard machine is that to show the applicability of the *connect* operation. Again we take the generic proof obligation from the machine configuration and instantiate it using the specification of the train yard machine. The result of this process is the generation of the following proof obligation.

$$\begin{aligned}
& \spadesuit \text{trains} \in \mathbb{P}(\infty) \\
& \text{trains} \in \mathbb{P}(\spadesuit \text{trains}) \\
& \bowtie \leftrightarrow \bowtie [\spadesuit \text{trains}, \mathbb{P}(\spadesuit \text{trains})] \in \\
& \quad \{ \mathbb{P}(\{ _ \text{VAR27} : \spadesuit \text{trains} \times \mathbb{P}(\spadesuit \text{trains}) \\
& \quad \bullet \forall _ \text{GI0001_GI0002} \bullet (_ \text{GI0001} \in \spadesuit \text{trains} \\
& \quad \quad \wedge _ \text{GI0002} \in \mathbb{P}(\spadesuit \text{trains}) \\
& \quad \quad \wedge _ \text{VAR27} = \langle _ \text{GI0001}, _ \text{GI0002} \rangle) \} \} \} \\
& \bowtie \not\leftrightarrow \bowtie [\spadesuit \text{trains}] \in \bowtie \leftrightarrow \bowtie [\spadesuit \text{trains}, \mathbb{P}(\spadesuit \text{trains})] \\
& \forall x \ a \bullet ((x \in \spadesuit \text{trains} \wedge a \in \mathbb{P}(\spadesuit \text{trains})) \\
& \quad \Rightarrow (\neg \langle x, a \rangle \in \bowtie \not\leftrightarrow \bowtie [\spadesuit \text{trains}] \\
& \quad \quad \wedge (\neg (\langle x, a \rangle \in \bowtie \not\leftrightarrow \bowtie [\spadesuit \text{trains}]) \\
& \quad \quad \wedge \neg (\neg (x \in a))) \wedge \neg (\neg (x \in a) \\
& \quad \quad \wedge \neg (\langle x, a \rangle \in \bowtie \not\leftrightarrow \bowtie [\spadesuit \text{trains}]))) \\
& \vdash? \\
& \exists _ u \ _ i \\
& \bullet _ u \in \{ \text{trainsInYard} : \mathbb{P}(\spadesuit \text{trains}) \bullet \text{trainsInYard} \in \mathbb{P}(\text{trains}) \} \\
& \quad \wedge _ i \in \{ \text{trainEnteringYard?} : \spadesuit \text{trains} \bullet \text{trainEnteringYard?} \in \text{trains} \} \\
& \quad \wedge \langle _ u, _ i \rangle \in \\
& \quad \{ \langle \text{trainsInYard}, \text{trainEnteringYard?} \rangle : \mathbb{P}(\spadesuit \text{trains}) \times \spadesuit \text{trains} \\
& \quad \bullet \text{trainsInYard} \in \mathbb{P}(\text{trains}) \wedge \text{trainEnteringYard?} \in \text{trains} \\
& \quad \wedge \langle \text{trainEnteringYard?}, \text{trainsInYard} \rangle \in \bowtie \not\leftrightarrow \bowtie [\spadesuit \text{trains}] \}
\end{aligned}$$

This proof obligation will be subject to the translation rules discussed in the last section, and a Isabelle/ZF theory file will be created. This file will contain the following lemma that is the Isabelle/ZF representation of our proof obligation.

```

lemma "
[|
ZFCarrierSet262 \<in> Pow ( Inf ) ;
ZFName261 \<in> Pow ( ZFCarrierSet262 ) ;

```

```

ZFName271
<in> { Pow ( { ZFName65 : ( ZFCarrierSet262 * Pow ( ZFCarrierSet262 ) )
  . (\<forall> ZFName66 ZFName67
    . ( ZFName66 \<in> ZFCarrierSet262
      & ZFName67 \<in> Pow (ZFCarrierSet262 )
      & ZFName65 = < ZFName66 , ZFName67 > ) ) } ) } ;
ZFName270 \<in> ZFName271 ;
( \<forall> ZFName88 ZFName89
  . ( ( ZFName88 \<in> ZFCarrierSet262
    & ZFName89 \<in> Pow (ZFCarrierSet262 ) )
  --> \<not> ( < ZFName88 , ZFName89 > \<in> ZFName270
    & \<not> ( \<not> ( ZFName88 \<in> ZFName89 ) ) )
    & \<not> ( \<not> ( ZFName88 \<in> ZFName89 )
    & \<not> ( < ZFName88 , ZFName89 > \<in> ZFName270 ) ) ) ) )
]
==>
( \<exists> ZFName284 ZFName285
  . ( ZFName284 \<in> { ZFName264 : Pow ( ZFCarrierSet262 )
    . ZFName264 \<in> Pow ( ZFName261 ) }
  & ZFName285 \<in> { ZFName268 : ZFCarrierSet262
    . ZFName268 \<in> ZFName261 }
  & < ZFName284 , ZFName285 >
  \<in> { < ZFName264 , ZFName268 >
    : ( Pow ( ZFCarrierSet262 ) * ZFCarrierSet262 )
    . ZFName264 \<in> Pow ( ZFName261 )
    & ZFName268 \<in> ZFName261
    & < ZFName268 , ZFName264 > \<in> ZFName270 } ) ) "

```

If we submit the theorem to Isabelle/ZF, and use either the ‘auto’ or ‘blast’ tactics, then the theorem prover is unable to establish a proof in any reasonable time period. If we use the ‘simp’ tactic then the following is output and one simplified goal remains.

```

ML> use_thy "trainYard_pob_1_0";
Loading theory "trainYard_pob_1_0"
### Interactive-only command "pr" (line 242 of
"/home/si/uni/software/frog/.isabelle/trainYard/trainYard_pob_1_0
/trainYard_pob_1_0.thy")
proof (prove): step 1

```

fixed variables: ZFCarrierSet262, ZFName261, ZFName271, ZFName270

goal (lemma, 1 subgoal):

```

1. [| ZFCarrierSet262 <= Inf; ZFName261 <= ZFCarrierSet262;
    ZFName271 =
    Pow({ZFName65: ZFCarrierSet262 * Pow(ZFCarrierSet262) .
      ALL ZFName66.
        ZFName66 : ZFCarrierSet262 &
        (ALL ZFName67.
          ZFName67 <= ZFCarrierSet262 &
          ZFName65 = <ZFName66, ZFName67>)});
    ZFName270 <=
    {ZFName65: ZFCarrierSet262 * Pow(ZFCarrierSet262) .
      ALL ZFName66.
        ZFName66 : ZFCarrierSet262 &
        (ALL ZFName67.
          ZFName67 <= ZFCarrierSet262 &
          ZFName65 = <ZFName66, ZFName67>)});
    ALL ZFName88 ZFName89.
      ZFName88 : ZFCarrierSet262 & ZFName89 <= ZFCarrierSet262 -->
      (<ZFName88, ZFName89> ~: ZFName270 | ZFName88 ~: ZFName89) &
      (ZFName88 : ZFName89 | <ZFName88, ZFName89> : ZFName270) |]
==> EX ZFName284.
    ZFName284 <= ZFCarrierSet262 &
    ZFName284 <= ZFName261 &
    (EX ZFName285.
      ZFName285 : ZFCarrierSet262 &
      ZFName285 : ZFName261 & <ZFName285, ZFName284> : ZFName270)

```

*** Failed to finish proof (after successful terminal method)

Isabelle/ZF is unable to discharge this goal with any of its automatic tactics. We examine now the outstanding goal to uncover why this is so. The above goal can be translated back to a \LaTeX representation to give the following.

$$\begin{aligned}
& \spadesuit \text{trains} \in \mathbb{P} \infty \\
& \text{trains} \in \mathbb{P} \spadesuit \text{trains} \\
& \vdash? \\
& \exists x \bullet x \in \mathbb{P} \spadesuit \text{trains} \wedge x \in \mathbb{P} \text{trains} \\
& \quad \wedge (\exists y \bullet y \in \spadesuit \text{trains} \wedge y \in \text{trains} \wedge y \notin x)
\end{aligned}$$

This proof obligation can quickly be seen as valid through observation alone. In this representation we have not relied upon the deep embedding of the ‘ \notin ’ operator. We consider whether it is this level of embedding that is preventing the discharge and translate the proof obligation back to the Isabelle/ZF syntax (abbreviating the variable names for brevity).

```
lemma "
  []
  CS : Pow(Inf) ;
  T : Pow(CS)
  []
  ==>
  EX x y. x : Pow(CS) & x : T & y : CS & y : T & y ~: x"
```

Passing to Isabelle/ZF, and instructing the theorem prover to use the ‘infinity’ rule and ‘auto’ tactic, produces the following output.

```
goal (lemma, 1 subgoal):
  1. [] CS <= Inf; T <= CS; 0 : Inf; ALL y:Inf. succ(y) : Inf []
     ==> EX x. x <= CS & x : T & (EX y. y : CS & y : T & y ~: x)
*** Failed to finish proof (after successful terminal method)
```

Despite using Isabelle/ZF’s native operators we have been unable to progress our proof. From this we can begin to see why our proof attempts are becoming blocked.

The Problem With Carrier Sets in Isabelle/ZF

The first cause of blockage derives from the definition of Z ’s carrier sets. In the Z methodology it is implicit that a carrier set is non-empty and it inherits behaviour from its use as a Z ‘type’. When we translate to Isabelle/ZF the carrier set loses its privileged status, it becomes a set with the same properties as any other. In order to convey the carrier set’s distinct attributes to Isabelle/ZF, we describe the carrier set as a subset of ‘the’ infinite set in the hope that Isabelle/ZF will be able to deduce that the carrier set is also infinite. Clearly however, the previous example shows that this does not produce an immediately satisfactory result.

After some experimentation we constructed the following proof obligation that clearly resembles the one above.

```
lemma "
  []
  CS = Inf
  []
  ==>
  EX x y . x : Pow(CS) & y : CS & y ~: x"
```

When we submitted this to Isabelle/ZF, and instructed the theorem prover to use the ‘infinity’ rule and ‘auto’ tactic, the following output was produced.

```
goal (lemma):
No subgoals!
lemma ?CS = Inf ==> EX x y. x : Pow(?CS) & y : ?CS & y ~: x
val it = () : uni
```

Hence, the obligation has been discharged successfully. We made a small change to the proof obligation in an attempt to determine where our blockage occurred specifically.

```
lemma "
  []
  CS : Pow(Inf)
  []
  ==>
  EX x y . x : Pow(CS) & y : CS & y ~: x"
```

This proof obligation was also submitted to Isabelle/ZF, and the theorem prover instructed to use the ‘infinity’ rule and ‘auto’ tactic. The following output was produced.

```
goal (lemma, 1 subgoal):
  1. [] CS <= Inf; 0 : Inf; ALL y:Inf. succ(y) : Inf []
      ==> EX x. x <= CS & (EX y. y : CS & y ~: x)
*** Failed to finish proof (after successful terminal method)
```

The theorem prover was no longer able to discharge the obligation. This led us to conclude that Isabelle/ZF is unable to deduce that the subset of an infinite set can itself be infinite. Upon further investigation we discovered that the infinity constant is really just a constant with the following properties.

$$0 \in \text{Inf}; \forall y : \text{Inf} \bullet \text{succ}(y) : \text{Inf}$$

The only implementation of an infinite set within the standard Isabelle/ZF libraries is used for representing the natural numbers.

An entire theory is used to establish the constants and rules that are required to describe the natural numbers and reason about them. An example of the necessary definitions is shown below.

```
theory Nat = OrdQuant + Bool:

constdefs
  nat :: i
    "nat == lfp(Inf, %X. {0} Un {succ(i). i:X})"

  quasinat :: "i => o"
    "quasinat(n) == n=0 | (\<exists>m. n = succ(m))"

...
```

The existence of this theory allowed us to conclude that we would require something similar for each of the carrier sets contained within a particular proof obligation. The extent to which we could automate the creation of such a theory for each and every carrier set is yet to be fully investigated. While it would be possible to create a generic theory that could provide the rules for handling a given set of situations, it would be extremely difficult to predict the range of situations for which rules could potentially be required. Furthermore, even if it were possible to create a theory that acted as a panacea we would still be required to establish the order in which its rules must be applied. Isabelle/ZF is particular about the application of rules. An attempt to apply a rule in a situation to which it does not apply generally brings about the immediate termination of the proof attempt. This problem

also occurs in the second issue blocking automatic, mechanical verification with Isabelle/ZF.

The Problems With Deep Embedding in Isabelle/ZF

We have chosen to use a deep embedding of our proof obligations to allow users as much flexibility as possible when using our tools. Instead of being forced to use the standard prelude and mathematical toolkits of Z, users are free to use their own standard definitions. Therefore, when we create a proof obligation we must ensure that Isabelle/ZF uses the same definition of operators that we have defined in our specification. We do this by including the definition in the antecedents of our proof obligation. For example, if we re-examine the provable proof obligation of the last section and replace Isabelle/ZF's native '~:' operator with the definition from the Z standard set toolkit we could generate the following proof obligation.

```
lemma "
[]
CS = Inf ;
REL \<in> { Pow ( { a : ( CS * Pow ( CS ) )
      . ( \<forall> b c
          . ( b \<in> CS & c \<in> Pow ( CS )
            & a = < b , c > ) ) } ) } ;
NOTIN \<in> REL ;
( \<forall> d e
  . ( ( d \<in> CS & e \<in> Pow ( CS ) )
    --> \<not> ( < d , e > \<in> NOTIN
      & \<not> ( \<not> ( d \<in> e ) ) )
    & \<not> ( \<not> ( d \<in> e )
      & \<not> ( < d , e > \<in> NOTIN ) ) ) )
[]
==>
EX x y . x : Pow(CS) & y : CS & <y,x> : NOTIN"
```

Submitting this proof obligation, with one of the automatic tactics, to Isabelle/ZF, perhaps unsurprisingly, fails to produce a successful result. The simplification tactic is however, able to reduce the obligation, leaving the following goal.

```

goal (lemma, 1 subgoal):
1. [| NOTIN <=
   {a: Inf * Pow(Inf) .
   ALL b. b : Inf & (ALL c. c <= Inf & a = <b, c>)};
   ALL d e.
   d : Inf & e <= Inf -->
   (<d, e> ~: NOTIN | d ~: e) & (d : e | <d, e> : NOTIN);
   0 : Inf; ALL y:Inf. succ(y) : Inf |]
==> EX x. x <= Inf & (EX y. y : Inf & <y, x> : NOTIN)
*** Failed to finish proof (after successful terminal method)

```

There is no easy solution to this problem and at present automatic mechanical discharge of proofs using Isabelle appears to be out of our reach. We must rely therefore, on a skilled user of Isabelle/ZF who is able to manipulate the theorem prover to produce the required results. As hypotheses are created for each and every one of our operators, however, any non-trivial proof obligation quickly becomes unwieldy both for the human prover and the theorem prover itself. Unfortunately Isabelle (generally) is considerably better at handling small deep proofs than the large shallow proofs that we will inevitably create.

Going forward we have two proposals that may be able to alleviate the headaches caused by the flexibility of the tool. Both involve directives being incorporated in the toolkits – standard or otherwise – that instruct our tool in the generation of proof obligations involving the operators contained within.

Consider, for example, the following definition of the ‘ \notin ’ operator that is included in the set toolkit of the standard Z toolkits.

relation(\notin)

$$\frac{\frac{[X]}{_ \notin _ : X \leftrightarrow \mathbb{P} X}}{\forall x : X; a : \mathbb{P} X \bullet x \notin a \Leftrightarrow \neg x \in a}$$

Our first proposal would allow the incorporation of rules in this definition that the tool could include in the theory file of any proof obligation in which

the operator was used. For example, we could include the following directive in our definition's operator template paragraph.

```
\begin{zed}
\relation ( \_ \notin \_)
% DEFINE LEMMA ISABELLE "notinLemma" {
% lemma notinLemma : "[| P |] ==> Q"
% apply (tactics)
% done
% }
\end{zed}
```

This directive could be parsed with the operator and stored with its internal definitions. When we came to generate the proof obligations for a construct containing the operator, the lemma (and its proof) would be included in the theory file. An algorithm could then be constructed to apply the incorporated rules depending upon their positions within the file (remembering of course to take into account the inter-reliance of operators).

A second alternative is to take greater advantage of Isabelle/ZF's native operators by providing translations where it is possible to do so. For instance, we may incorporate the following directive.

```
\begin{zed}
\relation ( \_ \notin \_)
% EMBED ISABELLE "#1 </notin> #2"
\end{zed}
```

This would lead to a semi-shallow embedding within the theorem prover, where native operators are used wherever possible. When constructing the proof obligation, if a reference was made to an operator with an embedding-translation it could be simply substituted, with the arguments (#1, #2) to the Z operator being inserted into the theorem prover-specific equivalent. While the example above uses only a single native element to represent an operator, it could be represented by any combination of native operators that can provide equivalent functionality.

In summary, users of our tools are currently required to attempt the majority of their proof obligations manually (although of course they have the assistance of the Isabelle/ZF automated theorem prover). In the future we would hope to overcome some of these issues and provide a more automated approach to verification. This could be achieved through the implementation of the ideas above, or through the use of a tool that attempts broad exhaustive proofs with a resolution theorem prover (such as Meng's [MP04, MQPss] interface to Vampire).

6.6 Concluding Remarks

In this chapter we have discussed the generation and discharge of a construct's proof obligations. We created a ZF language based on Zermelo-Fraenkel set theory that served as an intermediary between Z and the syntaxes of theorem provers. This ZF language is used in the semantic model of a construct. The semantic model is formed from a set of semantic bindings that can be used to determine the meaning of variables bound within that model. We have presented a collection of rules for deriving a construct's semantic model from its Z definition. A construct's semantic model is used to instantiate the generic proof obligations found in that construct's configuration. This is achieved by parsing the tree associated with the generic proof obligation, and at each stage applying of rules that query the semantic model of the construct to generate a proof obligation in the ZF language. After investigating the properties of a number of theorem provers we chose to proceed – initially – with Isabelle/ZF. Isabelle/ZF supports the ZF set theory natively and is based on first-order logic. Translation from our ZF language to the syntax of Isabelle/ZF is a relatively simple task. Discharging the proof obligations however, is a task that will require a skilled user and significant manual resources.

Chapter 7

Frog: Architecture and Implementation

This chapter describes the architecture and implementation of the Frog tool. We begin by describing the structure of the tool and proceed to explain how the components introduced in the previous chapters can be integrated to form an application capable of processing Z specifications, incorporating constructs and generating proof obligations that can show that those constructs are consistent. We then examine the user's perspective of Frog and show how the system is used. We describe how a user is able to configure the tool, create new specifications and prove the validity of constructs within that specification. We then proceed to give a general overview of the implementation of the application, examining how the processes described earlier in this document are converted into working tools, and how they can be integrated whilst maintaining a modular approach for easy extensibility.

7.1 Introduction

Frog is a tool that can be used in the formal creation and verification of specifications. A meta-language is used to describe machines and relationships, but the majority of a specification is presented in the Z notation. The notion of a machine or a relationship is configurable within Frog. Whilst it is possible to relate machines through long established techniques such as refinement, it is also possible to do so through newer methods such as retrenchment. In fact machines can be related through any method expressible in Frog's construct configuration language (Frog-CCL). Similarly, different types of machines can also be expressed. The configuration language allows full control over the clauses a construct (machine or relationship) can possess and the way in which the specification of those clauses can be used to verify the construct. Proof obligations are created from a construct's configuration and specification in the Zermelo-Fraenkel (ZF) set theory. The tool provides an interface to the Isabelle/ZF interactive theorem prover, where a user can attempt to verify a construct's proof obligations.

Each of the major elements of the Frog tool has been described in previous chapters. In chapter 4 we presented details of a Z parser that allows us to transform a specification in the \LaTeX notation described in [ISO02] into a typed abstract syntax tree (AST). In chapter 5 we introduced Frog's construct configuration language and showed how the \LaTeX notation could be extended to allow us to express machines and relationships within a Z specification. In chapter 6 we showed how the typed AST of a construct and the construct's configuration could be used to produce a semantic model, from which we could derive the construct's proof obligations. We then showed how those proof obligations could be translated to a format suitable for input to Isabelle/ZF. The remainder of this chapter describes how we brought these parts together into a single tool.

7.2 Architecture

It was our intention to make Frog as flexible and extensible as possible. Where possible we wanted to eliminate reliance between components and encourage a modular design. Figure 7.1 illustrates how the major components of Frog that we have discussed in the earlier chapters of this document are integrated around a central graphical user interface. In our diagram, we describe some of the major components as factories, but this syntax should not be confused with the factory referred to in the abstract factory or factory method design patterns [GHJV95, Mar02] that are commonly used in the design of object oriented software. In our design, factories can be considered an interface (in the traditional, rather than object-oriented sense) between the GUI and each of our modules (this will be discussed in more detail in section 7.4).

The structure of the tool presented in the figure illustrates that there are four principal components to the application (note that, there are other minor components that – for clarity – have not been included within this diagram). The first of these components is the set of configuration parsers, this set includes a number of parsers that – amongst other things – parse the configuration file Frog accepts on startup and processes the character dictionary configuration. We have no overall configuration factory, but a factory for each of the parsers, which is evoked from the GUI at the appropriate time. Note that, the information flow between the configuration parsers and the GUI is unidirectional. Once an event in the GUI leads to a configuration parser being called, it simply extracts the required information from the appropriate file, loads the appropriate objects with that information and passes that information back to the GUI where it is processed as required. One of the configuration parsers that belongs to this set is the implementation of the Frog-CCL parser described in chapter 5.

The second of the tool's major components is the parsing factory. The parsing factory implements the behaviour described in chapter 4; that is, given a file that contains Z sections (or constructs) and a Frog specification,

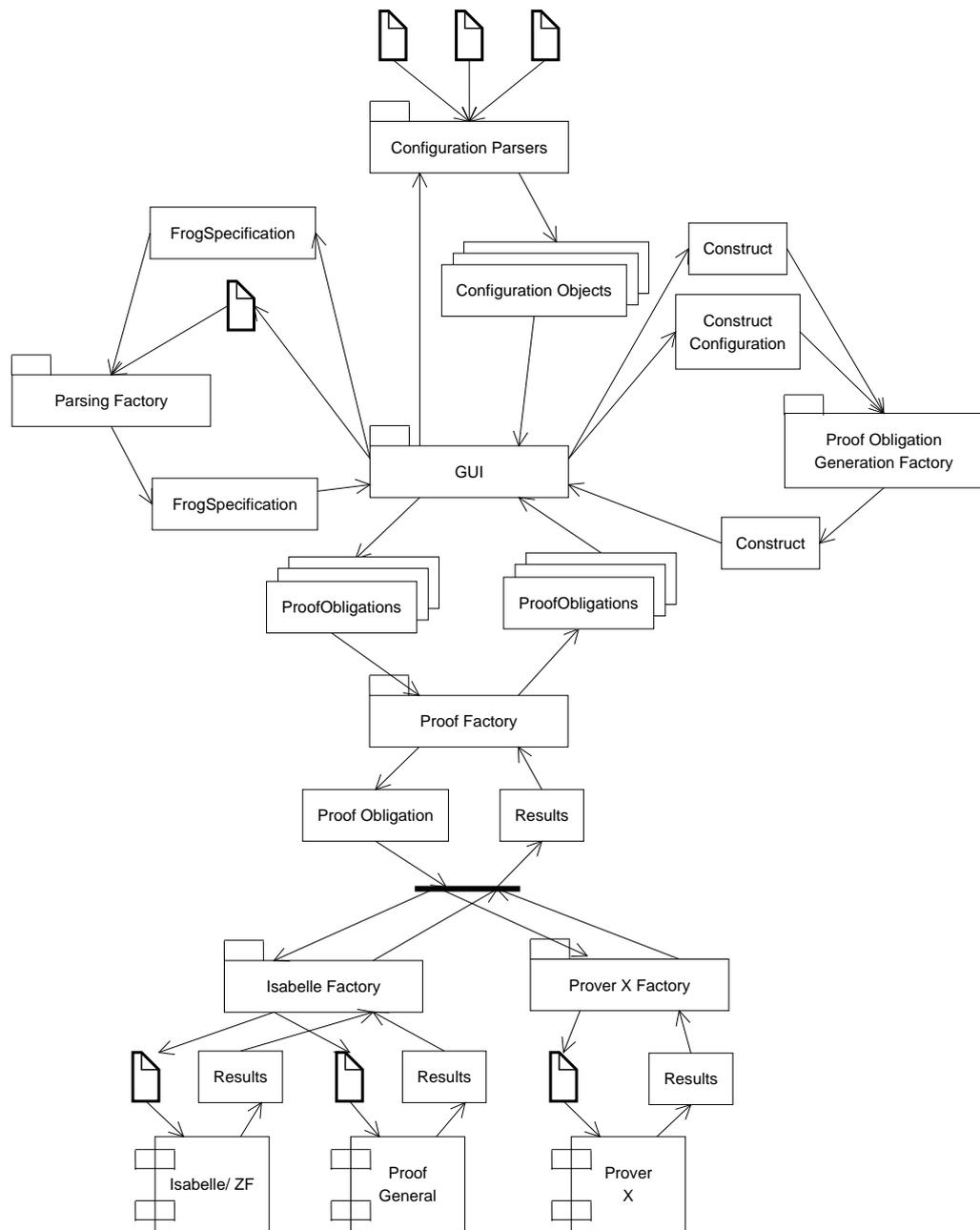


Figure 7.1: Overview of Frog Architecture

the factory will use the configuration information belonging to the specification to syntax check, syntax transform and type-check the file. If successful

it adds the appropriate constructs, sections and paragraphs to that specification. Note that, all communication between the GUI and the parsing factory is performed through the specification. If errors are generated in the parsing process, then they are attached to the appropriate place in the specification, rather than being passed directly back to the GUI. Once the GUI receives the specification back from the parsing factory, it refreshes its view of that specification and any errors are immediately apparent.

The next component is responsible for the generation of proof obligations and automates the processes described in sections 6.2.2 and 6.3. Unsurprisingly, this component is known as the proof obligation generation factory. Once the GUI invokes the factory by passing it the selected construct and its configuration, the proof obligation generation factory will attempt to generate the semantic binding environment for the construct. It will then use that information to instantiate the generic proof obligations contained in the construct's configuration. Again all communication between the modules is performed through the given construct object, with either the generated proof obligations or appropriate errors being attached to the construct. When the proof obligations have been generated, the GUI will refresh its view of the specification which will take into account any changes to the construct object.

The final component is that responsible for the interaction with theorem provers described in section 6.5. This component is slightly more complex as it involves a generic factory and specific factories for each supported theorem prover (in fact there is currently only a single theorem prover supported, but the architecture has been designed so that more than one can be easily used). The GUI interacts with the generic proof factory that takes the set of proof obligations provided and passes them one at a time to the selected theorem prover's factory. Assuming the Isabelle factory is being used, the proof obligation is then translated into the appropriate Isabelle syntax and a suitable theory file created. That file is then used to invoke Isabelle/ZF and the results are retrieved by the Isabelle factory. The Isabelle factory then returns the results to the proof factory (having possibly invoked Isabelle's interactive proof assistant – Proof General – beforehand). The proof factory

then processes the results, altering the status of the proof obligation and attaching any useful output from the theorem prover. Once again, when the GUI receives the processed proof obligations, it will refresh its view of the specification, and any changes will be reflected.

As might be expected from reading the preceding paragraphs of this section, the GUI itself has little functionality; it simply provides a graphical view of the current specification and allows users to invoke the various modules on that specification. However, the GUI module also incorporates the construct manager and this is used to calculate dependencies within a specification.

We mentioned earlier that the modular design of Frog was used to allow the greatest flexibility. In our initial architecture, we have only made provision for the use of multiple theorem prover factories. In addition, we have made it the user's responsibility to ensure that when using multiple theorem provers, they do so in a sensible way. As different theorem provers use different logics it would not usually make much sense for a user to use more than one theorem prover within a given Frog specification. As flexibility is one of our principal aims, however, we have not currently placed any constraint on the use of multiple theorem provers.

In the future, we hope to have a choice of modules at each stage of the specification and verification process. In essence, we would like to see a number of parsing factories, proof obligation generation factories and proof factories. As the number of choices increases, however, so does the number of nonsensical combinations. In order to introduce some sense to this whilst maintaining flexibility, we foresee the introduction of an additional configuration file that defines valid paths through the modules. For example, a B parser factory could be used with a B proof obligation generator factory whose proof obligations could be discharged by one or more proof factories interfacing to B-Tool, Atelier-B or RODIN (see section 3.2.1). These combinations could become quite complex so would probably be best represented in a structure such as a directed acyclic graph. The tool would come with a configuration representing the default paths through the graphs, but users could override those defaults where they feel it is appropriate to do so. Users would also be able to create their own modules and indicate to Frog their

compatibility with existing modules.

7.3 The User View

Having looked at the structure of the Frog application, we now examine the way in which it is used. That is, we shall assume the perspective of a user, and investigate how the components are integrated to make a practical and useful tool.

7.3.1 Configuration

When launching the tool, Frog accepts a configuration file. However, this file only contains the system information that Frog requires to run (for example, the location of the directory in which the images required for its icons are kept).

Generally, all configuration is performed at the specification level and is associated directly with a specification. When a user creates a new specification, Frog loads the configuration to be used with that specification. Note that, this configuration is now associated directly with the specification and any link with the configuration files is severed. Any changes made to the configuration files will not affect the configuration of existing specifications using those configuration files. We felt that the configuration needed to be linked directly to the specification, as the resolution of alterations would be far too difficult to handle automatically. This is especially true as the rules governing this resolution would likely change between specific instances. For example, it may be that we wish to change the Isabelle/ZF operator that we use to perform a specific user defined operator; if this change is made, do we use the new operator only in new proof attempts, in newly generated proof obligations or do we re-examine all existing proofs, substitute the operator, and attempt to reprove? If in the future, we feel that there is a case for altering configuration during a specification's lifecycle, we will provide tools to alter the particular specification's configuration directly; allowing a link between a specification and a configuration file leaves open the possibility

that old specifications become unusable when the configuration changes.

When we create a new specification, all of the available construct configurations (see chapter 5) are loaded from the appropriate locations, and presuming they are parsed successfully, will be associated with the new specification. If there are any errors in the parsing of a particular construct configuration then that construct will not be able to be used in the new specification. We have considered the creation of a tool that will allow users to edit¹ and test their construct configuration before creating a specification, but resource constraints have prevented the development of such a tool to date. As we have mentioned previously however, the configuration of a specification will no longer have a link to the underlying configuration files. Hence, we felt it was mandatory to include a tool that allows users to examine the configuration that they are currently using. This tool – the construct configuration browser – also allows users to ensure that their construct configuration has been parsed correctly (that is that both tool and user have the same understanding of the configuration). The construct configuration browser can be seen in figure 7.2 on the facing page and provides details of every aspect of a particular construct’s configuration.

It is also necessary at this stage to load the character dictionary (see section 4.6). Again the specified file is retrieved and Frog attempts to parse it; any failure in the parsing of this file will not prevent a new specification being created, but will restrict the characters available. Again we have considered a tool that allows users to experiment with their character dictionary configuration, but at present this tool is not considered a top priority. Similarly, we have not felt it worth the effort to create a browser for the character dictionary as we don’t believe that the configuration is likely to change significantly; having said that, when resources allow, we would like to include such a tool. Also there exists the possibility that such a tool could allow users to add characters to a specification’s character dictionary dynamically (altering existing characters would not be allowed for the reasons stated above).

¹Currently, it is necessary to edit construct configuration files outside of the Frog tool.

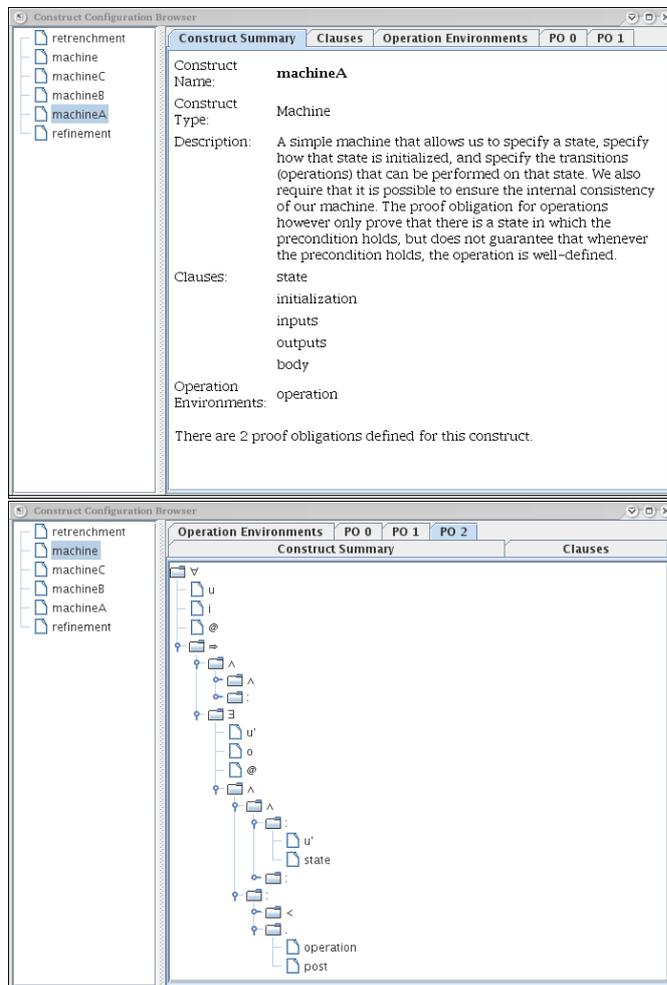


Figure 7.2: The construct configuration browser

7.3.2 The Development Process

Once a specification's configuration has been loaded successfully the user is prompted to specify a file in which to store the new specification and is given the option to load *Z*'s standard mathematical toolkits (see section 4.4.2). At this stage the user can select to load all of the standard toolkits (the default option) or to deselect them as required (it is also possible to deselect the inclusion of the standard prelude section, but no guarantee can be made of the tool's ability to perform in this circumstance). The tool forces the user to maintain integrity between the toolkits; that is, a user cannot select one

toolkit unless all of the toolkits on which it depends have also been selected. Note that, should a user choose not to load a toolkit at this stage they will be free to load the toolkits (in the same way as any other Z file) at some future point. However, when we load a toolkit at specification creation time it is afforded a special ‘toolkit’ status which allows it to be treated slightly differently to other files (for example, we do not normally make toolkit files available for editing within the tool – freeing space for the important files of the specification). Once the user has chosen which toolkits to include, the new specification is created, the toolkits parsed and type checked and the appropriate sections and paragraphs are added to the specification.

At this point the user may choose to add one or more files to their specification. Each file may contain either one or more (possibly anonymous) Z sections or a single construct. Files can be added by either creating a new file within a specification or opening an existing file. Figure 7.3 shows both a Z section and a machine construct open for editing.

Only basic editing facilities are provided (although certain JVM implementations provide implicit tools that can be used on certain operating systems). We would advise that, at present, the editing facilities are used principally for making corrections to existing specifications as our editor lacks many of the useful services that the modern user relies upon (for example, there is no search and replace facility). When time allows we hope to perform research into obtaining a more powerful editor that we can integrate into our tool. For now however, we feel that our basic editor is sufficient for our needs.

Once the editing process is complete, the user submits their file for syntax checking. An overview of this process is shown in figure 4.3 on page 131; the file is parsed, transformed and finally type checked. If a file comes through this process successfully its sections and constructs are added to the specification. If any errors are found they are reported back to the user; sections and paragraphs may be added to the specification, but this depends on how far the process progressed before errors were unearthed.

Frog provides three views of the current specification. The first of these is the file view which is what was seen in figure 7.3 on the facing page. Files

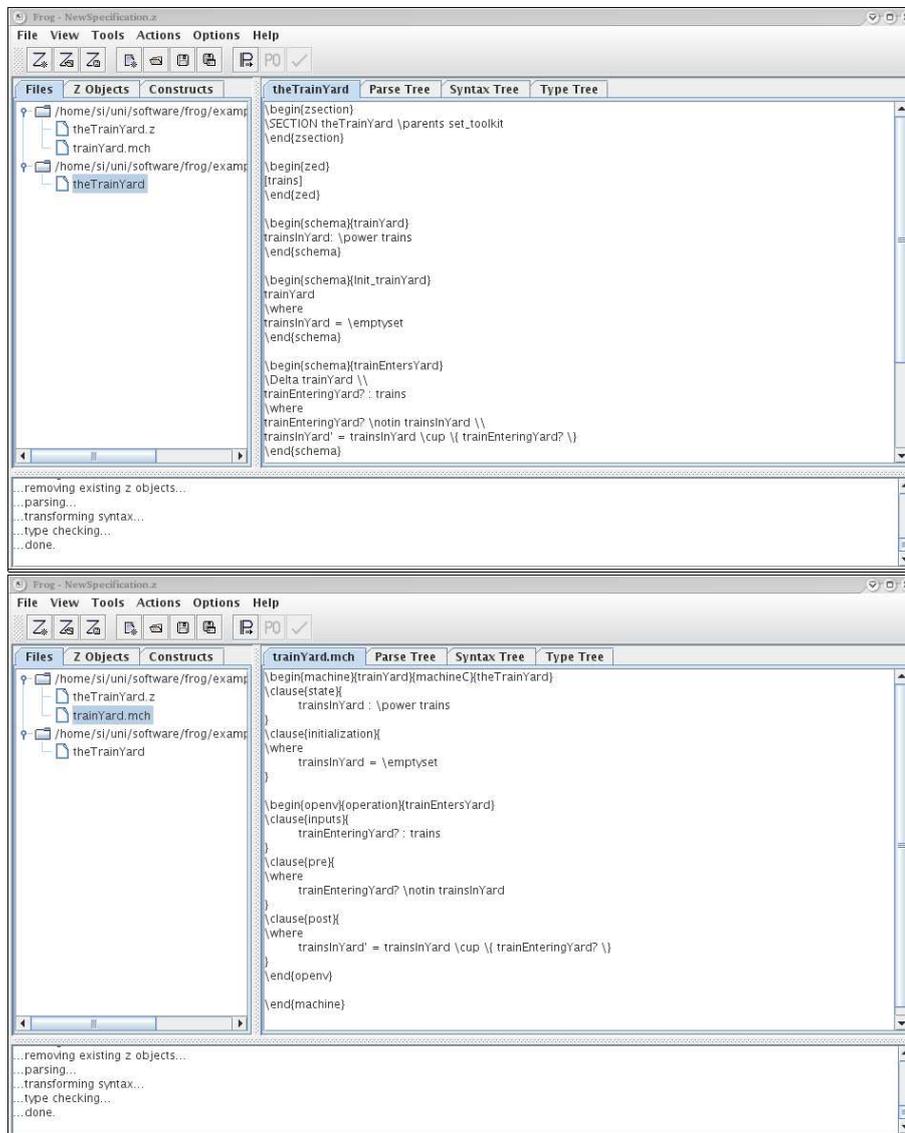


Figure 7.3: Editing files in a specification

are grouped by directory, and clicking on a file will allow it to be edited in the main window. When time allows, we will alter this view so that it is easy to distinguish between files that have been type checked and have not been changed since that type check, files that have been type checked but have since been altered and files that have not been type checked. Once a file has been type checked, a number of tabs appear in the main window

that allow the user to examine how it has been parsed. As well as the tab that allows the user to edit their file, there is a tab that shows the abstract syntax tree (AST) produced in the syntax checking phase of the parsing process, a tab that shows the AST after it has been subjected to syntactic transformation and finally a tab that shows the transformed AST annotated with the appropriate types. Examples of the trees presented in some of these tabs are shown in figure 7.4 on the next page. We make all of this information available to the user; firstly, it is extremely useful when trying to discover bugs in your specification – the tree view allows you to trace the problem back through the various phases of parsing. Secondly, this information is helpful when configuring new constructs and indeed when making extensions to the Frog tool itself. We believe that – while in a commercial product we may have a different aim – here the goal is to make as much information available to the user in order to encourage experimentation.

The second view is the Z object view, which allows us to browse all the Z objects that have been added to the specification (only successfully type checked objects are added to the specification). We present a list of sections which each contain their component paragraphs. Clicking on a section or paragraph will present the user with an HTML representation of that section or paragraph (a second tab is provided in the main window that allows a user to examine the parse tree of a paragraph individually). The HTML representation of the Z objects (see figure 7.5 on page 428) is a close approximation of the \LaTeX rendering outlined in [ISO02]. There are some limitations, but we feel that the meaning of the Z is clear in our representation, and certainly provides an advantage over the non-rendered \LaTeX source. The user is able to save the HTML representation of a section or paragraph. At present this is the only alternative representation that Frog translates to, but clearly an ANTLR tree walker could be easily constructed to pass over any of our ASTs and produce specifications in a number of different syntaxes.

Note that the paragraphs and sections created from a construct are available in this view. Again this allows users to ensure that their construct configurations are performing correctly.

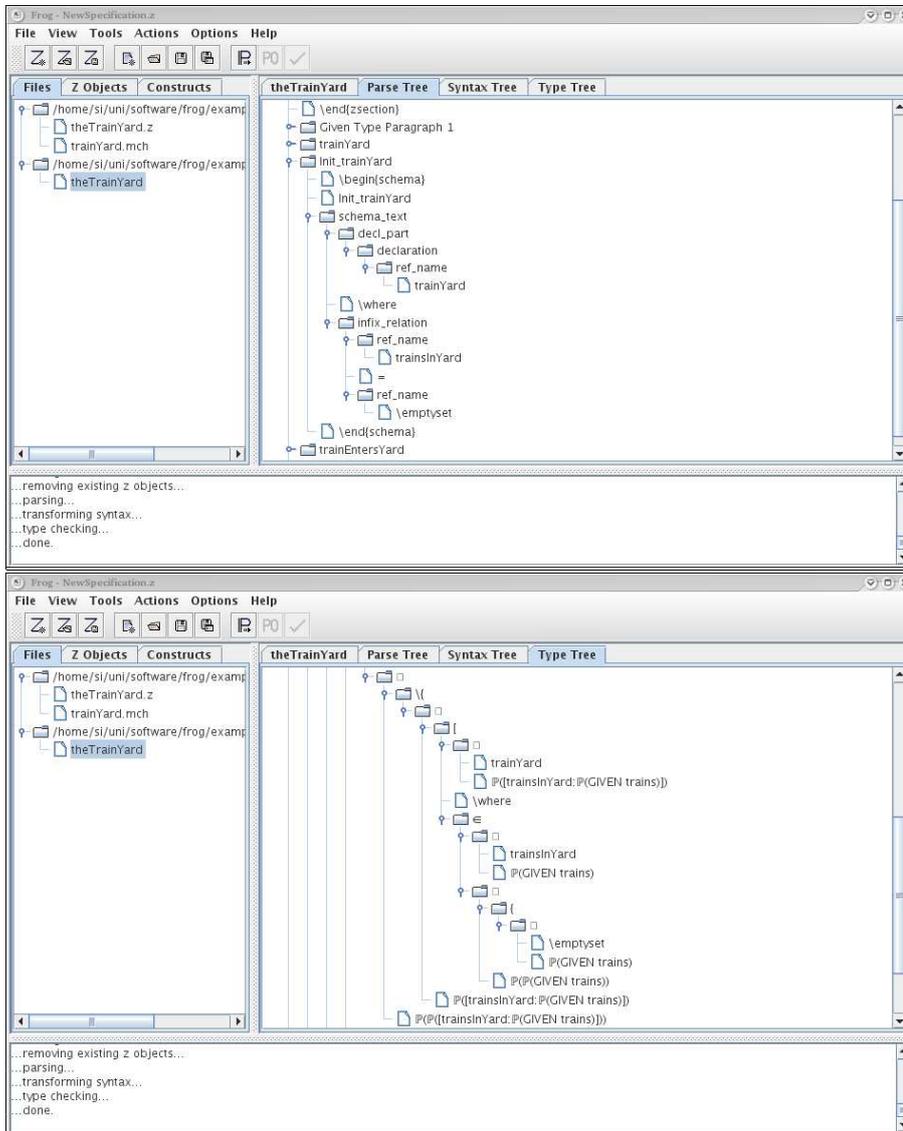


Figure 7.4: Viewing the ASTs belonging to a file

The final view is the construct view which allows us to browse the constructs that have been added to a specification. Clicking on a construct causes its HTML representation to be presented in the main window (again, a second tab details the construct's own parse tree). The HTML representation is similar to the \LaTeX rendering presented in this document (see figure 7.6). The principal difference – which can be observed in the figure – is that

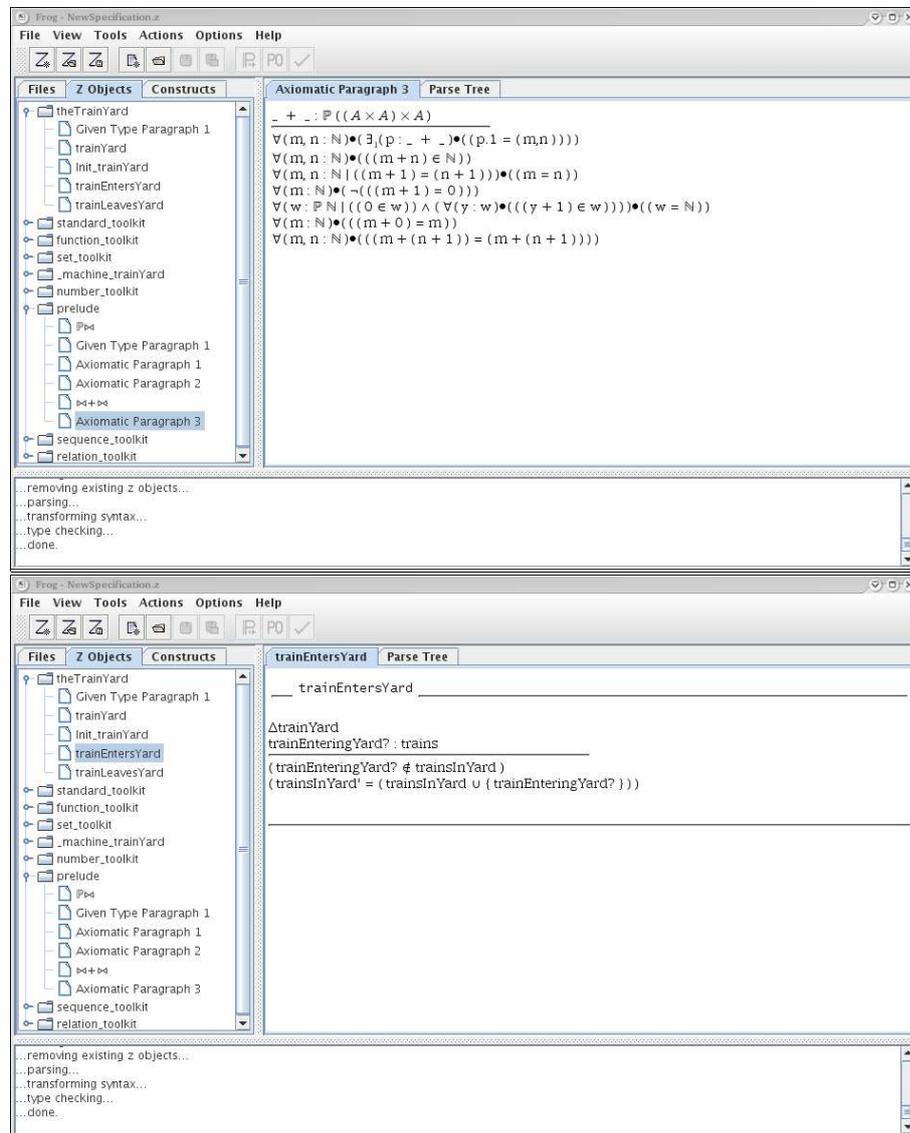


Figure 7.5: The HTML representation of paragraphs in the Z object view

we incorporate the implicitly referenced schemas (with a ϕ prefix to indicate that they refer to clauses) of each clause where the designated content is a schema text. Obviously this cannot be incorporated in the \LaTeX rendering as the construct configuration is not available.

Once a user has selected a construct in the construct view, two further options become available to them. These options allow the user to either generate the

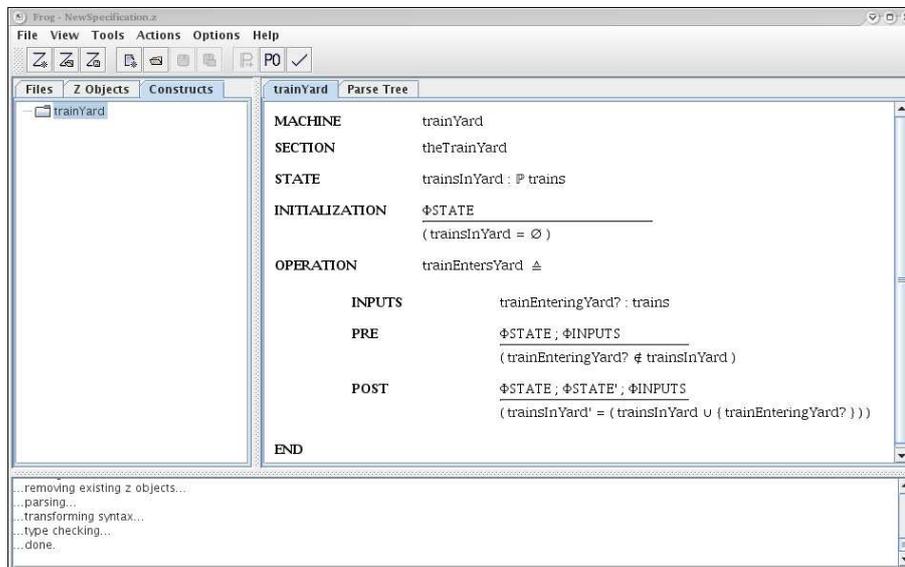


Figure 7.6: The HTML representation of constructs in the construct view

proof obligations for that construct, or to attempt to prove that construct (proof obligations will be automatically generated if we attempt to prove without any obligations).

7.3.3 The Proof Process

Generating proof obligations is typically the next step in the lifecycle of a construct. This is done on a per construct basis; that is, if a construct's proof depends on other constructs then we only prove our construct with the assumption that those other constructs are valid – it is not necessary to wait until we have proved those constructs. Generating the proof obligations for a construct simply involves selecting a construct and pressing the 'generate proof obligations' button. Frog then generates the semantic binding environment for that construct, examines its construct configuration and instantiates the generic proof obligations (where appropriate) to create a set of theorem prover-independent proof obligations. The proof obligations are added to the construct view as children of their construct. Each proof obligation is assigned a name; this name comprises the construct name, the string '_pob_' and a string that uniquely identifies the proof obligation. If the proof

obligation is construct level, then a single digit is used that corresponds with the identity of the proof obligation in the construct configuration browser. If the proof obligation is operation level, then one digit is used to identify the generic proof obligation and the second provides an index corresponding to the ordered list of operations belonging to the construct. For example, the machine *trainYard* has three proof obligations shown in figure 7.7. The first *trainYard_pob_0* is an instance of machine proof obligation zero — the machine initialization proof obligation. The remaining proof obligations, *trainYard_pob_1_0*, and *trainYard_pob_2_0* are instances of the machine’s operation level proof obligations – termination and correctness – that apply to the *trainEntersYard* operations of the construct. If we had defined the *trainLeavesYard* operation within this machine, we would have two additional obligations: *trainYard_pob_1_1*, and *trainYard_pob_2_1*. Clicking on a proof obligation brings a summary of that obligation into the main window. For each, proof obligation the summary contains its name, its proved status, and an HTML representation of the theorem-prover independent proof obligation. Note that, we use the ‘♠’ prefix with type names to indicate the appropriate carrier set.

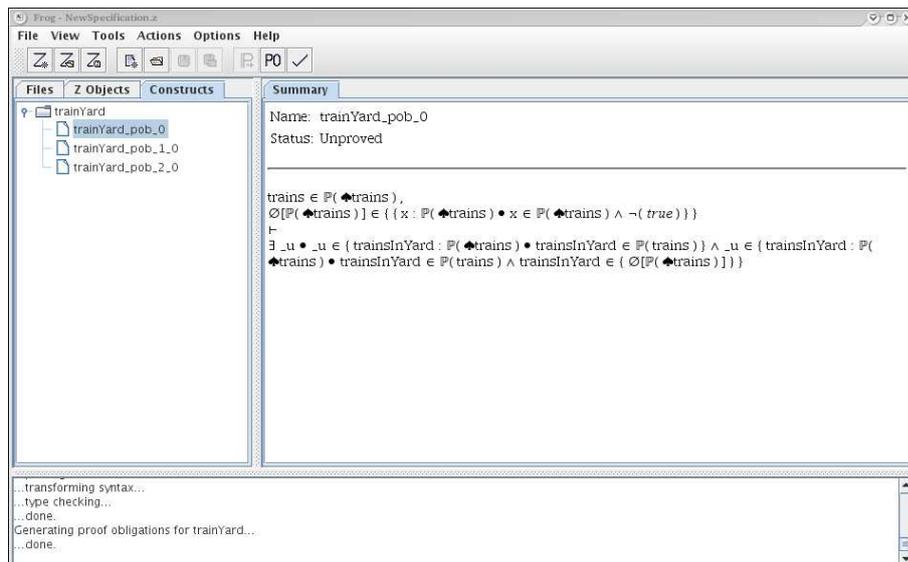


Figure 7.7: Proof obligations

Once the proof obligations have been generated, the next logical step is to attempt to discharge those obligations. The proof process is outlined in figure 6.1 on page 301. A user can choose to attempt the proof of a single obligation or to attempt the proof of all obligations belonging to a given construct (by default the tool will select all currently unproved obligations). The tool also asks the user which theorem provers they would like to attempt the proof with. In theory, and for maximum flexibility, the user could select different theorem provers for different obligations, but in practice the different logics used by those provers mean that most sensible developments will use only a single theorem prover. Currently, Frog only supports the Isabelle/ZF theorem prover, but the intention has always been to provide support for a number of theorem provers, making it possible to leverage their different strengths.

Assuming therefore, that the user has chosen to use Isabelle/ZF to prove their proof obligations, they will then be presented with a number of theorem prover specific options such as those shown in figure 7.8 (although the options in this instance are specific to Isabelle/ZF, it is foreseen that the options for other theorem provers will be of a similar nature). The principal choice here is to decide whether to use one of the ‘automated’ proof theories (‘auto’, ‘blast’ or ‘simp’), or specify more specialized tactics. These choices are made per proof obligation, so it is possible for a user to use an automated tactic for one obligation, and their own tactics for another.

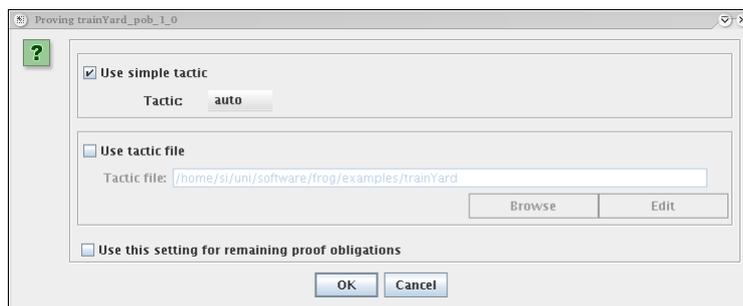


Figure 7.8: Isabelle/ZF proof options

If the user choose to use their own tactics they can simply choose an existing tactic file, or they can create or edit a new tactic file. Frog provides a

simple tactic editor (figure 7.9) for this purpose. This tactic editor provides a window where the user can see the Isabelle/ZF representation of their proof obligation, and a second window where they can enter the tactics necessary for the discharge of the obligation.

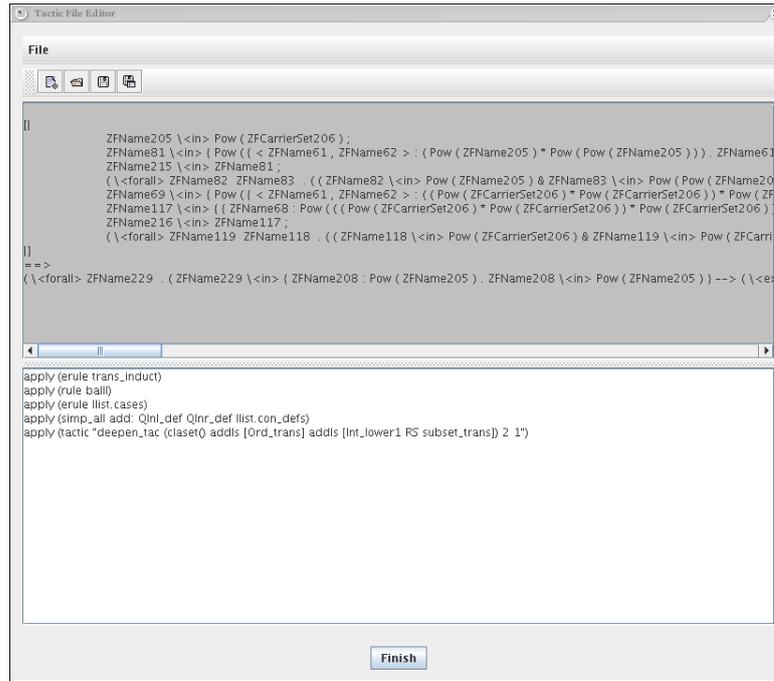


Figure 7.9: Tactic file editor

Once the user has finalized the tactics that they wish to use, the Isabelle theory – augmented with the selected tactics – is submitted to Isabelle/ZF. If the lemma contained within the theory can be proved, then we progress to the next proof obligation. If the lemma cannot be proved then Frog offers to open Proof General (Isabelle’s interactive proof assistant). Once the user has finished with Proof General, Frog asks them whether the proof obligation has been discharged (there is no way in which Frog can determine whether the obligation has been verified without the user’s intervention), before moving on to any remaining proof obligations. Of course, once a user has used Proof General to determine the required strategy to prove their obligation, the user can simply save the required tactics in a file and apply that tactic file to the obligation through Frog.

Once the user has attempted to prove all of the selected proof obligations, they are returned to the construct view. The summary tab for each of the proof obligations – where a proof has been attempted – has been augmented with two further tabs, the first gives the Isabelle/ZF representation of the proof obligation, and the second gives the output Isabelle returned when we attempted to discharge it. Furthermore, the summary tab of the proof obligation is updated, not only updating the proof status, but indicating the theorem prover with which a proof was attempted (and detailing which automated tactic or tactic file was used in the proof). This can be seen in figure 7.10 on the following page. As before, we would like this screen to give an immediate overview of which constructs and proof obligations have been proved, which have been attempted unsuccessfully and which are awaiting an attempt. Unfortunately, time constraints have prevented the implementation of this, but we foresee the use of colour to differentiate the various states in the construct browser.

This completes the description of the core functionality of the tool. There are, however, a number of further aspects that must be examined, particularly how we handle changes to constructs which have already been proved.

7.3.4 Altering and Removing Files: Maintaining Integrity

The various elements of a specification will usually have a reliance on other parts of that specification. For example, a Z section will have a dependence on any parent sections and a relationship has a dependence on the machines it relates. In order that a user can see the dependencies within their specification we provide a dependency browser. This browser allows a user to view the (non-construct) sections and constructs that belong to a specification, together with details of how they are related. The dependencies in a specification containing some of the examples we have discussed in this thesis can be seen in figure 7.11 on page 435.

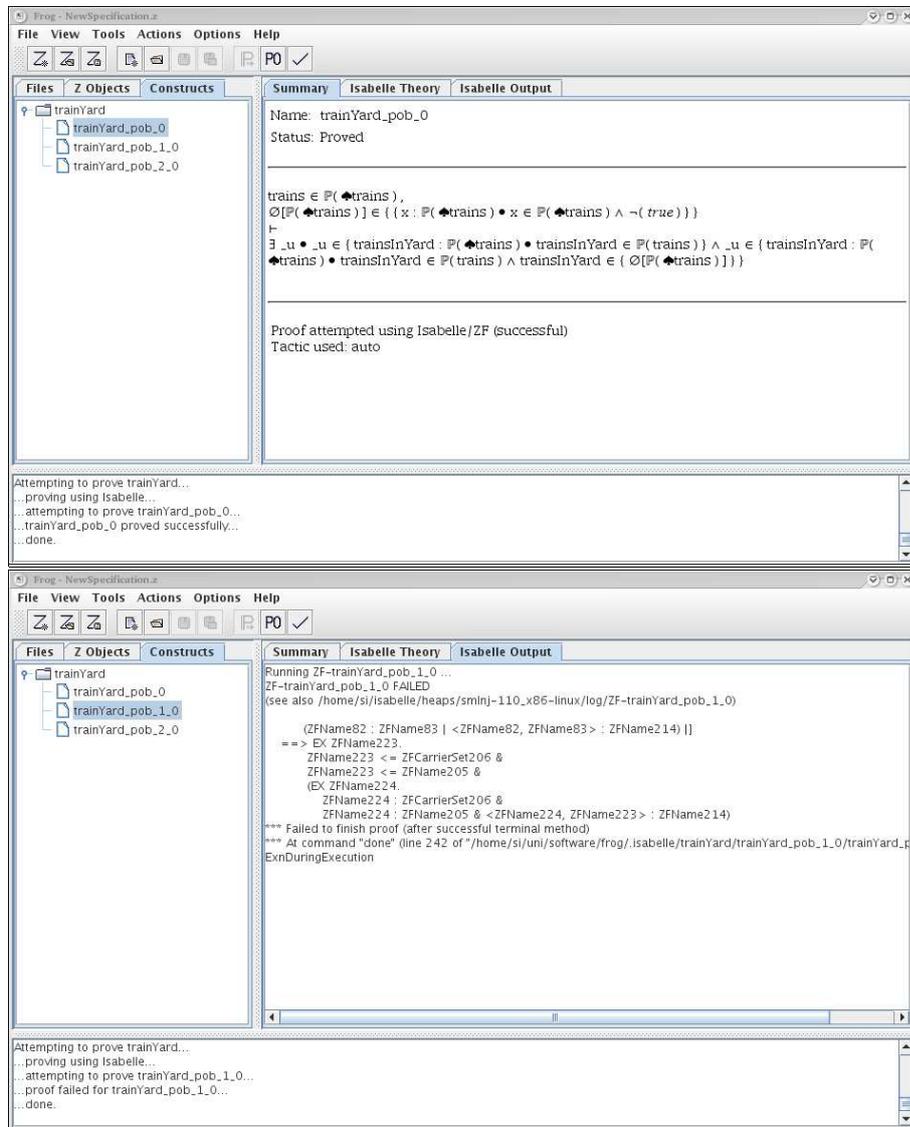


Figure 7.10: Results of proof process

Frog specifications typically have two types of relationships between constructs and sections. A parent-child relationship, or a source machine - relationship - target machine arrangement. Clearly, when an object (that is a section or construct) depends upon another, then a change to that (relied upon) object could have significant consequences for the (relying) object.

If a change is made to a file but it has not been type-checked (that is, the

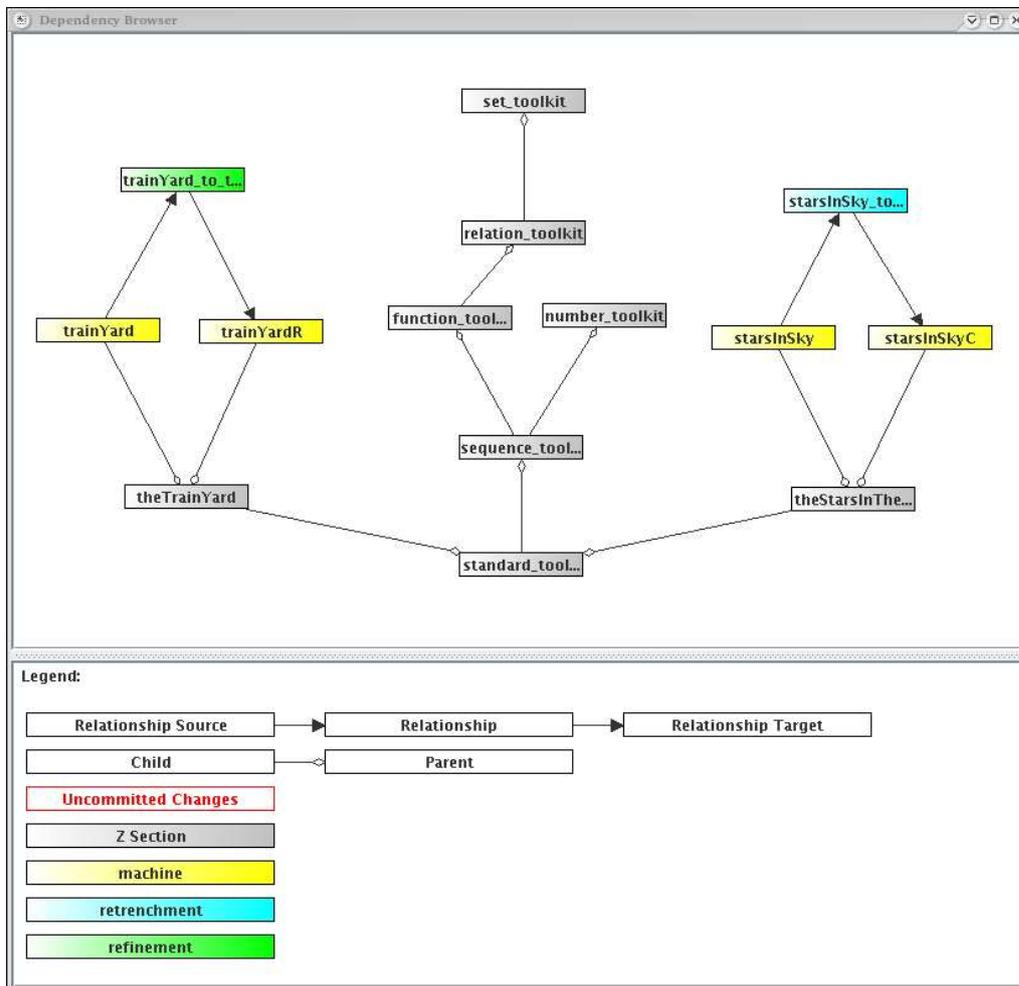


Figure 7.11: Dependency Browser

changes have not been committed to the specification) then we simply highlight the objects defined within that file in the dependency browser (changing the text colour to red). As these changes are considered to be pending they do not actually affect the meaning of the specification and this highlight is considered to be a warning.

However, if the changes to a file have been committed, there clearly exists the possibility that the changes will invalidate one or all of the objects that depend upon the objects defined within that file. These changes do not just affect the proofs of constructs, but can alter the way in which they

are parsed. Consider for example, a machine that uses an operator whose operator template is altered in the parent section such that the operator becomes postfix rather than prefix.

When a user re-submits a file for syntax checking we make a note of the dependencies before removing the objects contained within the specification. Once syntax checking is complete (and successful) we attempt to re-establish the links between objects, but of course this does not eliminate the possibility of discrepancies. Where an object has been re-submitted therefore, we changes all the appropriate arrows in the dependency browser to red until the dependent constructs have also been re-submitted. We also add a note to all proof obligations (whether verified or not) that the obligation is out of date, and that verifying the obligation will not necessarily verify the construct. Unlike some tools we do not automatically re-submit objects that rely on another re-submitted object. For maximum flexibility we rely on the user to ensure that their specifications maintain integrity. As any lack of dependency should be easily identifiable in the dependency browser, we do not feel that this should be a major issue, but allows the user to exhibit complete control over their specification.

When a user re-submits their file and it fails syntax checking or when a user removes a file from a specification, the results are slightly different as we are unable to re-establish the links between objects. For instance, a specification may contain two machines, A and B , where A is the parent of B . If the user removes the file containing A from the specification, then B is dependent on a non-existent construct. In the dependency browser we colour red the boxes of all objects that are dependent on non-existent objects (and their ancestors). If following the removal, the user should add a new file that redefines a machine A , we will not automatically re-establish the link (as they may just share a common name) until machine B is re-submitted.

7.3.5 Other Tools

As our main aim has been to get the core functionality of the tool working effectively and efficiently, we have not yet given much time to the development

of supporting tools.

The nature of the abstract syntax tree that we produce when parsing our Z specifications and constructs makes it easy to create ANTLR tree walkers that can produce alternative versions of our specifications in different syntaxes. This also has the advantage that we can use one tree walker to create representations of our specification at any stage of its parsing. Hence, we can produce typed versions (for input to CZT) as easily as translating plain specifications. Currently, our tool only produces specifications in HTML (that we also use for displaying our constructs within the tool), but we would like to produce translators to the ZML and ASCII (email) syntaxes relatively quickly. An example of the HTML produced by the tool is shown in figure 7.12 on the following page.

We have also begun work on an interactive, HTML help system (figure 7.13). These pages are generated from a raw format by a number of Perl [SC97, Wal00] scripts. We use the raw format so that the help files can be easily translated into a selection of alternate formats (for example, L^AT_EX). Presently, the help pages are only sparsely populated with tips and tricks that we have found useful whilst using the tool to verify our case studies. We hope that a more comprehensive guide to the system will be completed shortly.

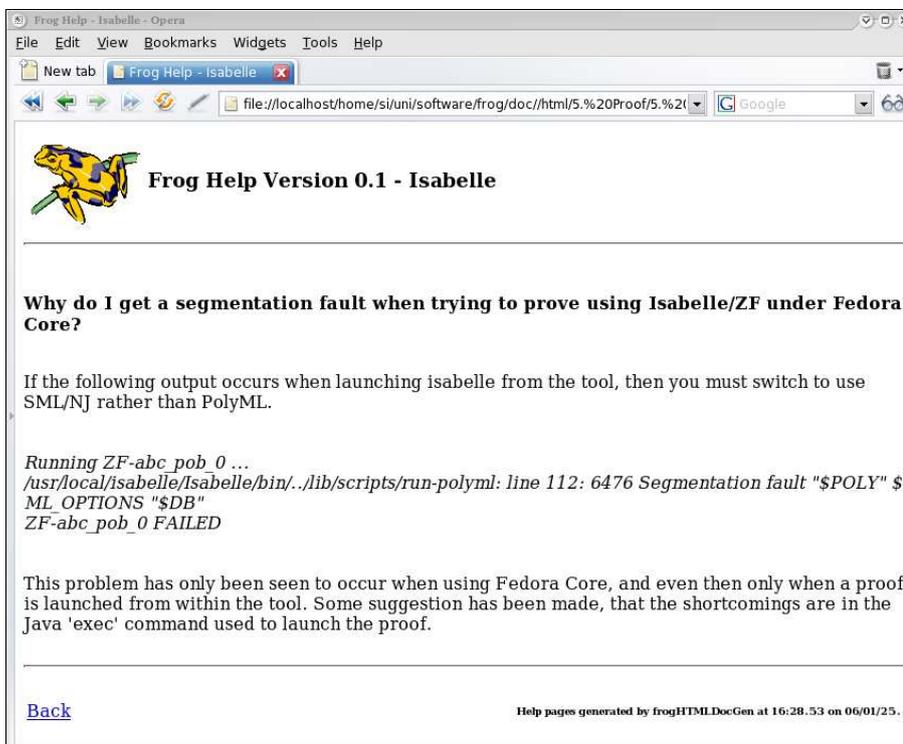


Figure 7.13: Help System

7.4 Implementation

The tool is implemented by more than two hundred Java classes and relies upon an assortment of scripts and libraries. In this thesis, we do not have the space, nor the inclination, to describe the functionality or design of each of

these. Instead we use the following sections to present an abstract description for each of the tools principal components. In the remainder of this section we shall outline some of our application wide choices.

Implementation Language

Frog is implemented primarily in the Java programming language [Dar01, Fla02, Fla04] and makes heavy use of the ANTLR language recognition software that we described in section 4.6.2. Initially, our choice of Java was motivated by the ‘write once, run anywhere’ tag line as we sought to make the tool available to the widest audience possible. As most software developers are aware however, Sun’s claims are not to be wholly believed and there are instances where provision needs to be made for the particular operating system in which the user runs a program. For this reason – and the more important restrictions placed on us by the availability of portable theorem provers – we have chosen to concentrate our efforts and ensure the tool is viable on the Linux [Sch04, SFW05] platform. Despite this drawback, the advantages provided by Java’s simplicity and the ever increasing number of freely available classes belonging to the class library², make us believe that we have made the correct choice as one of our principal goals has been to make our tool easily configurable and extensible. Of course, Java’s widely-publicized Achilles’ heel is its lack of speed (and intensive memory use). If we were developing a tool for commercial use this may be a problem, but as we see this tool as (currently) being used in research environments for experimentation we feel that the relatively poor performance (in practice Frog responds at roughly the same speeds as most modern applications and waiting is only required when syntax checking particularly large or complex specifications) is more than outweighed by the clarity of the language and the programs in which it is used.

²Growing to over three thousand classes in J2SE 5.0.

The Influence of the Parser Generator

The choice of ANTLR as the parser generator has influenced our implementation. We discussed in section 4.8 that this choice has led to a significant difference between Frog’s implementation and that of CZT. All the nodes of an ANTLR AST have the same class and differ only in their token type. With CZT the tree is an abstract concept and is comprised of Java interfaces, classes and the relationships between them. When we wish to process an ANTLR AST we use a tree walker which processes the nodes depending upon their token type. An ANTLR tree walker provides numerous, documented facilities for manipulating the structure of the tree so we only create instances of Java classes (to represent a specification) when all tree manipulation has been completed. CZT uses its own variation on the classical visitor design pattern [GHJV95, Mar02] to perform the functions equivalent to the tree walker. We feel that this approach is a lot clumsier in dealing with tree manipulation. Once manipulation is over we use the Java classes simply for the generation of proof obligations, we had decided not use a visitor pattern as the generation process is atomic and is only ever completed once for a given class³. We could have generated the proof obligations using a further tree walker, but we felt that it was easier to retrieve all of the required information when using the object structure. In any case we feel that the presence of these classes makes the tool flexible and more extensible in the future; allowing a future developer to manipulate either the ANTLR AST or the equivalent object structure.

Logging

Java’s utility package includes a ‘Logger’ class that allows a developer to record progress in a tool. This is useful for debugging as it allows the developer to see the flow of control through their program and to examine the values of variables throughout their lifecycle. The logging level of a Logger can be set dynamically and on each logging call, this level is compared to the

³Adding a visitor pattern to our design and implementation would, of course, be a relatively simple task. It is certainly our intention to ensure that our classes implement this pattern in the future.

level of the call. When the level of the call equals or exceeds that of the Logger some action will be performed. While Sun claims that this check is cheap to perform, we required extensive use of logging within Frog and felt that a static logging system would prevent these overheads affecting our tool's performance (clearly, these checks are required even when in production mode). With this in mind we decided to create an additional pre-processor for our ANTLR and Java source files using Perl. This pre-processor examines each of the target files and replaces appropriate directives with Java commands that allow us to log a variety of events. The actual code that we generate is highly configurable and could include simple output to the terminal, the use of a logging frame within the tool itself or a more complex tracking facility. For example, the following directives could be used in our ANTLR type checker.

```
//DEBUG TYPE_ENVIRONMENTS,LINE,+
//DEBUG TYPE_ENVIRONMENTS,TEXT,Section type environment created.
//DEBUG TYPE_ENVIRONMENTS,VAR,currentTypeEnvironment
```

Each directive contains fields: the flag which is required to be set for the command to be logged (`TYPE_ENVIRONMENTS`), the type of logging action to be performed (`VAR`) and the argument to that logging action (`currentTypeEnvironment`). In the simplest instance these directives could be converted by the pre-processor to commands that echo output to the terminal as follows.

```
//DEBUG TYPE_ENVIRONMENTS,LINE,+
System.out.println("++++");
//DEBUG TYPE_ENVIRONMENTS,TEXT,Section type environment created.
System.out.println("00000 - Section type environment created.");
//DEBUG TYPE_ENVIRONMENTS,VAR,currentTypeEnvironment
System.out.println("00001 - currentTypeEnvironment= "
                    + currentTypeEnvironment);
```

Running the pre-processor generates a copy of our source files – that incorporated the relevant logging commands – that are then used to actually build

the tool. We created a number of categories and sub-categories for our logging system that allowed us to maintain control of exactly what was logged. The logging choices are specified at compile-time by passing the Frog build tool a configuration file that indicates which logging flags have been set.

Testing

Java assertions were used in the source code of the tool and activated in the testing phase to ensure that any assumptions that we made in the implementation were valid. Each assertion contains a boolean expression that must evaluate to true when the assertion executes. If an assertion is found not to be true the system will throw an error. This verification allows us to be more confident about our assumptions and reduces the likelihood of unexpected errors. Assertions can be enabled or disabled at runtime so will be ignored when running the tool in production mode.

Unit testing was performed for critical classes with the JUnit [BG00a] testing facility. JUnit is a simple, open-source framework that extends Java and enables a developer to write repeatable tests. The unit tests are created in classes distinct from the source code, where JUnit allows us to perform operations with instances of our classes and make assertions about our expected results.

We used our tool to process all of the examples within this thesis and further examples from [ISO02] (and other resources). We subjected our tool to further evaluation by conducting a less trivial case study. The results of that case study are presented in chapter 8.

7.4.1 GUI and Construct Management

The GUI and the incorporated construct manager form the heart of the tool. From here all tasks are executed and the structure of a specification managed.

Designing a GUI

The GUI itself is fairly primitive and is designed to allow access to all of a specification's data without producing too much clutter. The GUI is implemented with the Swing libraries, and assumes the look-and-feel of the platform on which it is run. As with most interfaces the construction of the GUI was tedious and long-winded, but presented few significant challenges. In order to achieve the most attractive user interface, we provided an option to take advantage of Java 1.5's undocumented anti-aliasing attribute for text contained within Swing components. We have described how the GUI is used in section 7.3 and we will not go deeper into its implementation here.

The Construct Manager

We have briefly described elements of the construct manager's functionality in places throughout this thesis. We will present here a short synopsis of that information. We showed in figure 5.1 on page 228 the structure of a Frog specification. Only one specification can be open at any one time, and the notion is equivalent to that of a project in an integrated development environment. A specification can consist of one or many files. These files can contain either Z sections or construct definitions. We maintain the concept of the file within our specifications in order for us to keep track of inter-object dependencies. As the file is not a genuine syntactic block, we also consider each Z section and construct to be directly a member of the specification. Each construct is also related to its equivalent Z section and each section is formed from a number of paragraphs. The contents of the paragraph depend on its nature. For example, a paragraph can comprise expressions, predicates or a list of given types. Files, sections and paragraphs are all stored with the AST that forms their specification.

The construct manager is principally used to ensure that the specification maintains this structure and that internal integrity is preserved. In section 7.4.3 we will describe how the construct manager deals with dependency when sections and constructs are added to and removed from a specification. Files are added to and removed from specifications directly through the GUI.

When we add a file we create a new instance of `FrogFile` associated with the logical file (through a string representation of its file system location) and add it to the current specification. The constructs and sections belonging to that file are not added until it is submitted for syntax checking. When we remove a file from a specification we will calculate any dependencies (using the method described in section 7.4.3) and if necessary warn the user that they are removing a file on which other objects depend.

Files that contain toolkits are handled slightly differently to standard files. When a new specification is created, the user is given the option to include some or all of the toolkits that belong to Z 's standard mathematical toolkit. Logic in the GUI ensures that if a toolkit is selected, then so are the toolkits on which it depends. The toolkits are \LaTeX files like any other Z specification and can be stored anywhere within a file system (by default a set of the toolkits defined in [ISO02] is maintained within Frog's directory structure). When the user selects which toolkits they require, these files are retrieved from the file system and parsed in the standard fashion. However, instead of `FrogFile` instances these files are created as instances of the `FrogToolkit` class (a subclass of `FrogFile`). Instances of the `FrogToolkit` class will not be presented to the user in the file browser and hence will be unavailable for editing. If a user loads a toolkit at a later stage we are unable to determine this property and so it will be stored as an instance of `FrogFile`. In the future we will create an additional option that will allow a user to open a file as a toolkit. This means that they would be able to load their own versions of the standard toolkits at a later stage or create application-specific toolkits that are created as instances of `FrogToolkit`.

The construct manager also allows the user to save their specification as a whole. This process uses Java's serialization facility, which allows a developer to encode an entire object and save that encoding to a file. When serializing an object, Java will also store any child objects where those objects belong to classes that implement the `Serializable` interface. This means that when we attempt to serialize an instance of `FrogSpecification` all of the data belonging to that specification, its files, sections and constructs will be saved into a single file (assuming that we have correctly indicated the appropriate classes

as Serializable). Hence, when we re-load a specification we simply need to set the current specification to be the one found in the file and we will be in exactly the same position as when we saved the file. The only additional task is the need to check that the files referred to by the instances of FrogFile still exist (we do not need to worry about instances of FrogToolkit as these files cannot be edited within the tool and hence we retain no connection to the logical file). Presuming these files exist we then need to check that their contents match the contents we have stored with the appropriate instance of FrogFile, and if not, prompt the user to ask which definition they would prefer to use. Note that, when we save a specification we store a copy of the text in the editor window as part of the FrogFile. Therefore, there may exist changes to the definition of a FrogFile that were saved as part of the specification, but that have not been saved to the logical file.

The Dependency Browser

An additional component of the GUI is the dependency browser. The dependency browser is created using the JGraph swing component [Ald03]. The JGraph package is available under the Lesser GPL, so we are able to freely distribute it with Frog. JGraph is based on the mathematical theory of networks – graph theory – and the Swing user interface library, which defines the architecture. We create the vertices by examining the constructs and sections (obviously we do not re-examine sections that are associated with constructs) belonging to the current specification. We omit the prelude section as every other section depends upon it, and these dependencies would needlessly complicate our graph. We have an array of colours and these are assigned in turn to each construct type that arises (for these purposes, a Z section is considered a construct type). If the definition of a construct or section in the editor is different to that which was stored when the construct was committed we colour its text red (this is determined at the time of drawing the graph; there is no ‘isChanged’ flag maintained by the construct manager). To create the edges of our graph we examine each of the vertices in turn. For machines and Z sections we simply examine their list of parent

sections. For each parent section we will do one of three things depending upon the state of the relationship. If the parent section is present and we have not flagged it as being re-submitted we create an edge to the vertex representing the parent section (or its associated construct). If the parent section is present, but we have flagged it as being re-submitted, we still create the edge but colour it red. If the parent section is not present then we change the colour of the construct or Z section's box to red to indicate that it has missing dependencies. The process for relationship constructs is exactly the same, except that we only consider the source and target machine. We also give the generated edge a different arrowhead.

Other GUI Components

The GUI also contains three ANTLR generated tree parsers that are used to format the ASTs belonging to sections, paragraphs and constructs. These parsers produce an HTML representation of the AST that produces an approximate rendering of the original \LaTeX specification. The HTML is a subset of standard HTML that is suitable for the standard Swing JEditorPane component.

7.4.2 Configuration

Frog requires two forms of configuration: configuration for a specification and configuration for the tool itself.

Tool Configuration

The tool configuration is performed by passing a filename as an argument to the tool when starting up. The file referenced by this name will contain tool-wide settings and the location of other files that the tool requires when running. Due to its platform independence Java does not keep track of the directory in which a program is installed, so this configuration file is required – for example – to retrieve the location of the image files that form the icons on its GUI's buttons. The locations of the specification-specific configuration files are also found from this tool configuration file. This configuration, like

all of the configuration files in the system is read through the use of an ANTLR parser that is invoked from the appropriate configuration factory. In this instance, the GUI calls the configuration factory upon the start of the tool.

Specification Configuration

When a new specification is created the GUI invokes two further configuration factories. The first is used to fetch the configuration of the constructs. The GUI calls the configuration factory passing the newly created specification and the location of the configuration files for machines and relationships. The configuration factory invokes the Frog-CCL parser that we described in section 5.2 which uses the definitions provided in the configuration files to create instances of the ConstructConfiguration class (as well as instances of the various classes that define the clauses, operation environments and proof obligations of the construct). The parser groups these construct configurations to form a dictionary (that is a mapping from the construct type names to their configurations) and returns this dictionary to the configuration factory. Assuming that the construct configurations are valid (if not, warnings will be created in the GUI), the dictionary is added to the specification, and the configurations contained within are fixed for the life of that specification.

The second configuration factory used when a new specification is created is that used to load the character dictionary. This follows a similar pattern to the construct configuration factory. The GUI calls the configuration factory passing the newly created specification and the location of the configuration file for the character dictionary. The configuration factory invokes an ANTLR parser that process the contents of that file to create instances of the FrogCharacter class described in section 4.6.3. The parser collates these objects to create the character dictionary which it returns to the configuration factory. Assuming that the dictionary has been created correctly (again any errors will be echoed in the GUI), the dictionary is added to the specification and these mappings are then permanently attached to that specification.

The Construct Configuration Browser

The construct configuration browser retrieves the construct configuration dictionary from the current specification. It uses the hashtable defined in that dictionary to create an instance of `JTree`. This tree allows a user to select which construct configuration they wish to view. When the user makes a selection the viewing pane is populated with the information derived from the associated instance of `ConstructConfiguration`. This information is essentially that provided in the configuration file formatted using HTML. For the proof obligations, the AST is presented in tree format using ANTLR's `JTreeASTPanel` class.

7.4.3 Parsing and Syntax Checking

The process of parsing and syntax checking is handled by the parsing factory. However, before we can parse a file that has been re-submitted we first remove the objects defined in that file from the specification the GUI passes to the parsing factory. This removal is handled by the construct manager. The parsing factory only ever introduces objects to a specification. That is, it takes the file and specification passed to it by the GUI, parses the file and adds the objects in that file to the specification. When syntax checking is complete it returns the specification to the GUI, where the construct manager attempts to re-establish dependencies.

Resolving Initial Dependencies

When the GUI receives a request to submit a file for syntax checking, the first step is to determine whether other Frog objects (constructs and sections) are dependent on that file. Obviously, if this is the first time that the file has been submitted there will be no dependencies to resolve, and we proceed straight to the syntax checking.

In the instance that the file has previously been submitted for syntax checking, the construct manager will be maintaining a list of incoming dependencies. This list, that is unique to the file, will keep track of all sections

that cite one of the Frog objects contained in the file. Note that, at a low level all dependencies between objects are of the parent-child variety and are between sections. Dependencies are handled for the sections that correspond to constructs rather than the constructs themselves (obviously these are propagated to the higher level as required). For example, while a relationship construct may appear to relate two machine constructs, we actually have a section with two parent sections (see section 5.3.3). If the list is empty (that is, the contents of the file have no dependencies) the file's frog objects are simply removed from the specification and syntax checking can commence.

When the list is not empty, the construct manager needs to inform all reliant sections that the file is being removed. Note that, the re-submission of a file for syntax checking should really be considered as a removal of the old definition of the file's contents followed by the immediate introduction of the objects created from the new definition. We do not attempt to re-use existing instances of Construct or ZSection paragraphs, but simply discard the old objects and create new instances as part of the syntax checking process. When a section receives notification that an object on which it depends is being removed it deletes the appropriate objects from its list of parents (and where necessary from the relevant machine's parents or relationship's source and target) and adds its name to a locally maintained list of failed dependencies (a user can view the list of failed dependencies in the overview of a Frog object presented in the GUI). We do this at this stage to ensure that there no references within our specification to the constructs being removed from the specification.

Once we have removed references to the objects belonging to a file, we remove the objects themselves from the specification. The list of the file's dependants is maintained by the construct manager and will be retained so that we can attempt to re-establish the dependencies following a successful syntax check. The syntax checking can now proceed without fear of encountering duplicate objects.

Syntax Checking

Once a file's incoming dependencies have been resolved, it is passed along with the current specification (minus any removed file) to the parsing factory where the syntax checking process will proceed. The existing specification is required by the parsing factory so that – for example – parent relationships can be checked, operator's behaviour recalled and specification-specific configuration retrieved. We have attempted to make our parser as generic as possible, but clearly we have made some decisions in our parser design that require our specification to have some specific attributes. For example, we rely on the structure of files, sections and paragraphs that we have outlined and we require a valid character dictionary (see section 4.6.3) to be associated with a specification.

The parsing factory will progress through each of the phases of the syntax checking process, calling the relevant ANTLR generated tools with the appropriate arguments. As we have described the rules that govern the generation of these tools in sections 4.6 and 5.3.1, we will not detail them at this point. However, it is important to note that a side effect of the parsing stage of the syntax checking process is the creation of instances of `Construct`, `ZSection` and `ZParagraph` (or their sub-classes) that are added to the specification object. These objects are created as we parse the definitions in the specification and are used to store information at the appropriate level. Note that the tools completing each phase of the syntax checking process attach any errors raised to the specification itself. Errors are recorded at the paragraph, section (construct) and file level and are attached to the appropriate object. The parsing factory will determine whether errors occur by examining the specification after each phase. If the parsing factory finds errors it will return the specification to the construct manager rather than proceeding to the next phase. The construct manager can similarly determine the presence of errors from the specification and does not rely on any direct message from the parsing factory. If no errors are found the phases of syntax checking are performed in turn until the process is complete. Once syntax checking terminates successfully the parsing factory will simply return the

specification to the construct manager.

Re-establishing Dependencies

Once the syntax checking process has been completed any errors are reported in the GUI by the construct manager. Although a file may contain several, distinct sections, the presence of an error in any of them will prevent the use of all the sections. If no errors have been found, the construct manager will record any dependencies that objects in the file possess. It will then attempt to re-establish dependencies upon objects in the file. In order to re-establish a relationship the sections must have exactly the same name as they did before re-submission. If sections do have the same name then they are added back to the parent sections of the relying sections (and their related constructs). Of course, this re-establishment guarantees nothing about the validity of the relationship. A user may attempt to create proof obligations for a dependent without first re-submitting it for syntax checking, but the results cannot be guaranteed. The fact that the link is tenuous is indicated in the dependency browser.

Java Object View

We should note that one of the side effects of the syntax checking process is the creation of a Java object structure that reflects the shape of the final AST. We generate this structure as part of the type checking phase so that we do not need to process the syntactically transformed AST more than once. The structure that we use is similar to that created by the parsing process in CZT. However, we only ever use the object representation once all transformation is complete, so we define classes solely to represent Z objects that belong to the annotated syntax defined in chapter 10 of [ISO02]. That is, we only create the object view once our specification has been fully parsed. We can illustrate a typical object view by examining a sample section.

```
section mySection parents standard_toolkit
```

<i>mySchema</i>
$a : \mathbb{N}$
$a = 0$

Following translation the schema definition paragraph is represented by the following axiomatic description paragraph.

| [*mySchema* : {[$a : \mathbb{N}$ | $a \in \{number_literal_0\}$]}]

An object view for this paragraph is shown in figure 7.14.

```

ZSectionInheriting — (mySection)
├─ ZParagraphAxiomaticDesc — [mySchema : {[ $a : \mathbb{N}$  |  $a \in \{number\_literal\_0\}$ ]}]
├─ ZExpressionVarConstr — [mySchema : {[ $a : \mathbb{N}$  |  $a \in \{number\_literal\_0\}$ ]}]
│   └─ ZVariable — mySchema
│       └─ ZExpressionSetExtension — {[ $a : \mathbb{N}$  |  $a \in \{number\_literal\_0\}$ ]}
│           └─ ZExpressionSchemaConstr — [ $a : \mathbb{N}$  |  $a \in \{number\_literal\_0\}$ ]
│               └─ ZExpressionVarConstr — [ $a : \mathbb{N}$ ]
│                   └─ ZVariable — a
│                       └─ ZExpressionReference —  $\mathbb{N}$ 
│                           └─ ZPredicateMembership —  $a \in \{number\_literal\_0\}$ 
│                               └─ ZExpressionReference — a
│                                   └─ ZExpressionSetExtension — {number\_literal\_0}
│                                       └─ ZExpressionReference — number\_literal\_0

```

Figure 7.14: Object View for *mySection*

As we create each of these objects, we assign an instance of one of the type classes defined in table 4.4 on page 193. We now have an object view of our typed AST; this can be used alongside or in place of that AST to perform further tasks.

7.4.4 Generating Proof Obligations

The process of generating proof obligations is managed by the proof obligation generation factory. The user selects a construct from the GUI and requests that proof obligations are generated. The construct manager will first examine whether proof obligations already exist for the construct. If so, it will prompt the user to confirm that they wish to discard the existing obligations and create new ones from the current definition of the specification. While it may seem odd that it would be necessary to re-create proof obligations, it is necessary to do so when a relied-upon construct's definition has changed. The construct manager will only pass a construct with no obligations to the proof obligation generation factory. If a user chooses not to discard their existing proof obligations then the process will not proceed.

As with the parsing factory we have attempted to ensure that the proof obligation factory maintains independence from the rest of our implementation. This was one reason that we chose to use the Java object view to create proof obligations rather than using the AST generated by our parser. It is easier for an external tool to recreate our object view than it would be to generate an ANTLR AST that has the same format as that generated by our parsing factory.

In the Frog tool, the Java object view of a specification is only used to create our semantic model. Therefore, we included the code to perform this task directly in the classes that defined the object structure. Upon reflection however, we have come to feel that our choice to include the code for generating the semantic model within these classes was poor. With an implementation of the visitor design pattern we could have maintained a visitor within the proof obligation factory leading to a greater distinction between the structure and the tasks performed upon that structure. We intend to rectify this approach in future versions of the tool.

Creating the Semantic Model

Once any existing proof obligations have been disassociated with a construct, the construct manager passes the construct to the proof obligation generation

factory. The first stage in the generation of the proof obligations for a construct is the creation of its semantic model (any existing semantic model is discarded as the generation relies upon parent sections). This process is performed by the semantic binding sub-factory. The construct's semantic model is created by retrieving its associated Z section and then determining that section's semantic binding environment through implementations of the functions defined in table 6.2 on page 316. The function $S^s()$ was implemented by all classes extending ZSection, $S^d()$ by those extending ZParagraph, $S^p()$ by those extending ZPredicate and $S^e()$ by those extending ZExpression. The function $ZF()$ simply required the invocation of the constructor of the ZF classes outlined in table 6.1 on page 305, and $\tau()$ just required the retrieval of the type objects already associated with each instance of the Z classes. The remaining functions (such as the replacement and querying functions) were implemented as static methods of the semantic binding factory itself.

Instantiating the Proof Obligations

Assuming the semantic model was created correctly, the proof obligation generation factory then passes the construct (with attached semantic model) to the proof obligation instantiation sub-factory. From here, the construct's configuration is retrieved and the list of generic proof obligations established. For each of the generic proof obligations, the proof obligation instantiation sub-factory determines whether the proof obligation operates at the construct or operation environment level. If the proof obligation is defined at the construct level, the factory invokes the ANTLR tree walker upon its AST whereupon it is instantiated by the process described in section 6.3. If the proof obligation is defined at the operation environment level, the factory invokes the ANTLR tree walker upon its AST once for each suitable operation environment belonging to the construct (for example, for every set of matched operation environments in a relationship). The factory indicates to the tree walker that the proof obligation is defined at the operation environment level and specifies which of the operation environments to process. As each proof obligation is created it is attached to the construct in an unproved state.

When all proof obligations have been instantiated (for all suitable operation environments), the construct is passed back to the construct manager with the newly created proof obligations attached. The construct manager then instructs the GUI to update its view of the construct to reflect this addition.

7.4.5 Interfacing with Theorem Provers

All interaction with external theorem provers is handled through the proof factory. The only inputs to the proof factory are proof obligations – that is, instances of our `ProofObligation` class. These proof obligations are passed to the proof factory by the construct manager when the user selects the obligations they require from the GUI. Whilst we only use our proof factory to translate and submit our construct’s proof obligations, the tool would be equally well equipped to process any instance of the `ProofObligation` class that incorporated hypotheses and goals in our ZF language.

The proof factory processes each of the obligations it has received in turn. The factory has its own GUI that it launches for every proof obligation. This GUI allows the user to select one or more theorem provers with which to attempt a discharge of the proof obligation (note again that Frog does not restrict a user to a single theorem prover for a development, assuming that the user is aware of the consequences resulting from using multiple theorem provers). Once the user has selected their required theorem provers, the proof obligation is passed to the factory that processes theorems for the appropriate prover. At present only the Isabelle/ZF factory has been implemented.

Translating the Proof Obligations

The proof factory passes the proof obligation to the required theorem prover’s factory. The `ProofObligation` class contains a list of hypotheses and a statement that are instances of the aforementioned ZF language classes. The first stage in the preparation of the proof obligation for submission to the theorem prover is the translation into the required syntax. This process is described for the Isabelle/ZF theorem prover in section 6.5.1. Each theorem prover’s

factory will have its own GUI that allows a user to select options that are specific to that prover. In the case of Isabelle/ZF this allows the user to select one of the automatic tactics or use their own tactic file. Once a theory file has been generated, and it is ready for submission to the theorem prover, we attach an object to the ProofObligation instance that provides a representation of the theorem-prover specific proof obligation. For example, when using Isabelle/ZF we create an instance of ProofObligationIsabelleZF that contains the text of the theory file and maintains the status of the obligation contained within (that is, whether it has been successfully proved or not). This class will also store any output resulting from the theory's submission to the prover.

Interacting with Theorem Provers

Once the translation process is complete, the object containing the theorem-prover specific proof obligation is processed by the theorem prover's factory. For Isabelle/ZF this process involves creating a directory on the file system which is then populated with a number of files. The creation of the directory is performed with an Isabelle-specific tool that automatically creates files required for Isabelle/ZF to run effectively. Frog must also create two files to instruct Isabelle/ZF how to proceed. The first of these files is the theory file, the contents of which are retrieved from the ProofObligationIsabelleZF instance. The second is an ML file that instructs Isabelle/ZF how to process the directory – here we simply require an attempt to establish our theorem. Once these files have been created, the factory instructs Isabelle/ZF to process the directory and records any output. When Isabelle is complete the factory examines the return code to determine whether the proof was successful, storing the result (and any additional output) within the instance of ProofObligationIsabelleZF. If the attempt was unsuccessful the factory will offer to open any interactive proof assistant – in this instance Proof General – and the user can attempt the proof directly through that tool.

Once all proof attempts are complete the proof obligation is returned to the proof factory, which then continues to process any remaining proof

obligations. When all proof obligations have been processed by all the required theorem provers, they are returned to the construct manager. The construct manager then instructs the GUI to refresh its view of the proof obligations whereby the success or failure of the process (and any output from the theorem provers) is presented in the GUI.

7.4.6 Error Handling

Unfortunately, the handling of errors is a weak spot in the current Frog implementation. The problem resides principally in dealing with errors that occur during type checking. These troubles are a result of the syntactic transformation of a specification. By the time a specification reaches the type checking phase the AST presented bears little resemblance to the original L^AT_EX representation. When we are unable to type a variable there exists the possibility that we can retroactively calculate the position of that variable in the specification. With expressions and predicates however, this becomes more complex. For example, consider the following Z paragraph.

<i>badlyTypedSchema</i>
$a : \mathbb{N}$
$b : \textit{trains}$
$a = b$

Syntax transformation converts this specification to the following axiomatic description paragraph.

$$| \textit{badlyTypedSchema} : \{ \{ [a : \mathbb{N}] \wedge [b : \textit{trains}] \mid a \in \{b\} \} \}$$

The type checking process progresses well until we come to apply our rules to the membership predicate, $a \in \{b\}$. When we try to apply the rules for typing a membership predicate we realize that the types of the component expressions are incompatible: a has the type GIVEN \mathbb{A} and $\{b\}$ the type $\mathbb{P}(\text{GIVEN } \textit{trains})$. We now know there is an error, but how do we inform the

user where this occurred in the original specification? We are not even able to determine whether the membership predicate was originally a membership predicate, an equality relation or the result of a more complex transformation.

Frog's current approach is to try and provide a back trace so that the user can identify where in a specification their problem occurs. Understanding this trace will require an examination of the transformed AST. For example, Frog produces the following error messages for the above schema.

```
badlyTypedSchema.z, badlyTypedSchema:
    Types in membership predicate are inconsistent
badlyTypedSchema.z, badlyTypedSchema:
    - GIVEN \arithmos
badlyTypedSchema.z, badlyTypedSchema:
    - GIVEN trains
badlyTypedSchema.z, badlyTypedSchema:
    Predicate cannot be typed in schema construction
badlyTypedSchema.z, badlyTypedSchema:
    Element could not be typed in set extension
badlyTypedSchema.z, badlyTypedSchema:
    Set cannot be typed in variable construction
badlyTypedSchema.z, badlyTypedSchema:
    Expression cannot be typed in axiomatic description paragraph
```

By examining the error messages from bottom to top the user can identify the paragraph, then work through the tree by finding the appropriate expressions or predicates until they discover the typing error.

Obviously, this is not the ideal solution. In order to report errors correctly we would need some way of associating an expression with its original definition in the \LaTeX source. Unfortunately, as our resources have been limited we have not been able to devote the time required to devise such a system. Whilst not ideal we feel that our current system gives a user enough information to be able to identify their error, and we don't imagine that the problem will significantly reduce the usability of the tool.

Errors produced in the lexing and parsing phases are reported in the expected way. The error message is associated with the appropriate Frog object and the position of the offending token in the input file recorded. Similarly,

errors arising during the processing of configuration files, or in other input and output to the file system, are reported via the GUI in a conventional manner. The majority of other processes that are used by Frog do not lead to predictable errors. For example, if an error occurs during syntax transformation or proof obligation generation it will be due to a problem beyond the tool's control – whether this be due to a bug or a problem with the underlying system.

7.5 Concluding Remarks

In this chapter we have discussed the architecture and implementation of Frog. Frog is a tool that can be used to formally describe machines and the relationships between them. Machines and relationships are configured using Frog-CCL and specified with a combination of the Z notation and a Frog meta-language. Frog has been designed to be flexible and extensible. Its architecture breaks the tool into distinct components that handle their individual tasks independently. These components handle configuration, construct management, parsing of specifications, generation of proof obligations and interfacing with theorem provers. Frog also has a number of minor features that further aid a user in creating their specifications. The majority of the tool has been implemented in Java and ANTLR, and the tool's developers have endeavoured to apply solid design principles throughout. Whilst the tool is able to perform its core tasks effectively and efficiently, there remains much opportunity for extensions and evolution. One of the overriding considerations when implementing the tool was to allow this to occur with the minimum of fuss.

Chapter 8

Case Study

This chapter describes the mechanization of an existing retrenchment case study. Banach and Poppleton [BP02] used a telephone system to illustrate how retrenchment could be used effectively as a specification constructor. Beginning with a specification of a vanilla system, they showed how retrenchment could be used to formally relate this specification to others incorporating additional features. Furthermore, they showed how to create an integrated specification that could be considered a retrenchment of all the previous specifications. In the following sections we show how Frog could be used to provide mechanized support to this process. We begin by introducing the telephone system and describe the relationship between our work and that of Banach and Poppleton. We then consider each of the models and relationships in turn. We demonstrate the support that Frog gives, from the syntax checking of the specification, to the generation of proof obligations suitable for input to the Isabelle/ZF theorem prover.

8.1 Introduction

This case study describes the construction of a specification for a telephone system. We will begin by specifying a system that simply allows its users to make calls; this system is known as POTS – the Plain Old Telephone Service. We will then gradually introduce some additional features. We will show how the construction of this specification can be performed mechanically with Frog. An overview of the specification’s construction can be seen in figure 8.1.

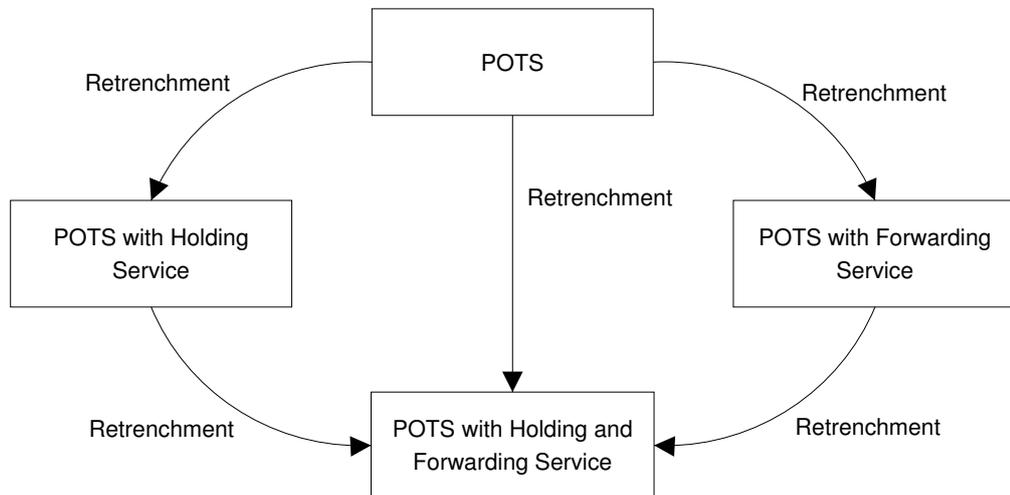


Figure 8.1: Overview of telephone system requirements

The case study is based on previous work that was presented in [BP02]. The specification presented in Banach and Poppleton’s paper was produced by hand and defined in a Z-like notation. In our mechanization, we shall attempt to retain identical behaviour to the specification presented there. As we are using ISO Z and Frog’s construct-based approach there will be notational differences. Some of these differences have led to small changes to some of the specification’s minor details. However, these changes have not affected the meaning of the models or relationships specified and should not distract – or even be apparent to – the reader unless making low level comparisons. Additionally there were instances where the behaviour defined in the original specification is not what a reader may immediately expect.

We have retained faith with the meaning of the original specification, seeking to preserve the ability of the reader to make a direct comparison between the two case studies. Footnotes have been inserted to explain these quirks so as not to confuse the reader.

The first step in the modelling of the phone call system is the specification of the basic requirements of the phone. That is, the ability to connect to, and disconnect from, other phones in the system. When one of the system's users attempts to make a phone call in this basic machine, the call will be successful only if the target phone is not already connected to another phone.

This basic machine will then be augmented. We will create a second machine that will incorporate an additional requirement of the telephone system that will allow a phone to be (optionally) registered for a holding service. This holding service will alter behaviour when an attempt is made to connect to a phone that is already involved in a connection. In this model when a connection is sought to a busy phone that has been registered for the holding service, the calling phone will be instructed to wait until the target phone becomes free. We do not seek to model the behaviour when the phone becomes free, we just incorporate the concept that an attempt to make a call can lead to three results: success, failure or holding. We will then create a retrenchment relationship that can illustrate the relationship between the basic model of the telephone system and that enhanced with the holding system. We will show how Frog is able to verify that the relationship is syntactically correct through syntax checking. Frog will then be used to generate proof obligations that will show the relationship is semantically sound.

A third machine will then be described. This machine will show that instead of the holding service, the telephone could be enhanced through the use of a forwarding service. The option of call forwarding is added to our telephone system and again leads to different behaviour when an attempt is made to connect to a phone that is already involved in a connection. In this instance, if an attempt is made to connect to a busy phone, and that phone is registered for the forwarding service, then the system will retrieve the registered forwarding number and attempt to make a connection with the

number's associated phone. Again, we will then show how a retrenchment relationship can be used to relate the model of the basic telephone system and the system that incorporates the forwarding service.

Finally, a machine that integrates both of the services will be presented. This machine will specify the behaviours required for the two services and also detail how their intersection is handled (that is, where a user of the telephone system is registered for both services). We will then present retrenchment relationships that will show how this machine relates to the two single-service machines and also how it relates to the model of the basic telephone system. A diagram illustrating the constructs required to produce our specification is seen in figure 8.2 on the next page.

Throughout the case study we will use the example machine and retrenchment configurations that we defined in examples 5.4 and 5.6 respectively. In fact, these configurations form an integral part of the case study. The distinguishing feature of Frog is its ability to combine configurations and specifications to provide flexible, mechanical support to the construction of a specification. Hence a specification is meaningless without its associated configuration.

8.2 Types and Operators

Before specifying the constructs that form our specification we will define some types and operators for use throughout. Below we define a section that will become a common ancestor of all our constructs.

```
section telephoneSection parents standard_toolkit
```

We define a section, *telephone_section*, that inherits all of the usual behaviour from the standard toolkit (which provides an interface to the entire Z mathematical toolkit).

```
[NUM]
```

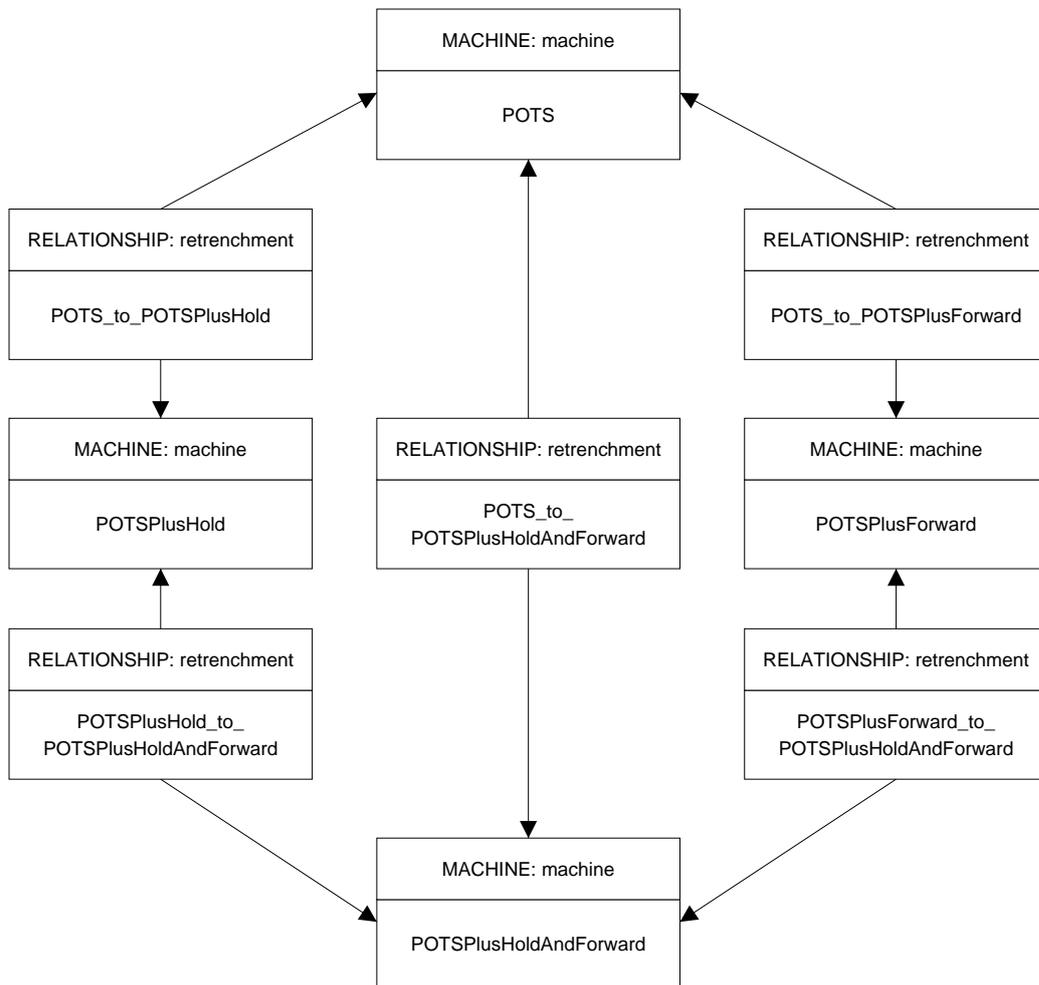


Figure 8.2: Overview of telephone system constructs

We then begin our specification by defining a given type paragraph which declares a single type, *NUM*. This type will represent the set of all possible phone numbers.

$$CALL_SUCCESS ::= successful \mid failed \mid held \mid optional_forward$$

We then declare a free type paragraph which allows the definition of an enumerated set, *CALL_SUCCESS*. This set is used to represent the status report given when trying to connect a call. The meanings of the possible statuses are given in table 8.1 on the following page.

Table 8.1: Possible status reports when connecting a call

Status	Definition
<i>successful</i>	A connection is made.
<i>failed</i>	No connection is made.
<i>held</i>	No connection is made, but the caller is given a hold message.
<i>optional_forward</i>	No connection is made, but the caller is given a message indicating that they could choose to be forwarded to an alternative number.

Having defined the necessary types, we declare a number of operators that will abbreviate our construct specifications. In [BP02] these operators are functions, $free(n)$ and $busy(n)$ that refer directly to the state of the telephone system (that is, the $calls$ variable). Here we define the operators as infix relations that will determine whether a number is free in any partial injection (see C.9) over the number type.

relation($_isFreeIn _$)

$$\frac{_isFreeIn _ : NUM \leftrightarrow (NUM \rightsquigarrow NUM)}{\forall x : NUM; y : NUM \rightsquigarrow NUM} \bullet x \text{ isFreeIn } y \Leftrightarrow x \notin \text{dom } y \wedge x \notin \text{ran } y$$

Firstly, we define a relation, $isFreeIn$. This operator will allow us to test whether a phone (identified by its phone numbers) is currently in the process of making a call. For a given phone number and set of currently active calls, the result of the application of the $isFreeIn$ operator will be true if the given phone number appears in neither the domain (see C.2) nor range (see C.11) of that set of currently active calls.

relation($_ \text{isBusyIn } _$)

$$\left| \begin{array}{l} _ \text{isBusyIn } _ : NUM \leftrightarrow (NUM \rightsquigarrow NUM) \\ \hline \forall x : NUM; y : NUM \rightsquigarrow NUM \\ \bullet x \text{ isBusyIn } y \Leftrightarrow \neg (x \text{ isFreeIn } y) \end{array} \right.$$

The `isBusyIn` operator is simply the negation of the `isFreeIn` operator. That is, for a given phone number and the set of currently active calls, the result of the application of the `isBusyIn` operator will be true if the given phone number is not free in the set of currently active calls.

8.3 The Plain Old Telephone Service

Now that we have defined our parent section, we begin the specification of the constructs with the definition of the machine *POTS*. This machine will define our most basic model of the telephone system, with neither call holding nor call forwarding service defined. The definition of our machine is given below.

MACHINE *POTS*
 TYPE *machine*
 SECTION *telephoneSection*
 STATE

$$\begin{array}{l} \text{calls} : NUM \rightsquigarrow NUM \\ \hline \text{dom calls} \cap \text{ran calls} = \emptyset \end{array}$$

INITIALIZATION

$$\begin{array}{l} \hline \text{calls} = \emptyset \end{array}$$

OPERATION *connect* $\hat{=}$

INPUTS

$source? : NUM$
 $target? : NUM$

OUTPUTS

$connection! : CALL_SUCCESS$

PRE

—————
 $source? \text{ isFreeIn } calls$

POST

—————
 $(target? \text{ isFreeIn } calls \wedge source? \neq target?)$
 $\Rightarrow (calls' = calls \cup \{source? \mapsto target?\})$
 $\wedge connection! = successful)$

$(target? \text{ isBusyIn } calls \vee source? = target?)$
 $\Rightarrow (calls' = calls \wedge connection! = failed)$

END *connect*

OPERATION *disconnect* $\hat{=}$

INPUTS

$source? : NUM$

PRE

—————
 $source? \text{ isBusyIn } calls$

POST

—————
 $calls' = \{source?\} \triangleleft calls \triangleright \{source?\}$

END *disconnect*

END *POTS*

The machine defines a single state variable, *calls*, which represents all the current calls in the system and is a partial injection between phone numbers. A phone can only ever be involved in one call at any time so if it appears in *calls* as a caller, it cannot also appear as a callee, and vice versa. The set of current calls is initialized to the emptyset. Two operations are defined for the machine; to connect, and disconnect a call.

The first operation, *connect*, models the behaviour of the system when connecting a call. There are two inputs to the operation: the phone numbers of caller and callee respectively. The operation has a single output, the possible values for this output were described in table 8.1. In the precondition of the operation we state the following: the calling phone must be free in order to make a call. We then define the behaviour of the operation: if the callee is free – and we have not tried to call our own phone – the call will be connected and success will be reported, if the callee is busy the call will not be connected and failure will be reported.

Note that, the model we define is fairly primitive, we – and the original case study – do not seek to specify all situations or behaviour. For instance, we have not covered the situation where a caller is trying to make a call to someone who has picked up the phone, but has not yet been connected. In the real world we may expect a failed call, in our model a target is only busy if they are actually connected. That is target number is only busy if it has formed a connection to another number, at all other times it is free. In our basic model this means that when a caller tries to make a call to a busy number, their own number is free throughout.

The second operation, *disconnect*, models the behaviour of the system when disconnecting a call. There is a single input to the operation, this is the number of the phone which is disconnecting the call (note that the caller or callee may terminate the connection and that *source?* in this instance refers to the source of the termination, rather than source of the call). In the

body of the operation we state the following: the terminating phone must be currently connected, and any call involving the terminating phone will be disconnected.

8.4 Introducing the Holding Service

We now consider the introduction of the holding service. The holding service allows a subscriber to ensure that they do not miss important calls. When someone makes a call to the subscriber, and the subscriber's phone is busy, the caller will receive a message informing them that the subscriber is currently busy, but if they continue to hold they will be connected as soon as possible. In this example we model only the passing of the hold message, we do not consider the modelling of the connection once the holding period expires.

8.4.1 Specifying the Extended Machine

The machine *POTSPlusHold*, defined below shows how the original phone system machine (defined in section 8.3 above) can be extended to incorporate the required behaviour for this service.

MACHINE *POTSPlusHold*
 TYPE *machine*
 SECTION *POTS*
 STATE

POTS.state
holdingTable : \mathbb{P} NUM

INITIALIZATION

POTS.initialization

holdingTable = \emptyset

OPERATION *connect* $\hat{=}$

INPUTS

 $source? : NUM$ $target? : NUM$

OUTPUTS

 $connection! : CALL_SUCCESS$

PRE

 $source? \text{ isFreeIn } calls$

POST

 $(target? \text{ isFreeIn } calls \wedge source? \neq target?)$
 $\Rightarrow (calls' = calls \cup \{source? \mapsto target?\})$
 $\wedge connection! = successful)$ $((target? \text{ isBusyIn } calls \vee source? = target?) \wedge target? \in holdingTable)$
 $\Rightarrow (calls' = calls \wedge connection! = held)$ $((target? \text{ isBusyIn } calls \vee source? = target?) \wedge target? \notin holdingTable)$
 $\Rightarrow (calls' = calls \wedge connection! = failed)$ $holdingTable' = holdingTable$ END *connect*OPERATION *disconnect* $\hat{=}$

INPUTS

 $source? : NUM$

PRE

 $POTS.disconnect.pre$

POST

POTS.disconnect.post

holdingTable' = holdingTable

END *disconnect*

OPERATION *registerForHold* $\hat{=}$

INPUTS

regNumber? : NUM

PRE

regNumber? \notin holdingTable

POST

holdingTable' = holdingTable \cup {regNumber?}

calls' = calls

END *registerForHold*

OPERATION *cancelHold* $\hat{=}$

INPUTS

regNumber? : NUM

PRE

regNumber? \in holdingTable

POST

$$\textit{holdingTable}' = \textit{holdingTable} \setminus \{\textit{regNumber}?\}$$

$$\textit{calls}' = \textit{calls}$$

END *cancelHold*

END *POTSPlusHold*

Note that we define the parent section of this machine to be that corresponding to our previous machine (*POTS*). This enables us to easily re-use behaviour which has not changed.

The *POTSPlusHold* machine inherits the state of the *POTS* machine (that is, the set of current calls) and defines an additional state variable that enables us to track the numbers of those phones that have been registered for the holding service. Our inherited variable is initialized as before, and the set of phones numbers registered for the holding service is initialized as the emptyset. Four operations are declared: the connect and disconnect operations, and two further operations that allow a phone to register for, or cancel, the holding service.

The first operation, *connect*, declares the same inputs and outputs as before. In the precondition of the operation we state the following: the calling phone must be free in order to make a call. In the postcondition we specify that if the callee is free – and we have not tried to call our own phone – the call will be connected and success reported, if the callee is busy and has registered for the holding service, the call will not be connected but a *held* status will be reported¹, and if the callee is busy and has not registered for the holding service, the call will not be connected and the failure will be reported.

Note again, the behaviour defined in our operation – and that of the original case study – could be considered counter-intuitive as a caller is still

¹Note that, in this trivial example, we do not model the held state nor its resolution.

only busy when they make a successful call. That is, if a caller tries to make a call to a number that is busy but has the holding service – and so is held – they themselves are still considered to be free.

The second operation, *disconnect*, has practically identical behaviour to the operation in the *POTS* machine, the only difference being the necessity to note the lack of change in the state of the holding table.

The third operation, *registerForHold*, allows a phone number to be registered for the holding service. It takes a single input that should be the number of the phone to be registered for the service. The precondition of the operation states that: the phone must not already be registered for the holding service, and the postcondition that the phone's number is added to the set of numbers registered for the holding service.

The final operation, *cancelHold*, allows a phone to cancel its subscription to the holding service. Again, the operation takes a single input, and this should be the number of the phone for which the holding service is to be cancelled. In the precondition of the operation we state the following: the phone must always be registered for the holding service, and in the postcondition that the phone's number is removed from the set of numbers registered for the holding service.

8.4.2 Specifying the Relationship

We now consider the specification of the retrenchment relationship between our two models (that is, the model describing the plain old telephone service and the model in which the holding service is introduced). This relationship will allow us to prove that the models behave identically, except in circumstances made clear in the specification of the retrenchment. The definition of the relationship is shown below.²

RELATIONSHIP *POTS_to_POTSplusHold*
 TYPE *retrenchment*

²This is an instance where we believe an oversight was made in the original case study. When specifying the concedes relation for this relationship, no consideration was given to the case where source and target number are identical. As we have previously specified, we will follow the original case study despite this oversight. The original relation was defined

FROM *POTS*
 TO *POTSPlusHold*
 RETRIEVE

$$POTS \gg \text{calls} = POTSPPlusHold \gg \text{calls}$$

RAMIFICATIONS *connect* $\hat{=}$
 WITHIN

$$POTS \gg \text{source?} = POTSPPlusHold \gg \text{source?}$$

$$POTS \gg \text{target?} = POTSPPlusHold \gg \text{target?}$$

OUTPUT

$$POTS \gg \text{connection!} = POTSPPlusHold \gg \text{connection!}$$

CONCEDES

$$POTSPPlusHold \gg \text{target?} \text{ isBusyIn } POTSPPlusHold \gg \text{calls}$$

$$POTSPPlusHold \gg \text{target?} \in POTSPPlusHold \gg \text{holdingTable}$$

$$POTS \gg \text{calls} = POTSPPlusHold \gg \text{calls}'$$

$$POTSPPlusHold \gg \text{calls} = POTSPPlusHold \gg \text{calls}'$$

$$POTS \gg \text{connection!} = \text{failed}$$

$$POTSPPlusHold \gg \text{connection!} = \text{held}$$

END *connect*

RAMIFICATIONS *disconnect* $\hat{=}$
 WITHIN

$$POTS \gg \text{source?} = POTSPPlusHold \gg \text{source?}$$

END *disconnect*

END *POTS_to_POTSPPlusHold*

as follows.

$$C_{CH,connect_n}(u', v', o, p; i, j, u, v) =$$

$$(\text{busy}(j) \wedge j \in \text{holtab} \wedge u' = u \wedge v' = v \wedge o = NO \wedge p = (\text{"Our...hold."})^{100})$$

Comparing the definition for this relation to the definition of the concedes clause in our relationship, we can see that the only differences are notational.

In the retrieve clause we specify that the set of current calls in each machine should be identical. We then consider any exceptions to this general rule by specifying the ramifications for each operation. No ramifications are required for the operations concerning the administration of the set of numbers registered for the holding service; these operations have no equivalent in the model of the plain old telephone service.

We consider first, the *connect* operation. The within clause specifies that the inputs to the operation in either machine must be identical. The output clause specifies that in normal circumstances the outputs of the operation in both machines will also be identical. The concedes clause must therefore, handle those situations in which either the set of current calls in each machine, or the outputs from the operation of each machine can allowably differ. For the *connect* operation of these two machines, there is no situation in which the set of current calls in each machine may differ. The operations' outputs differ, however, when an attempt is made to make a call to a busy phone which has also been registered for the holding service. The concedes clause therefore, details this situation, ensures that the set of current calls is still identical in each machine, and specifies the allowable values for the output of the operation in either machine.

The *disconnect* operation is much less complex as the behaviour is identical in both machines. As we have no guarantee that the inputs to the operation of the two machines will be identical however, we must specify this in the within clause.

8.4.3 Syntax Checking the Relationship

In Frog we created a new specification for our telephone system. We then added the file containing our general section (defined in section 8.2) and submitted it for syntax checking. We then imported the files containing the specifications for the vanilla machine and the version extended with the holding service. These machines were submitted for syntax checking and any errors resolved. Once our machines were syntactically correct we added the

file containing the relationship we have specified in the previous section. We will now analyse the results produced at various stages of this process.

Note that, in order to present these results within this thesis we created some simple ANTLR tree walkers that were able to produce a \LaTeX representation of the abstract syntax trees produced by the various phases of the syntax checking process. These tree walkers were produced in a ‘quick-and-dirty’ manner specifically for this case study (although the functionality remains in the tool). As such they were only capable of generating a string of tokens, and the formatting of the results was performed manually. Nevertheless the representations in the following sections should be considered faithful to the model possessed by the tool; the only alterations being the addition of white space.

Translating to the Extended Z Syntax

The first phase of the syntax checking process involves the transformation of constructs specified in the extended Z notation into the standard Z notation. Although this transformation occurs in the syntax transformation phase, Frog also produces a version of the transformed in the standard Z language (rather than in a language conforming to the annotated syntax). This is done to allow a user to ensure that their constructs are being translated into Z in the way that they expect. When creating new construct configurations it is useful to be able to directly compare the extended Z and standard Z specifications.

```
section _relationship_POTS_to_POTSPlusHold
parents _machine_POTS, _machine_POTSPlusHold
```

In this instance a new section is created that has two parents: the sections that are produced from the source and target machine of the relationship.

<pre> _POTS_to_POTSPlusHold_retrieve _POTS_state _POTSPlusHold_state (POTS>>calls = POTSPPlusHold>>calls) </pre>
--

A schema definition paragraph is then declared for each clause within the relationship. The declaration part of the schema text includes references to the schemas of the source and target machines that have been automatically imported through the references in the clause's relation attribute.

The \gg notation allows us to distinguish between the variables of source and target machine and the appropriate dereference will be automatically prepended to any variables imported from the referenced schemas. Note that, although the state of *POTSPlusHold* refers directly to that of *POTS*, making reference to the clause of another machine will result in a *copy* of that clause's schema being used. Hence, *_POTS_state* and *_POTSPlusHold_state* refer to distinct schemas, and the reader should not be worried that *POTS>>calls* and *POTSPlusHold>>calls* are actually references to the same variable.

<pre> _POTS_to_POTSPlusHold_connect_within _POTS_state _POTSPlusHold_state _POTS_connect_inputs _POTSPlusHold_connect_inputs (POTS>>source? = POTSPPlusHold>>source?) (POTS>>target? = POTSPPlusHold>>target?) </pre>

```

_POTS_to_POTSPlusHold_connect_output
_POTS_state
_POTSPlusHold_state
_POTS_state'
_POTSPlusHold_state'
_POTS_connect_inputs
_POTSPlusHold_connect_inputs
_POTS_connect_outputs
_POTSPlusHold_connect_outputs
-----
(POTS»connection! = POTSPlusHold»connection!)

```

```

_POTS_to_POTSPlusHold_connect_concedes
_POTS_state
_POTSPlusHold_state
_POTS_state'
_POTSPlusHold_state'
_POTS_connect_inputs
_POTSPlusHold_connect_inputs
_POTS_connect_outputs
_POTSPlusHold_connect_outputs
-----
(POTSPlusHold»target? isBusyIn POTSPlusHold»calls)
(POTSPlusHold»target? ∈ POTSPlusHold»holdingTable)
(POTS»calls = POTS»calls')
(POTSPlusHold»calls = POTSPlusHold»calls')
(POTS»connection! = failed)
(POTSPlusHold»connection! = held)

```

```

_POTS_to_POTSPlusHold_disconnect_within
_POTS_state
_POTSPlusHold_state
_POTS_disconnect_inputs
_POTSPlusHold_disconnect_inputs
-----
(POTS»source? = POTSPlusHold»source?)

```

We can see that the syntax of each paragraph is practically identical to that of the appropriate clause, the only difference being the specification of the appropriate clauses, and the introduction of brackets that Frog uses to explicitly indicate the associativity and precedence of operators. We can clearly see that each paragraph is a faithful translation of its clause.

Transforming to the Annotated Syntax

The next stage in the syntax checking process is the conversion into the annotated Z syntax. As we mentioned above, in practice this happens in conjunction with the transformation to the standard Z syntax.

```
section _relationship_POTS_to_POTSPlusHold
parents _machine_POTS, _machine_POTSPlusHold
```

There is no change to the section definition, but we can begin to see the results of the transformation in the description of the paragraphs; most notably in their progression from schema definition paragraphs to axiomatic description paragraphs. Note that, Frog's formatting tools are enthusiastic in the introduction of parentheses to effectively communicate its application of associativity and precedence. Although in practice this may not be necessary, it is a result of the translation from the AST where the results of the resolution of associativity and precedence are implicit.

```
AX
[_POTS_to_POTSPlusHold_retrieve :
{ [_POTSPlusHold_state  $\wedge$  _POTS_state |
(POTS  $\gg$  calls)  $\in$  ({POTSPlusHold  $\gg$  calls})}]]
END
```

AX

$[_{POTS_to_POTSPlusHold_connect_within} :$
 $\{[_{POTSPlusHold_connect_inputs} \wedge _{POTS_connect_inputs}$
 $\wedge _{POTSPlusHold_state} \wedge _{POTS_state} |$
 $((POTS \gg source?) \in (\{POTSPlusHold \gg source?\}))$
 $\wedge ((POTS \gg target?) \in (\{POTSPlusHold \gg target?\}))\}]$
 END

AX

$[_{POTS_to_POTSPlusHold_connect_output} :$
 $\{[_{POTSPlusHold_connect_outputs} \wedge _{POTS_connect_outputs}$
 $\wedge _{POTSPlusHold_connect_inputs} \wedge _{POTS_connect_inputs}$
 $\wedge _{POTSPlusHold_state'} \wedge _{POTS_state'}$
 $\wedge _{POTSPlusHold_state} \wedge _{POTS_state} |$
 $(POTS \gg connection!) \in (\{POTSPlusHold \gg connection!\})\}]$
 END

AX

$[_{POTS_to_POTSPlusHold_connect_concedes} :$
 $\{[_{POTSPlusHold_connect_outputs} \wedge _{POTS_connect_outputs}$
 $\wedge _{POTSPlusHold_connect_inputs} \wedge _{POTS_connect_inputs}$
 $\wedge _{POTSPlusHold_state'} \wedge _{POTS_state'}$
 $\wedge _{POTSPlusHold_state} \wedge _{POTS_state} |$
 $(((((POTSPlusHold \gg target?, POTSPlusHold \gg calls)) \in (\boxtimes isBusyIn \boxtimes)))$
 $\wedge ((POTSPlusHold \gg target?) \in (POTSPlusHold \gg holdingTable)))$
 $\wedge ((POTS \gg calls) \in (\{POTS \gg calls'\}))$
 $\wedge ((POTSPlusHold \gg calls) \in (\{POTSPlusHold \gg calls'\}))$
 $\wedge ((POTS \gg connection!) \in (\{failed\}))$
 $\wedge ((POTSPlusHold \gg connection!) \in (\{held\}))\}]$
 END

AX

$[_{POTS_to_POTSPlusHold_disconnect_within} :$
 $\{[_{POTSPlusHold_disconnect_inputs} \wedge _{POTS_disconnect_inputs}$
 $\wedge _{POTSPlusHold_state} \wedge _{POTS_state} |$
 $(POTS \gg source?) \in (\{POTSPlusHold \gg source?\})\}]$
 END

The transformations in this instance are relatively minor, and an observer can quickly establish that the new specification is a faithful translation of its predecessor and conforms to the annotated Z syntax specified in chapter 10 of [ISO02].

Type Checking

The final stage of the syntax checking process is the type checking phase, where the paragraphs and expressions of the transformed specification are decorated with the relevant type. Where a variable, expression or paragraph cannot be typed, an error is generated. Mechanical type checking typically uncovers numerous typing errors upon the initial submission of a specification. This is particularly true when specifying relationships where users (particularly the author) frequently forget to disambiguate variables with the same name in source and target machines.

```
section _relationship_POTS_to_POTSPlusHold
parents _machine_POTS, _machine_POTSPlusHold
```

Again, there is no change to the declaration of the section, as sections are not assigned a type.

```
AX ((([_POTS_to_POTSPlusHold_retrieve : (({(((
  ((_POTSPlusHold_state)°P([calls : P((GIVEN NUM) × (GIVEN NUM));
    holdingTable : P(GIVEN NUM)))))
  ∧ ((_POTS_state)°P([calls : P((GIVEN NUM) × (GIVEN NUM))]))
  )°P([POTSPlusHold≫calls : P((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫holdingTable : P(GIVEN NUM);
    POTS≫calls : P((GIVEN NUM) × (GIVEN NUM))]))))
|
```

```

(((POTS>>calls)°P((GIVEN NUM) × (GIVEN NUM)))
 ∈ ((({((POTSPlusHold>>calls)°P((GIVEN NUM) × (GIVEN NUM)))
 }°P(P((GIVEN NUM) × (GIVEN NUM))))))
 ]°P([POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
      POTSPlusHold>>holdingTable : P(GIVEN NUM);
      POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))]))
 }°P(P([POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
        POTSPlusHold>>holdingTable : P(GIVEN NUM);
        POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))]))))
 ]°P([_POTS_to_POTSPlusHold_retrieve :
      P([POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
        POTSPlusHold>>holdingTable : P(GIVEN NUM);
        POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))]))))
 END
 °[_POTS_to_POTSPlusHold_retrieve :
   P([POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
      POTSPlusHold>>holdingTable : P(GIVEN NUM);
      POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))]))]

```

The changes become apparent in the type checking of the paragraph that corresponds to the retrieve clause of our relationship. The first thing that we notice is that the size of the paragraph's definition has greatly increased. This is due to the appending of a type to every expression within the paragraph (and the paragraph itself). The other change is the expansion of the schema references, which has led to the explicit declaration of the two machine's state variables. As we mentioned previously this expansion also involves disambiguating between identically named variables. A close examination of the paragraph and its expressions can allow us to determine that it has been typed correctly and hence the clause is accepted as syntactically correct by Frog.

As the complexity of the clauses – and the number of imported schemas – increases, the more apparent the growth of the specification. The typed version of the specification for the within clause, below, is considerably larger than that of the untyped version. Again a close examination of the paragraph allows us to determine that the typing is consistent and Frog accepts the clause as syntactically correct.


```

} )% P(P([POTSPlusHold>>source? : GIVEN NUM;
          POTSPlusHold>>target? : GIVEN NUM;
          POTS>>source? : GIVEN NUM;
          POTS>>target? : GIVEN NUM;
          POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
          POTSPlusHold>>holdingTable : P(GIVEN NUM);
          POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))]))))
)]% P([_POTS_to_POTSPlusHold_connect_within :
      P([POTSPlusHold>>source? : GIVEN NUM;
        POTSPlusHold>>target? : GIVEN NUM;
        POTS>>source? : GIVEN NUM; POTS>>target? : GIVEN NUM;
        POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
        POTSPlusHold>>holdingTable : P(GIVEN NUM);
        POTS>>calls : P((GIVEN NUM) × (GIVEN NUM))])))]

```

END

```

%[_POTS_to_POTSPlusHold_connect_within :
  P([POTSPlusHold>>source? : GIVEN NUM;
    POTSPlusHold>>target? : GIVEN NUM;
    POTS>>source? : GIVEN NUM; POTS>>target? : GIVEN NUM;
    POTSPlusHold>>calls : P((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold>>holdingTable : P(GIVEN NUM);
    POTS>>calls : P((GIVEN NUM) × (GIVEN NUM)))]

```

AX

```

[_POTS_to_POTSPlusHold_connect_output :
  {[_POTSPlusHold_connect_outputs ∧ _POTS_connect_outputs
  ∧ _POTSPlusHold_connect_inputs ∧ _POTS_connect_inputs
  ∧ _POTSPlusHold_state' ∧ _POTS_state'
  ∧ _POTSPlusHold_state ∧ _POTS_state |
  (POTS>>connection!) ∈ ({POTSPlusHold>>connection!})}]

```

END

```

⊙[_POTS_to_POTSPlusHold_connect_output :
  ℙ([POTSPlusHold≫connection! : GIVEN CALL_SUCCESS;
    POTS≫connection! : GIVEN CALL_SUCCESS;
    POTSPlusHold≫source? : GIVEN NUM;
    POTSPlusHold≫target? : GIVEN NUM;
    POTS≫source? : GIVEN NUM; POTS≫target? : GIVEN NUM;
    POTSPlusHold≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫holdingTable' : ℙ(GIVEN NUM);
    POTS≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫calls : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫holdingTable : ℙ(GIVEN NUM);
    POTS≫calls : ℙ((GIVEN NUM) × (GIVEN NUM))])])

```

AX

```

[_POTS_to_POTSPlusHold_connect_concedes :
  {[_POTSPlusHold_connect_outputs ∧ _POTS_connect_outputs
  ∧ _POTSPlusHold_connect_inputs ∧ _POTS_connect_inputs
  ∧ _POTSPlusHold_state' ∧ _POTS_state'
  ∧ _POTSPlusHold_state ∧ _POTS_state |
  ((((((POTSPlusHold≫target?, POTSPlusHold≫calls)) ∈ (∞ isBusyIn ∞))
  ∧ ((POTSPlusHold≫target?) ∈ (POTSPlusHold≫holdingTable)))
  ∧ ((POTS≫calls) ∈ ({POTS≫calls'})))
  ∧ ((POTSPlusHold≫calls) ∈ ({POTSPlusHold≫calls'})))
  ∧ ((POTS≫connection!) ∈ ({failed})))
  ∧ ((POTSPlusHold≫connection!) ∈ ({held}))}]
END

```

```

⊙[_POTS_to_POTSPlusHold_connect_concedes :
  ℙ([POTSPlusHold≫connection! : GIVEN CALL_SUCCESS;
    POTS≫connection! : GIVEN CALL_SUCCESS;
    POTSPlusHold≫source? : GIVEN NUM;
    POTSPlusHold≫target? : GIVEN NUM;
    POTS≫source? : GIVEN NUM; POTS≫target? : GIVEN NUM;
    POTSPlusHold≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫holdingTable' : ℙ(GIVEN NUM);
    POTS≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫calls : ℙ((GIVEN NUM) × (GIVEN NUM));
    POTSPlusHold≫holdingTable : ℙ(GIVEN NUM);
    POTS≫calls : ℙ((GIVEN NUM) × (GIVEN NUM))])])

```

```

AX
[_POTS_to_POTSPlusHold_disconnect_within :
{[_POTSPlusHold_disconnect_inputs  $\wedge$  _POTS_disconnect_inputs
 $\wedge$  _POTSPlusHold_state  $\wedge$  _POTS_state |
(POTS $\gg$ source?)  $\in$  ({POTSPlusHold $\gg$ source?})}]]]
END
 $\circ$ [_POTS_to_POTSPlusHold_disconnect_within :
 $\mathbb{P}$ ([POTSPlusHold $\gg$ source? : GIVEN NUM;
POTS $\gg$ source? : GIVEN NUM;
POTSPlusHold $\gg$ calls :  $\mathbb{P}$ ((GIVEN NUM)  $\times$  (GIVEN NUM));
POTSPlusHold $\gg$ holdingTable :  $\mathbb{P}$ (GIVEN NUM);
POTS $\gg$ calls :  $\mathbb{P}$ ((GIVEN NUM)  $\times$  (GIVEN NUM)))]])

```

Once Frog has confirmed that all of our paragraphs have been typed correctly, we can conclude that our construct is also typed correctly. As we have determined that the specification is syntactically correct, the next stage is to provide the user with the means to verify the relationship. We do this through the generation of proof obligations.

8.4.4 Verifying the Relationship

We have discussed in chapter 6 the techniques that Frog uses to generate proof obligations. We have also mentioned the problems that have arisen from our decision to use a deep embedding of the Z semantics. The scale of the proof obligations that are generated by Frog can be illustrated through the examination of the – relatively simple – proof obligation required to show that our two machines can be initialized in states that satisfy our retrenchment relationship. This first proof obligation is shown on the following pages. Note that, in order to reduce the size of proof obligations, Frog uses the \spadesuit decoration to indicate that a set is carrier set. For example, we may have two sets NUM and $\spadesuit NUM$; the first of these is the given set we created in our given type paragraph, and the second is the carrier set associated with that given set. That is $\spadesuit NUM$ is equivalent to **GIVEN NUM**, but provides a slight, but significant, abbreviation when used repeatedly within a proof obligation. Again a tool was created to provide a L^AT_EX representation of the

proof obligation, but it was again necessary to manually format the results to produce a more readable version³.

$$\begin{aligned}
& \bowtie \cup \bowtie [NUM, NUM] \\
& \in \{ \mathbb{P} \{ _VAR27 : \spadesuit NUM \times \spadesuit NUM \bullet \forall _GI0001 _GI0002 \\
& \bullet (_GI0001 \in NUM \wedge _GI0002 \in NUM \\
& \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle) \} \} \\
& _VAR213 \in \spadesuit NUM \\
& _VAR217 \in \spadesuit NUM \\
& _VAR221 \in \spadesuit NUM \\
& _VAR225 \in \spadesuit NUM \\
& _VAR229 \in \spadesuit NUM \\
& _VAR233 \in \spadesuit NUM \\
& _VAR237 \in \spadesuit NUM \\
& _VAR241 \in \spadesuit NUM \\
& \spadesuit NUM \in \mathbb{P} \infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \bowtie \mapsto \bowtie [NUM, NUM] \\
& \in \{ \{ _VAR210 : \mathbb{P} \spadesuit NUM \times \spadesuit NUM \\
& \bullet \forall f \bullet (f \in \bowtie \cup \bowtie [NUM, NUM] \wedge \\
& \forall q p \bullet ((p \in f \wedge q \in f) \Rightarrow (\neg \exists _VAR214 \bullet \\
& \forall _VAR211 _VAR212 \bullet (\langle _VAR211, _VAR212 \rangle \\
& \in p \Leftrightarrow _VAR211 \in _VAR214) \wedge _VAR213 \in _VAR214 \wedge \\
& \exists _VAR218 \bullet \forall _VAR215 _VAR216 \bullet \\
& (\langle _VAR215, _VAR216 \rangle \in q \Leftrightarrow _VAR215 \in _VAR218) \\
& \wedge _VAR217 \in _VAR218 \wedge _VAR213 \in \{ _VAR217 \} \\
& \wedge \neg \exists _VAR222 \bullet \forall _VAR219 _VAR220 \bullet \\
& (\langle _VAR219, _VAR220 \rangle \in p \Leftrightarrow _VAR220 \in _VAR222) \\
& \wedge _VAR221 \in _VAR222 \wedge \exists _VAR226 \bullet \forall _VAR223 _VAR224 \bullet \\
& (\langle _VAR223, _VAR224 \rangle \in q \Leftrightarrow _VAR224 \in _VAR226) \\
& \wedge _VAR225 \in _VAR226 \wedge _VAR221 \in \{ _VAR225 \} \\
& \wedge \neg \exists _VAR230 \bullet \forall _VAR227 _VAR228 \bullet \\
& (\langle _VAR227, _VAR228 \rangle \in p \Leftrightarrow _VAR228 \in _VAR230) \\
& \wedge _VAR229 \in _VAR230 \wedge \exists _VAR234 \bullet \forall _VAR231 _VAR232 \bullet \\
& (\langle _VAR231, _VAR232 \rangle \in q \Leftrightarrow _VAR232 \in _VAR234) \\
& \wedge _VAR233 \in _VAR234 \wedge _VAR229 \in \{ _VAR233 \} \} \}
\end{aligned}$$

³Note also that we have used P and PPH to abbreviate the machine names within the proof obligation by taking advantage of Frog's aliasing functionality.

$$\begin{aligned}
& \wedge \neg \exists _VAR238 \bullet \forall _VAR235 _VAR236 \bullet \\
& (\langle _VAR235, _VAR236 \rangle \in p \Leftrightarrow _VAR235 \in _VAR238) \\
& \wedge _VAR237 \in _VAR238 \\
& \wedge \exists _VAR242 \bullet \forall _VAR239 _VAR240 \bullet \\
& (\langle _VAR239, _VAR240 \rangle \in q \Leftrightarrow _VAR239 \in _VAR242) \\
& \wedge _VAR241 \in _VAR242 \wedge _VAR237 \in \{_VAR241\}) \\
& \wedge _VAR210 = f \}} \\
\bowtie \cup \bowtie & [\{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \\
& \mathbb{P}\mathbb{P}\spadesuit NUM \bullet \forall _GI0005 _GI0006 \bullet \\
& (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _GI0006 \in \\
& \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _VAR35 = \langle _GI0005, _GI0006 \rangle)\}, \mathbb{P}\mathbb{P}\spadesuit NUM] \\
& \in \{\mathbb{P}\{_VAR27 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \times \mathbb{P}\mathbb{P}\spadesuit NUM \bullet \forall _GI0001 _GI0002 \bullet (_GI0001 \in \\
& \{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \bullet \forall _GI0005 _GI0006 \bullet (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge _GI0006 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _VAR35 = \\
& \langle _GI0005, _GI0006 \rangle)\} \wedge _GI0002 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle)\}\} \\
\bowtie \mapsto \bowtie & [\{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \\
& \mathbb{P}\mathbb{P}\spadesuit NUM \bullet \forall _GI0005 _GI0006 \bullet \\
& (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _GI0006 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge _VAR35 = \langle _GI0005, _GI0006 \rangle)\}, \mathbb{P}\mathbb{P}\spadesuit NUM] \\
& \in \{\{_VAR28 : \mathbb{P}\mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \bullet \forall f \bullet (f \in \bowtie \mapsto \bowtie [\{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \bullet \forall _GI0005 _GI0006 \bullet (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _GI0006 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge _VAR35 = \langle _GI0005, _GI0006 \rangle)\}, \mathbb{P}\mathbb{P}\spadesuit NUM] \\
& \wedge \forall x \bullet ((x \in \{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \bullet \forall _GI0005 _GI0006 \bullet (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _GI0006 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge _VAR35 = \langle _GI0005, _GI0006 \rangle)\} \Rightarrow (\exists_1 y \bullet y \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \wedge \langle x, y \rangle \in f)) \wedge _VAR28 = f)\}\} \\
& _VAR37 \in \mathbb{P}\mathbb{P}\spadesuit NUM \\
\bowtie \cap \bowtie & [\mathbb{P}\spadesuit NUM] \\
& \in \bowtie \mapsto \bowtie [\{_VAR35 : \mathbb{P}\mathbb{P}\spadesuit NUM \times \mathbb{P}\mathbb{P}\spadesuit NUM \\
& \bullet \forall _GI0005 _GI0006 \bullet (_GI0005 \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _GI0006 \\
& \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge _VAR35 = \langle _GI0005, _GI0006 \rangle)\}, \mathbb{P}\mathbb{P}\spadesuit NUM] \\
\forall b a \bullet & ((a \in \mathbb{P}\mathbb{P}\spadesuit NUM \wedge b \in \mathbb{P}\mathbb{P}\spadesuit NUM) \\
& \Rightarrow (\exists _FUN1 \bullet \forall _VAR36 \bullet (\langle \langle a, b \rangle, _VAR36 \rangle \\
& \in \bowtie \cap \bowtie [\mathbb{P}\spadesuit NUM] \Leftrightarrow _VAR36 \in _FUN1) \\
& \wedge _VAR37 \in _FUN1 \wedge _VAR37 \in \{\{_VAR38 : \mathbb{P}\spadesuit NUM \bullet \\
& \forall x \bullet (x \in \mathbb{P}\spadesuit NUM \wedge x \in a \wedge x \in b \wedge _VAR38 = x)\}\})) \\
\bowtie \mapsto \bowtie & [\bowtie \mapsto \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P}\spadesuit NUM] \\
& \in \{\mathbb{P}\{_VAR27 : \mathbb{P}\spadesuit NUM \times \spadesuit NUM \times \mathbb{P}\spadesuit NUM \\
& \bullet \forall _GI0001 _GI0002 \bullet (_GI0001 \in \bowtie \mapsto \bowtie [\spadesuit NUM, \spadesuit NUM] \\
& \wedge _GI0002 \in \mathbb{P}\spadesuit NUM \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle)\}\}
\end{aligned}$$

$$\begin{aligned}
& \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM] \\
& \in \{ \mathbb{P} \{ _VAR27 : \spadesuit NUM \times \spadesuit NUM \bullet \\
& \quad \forall _GI0001 _GI0002 \bullet (_GI0001 \in \spadesuit NUM \wedge _GI0002 \in \spadesuit NUM \\
& \quad \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle) \} \} \\
& \bowtie \rightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \in \{ \{ _VAR28 : \mathbb{P} \mathbb{P} \spadesuit NUM \times \spadesuit NUM \times \mathbb{P} \spadesuit NUM \\
& \quad \bullet \forall f \bullet (f \in \bowtie \leftrightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \quad \wedge \forall x \bullet ((x \in \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM]) \Rightarrow \\
& \quad (\exists_1 y \bullet y \in \mathbb{P} \spadesuit NUM \wedge \langle x, y \rangle \in f)) \wedge _VAR28 = f \} \} \\
& _VAR71 \in \mathbb{P} \spadesuit NUM \\
& _VAR75 \in \spadesuit NUM \\
& \text{dom}[\spadesuit NUM, \spadesuit NUM] \\
& \in \bowtie \rightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \forall r \bullet ((r \in \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM]) \Rightarrow (\exists _FUN2 \bullet \\
& _VAR70 \bullet (\langle r, _VAR70 \rangle \in \text{dom}[\spadesuit NUM, \spadesuit NUM] \\
& \Leftrightarrow _VAR70 \in _FUN2) \wedge _VAR71 \in _FUN2 \\
& \wedge _VAR71 \in \{ \{ _VAR72 : \spadesuit NUM \bullet \forall p \bullet (p \in r \wedge \exists _VAR76 \bullet \\
& \quad \forall _VAR73 _VAR74 \bullet (\langle _VAR73, _VAR74 \rangle \in p \Leftrightarrow _VAR73 \in _VAR76) \\
& \quad \text{land} _VAR75 \in _VAR76 \wedge _VAR72 = _VAR75) \} \} \} \\
& _VAR299 \in \mathbb{P} \spadesuit NUM \\
& \bowtie \leftrightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \in \{ \mathbb{P} \{ _VAR27 : \mathbb{P} \spadesuit NUM \times \spadesuit NUM \times \mathbb{P} \spadesuit NUM \\
& \quad \bullet \forall _GI0001 _GI0002 \bullet (_GI0001 \in \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM] \\
& \quad \wedge _GI0002 \in \mathbb{P} \spadesuit NUM \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle) \} \} \\
& \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM] \in \\
& \{ \mathbb{P} \{ _VAR27 : \spadesuit NUM \times \spadesuit NUM \\
& \quad \bullet \forall _GI0001 _GI0002 \bullet (_GI0001 \in \spadesuit NUM \wedge _GI0002 \in \spadesuit NUM \\
& \quad \wedge _VAR27 = \langle _GI0001, _GI0002 \rangle) \} \} \\
& \bowtie \rightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \in \{ \{ _VAR28 : \mathbb{P} \mathbb{P} \spadesuit NUM \times \spadesuit NUM \times \mathbb{P} \spadesuit NUM \\
& \quad \bullet \forall f \bullet (f \in \bowtie \leftrightarrow \bowtie [\bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \quad \wedge \forall x \bullet ((x \in \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM]) \Rightarrow \\
& \quad (\exists_1 y \bullet y \in \mathbb{P} \spadesuit NUM \wedge \langle x, y \rangle \in f)) \wedge _VAR28 = f \} \} \\
& _VAR78 \in \mathbb{P} \spadesuit NUM \\
& _VAR82 \in \spadesuit NUM \\
& \text{ran}[\spadesuit NUM, \spadesuit NUM] \\
& \in \bowtie \rightarrow \bowtie [\bowtie \rightarrow \bowtie [\spadesuit NUM, \spadesuit NUM], \mathbb{P} \spadesuit NUM] \\
& \forall r \bullet ((r \in \bowtie \leftrightarrow \bowtie [\spadesuit NUM, \spadesuit NUM]) \Rightarrow (\exists _FUN3 \bullet \\
& \quad \forall _VAR77 \bullet (\langle r, _VAR77 \rangle \in \text{ran}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR77 \in _FUN3) \wedge _VAR78 \in _FUN3 \\
& \quad \wedge _VAR78 \in \{ \{ _VAR79 : \spadesuit NUM \bullet \forall p \bullet (p \in r \wedge \exists _VAR83 \bullet \\
& \quad \forall _VAR80 _VAR81 \bullet (\langle _VAR80, _VAR81 \rangle \in p \Leftrightarrow _VAR81 \in _VAR83) \\
& \quad \wedge _VAR82 \in _VAR83 \wedge _VAR79 = _VAR82) \} \} \} \}
\end{aligned}$$

$$\begin{aligned}
& _VAR301 \in \mathbb{P} \spadesuit NUM \\
& _VAR303 \in \mathbb{P} \spadesuit NUM \\
& _VAR305 \in \mathbb{P} \spadesuit NUM \\
& _VAR307 \in \mathbb{P} \spadesuit NUM \\
& _VAR309 \in \mathbb{P} \spadesuit NUM \\
& _VAR311 \in \mathbb{P} \spadesuit NUM \\
& _VAR313 \in \mathbb{P} \spadesuit NUM \\
& _VAR315 \in \mathbb{P} \spadesuit NUM \\
& \emptyset[\mathbb{P} \spadesuit NUM] \in \\
& \quad \{ \{ _VAR29 : \mathbb{P} \spadesuit NUM \bullet \forall x \bullet \\
& \quad (x \in \mathbb{P} \spadesuit NUM \wedge \neg \text{true} \wedge _VAR29 = x) \} \} \\
& \emptyset[\mathbb{P} \spadesuit NUM \times \spadesuit NUM] \\
& \quad \in \{ \{ _VAR29 : \mathbb{P} \spadesuit NUM \times \spadesuit NUM \bullet \\
& \quad \forall x \bullet (x \in \mathbb{P} \spadesuit NUM \times \spadesuit NUM \wedge \neg \text{true} \wedge _VAR29 = x) \} \} \\
& \vdash? \\
& \forall _v \bullet \\
& (_v \in \{ \langle \text{calls}, \text{holdingTable} \rangle : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM \bullet \\
& \quad \text{calls} \in \bowtie \mapsto \bowtie [NUM, NUM] \\
& \quad \wedge \exists _FUN4 \\
& \quad \bullet (\forall _VAR298 \bullet (\langle \text{calls}, _VAR298 \rangle \in \text{dom}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR298 \in _FUN4) \wedge _VAR299 \in _FUN4) \\
& \quad \wedge \exists _FUN5 \\
& \quad \bullet (\forall _VAR300 \bullet (\langle \text{calls}, _VAR300 \rangle \in \text{ran}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR300 \in _FUN5) \wedge _VAR301 \in _FUN5) \\
& \quad \wedge \exists _FUN6 \\
& \quad \bullet (\forall _VAR302 \bullet (\langle \langle _VAR299, _VAR301 \rangle, _VAR302 \rangle \\
& \quad \in \bowtie \cap \bowtie [\mathbb{P}(\spadesuit NUM)] \\
& \quad \Leftrightarrow _VAR302 \in _FUN6) \wedge _VAR303 \in _FUN6) \\
& \quad \wedge _VAR303 \in \{ \emptyset[\mathbb{P}(\spadesuit NUM)] \} \wedge \text{calls} \in \{ \emptyset[\mathbb{P}(\spadesuit NUM \times \spadesuit NUM)] \} \\
& \quad \wedge \text{holdingTable} \in \mathbb{P}(NUM) \wedge \text{holdingTable} \in \{ \emptyset[\mathbb{P}(\spadesuit NUM)] \} \} \\
& \Rightarrow \exists _u \bullet _u \in \{ \text{calls} : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad \text{calls} \in \bowtie \mapsto \bowtie [NUM, NUM] \\
& \quad \wedge \exists _FUN7 \bullet \\
& \quad (\forall _VAR298 \bullet (\langle \text{calls}, _VAR298 \rangle \in \text{dom}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR298 \in _FUN7) \wedge _VAR299 \in _FUN7) \\
& \quad \wedge \exists _FUN8 \bullet \\
& \quad (\forall _VAR300 \bullet (\langle \text{calls}, _VAR300 \rangle \in \text{ran}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR300 \in _FUN8) \wedge _VAR301 \in _FUN8) \\
& \quad \wedge \exists _FUN9 \bullet \\
& \quad (\forall _VAR302 \bullet (\langle \langle _VAR299, _VAR301 \rangle, _VAR302 \rangle \\
& \quad \in \bowtie \cap \bowtie [\mathbb{P}(\spadesuit NUM)] \\
& \quad \Leftrightarrow _VAR302 \in _FUN9) \wedge _VAR303 \in _FUN9) \\
& \quad \wedge _VAR303 \in \{ \emptyset[\mathbb{P}(\spadesuit NUM)] \} \\
& \quad \wedge \text{calls} \in \{ \emptyset[\mathbb{P}(\spadesuit NUM \times \spadesuit NUM)] \} \}
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u , _v \rangle \in \{ \langle P \gg calls , \langle PPH \gg calls , PPH \gg holdingTable \rangle \} \\
& : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \bullet \\
& P \gg calls \in \bowtie \mapsto \bowtie [NUM , NUM] \\
& \wedge \exists _FUN10 \bullet \\
& \quad (\forall _VAR304 \bullet (\langle P \gg calls , _VAR304 \rangle \in \text{dom}[\spadesuit NUM , \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR304 \in _FUN10) \wedge _VAR305 \in _FUN10) \\
& \wedge \exists _FUN11 \bullet \\
& \quad (\forall _VAR306 \bullet (\langle P \gg calls , _VAR306 \rangle \in \text{ran}[\spadesuit NUM , \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR306 \in _FUN11) \wedge _VAR307 \in _FUN11) \\
& \wedge \exists _FUN12 \bullet \\
& \quad (\forall _VAR308 \bullet (\langle \langle _VAR305 , _VAR307 \rangle , _VAR308 \rangle \\
& \quad \in \bowtie \cap \bowtie [\mathbb{P}(\spadesuit NUM)] \\
& \quad \Leftrightarrow _VAR308 \in _FUN12) \wedge _VAR309 \in _FUN12) \\
& \wedge _VAR309 \in \{ \emptyset [\mathbb{P}(\spadesuit NUM)] \} \\
& \wedge P \gg calls \in \{ \emptyset [\mathbb{P}(\spadesuit NUM \times \spadesuit NUM)] \} \\
& \wedge PPH \gg calls \in \bowtie \mapsto \bowtie [NUM , NUM] \\
& \wedge \exists _FUN13 \bullet \\
& \quad (\forall _VAR310 \bullet (\langle PPH \gg calls , _VAR310 \rangle \in \text{dom}[\spadesuit NUM , \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR310 \in _FUN13) \wedge _VAR311 \in _FUN13) \\
& \wedge \exists _FUN14 \bullet \\
& \quad (\forall _VAR312 \bullet (\langle PPH \gg calls , _VAR312 \rangle \in \text{ran}[\spadesuit NUM , \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR312 \in _FUN14) \wedge _VAR313 \in _FUN14) \\
& \wedge \exists _FUN15 \bullet \\
& \quad (\forall _VAR314 \bullet (\langle \langle _VAR311 , _VAR313 \rangle , _VAR314 \rangle \\
& \quad \in \bowtie \cap \bowtie [\mathbb{P}(\spadesuit NUM)] \\
& \quad \Leftrightarrow _VAR314 \in _FUN15) \wedge _VAR315 \in _FUN15) \\
& \wedge _VAR315 \in \{ \emptyset [\mathbb{P}(\spadesuit NUM)] \} \\
& \wedge PH \gg calls \in \{ \emptyset [\mathbb{P}(\spadesuit NUM \times \spadesuit NUM)] \} \\
& \wedge PPH \gg holdingTable \in \mathbb{P}(NUM) \wedge P \gg calls \in \{ PPH \gg calls \} \}
\end{aligned}$$

The size of this proof obligation illustrates why we have had such difficulty in using the – principally manual – Isabelle/ZF interactive theorem prover. We have used Frog to translate this proof obligation into the Isabelle/ZF syntax and have then applied Isabelle/ZF’s simplification tactic to produce a starting point for a verification. The results of this process can be seen in appendix H.

The relative scale of the proof obligation can be seen even more vividly when we translate the above proof obligation into its shallow-embedded

equivalent. For example, consider the following fragment.

$$\begin{aligned}
& \exists _FUN10 \bullet \\
& \quad (\forall _VAR304 \bullet (\langle P \gg calls _VAR304 \rangle \in \text{dom}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR304 \in _FUN10) \wedge _VAR305 \in _FUN10 \\
& \wedge \exists _FUN11 \bullet \\
& \quad (\forall _VAR306 \bullet (\langle P \gg calls _VAR306 \rangle \in \text{ran}[\spadesuit NUM, \spadesuit NUM] \\
& \quad \Leftrightarrow _VAR306 \in _FUN11) \wedge _VAR307 \in _FUN11 \\
& \wedge \exists _FUN12 \bullet \\
& \quad (\forall _VAR308 \bullet (\langle _VAR305, _VAR307 \rangle, _VAR308) \\
& \quad \in \bowtie \cap \bowtie [\mathbb{P}(\spadesuit NUM)] \\
& \quad \Leftrightarrow _VAR308 \in _FUN12) \wedge _VAR309 \in _FUN12) \\
& \wedge _VAR309 \in \{\emptyset[\mathbb{P}(\spadesuit NUM)]\}
\end{aligned}$$

In a shallow embedding we could express this fragment with the following expression.

$$\text{dom } P \gg calls \cap \text{ran } P \gg calls \in \{\emptyset\}$$

This not only makes the proof obligation easier for humans to comprehend, but significantly easier for theorem provers to manipulate. In fact the entire initialization proof obligation can be expressed as follows.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}^\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \vdash? \\
& \forall _v \bullet (_v \in \{\langle calls, holdingTable \rangle : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM \bullet \\
& \quad calls \in NUM \leftrightarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge calls \in \{\emptyset\} \wedge holdingTable \in \mathbb{P}(NUM) \\
& \quad \wedge holdingTable \in \{\emptyset\}\}) \\
& \Rightarrow \exists _u \bullet _u \in \{calls : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad calls \in NUM \leftrightarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge calls \in \{\emptyset\}\}
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u , _v \rangle \in \{ \langle P \gg calls , \langle PPH \gg calls , PPH \gg holdingTable \rangle \} \\
& : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \bullet \\
& P \gg calls \in NUM \leftrightarrow NUM \wedge \text{dom } P \gg calls \cap \text{ran } P \gg calls \in \{\emptyset\} \\
& \wedge P \gg calls \in \{\emptyset\} \wedge PPH \gg calls \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPH \gg calls \cap \text{ran } PPH \gg calls \in \{\emptyset\} \wedge PPH \gg calls \in \{\emptyset\} \\
& \wedge PPH \gg holdingTable \in \mathbb{P}(NUM) \wedge PPH \gg holdingTable \in \{\emptyset\} \\
& \wedge P \gg calls \in \{ PPH \gg calls \}
\end{aligned}$$

Careful examination of the two proof obligations will show that they are both equivalent and correct instantiations of our retrenchment, initialization proof obligation for the given specification. With limited resources available however, we have been unable to pursue a mechanical verification of our obligations. While unfortunate, we feel that even in the future our time would be better spent pursuing a general solution to this problem by interfacing to a more suitable theorem prover (as we discussed in section 6.4.11) than in producing a proof requiring extensive manual intervention for a specific example.

In order to save space and produce proof obligations more legible to the human reader, the remaining proof obligations in this case study will be presented in the abbreviated format that we have produced above. These proof obligations have been obtained through a semi-automatic translation of the \LaTeX representations produced by Frog. They should be considered an accurate equivalent to those produced by the tool.

The next proof obligation generated is at the operation environment level and is used to demonstrate applicability. The first instance of this is created for the *connect* operation and is shown below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P} \infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \vdash? \\
& \forall _u _v _i _j \bullet \langle _u , _i \rangle \in \{ \langle calls , \langle source , target? \rangle \} \\
& : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \bullet \\
& calls \in NUM \leftrightarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \wedge holdingTable \in \mathbb{P}(NUM) \\
& \wedge source? \in NUM \wedge target? \in NUM \wedge source? \text{ isFreeIn } calls
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v \rangle \in \{ \langle P \gg \text{calls}, \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle \} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{ \emptyset \} \\
& \wedge P \gg \text{calls} \in \{ \emptyset \} \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{ \emptyset \} \wedge PPH \gg \text{calls} \in \{ \emptyset \} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPH \gg \text{holdingTable} \in \{ \emptyset \} \\
& \wedge P \gg \text{calls} \in \{ PPH \gg \text{calls} \} \} \\
& \wedge \langle _u, _v, _i _j \rangle \in \{ \langle P \gg \text{calls}, \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle, \\
& \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \langle PPH \gg \text{source?}, PPH \gg \text{target?} \rangle \} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{ \emptyset \} \\
& \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{ \emptyset \} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPH \gg \text{source?} \in \text{NUM} \\
& \wedge PPH \gg \text{target?} \in \text{NUM} \wedge P \gg \text{source?} \in \{ PPH \gg \text{source?} \} \\
& \wedge P \gg \text{target?} \in \{ PPH \gg \text{target?} \} \} \\
& \Rightarrow \langle _v, _j \rangle \in \{ \langle \text{calls}, \langle \text{source}, \text{target?} \rangle \} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{ \emptyset \} \\
& \wedge \text{holdingTable} \in \mathbb{P}(\text{NUM}) \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{source?} \text{ isFreeIn } \text{calls} \}
\end{aligned}$$

Again, a close examination of the proof obligation will confirm to the reader that it has been correctly derived from the relationship's configuration and specification. The equivalent proof obligation for the *disconnect* operation is practically identical; the only difference being that the last term in the precondition of the target machine's operation will state that the source number must be busy in calls rather than free.

Both the initialization and applicability proof obligations will be very similar for all of the retrenchments within this case study. In the interests of brevity we will not examine these proof obligations for the remaining relationships. Similarly, the *disconnect* operation will be identical in all of our machines as they are all simply references to the operation defined in the *POTS* machine. As such, there seems little point in presenting proof obligations for this operation as the behaviour of the models will be intrinsically correct.

With this in mind we progress to consider the correctness proof obligation for the retrenchment of the *connect* operation. The abbreviated version of the proof obligation is shown below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}_\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \spadesuit CALL_SUCCESS \in \mathbb{P}_\infty \\
& CALL_SUCCESS \in \mathbb{P} \spadesuit CALL_SUCCESS \\
& \vdash? \\
& \forall _u _v _v' _i _j _p \bullet \\
& \langle _v _v' _j _p \rangle \in \{ \langle \langle calls _ holdingTable \rangle, \\
& \quad \langle calls' _ holdingTable' \rangle, \langle source? _ target? \rangle, connection! \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM) \\
& \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM) \\
& \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& calls \in NUM \rightsquigarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \wedge calls' \in NUM \rightsquigarrow NUM \wedge \text{dom } calls' \cap \text{ran } calls' \in \{\emptyset\} \\
& \wedge holdingTable \in \mathbb{P}(NUM) \wedge holdingTable' \in \mathbb{P}(NUM) \\
& \wedge source? \in NUM \wedge target? \in NUM \wedge connection! \in CALL_SUCCESS \\
& \neg (target? \text{isFreeIn } calls \wedge source \neq target? \wedge \neg (\\
& \quad calls' \in \{calls \cup \{source? \mapsto target?\} \} \wedge connection! \in \{successful\})) \\
& \wedge \neg (\neg (\neg (target? \text{isBusyIn } calls) \wedge \neg (source? \in \{target?\})) \\
& \quad \wedge target? \notin holdingTable) \wedge \neg (calls' \in \{calls\} \wedge connection! \in \{held\}) \\
& \wedge \neg (\neg (\neg (target? \text{isBusyIn } calls) \wedge \neg (source? \in \{target?\})) \\
& \quad \wedge target? \notin holdingTable \wedge \neg (calls' \in \{calls\} \wedge connection \in \{failed\})) \\
& \quad \wedge holdingTable' \in \{holdingTable\}) \\
& \wedge \langle _u _v \rangle \in \{ \langle P \gg calls _ PPH \gg calls _ PPH \gg holdingTable \rangle \} \\
& : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \bullet \\
& P \gg calls \in NUM \rightsquigarrow NUM \wedge \text{dom } P \gg calls \cap \text{ran } P \gg calls \in \{\emptyset\} \\
& \wedge PPH \gg calls \in NUM \rightsquigarrow NUM \wedge \text{dom } PPH \gg calls \cap \text{ran } PPH \gg calls \in \{\emptyset\} \\
& \wedge PPH \gg holdingTable \in \mathbb{P}(NUM) \wedge P \gg calls \in \{PPH \gg calls \} \\
& \wedge \langle _u _v _i _j \rangle \in \{ \langle P \gg calls _ PPH \gg calls _ PPH \gg holdingTable \rangle, \\
& \quad \langle P \gg source? _ P \gg target? \rangle, \langle PPH \gg source? _ PPH \gg target? \rangle \} \\
& : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \\
& \times (\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \bullet
\end{aligned}$$

$$\begin{aligned}
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPH \gg \text{source?} \in \text{NUM} \\
& \wedge PPH \gg \text{target?} \in \text{NUM} \wedge P \gg \text{source?} \in \{PPH \gg \text{source?}\} \\
& \wedge P \gg \text{target?} \in \{PPH \gg \text{target?}\} \\
& \wedge \langle _u, _i \rangle \in \{\langle \text{calls}, \langle \text{source}, \text{target?} \rangle \rangle\} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{source?} \text{ isFreeIn } \text{calls} \\
& \Rightarrow \\
& \exists _u' _o \bullet \langle _u, _u', _i, _o \rangle \in \{\langle \text{calls}, \\
& \text{calls}', \langle \text{source?}, \text{target?} \rangle, \text{connection!} \rangle\} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls}' \cap \text{ran } \text{calls}' \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{connection!} \in \text{CALL_SUCCESS} \\
& \neg (\text{target} \text{ isFreeIn } \text{calls} \wedge \text{source?} \neq \text{target?} \wedge \neg (\\
& \text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{target?}\} \wedge \text{connection!} \in \{\text{successful}\}\}) \\
& \wedge \neg (\neg (\neg (\text{target?} \text{ isBusyIn } \text{calls}) \wedge \neg (\text{source?} \in \{\text{target?}\})) \\
& \wedge \neg (\text{calls}' \in \{\text{calls}\} \wedge \text{connection!} \in \{\text{failed}\})) \\
& \wedge ((\langle _u', _v' \rangle \in \{\langle P \gg \text{calls}, \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle \rangle\} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge P \gg \text{calls} \in \{PPH \gg \text{calls}\} \\
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{\langle P \gg \text{calls}, \\
& \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle, P \gg \text{calls}', \langle \\
& PPH \gg \text{calls}', PPH \gg \text{holdingTable}' \rangle, \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \\
& \langle PPH \gg \text{source?}, PPH \gg \text{target?} \rangle, P \gg \text{connection!}, PPH \gg \text{connection!} \rangle\} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times (\mathbb{P} \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P} \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPH \gg \text{calls}' \cap \text{ran } PPH \gg \text{calls}' \in \{\emptyset\}
\end{aligned}$$

$$\begin{aligned}
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPH \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPH \gg \text{source?} \in \text{NUM} \\
& \wedge PPH \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPH \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge P \gg \text{connection!} \in \{PPH \gg \text{connection!}\} \\
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \langle P \gg \text{calls}, \\
& \quad \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle, P \gg \text{calls}', \langle \\
& \quad PPH \gg \text{calls}', PPH \gg \text{holdingTable}' \rangle, \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \\
& \quad \langle PPH \gg \text{source?}, PPH \gg \text{target?} \rangle, P \gg \text{connection!}, PPH \gg \text{connection!} \rangle \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P} \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P} \spadesuit \text{NUM}) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPH \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPH \gg \text{calls}' \cap \text{ran } PPH \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPH \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPH \gg \text{source?} \in \text{NUM} \\
& \wedge PPH \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPH \gg \text{connection!} \in \text{CALL_SUCCESS} \wedge PPH \gg \text{target?} \text{ isBusyIn calls} \\
& \wedge PPH \gg \text{target?} \in PPH \gg \text{holdingTable} \wedge P \gg \text{calls} \in \{P \gg \text{calls}'\} \\
& \wedge PPH \gg \text{calls} \in \{PPH \gg \text{calls}'\} \wedge P \gg \text{connection!} \in \{\text{failed}\} \\
& \wedge PPH \gg \text{connection} \in \{\text{held}\}
\end{aligned}$$

Once more a close inspection of the proof obligation reveals it to be the required instantiate of the configuration's generic proof obligation with the information retrieved from the construct's specification. By this point even the abbreviated proof obligations are starting to grow, and as the ratio in size between this proof obligation and the one produced through the simplification of the one passed to Isabelle/ZF is roughly equivalent to those produced for initialization, we can see how non-trivial proof obligations quickly become unwieldy for any system that is not at least partially automated.

8.5 Introducing The Forwarding Service

We now consider the introduction of the forwarding service. The forwarding service is another provision for a subscriber that allows them to ensure that they do not miss important calls. When someone makes a call to the subscriber, and the subscriber's phone is busy the system will attempt to forward the call to another number registered by the subscriber.

8.5.1 Specifying the Extended Machine

The machine, *POTSPlusForward*, defined below, shows how the original phone system machine (defined in section 8.3) can be extended to incorporate the behaviour for the forwarding service.

MACHINE *POTSPlusForward*

TYPE *machine*

SECTION *POTS*

STATE

POTS.state

forwardingTable : *NUM* \rightsquigarrow *NUM*

$(\textit{forwardingTable}^+) \cap \text{id } \textit{NUM} = \emptyset$

INITIALIZATION

POTS.initialization

forwardingTable = \emptyset

OPERATION *connect* $\hat{=}$

INPUTS

source? : *NUM*

target? : *NUM*

OUTPUTS

connection! : *CALL_SUCCESS*

PRE

 $source? \text{ isFreeIn } calls$

POST

 $forwardingNumber : NUM$

 $(target? \text{ isFreeIn } calls \wedge source? \neq target?)$
 $\Rightarrow (calls' = calls \cup \{source? \mapsto target?\})$
 $\wedge connection! = successful)$
 $(target? \text{ isBusyIn } calls \wedge source? \neq target?)$
 $\wedge (target?, forwardingNumber) \in forwardingTable^+$
 $\wedge forwardingNumber \text{ isFreeIn } calls$
 $\wedge forwardingNumber \neq source?)$
 $\Rightarrow (calls' = calls \cup \{source? \mapsto forwardingNumber\})$
 $\wedge connection! = successful)$
 $((target? \text{ isBusyIn } calls$
 $\wedge (target? \notin \text{dom } forwardingTable$
 $\vee ((target?, forwardingNumber) \in forwardingTable^+$
 $\wedge (forwardingNumber \text{ isBusyIn } calls$
 $\vee forwardingNumber = source?)))$
 $\vee source? = target?))$
 $\Rightarrow (calls' = calls \wedge connection! = failed)$
 $forwardingTable' = forwardingTable$
END *connect*OPERATION *disconnect* $\hat{=}$

INPUTS

 $source? : NUM$

PRE

 $POTS.disconnect.pre$

POST

POTS.disconnect.post

$forwardingTable' = forwardingTable$

END *disconnect*

OPERATION *registerForForward* $\hat{=}$

INPUTS

$regSource? : NUM$

$regTarget? : NUM$

POST

$((forwardingTable \oplus \{regSource? \mapsto regTarget?\})^+ \cap id\ NUM = \emptyset)$
 $\Rightarrow forwardingTable' = forwardingTable$
 $\oplus \{regSource? \mapsto regTarget?\}$

$((forwardingTable \oplus \{regSource? \mapsto regTarget?\})^+ \cap id\ NUM \neq \emptyset)$
 $\Rightarrow forwardingTable' = forwardingTable$

$calls' = calls$

END *registerForForward*

OPERATION *cancelForward* $\hat{=}$

INPUTS

$regNumber? : NUM$

PRE

$regNumber? \in \text{dom } forwardingTable$

POST

$$forwardingTable' = \{regNumber?\} \triangleleft forwardingTable$$

$$calls' = calls$$

END *cancelForward*

END *POTSPlusForward*

Again, we define the parent section of the machine to be *POTS* allowing us to inherit unchanged behaviour directly.

The *POTSPlusForward* machine inherits the state of the *POTS* machine (that is, the set of current calls), and defines an additional state variable that enables us to store forwarding data. This data is in the form of a partial injection between phone numbers. Each pair in the forwarding data represents a link in a forwarding chain. Whilst most forwarding chains would typically involve a single link, there may exist chains with several links, and it is therefore necessary to use the non-reflexive transitive closure (see C.19) of our partial injection when attempting to determine a forwarding number for a given number⁴. The set of forwarding data is initialized to the empty-set. The state variable inherited from the *POTS* machine is initialized with the initialization schema also inherited from that machine. Four operations are specified in our extended model: the connect and disconnect operations present in the standard machine, and two additional operations that allow users to add and remove data from the forwarding table.

⁴Some may consider the use of non-reflexive transitive closure on the forwarding table to be unusual. It was used in the original case study and is intended to indicate that forwarding numbers will automatically be chained. For instance, if the number 1111 is the forwarding number for 0000 and 2222 the forwarding number for 1111; then a caller dialling 0000, and finding the phone busy, will be forwarded to 2222. What we might expect is that the forwarding will only be chained if the intermediate number is busy. That is, if 0000 is busy, a connection will be attempted with 1111 and only if that number is busy will there be an attempt to connect to 2222. We have noted previously our intention to maintain the meaning of the original case study and so we continue to use non-reflexive transitive closure in this machine.

The *connect* operation declares the same inputs and outputs as the un-enhanced model. In the precondition of the operation we again state that the calling phone must be free in order to make a call. In the postcondition we specify that if the callee is free – and we have not tried to call our own phone – the call will be connected and success reported, if the callee is busy and has registered for the call forwarding service, the system will determine the forwarding number required and if that number is free connect the call (to the forwarding number), and report success, if the forwarding number is busy, or the original callee has not registered for the forwarding service the call will not be connected and the failure will be reported.

The second operation, *disconnect*, has practically identical behaviour to the operation in the *POTS* machine, the only difference being the necessity to note the lack of change in the state of the forwarding table

The *registerForForward* operation allows the addition of forwarding data to the machine's forwarding table. If a number has already been registered for the forwarding service, its data will simply be overwritten (as each number can only appear once in the domain of a partial injection). The operation has two inputs: the number to be registered and the number to forward to. The body of the operation states the following: if the addition of the forwarding data will not produce cycles in the forwarding table then that data will be added, if cycles would result from that addition, then no data is added.

The *cancelForward* operation allows the removal of a forwarding rule from the machine's forwarding table. The operation has a single input which is the number of the subscribed phone for which the rule is to be cancelled. The precondition of the operation states that: the number to be removed from the forwarding table must already have forwarding rules associated with it, and the postcondition specifies that the row in the forwarding table that refers to the registered number will be removed.

8.5.2 Specifying the Relationship

We now consider the specification of the retrenchment relationship between the model of the plain old telephone service and the model in which the

forwarding service is introduced. This relationship will allow us to prove that the models behave identically, except in circumstances made clear in the specification of the retrenchment. The definition of the relationship is shown below.

RELATIONSHIP *POTS_to_POTSPlusForward*
 TYPE *retrenchment*
 FROM *POTS*
 TO *POTSPlusForward*
 RETRIEVE

$$POTS \gg \text{calls} = POTSPlusForward \gg \text{calls}$$

RAMIFICATIONS *connect* $\hat{=}$
 WITHIN

$$POTS \gg \text{source?} = POTSPlusForward \gg \text{source?}$$

$$POTS \gg \text{target?} = POTSPlusForward \gg \text{target?}$$

OUTPUT

$$POTS \gg \text{connection!} = POTSPlusForward \gg \text{connection!}$$

CONCEDES

$$POTSPlusForward \gg \text{target?} \text{ isBusyIn } POTSPlusForward \gg \text{calls}$$

$$\exists \text{ forwardingNumber} : \text{NUM}$$

- $((POTSPlusForward \gg \text{target?}, \text{forwardingNumber})$
 - $\in POTSPlusForward \gg \text{forwardingTable}^+$
 - $\wedge \text{forwardingNumber} \text{ isFreeIn } POTSPlusForward \gg \text{calls}$
 - $\wedge \text{forwardingNumber} \neq POTSPlusForward \gg \text{source?}$
 - $\wedge POTSPlusForward \gg \text{calls}'$
 - $= POTS \gg \text{calls}'$
 - $\cup \{POTSPlusForward \gg \text{source?} \mapsto \text{forwardingNumber}\}$

$$POTS \gg \text{connection!} = \text{failed}$$

$$POTSPlusForward \gg \text{connection!} = \text{successful}$$

END *connect*

RAMIFICATIONS *disconnect* $\hat{=}$

WITHIN

$POTS \gg source? = POTSPlusForward \gg source?$

END *disconnect*

END *POTS_to_POTSPlusForward*

In the retrieve clause we specify that the set of current calls in each machine should be identical. We then consider any exceptions to this rule by specifying the ramifications for each operation. No ramifications are required for the operations that concern the administration of the forwarding table; these operations have no equivalent in the model of the plain old telephone service.

Consider first, the *connect* operation. The within clause specifies that the inputs to the operations of both machines must be identical. The output clause specifies that in normal circumstances the outputs from the operations of both machines should also be identical. The concedes clause must, therefore, handle those situations in which either the set of current calls in each machine, or the outputs from the operation of each machine can allowably differ. In the *connect* operation we have one circumstance where the behaviour of the two operations in the two machines differs. When a callee is busy and has registered for the forwarding service, and the forwarding number available is both free and different to the original caller's number, a connection can be made in the machine providing the forwarding service where it cannot in the machine that provides only the standard service. The concedes clause therefore, details this circumstance, specifies the new link between the state of the set of calls following the execution of the operation in the two machines, and specifies the allowable values for the outputs of the operation in each machine.

Again, the ramifications for the *disconnect* operation are much simpler. The behaviour of the operation is identical in both machines, and we use the within clause simply to ensure that the inputs to the operation are the same for both machines.

With the specification of this relationship we may wish to take advantage of Frog’s construct flexibility. We could declare a new type of relationship-type which allows the declaration of local variables in the concedes clause. We would do this by changing the content attribute of that clause from a predicate to a schema text. Making this change would give us the following specification for the concedes clause.

CONCEDES

forwardingNumber : NUM

$POTSPlusForward \gg target?$ isBusyIn $POTSPlusForward \gg calls$
 $((POTSPlusForward \gg target?, forwardingNumber)$
 $\in POTSPlusForward \gg forwardingTable^+$
 $forwardingNumber$ isFreeIn $POTSPlusForward \gg calls$
 $forwardingNumber \neq POTSPlusForward \gg source?$
 $POTSPlusForward \gg calls'$
 $= POTS \gg calls'$
 $\cup \{POTSPlusForward \gg source? \mapsto forwardingNumber\}$
 $POTS \gg connection! = failed$
 $POTSPlusForward \gg connection! = successful$

In effect, these representations have almost identical semantics (the only difference being the scope of the local variable), and the choice between them is aesthetic. When we use the above concedes clause in a proof obligation an existential quantification will be created for the local variable over the entire predicate. This choice between representations illustrates the flexibility that using Frog-CCL provides.

When creating this relationship we considered a possible extension to Frog-CCL that would allow some form of inheritance. This would entail one configuration possessing the properties of another, but with some extension or overriding a subset of the existing properties. We did not have the opportunity to investigate this concept thoroughly, but it is an idea that we will seek to pursue in the future.

8.5.3 Syntax Checking the Relationship

Once we had created the specification of the machine that incorporated the forwarding service and the relationship that showed it was a retrenchment of the *POTS* machine, we were ready to begin syntax checking the relationship (of course, we would also previously ensure that the machines involved in the relationship were syntactically consistent). After resolving the ubiquitous errors that will inevitably creep into any specification, Frog accepted our construct's specification and produced its equivalent, typed abstract syntax tree. We will not show the output of the various steps in the syntax checking process, but will just present the typed version of the paragraph associated with the concedes clause (which we imagine would be the most interesting to a reader). As in the previous section, we do not show the version annotated with every expression's type, but just the version that shows the paragraph's type. This is displayed below.

```

AX [_POTS_to_POTSPlusForward_connect_concedes :
  { [_POTSPlusForward_connect_outputs ∧ _POTS_connect_outputs
    ∧ _POTSPlusForward_connect_inputs ∧ _POTS_connect_inputs
    ∧ _POTSPlusForward_state' ∧ _POTS_state'
    ∧ _POTSPlusForward_state ∧ _POTS_state |
    ((((POTSPlusForward≫target?, POTSPlusForward≫calls))
      ∈ (⊗ isBusyIn ⊗))
    ∧ (¬ (∀([forwardingNumber : NUM]) •
      (¬ ((((POTSPlusForward≫target?, forwardingNumber)
        ∈ ((⊗+ POTSPlusForward≫forwardingTable))
        ∧ (((forwardingNumber, POTSPlusForward≫calls) ∈ (⊗ isFreeIn ⊗)))
        ∧ (((forwardingNumber, POTSPlusForward≫source?) ∈ (⊗ ≠ ⊗)))
        ∧ ((POTSPlusForward≫calls')
          ∈ ({⊗ ∪ ⊗ (POTS≫calls',
            {⊗ → ⊗ (POTSPlusForward≫source?, forwardingNumber)}))))))))))
      ∧ ((POTS≫connection!) ∈ ({failed}))
      ∧ ((POTSPlusForward≫connection!) ∈ ({successful})))]
  END
  ∘ [_POTS_to_POTSPlusForward_connect_concedes :
    ℙ([POTSPlusForward≫connection! : GIVEN CALL_SUCCESS;
      POTS≫connection! : GIVEN CALL_SUCCESS;
      POTSPlusForward≫source? : GIVEN NUM;
      POTSPlusForward≫target? : GIVEN NUM;
      POTS≫source? : GIVEN NUM; POTS≫target? : GIVEN NUM;
      POTSPlusForward≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
      POTSPlusForward≫forwardingTable' : ℙ((GIVEN NUM) × (GIVEN NUM));
      POTS≫calls' : ℙ((GIVEN NUM) × (GIVEN NUM));
      POTSPlusForward≫calls : ℙ((GIVEN NUM) × (GIVEN NUM));
      POTSPlusForward≫forwardingTable : ℙ((GIVEN NUM) × (GIVEN NUM));
      POTS≫calls : ℙ((GIVEN NUM) × (GIVEN NUM)))]

```

8.5.4 Verifying the Relationship

Once the relationship had been confirmed as syntactically correct, we were in a position to generate the proof obligations for the construct. Frog was able to generate these successfully for all of the generic proof obligations and for all of the relevant operation environments. Again these proof obligations were large in size and mechanical verification with Isabelle/ZF was impossible within our project's time constraints. Close, manual examination of the

proof obligations however, could allow the informed observer to conclude that the relationship was valid, and given enough space and time, a pen-and-paper proof could fairly easily be presented. An abbreviated version of the correctness proof obligation for the *connect* operation of our relationship is presented below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}_\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \spadesuit CALL_SUCCESS \in \mathbb{P}_\infty \\
& CALL_SUCCESS \in \mathbb{P} \spadesuit CALL_SUCCESS \\
& \vdash? \\
& \forall _u _v _v' _i _j _p \bullet \\
& \langle _v _v' _i _j _p \rangle \in \{ \langle \text{calls} _i _j _p \rangle, \\
& \quad \langle \text{calls}' _i _j _p \rangle, \langle \text{source?} _i _j _p \rangle, \text{connection!} \rangle \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \quad \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \quad \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& \exists \text{forwardingNumber} \bullet (\\
& \quad \text{calls} \in NUM \rightsquigarrow NUM \wedge \text{dom calls} \cap \text{ran calls} \in \{\emptyset\} \\
& \quad \wedge \text{calls}' \in NUM \rightsquigarrow NUM \wedge \text{dom calls}' \cap \text{ran calls}' \in \{\emptyset\} \\
& \quad \wedge \text{forwardingTable} \in NUM \rightsquigarrow NUM \\
& \quad \wedge \text{forwardingTable}' \in NUM \rightsquigarrow NUM \\
& \quad \wedge (\text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \quad \wedge (\text{forwardingTable}'^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \quad \wedge \text{source?} \in NUM \wedge \text{target?} \in NUM \\
& \quad \wedge \text{connection!} \in CALL_SUCCESS \\
& \quad \wedge \text{forwardingNumber} \in NUM
\end{aligned}$$

$$\begin{aligned}
& \neg (target? \text{isFreeIn } calls \wedge source \neq target? \wedge \neg (\\
& \quad calls' \in \{calls \cup \{source? \mapsto target?\} \wedge connection! \in \{successful?\}) \\
& \quad \wedge \neg (target? \text{isBusyIn } calls \wedge source? \neq target? \\
& \quad \wedge \langle target?, forwardingNumber \rangle \in (forwardingTable^+ \\
& \quad \wedge forwardingNumber \text{isFreeIn } calls \wedge forwardingNumber \neq source? \\
& \quad \neg (calls' \in \{calls \cup \{source? \mapsto forwardingNumber\} \\
& \quad \wedge connection! \in \{successful?\})) \wedge \neg (\neg (\neg (target? \text{isBusyIn } calls \\
& \quad \wedge \neg (\neg (target? \in \text{dom } forwardingTable) \wedge \neg (\\
& \quad \langle target?, forwardingNumber \rangle \in (forwardingTable^+ \\
& \quad \wedge \neg (\neg (forwardingNumber \text{isBusyIn } calls) \\
& \quad \wedge \neg (forwardingNumber \in \{source?\})))))) \\
& \quad \wedge \neg (source? \in \{target?\})) \wedge \neg (calls' \in \{calls\} \\
& \quad \wedge connection! \in \{failed?\}) \\
& \quad \wedge forwardingTable' \in \{forwardingTable\}) \\
& \wedge \langle _u, _v \rangle \in \{ \langle P \gg calls, \langle PPF \gg calls, PPF \gg forwardingTable \rangle \} \\
& \quad : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \bullet \\
& \quad P \gg calls \in NUM \leftrightarrow NUM \wedge \text{dom } P \gg calls \cap \text{ran } P \gg calls \in \{\emptyset\} \\
& \quad \wedge PPF \gg calls \in NUM \leftrightarrow NUM \wedge \text{dom } PPF \gg calls \cap \text{ran } PPF \gg calls \in \{\emptyset\} \\
& \quad \wedge PPF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \quad \wedge (PPF \gg forwardingTable^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge P \gg calls \in \{PPF \gg calls\} \\
& \wedge \langle _u, _v, _i, _j \rangle \in \{ \langle P \gg calls, \langle PPF \gg calls, PPF \gg forwardingTable \rangle, \\
& \quad \langle P \gg source?, P \gg target? \rangle, \langle PPF \gg source?, PPF \gg target? \rangle \} \\
& \quad : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \quad \times (\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad P \gg calls \in NUM \leftrightarrow NUM \wedge \text{dom } P \gg calls \cap \text{ran } P \gg calls \in \{\emptyset\} \\
& \quad \wedge PPF \gg calls \in NUM \leftrightarrow NUM \wedge \text{dom } PPF \gg calls \cap \text{ran } PPF \gg calls \in \{\emptyset\} \\
& \quad \wedge PPF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \quad \wedge (PPF \gg forwardingTable^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge P \gg source? \in NUM \wedge P \gg target? \in NUM \wedge PPF \gg source? \in NUM \\
& \quad \wedge PPF \gg target? \in NUM \wedge P \gg source? \in \{PPF \gg source?\} \\
& \quad \wedge P \gg target? \in \{PPF \gg target?\} \\
& \wedge \langle _u, _i \rangle \in \{ \langle calls, \langle source, target? \rangle \} \\
& \quad : \mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad calls \in NUM \leftrightarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge source? \in NUM \wedge target? \in NUM \wedge source? \text{isFreeIn } calls \\
& \Rightarrow
\end{aligned}$$

$$\begin{aligned}
& \exists _u' _o \bullet \langle _u, _u', _i, _o \rangle \in \{ \langle \text{calls}, \\
& \quad \text{calls}', \langle \text{source?}, \text{target?} \rangle, \text{connection!} \rangle \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& \text{calls} \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom calls} \cap \text{ran calls} \in \{\emptyset\} \\
& \wedge \text{calls}' \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom calls}' \cap \text{ran calls}' \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{connection!} \in \text{CALL_SUCCESS} \\
& \neg (\text{target isFreeIn calls} \wedge \text{source?} \neq \text{target?} \wedge \neg (\\
& \quad \text{calls}' \in \{ \text{calls} \cup \{ \text{source?} \mapsto \text{target?} \} \wedge \text{connection!} \in \{ \text{successful} \} \}) \\
& \wedge \neg (\neg (\neg (\text{target? isBusyIn calls}) \wedge \neg (\text{source?} \in \{ \text{target?} \})) \\
& \quad \wedge \neg (\text{calls}' \in \{ \text{calls} \} \wedge \text{connection!} \in \{ \text{failed} \})) \\
& \wedge (\langle _u', _v' \rangle \in \{ \langle P \gg \text{calls}, \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle \rangle \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \bullet \\
& P \gg \text{calls} \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{calls} \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom } \text{PPF} \gg \text{calls} \cap \text{ran } \text{PPF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \text{NUM} \rightsquigarrow \text{NUM} \\
& \wedge (\text{PPF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{calls} \in \{ \text{PPF} \gg \text{calls} \} \\
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \langle P \gg \text{calls}, \\
& \quad \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle, P \gg \text{calls}', \langle \\
& \quad \text{PPF} \gg \text{calls}', \text{PPF} \gg \text{forwardingTable}' \rangle, \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \\
& \quad \langle \text{PPF} \gg \text{source?}, \text{PPF} \gg \text{target?} \rangle, P \gg \text{connection!}, \text{PPF} \gg \text{connection!} \rangle \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \mathbb{P} \spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{calls} \in \text{NUM} \rightsquigarrow \text{NUM} \\
& \wedge \text{dom } \text{PPF} \gg \text{calls} \cap \text{ran } \text{PPF} \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \rightsquigarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{calls}' \in \text{NUM} \rightsquigarrow \text{NUM} \\
& \wedge \text{dom } \text{PPF} \gg \text{calls}' \cap \text{ran } \text{PPF} \gg \text{calls}' \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \text{NUM} \rightsquigarrow \text{NUM} \\
& \wedge (\text{PPF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{forwardingTable}' \in \text{NUM} \rightsquigarrow \text{NUM} \\
& \wedge (\text{PPF} \gg \text{forwardingTable}'^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge \text{PPF} \gg \text{source?} \in \text{NUM} \\
& \wedge \text{PPF} \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge \text{PPF} \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge P \gg \text{connection!} \in \{ \text{PPF} \gg \text{connection!} \} \\
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \langle P \gg \text{calls} , \\
& \quad \langle PPF \gg \text{calls} , PPF \gg \text{forwardingTable} \rangle, P \gg \text{calls}' , \langle \\
& \quad PPF \gg \text{calls}' , PPF \gg \text{forwardingTable}' \rangle, \langle P \gg \text{source?} , P \gg \text{target?} \rangle, \\
& \quad \langle PPF \gg \text{source?} , PPF \gg \text{target?} \rangle, P \gg \text{connection!} , PPF \gg \text{connection!} \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPF \gg \text{calls} \cap \text{ran } PPF \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPF \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPF \gg \text{calls}' \cap \text{ran } PPF \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPF \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge PPF \gg \text{forwardingTable}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPF \gg \text{forwardingTable}'^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPF \gg \text{source?} \in \text{NUM} \\
& \wedge PPF \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPF \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPF \gg \text{target} \text{ isBusyIn } PPF \gg \text{calls} \\
& \wedge \neg (\forall \text{forwardingNumber} : \text{NUM} \bullet \neg (\\
& \quad \langle PPF \gg \text{target?}, \text{forwardingNumber} \rangle \in (PPF \gg \text{forwardingTable}^+) \\
& \quad \wedge \text{forwardingNumber} \text{ isFreeIn } PPF \gg \text{calls} \\
& \quad \wedge \text{forwardingNumber} \neq PPF \gg \text{source?} \\
& \quad \wedge PPF \gg \text{calls}' \in \{P \gg \text{calls}' \cup \{PPF \gg \text{source?} \mapsto \text{forwardingNumber}\}\})) \\
& \wedge P \gg \text{connection!} \in \{\text{failed}\} \wedge PPF \gg \text{connection!} \in \{\text{successful}\}
\end{aligned}$$

8.6 Integrating the Holding and Forwarding Services

Finally, we consider the system in which both holding and forwarding services can be used. The integration of these services requires an arbitrary decision as to their precedence. In this example, if a subscriber to both services is found to be busy, we will assume that a call will be held, but that the caller will be given an option to have their call forwarded (if the forwarding number

is not busy).

8.6.1 Specifying the Extended Machine

The machine *POTSPlusHoldAndForward*, below, shows how all three of the previous machines (defined in sections 8.3, 8.4.1, and 8.5.1) can be extended to incorporate the behaviour of both the holding and forwarding service.

```
MACHINE POTSPlusHoldAndForward
TYPE machine
SECTION POTSPlusHold, POTSPlusForward
STATE
```

```
    POTSPlusHold.state
    POTSPlusForward.state
```

```
INITIALIZATION
```

```
    POTSPlusHold.initialization
    POTSPlusForward.initialization
```

```
OPERATION connect  $\hat{=}$ 
INPUTS
```

```
    source? : NUM
    target? : NUM
```

```
OUTPUTS
```

```
    connection! : CALL_SUCCESS
```

```
PRE
```

```
    source? isFreeIn calls
```

POST

$$\text{forwardingNumber} : \text{NUM}$$

$$\begin{aligned} & (\text{target? isFreeIn calls} \wedge \text{source?} \neq \text{target?}) \\ & \Rightarrow (\text{calls}' = \text{calls} \cup \{\text{source?} \mapsto \text{target?}\} \\ & \quad \wedge \text{connection!} = \text{successful}) \end{aligned}$$

$$\begin{aligned} & (\text{target? isBusyIn calls} \wedge \text{source?} \neq \text{target?} \\ & \wedge \text{target?} \in \text{holdingTable} \\ & \wedge (\text{target?} \notin \text{dom forwardingTable} \\ & \quad \vee ((\text{target?}, \text{forwardingNumber}) \in \text{forwardingTable}^+ \\ & \quad \wedge \text{forwardingNumber isBusyIn calls})) \\ & \Rightarrow (\text{calls}' = \text{calls} \wedge \text{connection!} = \text{held}) \end{aligned}$$

$$\begin{aligned} & (\text{target? isBusyIn calls} \wedge \text{source?} \neq \text{target?} \\ & \wedge \text{target?} \notin \text{holdingTable} \\ & \wedge (\text{target?}, \text{forwardingNumber}) \in \text{forwardingTable}^+ \\ & \wedge \text{forwardingNumber isFreeIn calls} \\ & \wedge \text{forwardingNumber} \neq \text{source?}) \\ & \Rightarrow (\text{calls}' = \text{calls} \cup \{\text{source?} \mapsto \text{forwardingNumber}\} \\ & \quad \wedge \text{connection!} = \text{successful}) \end{aligned}$$

$$\begin{aligned} & (\text{target? isBusyIn calls} \wedge \text{source?} \neq \text{target?} \\ & \wedge \text{target?} \in \text{holdingTable} \\ & \wedge (\text{target?}, \text{forwardingNumber}) \in \text{forwardingTable}^+ \\ & \wedge \text{forwardingNumber isFreeIn calls} \\ & \wedge \text{forwardingNumber} \neq \text{source?}) \\ & \Rightarrow (\text{calls}' = \text{calls} \wedge \text{connection!} = \text{optional_forward}) \end{aligned}$$

$$\begin{aligned} & ((\text{target? isBusyIn calls} \\ & \quad \wedge (\text{target?} \notin \text{holdingTable} \\ & \quad \vee ((\text{target?}, \text{forwardingNumber}) \in \text{forwardingTable}^+ \\ & \quad \wedge (\text{forwardingNumber isBusyIn calls} \\ & \quad \quad \vee \text{forwardingNumber} = \text{source?})))) \\ & \vee \text{source?} = \text{target?}) \\ & \Rightarrow (\text{calls}' = \text{calls} \wedge \text{connection!} = \text{failed}) \end{aligned}$$

$$\begin{aligned} \text{holdingTable}' & = \text{holdingTable} \\ \text{forwardingTable}' & = \text{forwardingTable} \end{aligned}$$

END *connect*

OPERATION *disconnect* $\hat{=}$
 INPUTS

source? : NUM

PRE

POTSPlusHold.disconnect.pre

POST

POTSPlusHold.disconnect.post

forwardingTable' = *forwardingTable*

END *disconnect*

OPERATION *registerForForward* $\hat{=}$
 INPUTS

regSource? : NUM

regTarget? : NUM

POST

POTSPlusForward.registerForForward.post

holdingTable' = *holdingTable*

END *registerForForward*

OPERATION *cancelForward* $\hat{=}$
 INPUTS

regNumber? : NUM

PRE

POTSPlusForward.cancelForward.pre

POST

POTSPlusForward.cancelForward.post

holdingTable' = holdingTable

END *cancelForward*

OPERATION *registerForHold* $\hat{=}$
 INPUTS

regNumber? : NUM

PRE

POTSPlusHold.registerForHold.pre

POST

POTSPlusHold.registerForHold.post

forwardingTable' = forwardingTable

END *registerForHold*

OPERATION *cancelHold* $\hat{=}$
 INPUTS

regNumber? : NUM

PRE

POTSPlusHold.cancelHold.pre

POST

POTSPlusHold.cancelHold.post

forwardingTable' = forwardingTable

END *cancelHold*

END *POTSPlusHoldAndForward*

On this occasion we define our machine to have two parent sections: *POTSPlusHold* and *POTSPlusForward*. This allows us to inherit behaviour from both machines (and indirectly *POTS* as well).

The state of the integrated machine is formed from the conjunction of the schemas retrieved from the state clauses of the machines which individually define the holding and forwarding services. Note that, schema conjunction will eliminate any duplicate declarations and that as we have declared previously, these schemas' references to *POTS.state* are not identical as copies are made when clauses are imported from another machine. Similarly, the initialization is the conjunction of their respective initialization schemas. The behaviour of the *disconnect*, *registerForForward*, *cancelForward*, *registerForHold* and *cancelHold* operations is also directly inherited (with small extensions to show that they do not cause changes to any other state) from the parent machines.

This leaves only the *connect* operation. The operation declares the same inputs and outputs as all the other versions of the operation. The precondition states that the caller must be free in order to make a call, the remainder of the operation's behaviour is summarized in table 8.2 below.

8.6.2 Specifying the Relationship: POTS with Holding Service to Integrated Machine

There are three retrenchment relationships involving the machine in which both the holding and forwarding services are implemented (we shall refer to this machine as the integrated machine). We consider first the retrenchment between the machine that implements solely the holding service and the integrated machine. This relationship will allow us to prove that the integrated model mirrors the behaviour of the machine that implements solely the holding service, except in circumstances made clear in the specification of the

Table 8.2: Behaviour of connect operation in *POTSPlusHoldAndForward*

Condition	Result
Callee is free.	A connection is made, success is reported.
Callee is busy. Called number is not caller's number. Callee registered for holding service. Callee not registered for forwarding service.	No connection made, caller held.
Callee is busy. Called number is not caller's number. Callee not registered for holding service. Callee registered for forwarding service. Forwarding number free.	A connection is made to forwarding number, success is reported.
Callee is busy. Called number is not caller's number. Callee registered for holding service. Callee registered for forwarding service. Forwarding number busy.	No connection made, caller held.
Callee is busy. Called number is not caller's number. Callee registered for holding service. Callee registered for forwarding service. Forwarding number free.	No connection made, caller held and given option to be put through to forwarding number.
All other circumstances.	No connection made, failure is reported.

retrenchment. The definition of the relationship is given below.

```

RELATIONSHIP POTSPlusHold_to_POTSPlusHoldAndForward
TYPE retrenchment
FROM POTSPlusHold AS PPH
TO POTSPlusHoldAndForward AS PPHF

```

RETRIEVE

$$\begin{aligned} PPH \gg \text{calls} &= PPHF \gg \text{calls} \\ PPH \gg \text{holdingTable} &= PPHF \gg \text{holdingTable} \end{aligned}$$

RAMIFICATIONS *connect* $\hat{=}$
WITHIN

$$\begin{aligned} PPH \gg \text{source?} &= PPHF \gg \text{source?} \\ PPH \gg \text{target?} &= PPHF \gg \text{target?} \end{aligned}$$

OUTPUT

$$PPH \gg \text{connection!} = PPHF \gg \text{connection!}$$

CONCEDES

$$\begin{aligned} &\exists \text{ forwardingNumber} : \text{NUM} \\ &\bullet (PPH \gg \text{holdingTable} = PPHF \gg \text{holdingTable} \\ &\quad \wedge PPHF \gg \text{target?} \text{ isBusyIn } PPHF \gg \text{calls} \\ &\quad \wedge (PPHF \gg \text{target?}, \text{ forwardingNumber}) \in \text{ forwardingTable}^+ \\ &\quad \wedge \text{ forwardingNumber} \text{ isFreeIn } \text{calls} \\ &\quad \wedge \text{ forwardingNumber} \neq PPHF \gg \text{source?}) \\ &(PPHF \gg \text{target?} \notin PPHF \gg \text{holdingTable} \\ &\wedge PPHF \gg \text{calls}' = \\ &\quad PPH \gg \text{calls}' \cup \{ PPHF \gg \text{source?} \mapsto \text{ forwardingNumber} \} \\ &\wedge PPH \gg \text{connection!} = \text{failed} \\ &\wedge PPHF \gg \text{connection!} = \text{successful}) \\ &\vee \\ &(PPHF \gg \text{target?} \in PPHF \gg \text{holdingTable} \\ &\wedge PPH \gg \text{calls}' = PPHF \gg \text{calls}' \\ &\wedge PPH \gg \text{connection!} = \text{held} \\ &\wedge PPHF \gg \text{connection!} = \text{optional_forward}) \end{aligned}$$

END *connect*

RAMIFICATIONS *disconnect* $\hat{=}$
WITHIN

$$PPH \gg \text{source?} = PPHF \gg \text{source?}$$

END *disconnect*

RAMIFICATIONS *registerForHold* $\hat{=}$
WITHIN

$$PPH \gg \text{regNumber?} = PPHF \gg \text{regNumber?}$$

END *registerForHold*

RAMIFICATIONS *cancelHold* $\hat{=}$
WITHIN

$$PPH \gg \text{regNumber?} = PPHF \gg \text{regNumber?}$$

END *cancelHold*

END *POTSPlusHold_to_POTSPlusHoldAndForward*

In the retrieve clause we specify that the set of current calls, and the set of numbers registered for the holding service in each machine should be identical. We then consider the exceptions to the general rule by specifying the ramifications for each operation. No ramifications are required for the operations concerning the administration of the forwarding table as these operations were not declared in the *POTSPlusHold* machine.

The *disconnect*, *registerForHold*, and *cancelHold* operations are identical in both machines, and it is necessary, therefore, only to ensure that the inputs to the operations in both machines are identical. This is done with the within clause of the ramifications for each operation.

The *connect* operation in the integrated machine, however, introduces the concept of the forwarding service to the behaviour specified in the *POTSPlusHold* machine. We must, therefore, identify the circumstances in which the behaviour of the two operations differs, and specify the correct behaviour in these circumstances in the concedes clause for the operation's ramifications. Examining the specifications of the two machines, we can see that the behaviour will differ when the callee is busy, they have registered for the forwarding service, and the forwarding number is free. Beyond that,

there are two circumstances to consider. Firstly, we examine the situation where the callee has also registered for the holding service; in this instance the set of current calls will not differ, but the user of the integrated machine will receive a hold message giving the option to be forwarded (rather than simply being held). Secondly, we examine the situation where the callee has not also registered for the holding service; in this instance the call will be connected to the forwarding number when using the integrated machine, and a success message will be generated (again, rather than simply being held). It is also necessary to specify in the concedes clauses that even in the exceptional circumstances, the set of numbers registered for the holding service is consistent between machines (despite the fact that it is unchanged in the operation of both machines).

8.6.3 Syntax Checking the Relationship

Once we had created the specification of the relationship, we used Frog to ensure that it was syntactically consistent. Again, some initial errors were discovered in our specifications that Frog caught and aided us in resolving. Frog accepted our construct's specification and produced its equivalent, typed abstract syntax tree. Again we will just present the typed version of the paragraph associated with the concedes clause. As in the previous sections, we do not show the version annotated with every expression's type, but just the version that shows the paragraph's type. This is displayed below.

$$\begin{aligned}
 \text{AX } & [_{POTSPplusHold_to_POTSPplusHoldAndForward_connect_concedes} : \\
 & \{[_{POTSPplusHoldAndForward_connect_outputs} \\
 & \wedge _POTSPplusHold_connect_outputs \\
 & \wedge _POTSPplusHoldAndForward_connect_inputs \\
 & \wedge _POTSPplusHold_connect_inputs \\
 & \wedge _POTSPplusHoldAndForward_state' \wedge _POTSPplusHold_state' \\
 & \wedge _POTSPplusHoldAndForward_state \wedge _POTSPplusHold_state} |
 \end{aligned}$$

and *cancelHold*. As with *disconnect* the retrenchment of the operations over the holding table is inherently correct as in both cases the operation is identical in source and target machine. Once again, the proof obligations generated were large in size and mechanical verification with Isabelle/ZF was impossible within our project's time constraints. Close, manual examination of the proof obligations however, could allow the informed observer to conclude that the relationship was valid. Should sufficient time be available, a pen-and-paper proof could fairly easily be presented. An abbreviated version of the correctness proof obligation for the *connect* operation of our relationship is presented below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}^\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \spadesuit CALL_SUCCESS \in \mathbb{P}^\infty \\
& CALL_SUCCESS \in \mathbb{P} \spadesuit CALL_SUCCESS \\
& \vdash? \\
& \forall _u _v _v' _i _j _p \bullet \\
& \langle _v, _v', _j, _p \rangle \in \{ \langle calls, holdingTable, forwardingTable \rangle, \\
& \quad \langle calls', holdingTable', forwardingTable' \rangle, \langle source?, target? \rangle, connection! \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \quad \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \quad \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& \exists forwardingNumber \bullet (\\
& \quad calls \in NUM \rightsquigarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge calls' \in NUM \rightsquigarrow NUM \wedge \text{dom } calls' \cap \text{ran } calls' \in \{\emptyset\} \\
& \quad \wedge holdingTable \in \mathbb{P}(NUM) \\
& \quad \wedge forwardingTable \in NUM \rightsquigarrow NUM \\
& \quad \wedge forwardingTable' \in NUM \rightsquigarrow NUM \\
& \quad \wedge (forwardingTable^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge (forwardingTable'^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge source? \in NUM \wedge target? \in NUM \\
& \quad \wedge connection! \in CALL_SUCCESS \\
& \quad \wedge forwardingNumber \in NUM
\end{aligned}$$

$$\begin{aligned}
& \neg (\text{target? isFreeIn calls} \wedge \text{source} \neq \text{target?} \wedge \neg (\\
& \text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{target?}\} \wedge \text{connection!} \in \{\text{successful}\}\}) \\
& \wedge \neg (\text{target? isBusyIn calls} \wedge \text{source?} \neq \text{target?} \\
& \wedge \text{target?} \in \text{holdingTable} \wedge \neg (\neg \text{target?} \in \text{dom forwardingTable}) \\
& \wedge \neg (\langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+)) \\
& \wedge \text{forwardingNumber isBusyIn calls}) \wedge \neg (\text{calls}' \in \{\text{calls}\} \\
& \wedge \text{connection!} \in \{\text{held}\})) \wedge \neg (\text{target? isBusyIn} \wedge \\
& \wedge \text{source?} \neq \text{target?} \wedge \text{target?} \notin \text{holdingTable} \\
& \wedge \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+)) \\
& \wedge \text{forwardingNumber isFreeIn calls} \wedge \text{forwardingNumber} \neq \text{source?} \\
& \wedge \neg (\text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{forwardingNumber}\} \\
& \wedge \text{connection!} = \text{successful}\}) \wedge \neg (\text{target? isBusyIn calls} \\
& \wedge \text{source?} \neq \text{target?} \wedge \text{target?} \in \text{holdingTable} \\
& \wedge \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+)) \\
& \wedge \text{forwardingNumber isFreeIn calls} \wedge \text{forwardingNumber} \neq \text{source?} \\
& \wedge \neg (\text{calls}' \in \{\text{calls}\} \wedge \text{connection!} \in \{\text{optional_forward}\})) \\
& \wedge \neg (\neg (\neg (\text{target? isBusyIn calls} \wedge \neg (\neg (\text{target?} \notin \text{holdingTable})) \\
& \wedge \neg (\langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+)) \\
& \wedge \neg (\neg (\text{forwardingNumber isBusyIn calls})) \\
& \wedge \neg (\text{forwardingNumber} \in \{\text{source?}\})))))) \\
& \wedge \neg (\text{calls}' \in \{\text{calls}\} \wedge \text{connection!} \in \{\text{failed}\})) \\
& \wedge \text{holdingTable}' \in \{\text{holdingTable}\}\} \\
& \wedge \text{forwardingTable}' \in \{\text{forwardingTable}\}\} \\
& \wedge \langle _u, _v \rangle \in \{\langle \langle \text{PPH} \gg \text{calls}, \text{PPH} \gg \text{holdingTable} \rangle, \\
& \langle \text{PPHF} \gg \text{calls}, \text{PPHF} \gg \text{holdingTable}, \text{PPHF} \gg \text{forwardingTable} \rangle \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \bullet \\
& \text{PPH} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPH} \gg \text{calls} \cap \text{ran PPH} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPHF} \gg \text{calls} \cap \text{ran PPHF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPH} \gg \text{holdingTable} \in \text{NUM} \wedge \text{PPHF} \gg \text{holdingTable} \in \text{NUM} \\
& \wedge \text{PPHF} \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{PPHF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPH} \gg \text{calls} \in \{\text{PPHF} \gg \text{calls}\} \\
& \wedge \text{PPH} \gg \text{holdingTable} \in \{\text{PPHF} \gg \text{holdingTable}\}
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v, _i _j \rangle \in \{ \langle \langle PPH \rangle \rangle \text{calls}, PPH \rangle \rangle \text{holdingTable}, \\
& \quad \langle PPHF \rangle \rangle \text{calls}, PPHF \rangle \rangle \text{holdingTable}, PPHF \rangle \rangle \text{forwardingTable} \}, \\
& \quad \langle PPH \rangle \rangle \text{source?}, PPH \rangle \rangle \text{target?} \}, \langle PPHF \rangle \rangle \text{source?}, PPHF \rangle \rangle \text{target?} \} \rangle \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \quad \times (\mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM))) \\
& \quad \times (\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad PPH \rangle \rangle \text{calls} \in NUM \mapsto NUM \\
& \quad \wedge \text{dom } PPH \rangle \rangle \text{calls} \cap \text{ran } PPH \rangle \rangle \text{calls} \in \{\emptyset\} \\
& \quad \wedge PPH \rangle \rangle \text{holdingTable} \in NUM \\
& \quad \wedge PPHF \rangle \rangle \text{calls} \in NUM \mapsto NUM \\
& \quad \wedge \text{dom } PPHF \rangle \rangle \text{calls} \cap \text{ran } PPHF \rangle \rangle \text{calls} \in \{\emptyset\} \\
& \quad \wedge PPHF \rangle \rangle \text{holdingTable} \in NUM \\
& \quad \wedge PPHF \rangle \rangle \text{forwardingTable} \in NUM \mapsto NUM \\
& \quad \wedge (PPHF \rangle \rangle \text{forwardingTable}^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge PPH \rangle \rangle \text{source?} \in NUM \wedge PPH \rangle \rangle \text{target?} \in NUM \wedge PPHF \rangle \rangle \text{source?} \in NUM \\
& \quad \wedge PPHF \rangle \rangle \text{target?} \in NUM \wedge PPH \rangle \rangle \text{source?} \in \{PPHF \rangle \rangle \text{source?}\} \\
& \quad \wedge PPH \rangle \rangle \text{target?} \in \{PPHF \rangle \rangle \text{target?}\} \\
& \wedge \langle _u, _i \rangle \in \{ \langle \langle \text{calls}, \text{holdingTable} \rangle \rangle, \langle \text{source?}, \text{target?} \rangle \} \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \times (\spadesuit NUM \times \spadesuit NUM) \bullet \\
& \quad \text{calls} \in NUM \mapsto NUM \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \quad \wedge \text{holdingTable} \in \mathbb{P} NUM \wedge \text{source?} \in NUM \wedge \text{target?} \in NUM \\
& \quad \wedge \text{source?} \text{ isFreeIn } \text{calls} \} \\
& \Rightarrow \\
& \exists _u' _o \bullet \langle _u, _u', _i, _o \rangle \in \{ \langle \langle \text{calls}, \text{holdingTable} \rangle \rangle, \\
& \quad \langle \text{calls}', \text{holdingTable}' \rangle, \langle \text{source?}, \text{target?} \rangle, \text{connection!} \rangle \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM) \\
& \quad \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P} \spadesuit NUM) \\
& \quad \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& \quad \text{calls} \in NUM \mapsto NUM \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \quad \wedge \text{calls}' \in NUM \mapsto NUM \wedge \text{dom } \text{calls}' \cap \text{ran } \text{calls}' \in \{\emptyset\} \\
& \quad \wedge \text{holdingTable} \in \mathbb{P}(NUM) \wedge \text{holdingTable}' \in \mathbb{P}(NUM) \\
& \quad \wedge \text{source?} \in NUM \wedge \text{target?} \in NUM \wedge \text{connection!} \in CALL_SUCCESS \\
& \quad \neg (\text{target?} \text{ isFreeIn } \text{calls} \wedge \text{source?} \neq \text{target?} \wedge \neg (\\
& \quad \text{calls}' \in \{ \text{calls} \cup \{ \text{source?} \mapsto \text{target?} \} \} \wedge \text{connection!} \in \{ \text{successful} \} \} \} \\
& \quad \wedge \neg (\neg (\neg (\text{target?} \text{ isBusyIn } \text{calls}) \wedge \neg (\text{source?} \in \{ \text{target?} \} \}))) \\
& \quad \wedge \text{target?} \notin \text{holdingTable} \wedge \neg (\text{calls}' \in \{ \text{calls} \} \wedge \text{connection!} \in \{ \text{held} \} \} \\
& \quad \wedge \neg (\neg (\neg (\text{target?} \text{ isBusyIn } \text{calls}) \wedge \neg (\text{source?} \in \{ \text{target?} \} \}))) \\
& \quad \wedge \text{target?} \notin \text{holdingTable} \wedge \neg (\text{calls}' \in \{ \text{calls} \} \wedge \text{connection} \in \{ \text{failed} \} \} \\
& \quad \wedge \text{holdingTable}' \in \{ \text{holdingTable} \} \}
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u', _v' \rangle \in \{ \langle \langle PPH \rangle\!\rangle \text{calls}, PPH \rangle\!\rangle \text{holdingTable} \rangle, \\
& \quad \langle PPHF \rangle\!\rangle \text{calls}, PPHF \rangle\!\rangle \text{holdingTable}, PPHF \rangle\!\rangle \text{forwardingTable} \rangle \rangle \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \quad \times (\mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM))) \bullet \\
& PPH \rangle\!\rangle \text{calls} \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPH \rangle\!\rangle \text{calls} \cap \text{ran } PPH \rangle\!\rangle \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \rangle\!\rangle \text{calls} \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPHF \rangle\!\rangle \text{calls} \cap \text{ran } PPHF \rangle\!\rangle \text{calls} \in \{\emptyset\} \\
& \wedge PPH \rangle\!\rangle \text{holdingTable} \in NUM \wedge PPHF \rangle\!\rangle \text{holdingTable} \in NUM \\
& \wedge PPHF \rangle\!\rangle \text{forwardingTable} \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \rangle\!\rangle \text{forwardingTable}^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \wedge PPH \rangle\!\rangle \text{calls} \in \{ PPHF \rangle\!\rangle \text{calls} \} \\
& \wedge PPH \rangle\!\rangle \text{holdingTable} \in \{ PPHF \rangle\!\rangle \text{holdingTable} \} \\
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \\
& \quad \langle \langle PPH \rangle\!\rangle \text{calls}, PPH \rangle\!\rangle \text{holdingTable} \rangle, \\
& \quad \langle PPHF \rangle\!\rangle \text{calls}, PPHF \rangle\!\rangle \text{holdingTable}, PPHF \rangle\!\rangle \text{forwardingTable} \rangle, \\
& \quad \langle PPH \rangle\!\rangle \text{calls}', PPH \rangle\!\rangle \text{holdingTable}' \rangle, \\
& \quad \langle PPHF \rangle\!\rangle \text{calls}', PPHF \rangle\!\rangle \text{holdingTable}', PPHF \rangle\!\rangle \text{forwardingTable}' \rangle, \\
& \quad \langle PPH \rangle\!\rangle \text{source?}, PPH \rangle\!\rangle \text{target?} \rangle, \langle PPHF \rangle\!\rangle \text{source?}, PPHF \rangle\!\rangle \text{target?} \rangle, \\
& \quad PPH \rangle\!\rangle \text{connection!}, PPHF \rangle\!\rangle \text{connection!} \rangle \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \quad \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \quad \times \mathbb{P}(\spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \\
& \quad \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \\
& \quad \times \spadesuit CALL_SUCCESS \times \spadesuit CALL_SUCCESS \bullet \\
& PPH \rangle\!\rangle \text{calls} \in NUM \leftrightarrow NUM \wedge \\
& \text{dom } PPH \rangle\!\rangle \text{calls} \cap \text{ran } PPH \rangle\!\rangle \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \rangle\!\rangle \text{calls} \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPHF \rangle\!\rangle \text{calls} \cap \text{ran } PPHF \rangle\!\rangle \text{calls} \in \{\emptyset\} \\
& PPH \rangle\!\rangle \text{calls}' \in NUM \leftrightarrow NUM \wedge \\
& \text{dom } PPH \rangle\!\rangle \text{calls}' \cap \text{ran } PPH \rangle\!\rangle \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \rangle\!\rangle \text{calls}' \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPHF \rangle\!\rangle \text{calls}' \cap \text{ran } PPHF \rangle\!\rangle \text{calls}' \in \{\emptyset\} \\
& \wedge PPH \rangle\!\rangle \text{holdingTable} \in \mathbb{P}(NUM) \wedge PPH \rangle\!\rangle \text{holdingTable}' \in \mathbb{P}(NUM) \\
& \wedge PPHF \rangle\!\rangle \text{holdingTable} \in \mathbb{P}(NUM) \wedge PPHF \rangle\!\rangle \text{holdingTable}' \in \mathbb{P}(NUM) \\
& \wedge PPHF \rangle\!\rangle \text{forwardingTable} \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \rangle\!\rangle \text{forwardingTable}^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \wedge PPHF \rangle\!\rangle \text{forwardingTable}' \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \rangle\!\rangle \text{forwardingTable}'^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \wedge PPH \rangle\!\rangle \text{source?} \in NUM \wedge PPH \rangle\!\rangle \text{target?} \in NUM \\
& \wedge PPHF \rangle\!\rangle \text{source?} \in NUM \wedge PPHF \rangle\!\rangle \text{target?} \in NUM \\
& \wedge PPH \rangle\!\rangle \text{connection!} \in CALL_SUCCESS \\
& \wedge PPHF \rangle\!\rangle \text{connection!} \in CALL_SUCCESS \\
& \wedge PPH \rangle\!\rangle \text{connection!} \in \{ PPHF \rangle\!\rangle \text{connection!} \} \\
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \\
& \quad \langle \langle PPH \gg \text{calls}, PPH \gg \text{holdingTable} \rangle, \\
& \quad \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, PPHF \gg \text{forwardingTable} \rangle, \\
& \quad \langle PPH \gg \text{calls}', PPH \gg \text{holdingTable}' \rangle, \\
& \quad \langle PPHF \gg \text{calls}', PPHF \gg \text{holdingTable}', PPHF \gg \text{forwardingTable}' \rangle, \\
& \quad \langle PPH \gg \text{source?}, PPH \gg \text{target?} \rangle, \langle PPHF \gg \text{source?}, PPHF \gg \text{target?} \rangle, \\
& \quad PPH \gg \text{connection!}, PPHF \gg \text{connection!} \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \mathbb{P}(\spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \\
& \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& PPH \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \\
& \text{dom } PPH \gg \text{calls} \cap \text{ran } PPH \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& PPH \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \\
& \text{dom } PPH \gg \text{calls}' \cap \text{ran } PPH \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls}' \cap \text{ran } PPHF \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPH \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPH \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge PPHF \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPHF \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{forwardingTable}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}'^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge PPH \gg \text{source?} \in \text{NUM} \wedge PPH \gg \text{target?} \in \text{NUM} \\
& \wedge PPHF \gg \text{source?} \in \text{NUM} \wedge PPHF \gg \text{target?} \in \text{NUM} \\
& \wedge PPH \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPHF \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge \neg (\forall \text{forwardingNumber} : \text{NUM} \bullet \neg (\neg (\neg \\
& PPH \gg \text{holdingTable} \in \{PPHF \gg \text{holdingTable}\} \\
& \wedge PPHF \gg \text{target?} \text{ isBusyIn } PPHF \gg \text{calls} \\
& \wedge \langle PPHF \gg \text{target?}, \text{forwardingNumber} \rangle \in (PPHF \gg \text{forwardingTable}^+) \\
& \wedge \text{forwardingNumber} \text{ isFreeIn } PPHF \gg \text{calls} \\
& \wedge \text{forwardingNumber} \neq PPHF \gg \text{source?} \\
& \wedge PPHF \gg \text{target?} \notin PPHF \gg \text{holdingTable} \\
& \wedge PPHF \gg \text{calls}' \in \{PPH \gg \text{calls}'\} \\
& \cup \{PPHF \gg \text{source?} \mapsto \text{forwardingNumber}\}) \\
& \wedge PPH \gg \text{connection} \in \{\text{failed}\} \wedge PPHF \gg \text{connection} \in \{\text{successful}\}) \\
& \wedge \neg (PPH \gg \text{holdingTable} \notin \{PPHF \gg \text{holdingTable}\} \\
& \wedge PPH \gg \text{calls}' \in \{PPHF \gg \text{calls}'\} \\
& \wedge PPH \gg \text{connection} \in \{\text{held}\} \\
& \wedge PPHF \gg \text{connection} \in \{\text{optional_forward}\})
\end{aligned}$$

8.6.4 Specifying the Relationship: POTS with Forwarding Service to Integrated Machine

We consider next the retrenchment between the machine that implements solely the forwarding service and the integrated machine. This relationship will allow us to prove that the integrated model mirrors the behaviour of the machine that implements solely the forwarding service, except in circumstances made clear in the specification of the retrenchment. The definition of the relationship is given below.

RELATIONSHIP *POTSPlusForward_to_POTSPlusHoldAndForward*
 TYPE *retrenchment*
 FROM *POTSPlusForward* AS *PPF*
 TO *POTSPlusHoldAndForward* AS *PPHF*
 RETRIEVE

$$PPF \gg \text{calls} = PPHF \gg \text{calls}$$

$$PPF \gg \text{forwardingTable} = PPHF \gg \text{forwardingTable}$$

RAMIFICATIONS *connect* $\hat{=}$
 WITHIN

$$PPF \gg \text{source?} = PPHF \gg \text{source?}$$

$$PPF \gg \text{target?} = PPHF \gg \text{target?}$$

OUTPUT

$$PPF \gg \text{connection!} = PPHF \gg \text{connection!}$$

CONCEDES

$PPF \gg forwardingTable = PPHF \gg forwardingTable$
 $PPHF \gg target? \text{ isBusyIn } PPHF \gg calls$
 $PPHF \gg target? \in PPHF \gg holdingTable$

$(\exists forwardingNumber : NUM$
 $\bullet ((PPHF \gg target?, forwardingNumber) \in forwardingTable^+$
 $\wedge forwardingNumber \text{ isFreeIn } PPHF \gg calls$
 $\wedge forwardingNumber \neq PPHF \gg source?$
 $\wedge PPF \gg calls' =$
 $PPHF \gg calls' \cup \{PPHF \gg source? \mapsto forwardingNumber\})$
 $\wedge PPF \gg connection! = successful$
 $\wedge PPHF \gg connection! = optional_forward)$
 \vee
 $((PPHF \gg target? \notin \text{dom } PPHF \gg forwardingTable$
 $\vee (\forall forwardingNumber : NUM$
 $\bullet ((PPHF \gg target?, forwardingNumber) \in forwardingTable^+$
 $\Rightarrow forwardingNumber \text{ isBusyIn } PPHF \gg calls)))$
 $\wedge PPF \gg calls' = PPHF \gg calls'$
 $\wedge PPF \gg connection! = failed$
 $\wedge PPHF \gg connection! = held)$

END *connect*

RAMIFICATIONS *disconnect* $\hat{=}$
 WITHIN

$PPF \gg source? = PPHF \gg source?$

END *disconnect*

RAMIFICATIONS *registerForForward* $\hat{=}$
 WITHIN

$PPF \gg regSource? = PPHF \gg regSource?$
 $PPF \gg regTarget? = PPHF \gg regTarget?$

END *registerForForward*

RAMIFICATIONS *cancelForward* $\hat{=}$

WITHIN

$$PPF \gg \text{regNumber?} = PPHF \gg \text{regNumber?}$$

END *cancelForward*

END *POTSPlusForward_to_POTSPlusHoldAndForward*

In the retrieve clause we specify that the set of current calls and the data in the forwarding table for each machine should be identical. We then consider the exceptions to the general rule by specifying the ramifications for each operation. No ramifications are required for the operations concerning the administration of the holding table as these operations were not declared in the *POTSPlusForward* machine.

The *disconnect*, *registerForForward*, and *cancelForward* operations are identical in both machines, and it is necessary, therefore, only to ensure that the inputs to the operations in both machines are identical. This is done with the within clause of the ramifications for each operation.

The *connect* operation in the integrated machine, however, introduces the concept of the holding service to the behaviour specified in the *POTSPlusForward* machine. We must therefore, identify the circumstances in which the behaviour of the two operations differs, and specify the correct behaviour in these circumstances in the concedes clause for the operation's ramifications. Examining the specifications of the two machines, we can see that the behaviour will differ when the callee is busy, and they have registered for the holding service. Beyond that, there are two circumstances to consider. Firstly, we examine the situation where the forwarding number is busy; in this instance the set of current calls will not differ, but the user of the integrated machine will receive a hold message (rather than simply a failure message). Secondly, we examine the situation where the forwarding number is free; in this instance the user of the integrated machine would receive a message giving the option to be connected to the forwarding number (rather than being connected straight to the forwarding number). It is also necessary to specify in the concedes clauses that even in the exceptional circumstances,

the set of data in the forwarding table is consistent between machines (despite the fact that it is unchanged in the operation of both machines).

8.6.5 Syntax Checking the Relationship

Again, upon creation of the relationship's specification, we used Frog to ensure that it was syntactically consistent. Errors were uncovered by Frog in the type checking process and resolved with its assistance. Eventually, Frog accepted our construct's specification and produced its equivalent, typed abstract syntax tree. We will just present the typed version of the paragraph associated with the *concedes* clause. As in the previous sections, we do not show the version annotated with every expression's type, but just the version that shows the paragraph's type. This is displayed below.

$$\begin{aligned}
& \mathbf{AX} \ [_POTSPlusForward_to_POTSPlusHoldAndForward_connect_concedes : \\
& \{ _POTSPlusHoldAndForward_connect_outputs \\
& \wedge _POTSPlusForward_connect_outputs \\
& \wedge _POTSPlusHoldAndForward_connect_inputs \\
& \wedge _POTSPlusForward_connect_inputs \\
& \wedge _POTSPlusHoldAndForward_state' \wedge _POTSPlusForward_state' \\
& \wedge _POTSPlusHoldAndForward_state \wedge _POTSPlusForward_state \mid \\
& (((PPF \gg forwardingTable) \in (\{ PPHF \gg forwardingTable \})) \\
& \wedge (((PPHF \gg target?, PPHF \gg calls) \in (\boxtimes \text{isBusyIn} \boxtimes))) \\
& \wedge ((PPHF \gg target?) \in (PPHF \gg holdingTable))) \\
& \wedge (\neg ((\neg (\neg (\forall ([forwardingNumber : NUM]) \bullet \\
& \quad (\neg (((((((PPHF \gg target?, forwardingNumber) \\
& \quad \quad \in (\boxtimes^+ PPHF \gg forwardingTable)))) \\
& \wedge (((forwardingNumber, PPHF \gg calls) \in (\boxtimes \text{isFreeIn} \boxtimes))) \\
& \wedge (((forwardingNumber, PPHF \gg source?) \in (\boxtimes \neq \boxtimes))) \\
& \wedge ((PPF \gg calls') \in (\{ (\boxtimes \cup \boxtimes (PPHF \gg calls', \\
& \quad \{ (\boxtimes \mapsto \boxtimes (PPHF \gg source?, forwardingNumber) \} \})))
\end{aligned}$$

$$\begin{aligned}
& \wedge ((PPF \gg \text{connection!}) \in (\{\text{successful}\})) \\
& \wedge ((PPHF \gg \text{connection!}) \in (\{\text{optional_forward}\})) \\
& \wedge (\neg (((\neg ((\neg (((PPHF \gg \text{target?}, (\text{dom } PPHF \gg \text{forwardingTable}))) \\
& \quad \in (\boxtimes \notin \boxtimes)))))) \\
& \wedge (\neg (\forall ([\text{forwardingNumber} : \text{NUM}]) \bullet \\
& \quad (\neg (((PPHF \gg \text{target?}, \text{forwardingNumber}) \\
& \quad \quad \in ((\boxtimes^+ PPHF \gg \text{forwardingTable})))) \\
& \quad \wedge (\neg (((\text{forwardingNumber}, PPHF \gg \text{calls}) \in (\boxtimes \text{isBusyIn } \boxtimes)))))) \\
& \wedge ((PPF \gg \text{calls}') \in (\{PPHF \gg \text{calls}'\})) \\
& \wedge ((PPF \gg \text{connection!}) \in (\{\text{failed}\})) \\
& \wedge ((PPHF \gg \text{connection!}) \in (\{\text{held}\})))] \\
& \text{END} \\
& \circ[_{\text{POTSPlusForward_to_POTSPlusHoldAndForward_connect_concedes}} : \\
& \quad \mathbb{P}([PPHF \gg \text{connection!} : \text{GIVEN CALL_SUCCESS}; \\
& \quad \quad PPHF \gg \text{connection!} : \text{GIVEN CALL_SUCCESS}; \\
& \quad \quad PPHF \gg \text{source?} : \text{GIVEN NUM}; PPHF \gg \text{target?} : \text{GIVEN NUM}; \\
& \quad \quad PPHF \gg \text{source?} : \text{GIVEN NUM}; PPHF \gg \text{target?} : \text{GIVEN NUM}; \\
& \quad \quad PPHF \gg \text{calls}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{holdingTable}' : \mathbb{P}(\text{GIVEN NUM}); \\
& \quad \quad PPHF \gg \text{calls}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{forwardingTable}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{calls} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{holdingTable} : \mathbb{P}(\text{GIVEN NUM}); \\
& \quad \quad PPHF \gg \text{calls} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{forwardingTable} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})))]
\end{aligned}$$

8.6.6 Verifying the Relationship

Once the relationship had been confirmed as syntactically correct, we progressed to the generation of the proof obligations for the construct. Frog generated these successfully by instantiating all of the generic proof obligations in the constructs configuration. For the operation environment level proof obligations this was done for all operation environments. In this instance we had four valid operation environments: *connect*, *disconnect*, *registerForForward* and *cancelForward*. Again, like *disconnect* the operations to alter the forwarding table were identical and their verification would be straightforward.

As can be seen in the proof obligation presented below, the increasing complexity of the relationships led to still larger proof obligations, and mechanical verification with Isabelle/ZF proved to be impossible within our project's time constraints. Close, manual examination of the proof obligations however, could allow the informed observer to conclude that the relationship was valid, and given enough space and time, a pen-and-paper proof could fairly easily be presented. An abbreviated version of the correctness proof obligation for the *connect* operation of our relationship is presented below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}^\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \spadesuit CALL_SUCCESS \in \mathbb{P}^\infty \\
& CALL_SUCCESS \in \mathbb{P} \spadesuit CALL_SUCCESS \\
& \vdash? \\
& \forall _u _v _v' _i _j _p \bullet \\
& \langle _v, _v', _j, _p \rangle \in \{ \langle \langle calls, holdingTable, forwardingTable \rangle, \\
& \quad \langle calls', holdingTable', forwardingTable' \rangle, \langle source?, target? \rangle, connection! \rangle \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& \exists forwardingNumber \bullet (\\
& \quad calls \in NUM \rightsquigarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge calls' \in NUM \rightsquigarrow NUM \wedge \text{dom } calls' \cap \text{ran } calls' \in \{\emptyset\} \\
& \quad \wedge holdingTable \in \mathbb{P}(NUM) \\
& \quad \wedge forwardingTable \in NUM \rightsquigarrow NUM \wedge forwardingTable' \in NUM \rightsquigarrow NUM \\
& \quad \wedge (forwardingTable^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge (forwardingTable'^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge source? \in NUM \wedge target? \in NUM \wedge connection! \in CALL_SUCCESS \\
& \quad \wedge forwardingNumber \in NUM \\
& \quad \neg (target? \text{ isFreeIn } calls \wedge source? \neq target? \wedge \neg (\\
& \quad \quad calls' \in \{calls \cup \{source? \mapsto target?\} \} \wedge connection! \in \{successful\})) \\
& \quad \wedge \neg (target? \text{ isBusyIn } calls \wedge source? \neq target? \\
& \quad \wedge target? \in holdingTable \wedge \neg (\neg target? \in \text{dom } forwardingTable) \\
& \quad \wedge \neg (\langle target?, forwardingNumber \rangle \in (forwardingTable^+)) \\
& \quad \wedge forwardingNumber \text{ isBusyIn } calls) \wedge \neg (calls' \in \{calls\})
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{connection!} \in \{\text{held}\}) \wedge \neg (\text{target? isBusyIn} \wedge \\
& \wedge \text{source?} \neq \text{target?} \wedge \text{target?} \notin \text{holdingTable} \\
& \wedge \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+) \\
& \wedge \text{forwardingNumber isFreeIn calls} \wedge \text{forwardingNumber} \neq \text{source?} \\
& \wedge \neg (\text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{forwardingNumber}\} \\
& \wedge \text{connection!} = \text{successful}\}) \wedge \neg (\text{target? isBusyIn calls} \\
& \wedge \text{source?} \neq \text{target?} \wedge \text{target?} \in \text{holdingTable} \\
& \wedge \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+) \\
& \wedge \text{forwardingNumber isFreeIn calls} \wedge \text{forwardingNumber} \neq \text{source?} \\
& \wedge \neg (\text{calls}' \in \{\text{calls}\} \wedge \text{connection!} \in \{\text{optional_forward}\}) \\
& \wedge \neg (\neg (\neg (\text{target? isBusyIn calls} \wedge \neg (\neg (\text{target?} \notin \text{holdingTable} \\
& \wedge \neg (\langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+) \\
& \wedge \neg (\neg (\text{forwardingNumber isBusyIn calls} \\
& \wedge \neg (\text{forwardingNumber} \in \{\text{source?}\})))))) \\
& \wedge \neg (\text{calls}' \in \{\text{calls}\} \wedge \text{connection!} \in \{\text{failed}\}) \\
& \wedge \text{holdingTable}' \in \{\text{holdingTable}\}) \\
& \wedge \text{forwardingTable}' \in \{\text{forwardingTable}\}) \\
& \wedge \langle _u, _v \rangle \in \{ \langle \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPHF} \gg \text{calls}, \text{PPHF} \gg \text{holdingTable}, \text{PPHF} \gg \text{forwardingTable} \rangle \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \bullet \\
& \text{PPF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPF} \gg \text{calls} \cap \text{ran PPF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPHF} \gg \text{calls} \cap \text{ran PPHF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{PPF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{holdingTable} \in \text{NUM} \\
& \wedge \text{PPHF} \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{PPHF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{calls} \in \{\text{PPHF} \gg \text{calls}\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \{\text{PPHF} \gg \text{forwardingTable}\} \\
& \wedge \langle _u, _v, _i, _j \rangle \in \{ \langle \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPHF} \gg \text{calls}, \text{PPHF} \gg \text{holdingTable}, \text{PPHF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPF} \gg \text{source?}, \text{PPF} \gg \text{target?} \rangle, \langle \text{PPHF} \gg \text{source?}, \text{PPHF} \gg \text{target?} \rangle \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet
\end{aligned}$$

$$\begin{aligned}
& PPF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPF \gg \text{calls} \cap \text{ran } PPF \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \text{NUM} \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge PPF \gg \text{source?} \in \text{NUM} \wedge PPF \gg \text{target?} \in \text{NUM} \wedge PPHF \gg \text{source?} \in \text{NUM} \\
& \wedge PPHF \gg \text{target?} \in \text{NUM} \wedge PPF \gg \text{source?} \in \{PPHF \gg \text{source?}\} \\
& \wedge PPF \gg \text{target?} \in \{PPHF \gg \text{target?}\} \\
& \wedge \langle _u, _i \rangle \in \{\langle \text{calls}, \text{holdingTable} \rangle, \langle \text{source?}, \text{target?} \rangle\} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{forwardingTable} \in \mathbb{P} \text{NUM} \wedge (\text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{source?} \text{ isFreeIn } \text{calls} \\
& \Rightarrow \\
& \exists _u' _o \bullet \langle _u, _u', _i, _o \rangle \in \{\langle \text{calls}, \text{forwardingTable} \rangle, \\
& \langle \text{calls}', \text{forwardingTable}' \rangle, \langle \text{source?}, \text{target?} \rangle, \text{connection!} \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& \exists \text{forwardingNumber} \bullet (\\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls}' \cap \text{ran } \text{calls}' \in \{\emptyset\} \\
& \wedge \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{forwardingTable}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge (\text{forwardingTable}'^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge \text{forwardingNumber} \in \text{NUM} \\
& \neg (\text{target?} \text{ isFreeIn } \text{calls} \wedge \text{source?} \neq \text{target?} \wedge \neg (\\
& \text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{target?}\} \wedge \text{connection!} \in \{\text{successful}\})) \\
& \wedge \neg (\text{target?} \text{ isBusyIn } \text{calls} \wedge \text{source?} \neq \text{target?} \\
& \wedge \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+) \\
& \wedge \text{forwardingNumber} \text{ isFreeIn } \text{calls} \wedge \text{forwardingNumber} \neq \text{source?} \\
& \neg (\text{calls}' \in \{\text{calls} \cup \{\text{source?} \mapsto \text{forwardingNumber}\}
\end{aligned}$$

$$\begin{aligned}
& \wedge \text{connection!} \in \{\text{successful}\}) \wedge \neg (\neg (\neg (\text{target? isBusyIn calls} \\
& \wedge \neg (\neg (\text{target?} \in \text{dom forwardingTable}) \wedge \neg (\\
& \langle \text{target?}, \text{forwardingNumber} \rangle \in (\text{forwardingTable}^+)) \\
& \wedge \neg (\neg (\text{forwardingNumber isBusyIn calls}) \\
& \wedge \neg (\text{forwardingNumber} \in \{\text{source?}\})))))) \\
& \wedge \neg (\text{source?} \in \{\text{target?}\}) \wedge \neg (\text{calls}' \in \{\text{calls}\} \\
& \wedge \text{connection!} \in \{\text{failed}\})) \\
& \wedge \text{forwardingTable}' \in \{\text{forwardingTable}\}) \\
\wedge \langle _u', _v' \rangle \in \{ \langle \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPHF} \gg \text{calls}, \text{PPHF} \gg \text{holdingTable}, \text{PPHF} \gg \text{forwardingTable} \rangle \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \bullet \\
& \text{PPF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPF} \gg \text{calls} \cap \text{ran PPF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPHF} \gg \text{calls} \cap \text{ran PPHF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{PPF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{holdingTable} \in \text{NUM} \\
& \wedge \text{PPHF} \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (\text{PPHF} \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge \text{PPF} \gg \text{calls} \in \{\text{PPHF} \gg \text{calls}\} \\
& \wedge \text{PPF} \gg \text{forwardingTable} \in \{\text{PPHF} \gg \text{forwardingTable}\} \\
\wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \\
& \langle \langle \text{PPF} \gg \text{calls}, \text{PPF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPHF} \gg \text{calls}, \text{PPHF} \gg \text{holdingTable}, \text{PPHF} \gg \text{forwardingTable} \rangle, \\
& \langle \text{PPF} \gg \text{calls}', \text{PPF} \gg \text{forwardingTable}' \rangle, \\
& \langle \text{PPHF} \gg \text{calls}', \text{PPHF} \gg \text{holdingTable}', \text{PPHF} \gg \text{forwardingTable}' \rangle, \\
& \langle \text{PPF} \gg \text{source?}, \text{PPF} \gg \text{target?} \rangle, \langle \text{PPHF} \gg \text{source?}, \text{PPHF} \gg \text{target?} \rangle, \\
& \text{PPF} \gg \text{connection!}, \text{PPHF} \gg \text{connection!} \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM})) \\
& \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& \text{PPF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \\
& \text{dom PPF} \gg \text{calls} \cap \text{ran PPF} \gg \text{calls} \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPHF} \gg \text{calls} \cap \text{ran PPHF} \gg \text{calls} \in \{\emptyset\} \\
& \text{PPF} \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \\
& \text{dom PPF} \gg \text{calls}' \cap \text{ran PPF} \gg \text{calls}' \in \{\emptyset\} \\
& \wedge \text{PPHF} \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom PPHF} \gg \text{calls}' \cap \text{ran PPHF} \gg \text{calls}' \in \{\emptyset\}
\end{aligned}$$

$$\begin{aligned}
& \wedge PPF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \wedge (PPF \gg forwardingTable^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPF \gg forwardingTable' \in NUM \leftrightarrow NUM \\
& \wedge (PPF \gg forwardingTable'^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPHF \gg holdingTable \in \mathbb{P}(NUM) \wedge PPHF \gg holdingTable' \in \mathbb{P}(NUM) \\
& \wedge PPHF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \gg forwardingTable^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPHF \gg forwardingTable' \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \gg forwardingTable'^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPF \gg source? \in NUM \wedge PPF \gg target? \in NUM \\
& \wedge PPHF \gg source? \in NUM \wedge PPHF \gg target? \in NUM \\
& \wedge PPF \gg connection! \in CALL_SUCCESS \\
& \wedge PPHF \gg connection! \in CALL_SUCCESS \\
& \wedge PPF \gg connection! \in \{PPHF \gg connection!\} \\
& \wedge \langle -u, -v, -u', -v', -i, -j, -o, -p \rangle \in \{ \\
& \quad \langle PPF \gg calls, PPF \gg forwardingTable \rangle, \\
& \quad \langle PPHF \gg calls, PPHF \gg holdingTable, PPHF \gg forwardingTable \rangle, \\
& \quad \langle PPF \gg calls', PPF \gg forwardingTable' \rangle, \\
& \quad \langle PPHF \gg calls', PPHF \gg holdingTable', PPHF \gg forwardingTable' \rangle, \\
& \quad \langle PPF \gg source?, PPF \gg target? \rangle, \langle PPHF \gg source?, PPHF \gg target? \rangle, \\
& \quad PPF \gg connection!, PPHF \gg connection! \} \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \\
& \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \\
& \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \times (\spadesuit NUM \times \spadesuit NUM) \times (\spadesuit NUM \times \spadesuit NUM) \\
& \times \spadesuit CALL_SUCCESS \times \spadesuit CALL_SUCCESS \bullet \\
& PPF \gg calls \in NUM \leftrightarrow NUM \wedge \\
& \text{dom } PPF \gg calls \cap \text{ran } PPF \gg calls \in \{\emptyset\} \\
& \wedge PPHF \gg calls \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPHF \gg calls \cap \text{ran } PPHF \gg calls \in \{\emptyset\} \\
& PPF \gg calls' \in NUM \leftrightarrow NUM \wedge \\
& \text{dom } PPF \gg calls' \cap \text{ran } PPF \gg calls' \in \{\emptyset\} \\
& \wedge PPHF \gg calls' \in NUM \leftrightarrow NUM \\
& \wedge \text{dom } PPHF \gg calls' \cap \text{ran } PPHF \gg calls' \in \{\emptyset\} \\
& \wedge PPF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \wedge (PPF \gg forwardingTable^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPF \gg forwardingTable' \in NUM \leftrightarrow NUM \\
& \wedge (PPF \gg forwardingTable'^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPHF \gg holdingTable \in \mathbb{P}(NUM) \wedge PPHF \gg holdingTable' \in \mathbb{P}(NUM)
\end{aligned}$$

$$\begin{aligned}
& \wedge PPHF \gg forwardingTable \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \gg forwardingTable^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPHF \gg forwardingTable' \in NUM \leftrightarrow NUM \\
& \wedge (PPHF \gg forwardingTable'^+) \cap id\ NUM \in \{\emptyset\} \\
& \wedge PPF \gg source? \in NUM \wedge PPF \gg target? \in NUM \\
& \wedge PPHF \gg source? \in NUM \wedge PPHF \gg target? \in NUM \\
& \wedge PPF \gg connection! \in CALL_SUCCESS \\
& \wedge PPHF \gg connection! \in CALL_SUCCESS \\
& \wedge PPF \gg forwardingTable \in \{PPHF \gg forwardingTable\} \\
& \wedge PPHF \gg target? \text{ isBusyIn } PPHF \gg calls \\
& \wedge PPHF \gg target? \in PPHF \gg holdingTable \\
& \wedge \neg (\neg (\neg (\forall forwardingNumber \in NUM \bullet \\
& \neg (\langle PPHF \gg target?, forwardingNumber \rangle \in (PPHF \gg forwardingTable^+)) \\
& \wedge forwardingNumber \text{ isFreeIn } PPHF \gg calls \\
& \wedge PPF \gg calls' \in \{PPHF \gg calls'\} \\
& \cup \{PPHF \gg source? \mapsto forwardingNumber\})) \\
& \wedge PPF \gg connection! \in \{successful\} \\
& \wedge PPHF \gg connection! \in \{optional_forward\})) \\
& \wedge \neg (\neg (\neg (PPHF \gg target? \notin \text{dom } PPHF \gg forwardingTable)) \\
& \wedge \neg (\forall forwardingNumber \in NUM \bullet \neg (\\
& (\langle PPHF \gg target?, forwardingNumber \rangle \in (PPHF \gg forwardingTable^+)) \\
& \wedge \neg (forwardingNumber \text{ isBusyIn } PPHF \gg calls)))) \\
& \wedge PPF \gg calls' \in \{PPHF \gg calls'\} \\
& \wedge PPF \gg connection! \in \{failed\} \wedge PPHF \gg connection! \in \{held\}
\end{aligned}$$

8.6.7 Specifying the Relationship: POTS to Integrated Machine

Finally, we consider the retrenchment between the machine modelling the plain old telephone service and the integrated machine. This relationship will allow us to prove that the integrated model mirrors the behaviour of the *POTS* machine, except in circumstances made clear in the specification of the retrenchment. The definition of the relationship is given below.

```

RELATIONSHIP POTS_to_POTSPlusHoldAndForward
TYPE retrenchment
FROM POTS AS P
TO POTSPlusHoldAndForward AS PPHF
RETRIEVE

```

$$P \gg \text{calls} = PPHF \gg \text{calls}$$

```

RAMIFICATIONS connect  $\hat{=}$ 
WITHIN

```

$$P \gg \text{source?} = PPHF \gg \text{source?}$$

$$P \gg \text{target?} = PPHF \gg \text{target?}$$

```

OUTPUT

```

$$P \gg \text{connection!} = PPHF \gg \text{connection!}$$

CONCEDES

$$\begin{aligned}
& PPHF \gg target? \text{ isBusyIn } PPHF \gg calls \\
& P \gg connection! = failed \\
& ((PPHF \gg target? \notin \text{dom } PPHF \gg forwardingTable \\
& \vee (\forall forwardingNumber : NUM \\
& \bullet ((PPHF \gg target?, forwardingNumber) \in forwardingTable^+ \\
& \Rightarrow forwardingNumber \text{ isBusyIn } PPHF \gg calls))) \\
& \wedge PPHF \gg target? \in holdingTable \\
& \wedge P \gg calls' = PPHF \gg calls' \\
& \wedge PPHF \gg connection! = held) \\
& \vee \\
& (\exists forwardingNumber : NUM \\
& \bullet ((PPHF \gg target?, forwardingNumber) \in forwardingTable^+ \\
& \wedge PPHF \gg target? \notin holdingTable \\
& \wedge forwardingNumber \text{ isFreeIn } PPHF \gg calls \\
& \wedge forwardingNumber \neq PPHF \gg source? \\
& \wedge P \gg calls' = PPHF \gg calls' \\
& \cup \{PPHF \gg source? \mapsto forwardingNumber\}) \\
& \wedge PPHF \gg connection! = successful) \\
& \vee \\
& (\exists forwardingNumber : NUM \\
& \bullet ((PPHF \gg target?, forwardingNumber) \in forwardingTable^+ \\
& \wedge PPHF \gg target? \in holdingTable \\
& \wedge forwardingNumber \text{ isFreeIn } PPHF \gg calls \\
& \wedge forwardingNumber \neq PPHF \gg source?) \\
& P \gg calls' = PPHF \gg calls' \\
& PPHF \gg connection! = optional_forward)
\end{aligned}$$

END *connect*

RAMIFICATIONS *disconnect* $\hat{=}$
WITHIN

$$PPF \gg source? = PPHF \gg source?$$

END *disconnect*

END *POTS_to_POTSPlusHoldAndForward*

In the retrieve clause we specify that the set of current calls for each machine should be identical. We then consider the exceptions to the general rule by specifying the ramifications for each operation. No ramifications are required for the operations concerning the administration of the holding table or the data in the forwarding table as these operations were not declared in the *POTS* machine.

The *disconnect* operation is identical in both machines, and it is necessary, therefore, only to ensure that the inputs to the operations in both machines are identical. This is done with the within clause of the ramifications for the operation.

The *connect* operation in the integrated machine, however, introduces the concepts of the holding and forwarding services to the behaviour specified in the *POTS* machine. We must, therefore, identify the circumstances in which the behaviour of the two operations differs, and specify the correct behaviour in these circumstances in the concedes clause for the operation's ramifications. Examining the specifications of the two machines, we can see that the behaviour may differ whenever the callee is busy. Beyond that, there are three circumstances to consider. Firstly, we consider the situation where the callee has registered for the holding service, but not the forwarding service (or they have registered for the holding service, and the forwarding number is busy); in these circumstances, the user of the integrated machine will be held (rather than receiving a failure). Secondly, we consider the situation where the callee has registered for the forwarding service, but not the holding service, and the forwarding number is not busy; in this situation, the user of the integrated machine will be connected to the forwarding number and receive a success message (again, rather than simply receiving a failure). Finally, consider the circumstance where the caller has registered for both services and the forwarding number is free; here, the user of the integrated machine will be given a message with the option of connecting to the forwarding number (again, rather than simply receiving a failure).

8.6.8 Syntax Checking the Relationship

Once we had created the specification for the final relationship, we again used Frog to validate its syntactical consistency. Errors were identified and resolved with Frog's assistance. When these were eliminated, Frog accepted our construct's specification and produced its equivalent, typed abstract syntax tree. We present the typed version of the paragraph associated with the concedes clause below. As in the previous sections, we do not show the version annotated with every expression's type, but just the version that shows the paragraph's type.

$$\begin{aligned}
& \text{AX } [_POTS_to_POTSPlusHoldAndForward_connect_concedes : \\
& \{[_POTSPlusHoldAndForward_connect_outputs \\
& \wedge _POTS_connect_outputs \\
& \wedge _POTSPlusHoldAndForward_connect_inputs \\
& \wedge _POTS_connect_inputs \wedge _POTSPlusHoldAndForward_state' \\
& \wedge _POTS_state' \wedge _POTSPlusHoldAndForward_state \\
& \wedge _POTS_state \mid \\
& (((PPHF \gg target?, PPHF \gg calls)) \in (\boxtimes \text{isBusyIn} \boxtimes)) \\
& \wedge ((P \gg connection!) \in (\{failed\})) \\
& \wedge (\neg ((\neg (\neg (((\neg ((\neg (\\
& ((PPHF \gg target?, (dom PPHF \gg forwardingTable))) \in (\boxtimes \notin \boxtimes))) \\
& \wedge (\neg (\forall ([forwardingNumber : NUM]) \bullet \\
& (\neg (((PPHF \gg target?, forwardingNumber) \\
& \quad \in ((\boxtimes^+ PPHF \gg forwardingTable))) \\
& \quad \wedge (\neg (((forwardingNumber, PPHF \gg calls)) \in (\boxtimes \text{isBusyIn} \boxtimes)))))))))) \\
& \wedge ((PPHF \gg target?) \in (PPHF \gg holdingTable))) \\
& \wedge ((P \gg calls') \in (\{PPHF \gg calls'\})) \\
& \wedge ((PPHF \gg connection!) \in (\{held\})) \\
& \wedge (\neg (\neg (\forall ([forwardingNumber : NUM]) \bullet \\
& (\neg ((((((PPHF \gg target?, forwardingNumber) \\
& \quad \in ((\boxtimes^+ PPHF \gg forwardingTable))) \\
& \quad \wedge (((PPHF \gg target?, PPHF \gg holdingTable)) \in (\boxtimes \notin \boxtimes)) \\
& \quad \wedge (((forwardingNumber, PPHF \gg calls)) \in (\boxtimes \text{isFreeIn} \boxtimes)) \\
& \quad \wedge (((forwardingNumber, PPHF \gg source?) \in (\boxtimes \neq \boxtimes)) \\
& \quad \wedge ((P \gg calls') \\
& \quad \in (\{(\boxtimes \cup \boxtimes (PPHF \gg calls', \\
& \quad \{(\boxtimes \mapsto \boxtimes (PPHF \gg source?, forwardingNumber))\}))))))
\end{aligned}$$

$$\begin{aligned}
& \wedge ((PPHF \gg \text{connection!}) \in (\{\text{successful}\}))))))))) \\
& \wedge (\neg (\neg (\forall ([\text{forwardingNumber} : \text{NUM}]) \bullet \\
& \quad (\neg (((((((PPHF \gg \text{target?}, \text{forwardingNumber}) \\
& \quad \in ((\bowtie^+ PPHF \gg \text{forwardingTable}))) \\
& \quad \wedge ((PPHF \gg \text{target?}) \in (PPHF \gg \text{holdingTable}))) \\
& \quad \wedge (((\text{forwardingNumber}, PPHF \gg \text{calls}) \in (\bowtie \text{isFreeIn } \bowtie))) \\
& \quad \wedge (((\text{forwardingNumber}, PPHF \gg \text{source?}) \in (\bowtie \neq \bowtie)))) \\
& \wedge ((P \gg \text{calls}') \in (\{PPHF \gg \text{calls}'\}))) \\
& \wedge ((PPHF \gg \text{connection!}) \in (\{\text{optional_forward}\})))))))]]) \\
& \text{END} \\
& \circledast [_POTS_to_POTSPlusHoldAndForward_connect_concedes : \\
& \quad \mathbb{P}([\text{PPHF} \gg \text{connection!} : \text{GIVEN CALL_SUCCESS}; \\
& \quad \quad P \gg \text{connection!} : \text{GIVEN CALL_SUCCESS}; \\
& \quad \quad PPHF \gg \text{source?} : \text{GIVEN NUM}; PPHF \gg \text{target?} : \text{GIVEN NUM}; \\
& \quad \quad P \gg \text{source?} : \text{GIVEN NUM}; P \gg \text{target?} : \text{GIVEN NUM}; \\
& \quad \quad PPHF \gg \text{calls}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{holdingTable}' : \mathbb{P}(\text{GIVEN NUM}); \\
& \quad \quad PPHF \gg \text{forwardingTable}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad P \gg \text{calls}' : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{calls} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad PPHF \gg \text{holdingTable} : \mathbb{P}(\text{GIVEN NUM}); \\
& \quad \quad PPHF \gg \text{forwardingTable} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})); \\
& \quad \quad P \gg \text{calls} : \mathbb{P}((\text{GIVEN NUM}) \times (\text{GIVEN NUM})))]])
\end{aligned}$$

8.6.9 Verifying the Relationship

Once the relationship had been confirmed as syntactically correct, Frog was able to generate these successfully for all of the generic proof obligations. In this instance the only relevant operation environments were *connect* and *disconnect*. Again, mechanical verification with Isabelle/ZF was impossible within our project's time constraints. Close, manual examination of the proof obligations however, could allow the informed observer to conclude that the relationship was valid, and given enough space and time, a pen-and-paper proof could fairly easily be presented. An abbreviated version of the correctness proof obligation for the *connect* operation of our relationship is presented below.

$$\begin{aligned}
& \spadesuit NUM \in \mathbb{P}_\infty \\
& NUM \in \mathbb{P} \spadesuit NUM \\
& \spadesuit CALL_SUCCESS \in \mathbb{P}_\infty \\
& CALL_SUCCESS \in \mathbb{P} \spadesuit CALL_SUCCESS \\
& \vdash? \\
& \forall _u _v _v' _i _j _p \bullet \\
& \langle _v, _v', _j, _p \rangle \in \{ \langle \langle calls, holdingTable, forwardingTable \rangle, \\
& \quad \langle calls', holdingTable', forwardingTable' \rangle, \langle source?, target? \rangle, connection! \rangle \\
& : (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \times (\mathbb{P}(\spadesuit NUM \times \spadesuit NUM) \times \mathbb{P}(\spadesuit NUM) \times \mathbb{P}(\spadesuit NUM \times \spadesuit NUM)) \\
& \times (\spadesuit NUM \times \spadesuit NUM) \times \spadesuit CALL_SUCCESS \bullet \\
& \exists forwardingNumber \bullet (\\
& \quad calls \in NUM \rightsquigarrow NUM \wedge \text{dom } calls \cap \text{ran } calls \in \{\emptyset\} \\
& \quad \wedge calls' \in NUM \rightsquigarrow NUM \wedge \text{dom } calls' \cap \text{ran } calls' \in \{\emptyset\} \\
& \quad \wedge holdingTable \in \mathbb{P}(NUM) \\
& \quad \wedge forwardingTable \in NUM \rightsquigarrow NUM \wedge forwardingTable' \in NUM \rightsquigarrow NUM \\
& \quad \wedge (forwardingTable^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge (forwardingTable'^+) \cap \text{id } NUM \in \{\emptyset\} \\
& \quad \wedge source? \in NUM \wedge target? \in NUM \wedge connection! \in CALL_SUCCESS \\
& \quad \wedge forwardingNumber \in NUM \\
& \quad \neg (target? \text{ isFreeIn } calls \wedge source? \neq target? \wedge \neg (\\
& \quad \quad calls' \in \{calls \cup \{source? \mapsto target?\}\} \wedge connection! \in \{successful\})) \\
& \quad \wedge \neg (target? \text{ isBusyIn } calls \wedge source? \neq target? \\
& \quad \wedge target? \in holdingTable \wedge \neg (\neg target? \in \text{dom } forwardingTable) \\
& \quad \wedge \neg (\langle target?, forwardingNumber \rangle \in (forwardingTable^+) \\
& \quad \wedge forwardingNumber \text{ isBusyIn } calls)) \wedge \neg (calls' \in \{calls\} \\
& \quad \wedge connection! \in \{held\})) \wedge \neg (target? \text{ isBusyIn } \wedge \\
& \quad \wedge source? \neq target? \wedge target? \notin holdingTable \\
& \quad \wedge \langle target?, forwardingNumber \rangle \in (forwardingTable^+) \\
& \quad \wedge forwardingNumber \text{ isFreeIn } calls \wedge forwardingNumber \neq source? \\
& \quad \wedge \neg (calls' \in \{calls \cup \{source? \mapsto forwardingNumber\} \\
& \quad \wedge connection! = successful)) \wedge \neg (target? \text{ isBusyIn } calls \\
& \quad \wedge source? \neq target? \wedge target? \in holdingTable \\
& \quad \wedge \langle target?, forwardingNumber \rangle \in (forwardingTable^+) \\
& \quad \wedge forwardingNumber \text{ isFreeIn } calls \wedge forwardingNumber \neq source? \\
& \quad \wedge \neg (calls' \in \{calls\} \wedge connection! \in \{optional_forward\})) \\
& \quad \wedge \neg (\neg (\neg (target? \text{ isBusyIn } calls \wedge \neg (\neg (target? \notin holdingTable) \\
& \quad \wedge \neg (\langle target?, forwardingNumber \rangle \in (forwardingTable^+) \\
& \quad \wedge \neg (\neg (forwardingNumber \text{ isBusyIn } calls) \\
& \quad \wedge \neg (forwardingNumber \in \{source?\})))))) \\
& \quad \wedge \neg (calls' \in \{calls\} \wedge connection! \in \{failed\})) \\
& \quad \wedge holdingTable' \in \{holdingTable\}) \\
& \quad \wedge forwardingTable' \in \{forwardingTable\})
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v \rangle \in \{ \langle P \gg \text{calls}, \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, \\
& \quad PPHF \gg \text{forwardingTable} \rangle \rangle \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \quad \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \text{NUM} \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{calls} \in \{ PPHF \gg \text{calls} \} \\
& \wedge \langle _u, _v, _i _j \rangle \in \{ \langle P \gg \text{calls}, \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, \\
& \quad PPHF \gg \text{forwardingTable} \rangle, \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \\
& \quad \langle PPHF \gg \text{source?}, PPHF \gg \text{target?} \rangle \} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \quad \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}))) \\
& \quad \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \text{NUM} \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPHF \gg \text{source?} \in \text{NUM} \\
& \wedge PPHF \gg \text{target?} \in \text{NUM} \wedge P \gg \text{source?} \in \{ PPHF \gg \text{source?} \} \\
& \wedge P \gg \text{target?} \in \{ PPHF \gg \text{target?} \} \\
& \wedge \langle _u, _i \rangle \in \{ \langle \text{calls}, \langle \text{source}, \text{target?} \rangle \rangle \} \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \bullet \\
& \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{source?} \text{ isFreeIn } \text{calls} \\
& \Rightarrow \\
& \exists _u' _o \bullet \langle _u, _u', _i, _o \rangle \in \{ \langle \text{calls}, \\
& \quad \text{calls}', \langle \text{source?}, \text{target?} \rangle, \text{connection!} \rangle \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \quad \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& \wedge \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls} \cap \text{ran } \text{calls} \in \{\emptyset\} \\
& \wedge \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } \text{calls}' \cap \text{ran } \text{calls}' \in \{\emptyset\} \\
& \wedge \text{source?} \in \text{NUM} \wedge \text{target?} \in \text{NUM} \wedge \text{connection!} \in \text{CALL_SUCCESS} \\
& \neg (\text{target} \text{ isFreeIn } \text{calls} \wedge \text{source?} \neq \text{target?} \wedge \neg (\\
& \quad \text{calls}' \in \{ \text{calls} \cup \{ \text{source?} \mapsto \text{target?} \} \wedge \text{connection!} \in \{ \text{successful} \} \}) \\
& \wedge \neg (\neg (\neg (\text{target?} \text{ isBusyIn } \text{calls}) \wedge \neg (\text{source?} \in \{ \text{target?} \}))) \\
& \wedge \neg (\text{calls}' \in \{ \text{calls} \} \wedge \text{connection!} \in \{ \text{failed} \}) \} \\
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u', _v' \rangle \in \{ \langle P \gg \text{calls}, \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, \\
& \quad PPHF \gg \text{forwardingTable} \rangle \rangle \\
& : \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \quad \times (\mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \text{NUM} \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{calls} \in \{ PPHF \gg \text{calls} \} \\
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \\
& \quad \langle P \gg \text{calls}, \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, PPHF \gg \text{forwardingTable} \rangle, \\
& \quad P \gg \text{calls}', \langle PPHF \gg \text{calls}', PPHF \gg \text{holdingTable}', PPHF \gg \text{forwardingTable}' \rangle, \\
& \quad \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \langle PPHF \gg \text{source?}, PPHF \gg \text{target?} \rangle, \\
& \quad P \gg \text{connection!}, PPHF \gg \text{connection!} \rangle \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \quad \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \quad \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \quad \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \quad \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls}' \cap \text{ran } PPHF \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPHF \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{forwardingTable}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}'^+) \cap \text{id } \text{NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPHF \gg \text{source?} \in \text{NUM} \\
& \wedge PPHF \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPHF \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge P \gg \text{connection!} \in \{ PPHF \gg \text{connection!} \} \\
\end{aligned}$$

$$\begin{aligned}
& \wedge \langle _u, _v, _u', _v', _i, _j, _o, _p \rangle \in \{ \\
& \quad \langle P \gg \text{calls}, \langle PPHF \gg \text{calls}, PPHF \gg \text{holdingTable}, PPHF \gg \text{forwardingTable} \rangle, \\
& \quad P \gg \text{calls}', \langle PPHF \gg \text{calls}', PPHF \gg \text{holdingTable}', PPHF \gg \text{forwardingTable}' \rangle, \\
& \quad \langle P \gg \text{source?}, P \gg \text{target?} \rangle, \langle PPHF \gg \text{source?}, PPHF \gg \text{target?} \rangle, \\
& \quad P \gg \text{connection!}, PPHF \gg \text{connection!} \} \\
& : (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM}) \times \mathbb{P}(\spadesuit \text{NUM} \times \spadesuit \text{NUM})) \\
& \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \times (\spadesuit \text{NUM} \times \spadesuit \text{NUM}) \\
& \times \spadesuit \text{CALL_SUCCESS} \times \spadesuit \text{CALL_SUCCESS} \bullet \\
& P \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls} \cap \text{ran } P \gg \text{calls} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls} \cap \text{ran } PPHF \gg \text{calls} \in \{\emptyset\} \\
& P \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \wedge \text{dom } P \gg \text{calls}' \cap \text{ran } P \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \gg \text{calls}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge \text{dom } PPHF \gg \text{calls}' \cap \text{ran } PPHF \gg \text{calls}' \in \{\emptyset\} \\
& \wedge PPHF \gg \text{holdingTable} \in \mathbb{P}(\text{NUM}) \wedge PPHF \gg \text{holdingTable}' \in \mathbb{P}(\text{NUM}) \\
& \wedge PPHF \gg \text{forwardingTable} \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge PPHF \gg \text{forwardingTable}' \in \text{NUM} \leftrightarrow \text{NUM} \\
& \wedge (PPHF \gg \text{forwardingTable}'^+) \cap \text{id NUM} \in \{\emptyset\} \\
& \wedge P \gg \text{source?} \in \text{NUM} \wedge P \gg \text{target?} \in \text{NUM} \wedge PPHF \gg \text{source?} \in \text{NUM} \\
& \wedge PPHF \gg \text{target?} \in \text{NUM} \wedge P \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPHF \gg \text{connection!} \in \text{CALL_SUCCESS} \\
& \wedge PPHF \gg \text{target isBusyIn } PPHF \gg \text{calls} \\
& \wedge P \gg \text{connection!} \in \{\text{failed}\} \wedge \neg (\neg (\neg (\neg (\neg (\neg (\\
& \quad PPHF \gg \text{target} \in \text{dom } PPHF \gg \text{forwardingTable} \\
& \quad \wedge \neg (\forall \text{forwardingNumber} : \text{NUM} \bullet \\
& \quad \langle PPHF \gg \text{target?}, \text{forwardingNumber} \rangle (PPHF \gg \text{forwardingTable}^+) \\
& \quad \wedge \neg (\text{forwardingNumber isBusyIn } PPHF \gg \text{calls})))))) \\
& \quad \wedge PPHF \gg \text{target?} \in PPHF \gg \text{holdingTable} \wedge P \gg \text{calls}' \in \{PPHF \gg \text{calls}'\} \\
& \quad \wedge PPHF \gg \text{connection!} \in \{\text{held}\}) \wedge \neg (\neg (\\
& \quad \forall \text{forwardingNumber} : \text{NUM} \bullet \neg (\text{forwardingNumber} \neq PPHF \gg \text{source?} \\
& \quad \wedge \langle PPHF \gg \text{target?}, \text{forwardingNumber} \rangle (PPHF \gg \text{forwardingTable}^+) \\
& \quad \wedge \neg (\text{forwardingNumber isFreeIn } PPHF \gg \text{calls})))))) \\
& \quad \wedge PPHF \gg \text{target?} \notin PPHF \gg \text{holdingTable} \\
& \quad \wedge P \gg \text{calls}' \in \{PPHF \gg \text{calls}' \cup \{PPHF \gg \text{source?} \mapsto \text{forwardingNumber}\}\} \\
& \quad \wedge PPHF \gg \text{connection!} \in \{\text{successful}\})))))) \\
& \quad \wedge \neg (\neg (\forall \text{forwardingNumber} : \text{NUM} \bullet \neg (\\
& \quad \langle PPHF \gg \text{target?}, \text{forwardingNumber} \rangle (PPHF \gg \text{forwardingTable}^+) \\
& \quad \wedge PPHF \gg \text{target?} \in PPHF \gg \text{holdingTable} \\
& \quad \wedge \text{forwardingNumber isFreeIn } PPHF \gg \text{calls} \\
& \quad \wedge \text{forwardingNumber} \neq PPHF \gg \text{source?} \wedge P \gg \text{calls}' \in \{PPHF \gg \text{calls}'\} \\
& \quad \wedge PPHF \gg \text{connection!} \in \{\text{optional_forward}\})))))) \}
\end{aligned}$$

8.7 Concluding Remarks

This chapter has described the mechanization, with Frog, of a case study examining the specification of a telephone system. This involved incrementally adding functionality to a vanilla specification and examining the relationships between the specifications produced. We began with a telephone system capable solely of connecting and disconnecting calls. We then introduced a system that incorporated a holding service and specified the retrenchment relationship between this system and its predecessor. We syntax checked this relationship mechanically with Frog uncovering a number of errors and producing a number of alternate specifications leading to a fully typed specification of our relationship. Proof obligations were generated for this relationship by Frog. The size of these proof obligations prevented us from verifying them mechanically with Isabelle/ZF, but we were able to establish manually that the obligations had been generated correctly from the relationship's configuration and specification. We then proceeded to introduce a new system that introduced a call forwarding service. Again we related the specification of this system to our vanilla system with a retrenchment relationship and were able to successfully type check it before generating manually verified proof obligations. Finally, we presented a fully integrated system that handled both additional services. We showed that Frog was able to type check the relationships between all of these systems and generate the proof obligations that could allow a user – given sufficient time – to verify the relationship in Isabelle/ZF.

Chapter 9

Conclusions

This chapter provides a retrospective of the thesis. We begin with a review of the thesis, highlighting the contribution that we believe the contained research makes. We describe the positioning of our work, the principal results and the supporting tools and case study. We then progress to evaluate our work. We discuss the extent to which our original goals have been satisfied and illustrate our successes and failures, considering any alternate approaches that we may have taken. Finally, our conclusions are presented and we discuss the avenues available for further research, both in the continued development of Frog and its integration with similar tool sets.

9.1 A Review of the Contribution Made by the Research Described in this Thesis

In this thesis we have described research that has led to the production of Frog, a tool capable of providing mechanical assistance in the construction of a specification using retrenchment. The creation of this tool has involved building one of the first parsers of the ISO Standard for the Z notation. In order to incorporate a construct within a Z specification we devised an extension to this notation, creating a configurable syntax that allowed the specification and flexible configuration of constructs. We have also created methods that allow the automatic generation of proof obligations from the combination of a construct's configuration and specification. Techniques for translating the generated proof obligations into a format suitable for external theorem provers were also devised.

The motivations for our research were described in chapters 1–3. We began our thesis by describing the circumstances that led to the need for mechanized support for retrenchment. Model-based specification was introduced and we briefly described how refinement was used to formally create concrete programs from abstract specifications. Forwards simulation was introduced as a method for proving correctness in stepwise refinement. We showed how a refinement relationship could be expressed in Z and described how the forward simulation rules could be adapted to provide a verification of this relationship. Despite refinement's success there are limitations to its use; we described these and introduced retrenchment a technique that was able to overcome some of those restrictions. The different forms of retrenchment (and its alternatives) were briefly described.

We progressed to examine the need for tools to support formal methods. We investigated the literature and revealed some of the principal reasons that formal methods' detractors have cited for its failure. We showed that the lack of sufficient, adequate tool support was one of the foremost reasons. Having established that tool support was essential for a formal technique, we looked at the different areas in which mechanical assistance could be a boon. We then considered retrenchment specifically and considered how best to

provide the mechanized support. Firstly, we examined some existing toolkits and assessed the potential for adaptation to retrenchment. Unfortunately, all of the freely available tools were each found lacking some essential feature. The possibility of embedding directly within a theorem prover was dismissed as the result would have needed to be tailored to each particular type of retrenchment. As a result of this, we established our requirement that the tool support for retrenchment should not only be able to handle all of retrenchment's different forms, but should be fully configurable to allow the specification of as many different types of relationship as possible. With this in mind, and having rejected the other alternatives, we presented the motivation for creating a stand-alone toolkit. The areas in which mechanical support could be provided were then re-examined and their use in a retrenchment toolkit evaluated.

Having established the need for a new tool we began to describe how such a toolkit could be built, and in chapters 4–6 we described the principal contribution made by our research. This began with the decision to use ISO standard Z [ISO02] as the principal notation within our tool. Having considered existing parsers, and establishing that there was no freely available tool that met our requirements, we came to the conclusion that we would need to implement our own parser. Our parser was one of the first to attempt an implementation of the human readable grammar presented in the ISO standard. This process required the resolution of this grammar's intrinsic ambiguity, but we used the ANTLR parser generator to produce a parser that – through the use of syntactic predicates – processed a grammar which was recognisably similar to the one presented in the standard. In places we were forced to make adaptations to our grammar in order to overcome – what we felt were – contradictions in the standard document. Unlike other attempts at parsing the ISO standard Z notation, we used a predicated-LL(k) grammar and we believe that this was the first project to use that approach.

Once we had described the construction of a parser for standard Z, we began to consider the incorporation of configurable, encapsulated constructs. We determined that a specification would consist of generic machines and

relationships. The machines were used to represent models, and the relationships the links between them. Both machines and relationships were designed to be fully configurable concepts, the only fixed attribute of each being whether it formed a cell or a vertex in a graph of a specification. We introduced Frog-CCL, a language that we devised for use in expressing the syntax and semantics of a construct. This language offered much flexibility, allowing users to significantly alter the structure of their constructs and specify the proof obligations that would need to be discharged to verify the construct. We showed how the \LaTeX representation of the Z notation could be extended to incorporate these generic constructs and described how a tool could use the configuration of a construct to transform a specification in this extended syntax into a Z section that satisfied the annotated grammar of the ISO standard. We illustrated the flexibility of Frog-CCL by providing sample configurations and specifications for machines, refinement and retrenchment. This fully configurable approach to constructs allows users to experiment with the nature of models and the relationships between them, and we believe it is a major asset of our tool.

Having established methods that would allow us to provide mechanical support for the specification of constructs (particularly retrenchment relationships), our thoughts turned to the creation of tools that could generate proof obligations to verify those constructs. The first stage of this process involved the generation of a semantic model from a specification. The semantic model was effectively a set of semantic bindings, each of which bound a variable to a set of hypotheses and goals that would be used as the building blocks for the generation of proof obligations. We defined a collection of functions that could be used to automatically generate a semantic model from a typed Z specification. We also devised a ZF language (based on Zermelo-Fraenkel set theory) that could serve as an intermediary between Z and the language of external theorem provers. We used this language to record the elements in our semantic model and also to store our proof obligations. A method of using the semantic model to instantiate the generic proof obligations contained within a construct's configurations was described.

Once we had shown how proof obligations could be created from a construct, we considered the ways in which we could assist a tool user in discharging those proof obligations. We had already discussed that deductive reasoning was the most sensible option, but now we confirmed the decision to interface to an external theorem prover. We examined a number of alternatives and chose to interface – initially – with Isabelle/ZF. We showed how a construct’s proof obligations could be easily translated from the syntax of the ZF language, to that of Isabelle/ZF. Unfortunately, the ability of Isabelle/ZF to discharge the proof obligations that we generated was not as great as we had hoped. We illustrated the main issues that prevented automated, mechanical discharge.

In chapter 7, we described the architecture and implementation of the Frog tool. Here we described how the elements we had described in the previous chapters were integrated to provide an umbrella toolkit that, whilst modular, supported a user from the creation of a specification to the discharge of its proof obligations. We showed the tool from the user’s perspective and then analysed some salient points of its implementation.

Finally, in chapter 8, we showed how Frog could be used to support the specification and verification of a non-trivial example. We examined an existing retrenchment case study concerning the modelling of a telephone system. The tool was capable of expressing all of the required machines and relationships. Frog showed its value by uncovering numerous typing errors and automatically generating the constructs’ proof obligations.

9.2 Evaluation, Conclusions and Future Research

In this last section we shall evaluate our success in meeting our specified goals, draw conclusions from our experience and discuss opportunities for future research leading from the completion of this project.

9.2.1 Evaluating the Research

In chapter 1 we specified some goals for our research. We will now evaluate the extent of our success in meeting those objectives.

Our first goal was the creation of a tool that was capable of parsing the Z notation in a way that conformed to the ISO standard. We believe that we have succeeded in this aim. The only caveats are that there some areas in which – as a result of using the draft version of the standard (before the final version became publicly available) — we used our own solutions to known, defined problems that were later explicitly resolved in the standard. There were also some areas in which we believed there were contradictions present and we were forced to seek a suitable compromise to produce a consistent tool.

We had not initially intended to support the entire Z notation, but the need to support the mathematical toolkits forced our hand. In fact, the parsing of the standard toolkit (and its ancestors) provides a significant test to any parser’s handling of the majority of the Z notation. We constructed test constructs that used all of the operators defined within the toolkits, and Frog was able to successfully parse and type all of them. The parser was further tested with our telephone case study. Far from not being able to handle these more extensive specifications, Frog was able to highlight numerous mistakes in our initial constructs and provide assistance in their repair.

The second objective that we highlighted, was the need to provide a mechanism for creating models and relationships using the Z notation. Furthermore, we made it a requirement that this system was configurable enough to be able to express machines and relationships with a range of properties. We believe that the design of – and mechanical support for – Frog-CLL more than satisfies this aim. It has allowed us a great degree of flexibility and power in the expression of a construct’s attributes and allows for the dynamic specification of the shape of proof obligations.

Together with the extension to the \LaTeX representation of the Z syntax, Frog-CCL allows a user to incorporate these constructs into Z specifications.

We have also outlined a system – that extends the syntax transformation process in [ISO02] – that transforms these constructs into a Z representation that will satisfy the annotated grammar of the ISO standard. This transformation ensures that it will be possible to export a Z representation of our constructs that can be processed by other Z tools. We have demonstrated the power of Frog-CCL by defining configurations for some example machines, a refinement relationship and a retrenchment relationship. One of our machines and the retrenchment were successfully used in our telephone system case study.

The next of our stated goals was that it would be necessary to be able to generate proof obligations from a construct’s configuration and specification. It was also required that these proof obligations could be fed to an external theorem prover for verification. Again, we feel that we have successfully fulfilled this obligation. We have defined a system that can extract a semantic model from a construct’s specification and that is able to use the semantic model to instantiate the generic proof obligations of its configuration. This system also allows a user to interface with the Isabelle/ZF interactive theorem prover. Although we would have liked to be able to provide assistance in the automatic discharge of proof obligations, Isabelle/ZF was more suited to the proof of complex, but small, theorems than the large, shallow theorems that we presented it. In retrospect, Isabelle/ZF was probably not the best choice of theorem prover, but its native support of Zermelo-Fraenkel set theory appeared to be very attractive when making our initial choice. In the future we will probably pursue a shallower embedding in a more suitable theorem prover. Having said this however, we feel that interfacing to Isabelle/ZF was certainly a valuable exercise and it will allow the interested user to experiment to a greater extent than any shallow embedding would (this is certainly a benefit when we consider that one of our overriding objectives was to produce a tool that was as flexible as possible).

Our final goal was that we would be able to illustrate that we had met the above objectives through the mechanization of a non-trivial retrenchment case study. We re-visited the retrenchment case study first presented in [BP02]. We were successfully able to parse and type check all of our

constructs and using a set of default configurations were able to produce accurate proof obligations whose discharge would verify the consistency of our specification. Our only disappointment was the inability – due to time constraints – to show that our proof obligations could be discharged mechanically with Isabelle/ZF. When we consider however, that the verification of proof obligations only verifies a construct is correct under the assumption that the generic proof obligations have been expressed correctly in the construct’s configuration, we feel that – even in the future – our resources would be better spent seeking a generic, automated approach to theorem proving, than in the manually assisted discharge of proof obligations for one specific configuration and specification combination.

9.2.2 Conclusions and Opportunities for Further Research

The research described in this thesis has been an attempt to provide mechanized support for retrenchment. We have satisfactorily met our objectives and built a tool that will provide a solid foundation for future retrenchment research. We feel therefore, that we are able to conclude that our research has been a success. Of course, while we may have met our objectives, there is still much work to be done in transforming Frog into a tool that is able to support the use of retrenchment in significant, complex project developments and we will have to significantly improve many areas before Frog could be considered for use in an industrial situation. This leaves much scope for future research and development.

Extending the Notations and Representations Available

Of significant interest is the ability to interface with other Z tools. For instance, we would like to investigate the possibility of integrating with CZT. Such an integration would allow us not only to take advantage of CZT’s tools, but also to use its translators to interact with yet more tools. For example, a B-Method translator is being developed for CZT that would allow our constructs to be used with existing B tools. Such an integration could

be achieved either by directly interfacing with CZT's class hierarchy or by extending Frog so that it is capable of handling more representations than the \LaTeX of [ISO02], and in particular the ZML syntax of CZT. We would suggest that the latter of these approaches would prove more fruitful, and indeed we would consider the ability to support more representations to be a major goal of the toolkit in the future, with the ability to directly support [ISO02]'s unicode¹ and ASCII standards an immediate aim. In fact, we would like to see Frog's decoupling from Z so that it is possible to use a wide range of notations, whether these be established syntaxes such as those belonging to B or VDM, or custom syntaxes required for specifying particular applications.

In the shorter term we would like to introduce an extra phase into Frog's parsing process. This would take the form of a pre-processing phase whose responsibilities would be twofold. Firstly, we would like to incorporate the ability to handle non-standard Z. There have been many extensions to Z and whilst [ISO02] seeks to promote uniformity these extensions can be very useful for particular applications. For instance, we mentioned in section 4.4.1 the difficulty we had in guaranteeing the properties of the prelude section. We would envisage a configuration file which contained a mapping from the custom syntax to the syntax of [ISO02], along with the implementation of a call-back which would then be called when parsing the Z object in the original syntax checking phase. For instance, the prelude problem may be solved by an entry such as the following.

```
EXTENDED_SYNTAX = \begin{section} \PRELUDE \end{section}
ISO_SYNTAX = \begin{section} \SECTION prelude \end{section}
CALLBACK = {
    ZSection currentSection = getCurrentSection();
    currentSection.setPrelude(true);
}
```

This would map the syntax so that the ISO-compatible parser were able to understand it, but also allow the prelude section to be uniquely identified.

¹Of course, providing the ability to create and edit unicode documents would require significant enhancement to Frog's text editing capabilities. However, simply being able to accept and output specification in the representation would provide a useful first step.

Obviously, there would be a limit to the power of the call-backs, but these could prove exceedingly useful and there is an argument that it would be possible to translate complete distinct notations such as B using such features.

The second responsibility of the pre-processing phase would be to handle the ordering of paragraphs. We indicated in section 4.5 that Frog requires paragraphs to be correctly ordered, but a pre-processing phase could reorder paragraphs to ensure that dependencies are handled correctly regardless of their order in the Z specification.

Decoupling and Enhancing Frog-CCL

An area in which we feel our work has been innovative, and where we would like to pursue further research, is in the configuration of proof obligations. In [FB07] we presented a summary of the research that we had undertaken into this area (including that described in this thesis) and propounded ways in which this configurable approach could be extended in the future. Many of these involve extending the expressiveness of Frog-CCL, but perhaps the most important is the decoupling of the language from Frog and from Z. Whilst the development of Frog has suited a tight coupling between Frog-CCL and Z, we feel that the configuration of proof obligations is something that could be incorporated in all tools. In order for this to be possible the language should not depend upon a particular toolkit or notation. The language as it stands would not be particularly useful for trying to specify the syntax changes or proof obligations in a B construct. Furthermore it would be necessary to define quite different rules for translating the constructs into those of the original syntax². This separation of concerns is likely to require a great deal of research if we are to achieve a language that is syntax-independent. The decoupling from Z is therefore likely to be a long term goal. More pressing is ensuring that any links to Frog are in name alone, this should be quite easy to achieve and we feel that the language (and its translation mechanisms) could be adapted for use with other Z tools with very little effort.

²Whether this would be possible is, of course, a separate issue requiring investigation.

Aside from these decoupling issues, there are also the aforementioned aims regarding the language's power. When examining the B-Method's constructs, there are two implicit capabilities that we cannot replicate with our current definition of Frog-CCL. The first of these is the ability to specify that a different syntax will be used within a clause; in the B-Method the syntax used within an abstract machine's clauses is different to that used within those of an implementation machine. The second is the ability to restrict the types of construct that can form relationships; for instance, an abstract machine cannot be considered to be the refinement of any other construct. We would like to incorporate both of these capabilities within Frog-CCL. The first requires the parsing improvements that we mentioned earlier in order that we have a number of syntaxes from which a construct's syntax could then be chosen. This could simply be a meta-syntax that it would be possible to pre-process into a standard notation or could be an entirely different notation. Allowing constructs to be configured with one of these syntaxes would be relatively simple. Translating the construct to the base notation however, would not only require the parser to support the processing of the clause's contents (which would be simple assuming the parser was already able to handle the chosen syntax), but would also require the construct's syntax to be reducible to that base notation. As we have mentioned previously, this ability is likely to require significant effort and in the short-term a suitable goal would be the ability to support user-defined syntaxes that could be mapped to the current Z notation (with the pre-processing strategy described above). The second capability would be fairly easy to implement by extending the configuration of each relationship so that it is necessary to specify which machines can form the source and which the target of that relationship. A system like this would be burdensome, however, where a relationship could be used with any type of machine and there were a number of possible machine configurations – it would be necessary to specify each of those machine configurations as a possible source and as a possible target. We felt implementing inter-construct compatibility in this way would place an unnecessary burden upon the user and did not envisage providing the feature in this way.

In addition to the extensions we have discussed above, we would like to consider allowing constructs' configurations to inherit from one another (we mentioned this briefly in section 8.5.2). The benefits of this are twofold. Firstly, where constructs are similar, the common configuration need only be specified once. Secondly, it provides a neat solution to the problem of defining inter-construct compatibility. The first of these benefits is particularly useful in the retrenchment world where there are many relationship types with a common theme. For instance, the difference between the primitive retrenchment relationship and an output retrenchment relationship (see section 2.5.1) is a single clause (and that clause's involvement in a proof obligation). Instead of repeating all the common clause definitions and proof obligations it would be better to simply define the differences. For example, the output retrenchment relationship might be defined as follows.

```

DEFINE RELATIONSHIP output_retrenchment EXTENDS retrenchment
  CLAUSES
    (
      NAME= output,
      LEVEL= ramifications,
      REQUIREMENT= OPTIONAL,
      CONTENT= PREDICATE,
      RELATION= <FROM_MACHINE(state),TO_MACHINE(state),
                FROM_MACHINE(state'),TO_MACHINE(state'),
                FROM_MACHINE(inputs),TO_MACHINE(inputs),
                FROM_MACHINE(outputs),TO_MACHINE(outputs)>
    )
  PROOF_OBLIGATIONS
    (
      OPERATION_LEVEL, CORRECTNESS,
      (
        ! u, v, v', i, j, p
        @ <v,v',j,p> : TO_MACHINE(operation.post)
          & <u,v> : ramifications.retrieve
          & <u,v,i,j> : ramifications.within
          & <u,i> : FROM_MACHINE(operation.pre)
          => ( # u',o @ <u,u',i,o> :

```

```

                                FROM_MACHINE(operation.post)
      & (( <u',v'> : retrieve
          & <u,v,u',v',i,j,o,p> :
                                ramifications.output )
        | <u,v,u',v',i,j,o,p> :
                                ramifications.concedes )
    )
  )
)
END

```

Instead of needing to repeat the clause definitions for retrieve, within and concedes we need to define only the new output clause. Where the definition of a clause changes, we would not need explicitly indicate that the clause had been overridden, but could simply redefine it and specify – within Frog-CCL – that the definition of the child is always preferred. The same rules would apply for operation environments. With proof obligations it would be necessary to mandate an identifier for each obligation so that it would be possible to determine whether the proof obligation was an addition or a replacement (as in the case above). If we extend our inheritance model in a similar way to Java's, so that every machine configuration implicitly inherits from a notional 'machine' super-configuration, the difficulties of specifying compatibility between constructs are also eased. Where a relationship could relate machines of any type we would restrict the source and target machine to be instances of the super-configuration, which by definition is any available machine. Similarly, if we had five types of abstract machine and we wished to restrict the source of a machine to being one of these we need only define an umbrella configuration in order to prevent us needing to specify the name of each. We believe that the use of inheritance would greatly increase the usefulness of Frog-CCL and we envisage it being an immediate goal as the project progresses. Once this was in place, there would be no disadvantage to implementing the ability to place restrictions on inter-construct compatibility, and we feel that this would be a sensible move.

We have discussed the advantages of a configurable approach. It should be noted, however, that other tools such as RODIN and Overture have made attempts to support customization through their ability to incorporate plug-ins. Obviously, the overhead for creating a new plug-in for every new construct is significantly greater than that for creating a new Frog-CCL configuration file, and we feel that our approach provides not only a resource benefit, but more freedom to experiment and use custom constructs for applications requiring additional rigour. We feel therefore, that the further research that we would undertake in this area would prove fruitful, and a decoupling from Frog would provide a language that could be widely used in model-based specification.

Facilitating Verification: Approaching an Automated Solution

Whilst we have discussed a number of worthwhile extensions to our research, we feel that the first piece of further work to be completed is an interface to an alternative, external theorem prover. Preferably this should be a prover that is better able to attempt the automatic discharge of proof obligations generated by Frog, but certainly one where the interactivity required is less. If we are able to provide a fully automated process from specification to verification, Frog's users will be able to make much better use of the tool. There are a number of tactics we could employ in seeking a better proving strategy, some of which we discussed in section 6.4.11.

A first approach to providing automated verification is to interface to a genuine automated theorem prover. The disadvantage with these theorem provers – as we highlighted when we looked at them initially – is the lack of direct support for set theory. As we mentioned, however, there may be tools that could translate the proof obligations for Isabelle/ZF into equivalent proof obligations that are suitable for theorem provers using first-order logic. An investigation into these tools would be a necessary first step in achieving a fully automated verification.

We discussed the problems of using a deep embedding with Isabelle/ZF in section 6.5.2 and presented two ways in which we could embed directives

within the definitions of Z 's mathematical toolkits. The first method would associate some lemmas and tactics with each operator; these lemmas and tactics would be applied by the tool to the proof obligation using Isabelle/ZF in the hope that the combined rules for each operator would automate that proof obligation's discharge. The difficulty with this method is in creating the lemmas and tactics for each operator. This would require a significant overhead and the extent to which the operator's directives could be composed in order to discharge a complex proof obligation would also need to be researched. The second method involves mapping the operators defined in Z 's mathematical toolkits to the native operators of Isabelle/ZF; in effect a shallow embedding of Z 's operators. This solution would make the proof obligations generated much simpler as it would not be necessary to define the required operators within the proof obligation. The disadvantages, however, are that there may be subtle semantic differences between the operator's definitions in Z and Isabelle/ZF, and also that while the proof obligation may not be so hard to discharge, it would still be far from an automated process.

Another approach to making Frog's proof obligations easier to discharge is to consider the use of a specialized Z theorem prover that is able to take advantage of Z 's schemas. While we initially wished to show that Frog could hook up to any suitable theorem prover – and used Isabelle/ZF to exhibit this proof of concept – our concern now is in providing facilities for discharging the proof obligations more easily. Whilst theorem provers such as CADiZ and Z/EVES may be unable to provide a fully automated solution, there can be little doubt that they could significantly facilitate the proof process.

Finally, we could investigate the use of theorem provers that use different forms of logic, particularly higher order logic. Whilst the use of these theorem provers may provide a solution that is no more automated than Isabelle/ZF, they do have significantly more documentation and many more users who are able to provide help to those struggling to discharge an obligation. If this route were to be pursued, the possibility of using a shallow embedding or associating lemmas and tactics with Z operators could also be investigated, and such a combination may provide a powerful solution.

It is clear that each of the solutions we have proposed has its advantages

and disadvantages. Our experience has shown it is hard to predict the best solution. We feel that it would be necessary to create prototypes – perhaps capable of handling a small subset of operators – for each of the above solutions and to evaluate the success of each. Whichever option is taken, there is likely to be a requirement for significant research in order to provide a quality solution.

Augmenting Frog

As with any tool, there are countless ways in which the Frog toolkit itself could be improved. There are many desirable features and augmentations that could be added to the tool set. For instance, in section 3.4 we discussed a number of features that resource constraints did not allow us to incorporate into Frog’s first iteration – such as an animator, model checker or test-case generator – that would provide significant value when creating specifications. An implementation of one, or more, of these in the future is obviously something the tool’s users would be likely to find useful. Perhaps more desirable, however, would be the ability for Frog to interface with existing implementations of these features. We suggested earlier that compatibility with CZT would be a good first step as its developers already have plans to integrate with many existing toolkits, each of which provides a different tool set and augments the total set of available tools. In the future – and although we envision Frog becoming open-source software – we would like to define extension points (APIs) that allow users to hook in their own features. Eclipse has shown that being able to incorporate plug-ins through extension points can significantly improve the power of an application, as users are able to tailor the generic solution provided by the tool to handle their specific needs. The use of such plug-ins gives a good fit with Frog’s goals of flexibility and configurability.

As well as new features, there are opportunities to enhance the existing toolkit. Like most investigative projects of this nature a significant amount of learning takes place throughout the development cycle and much more is known at the end than at the beginning. There are likely to be many areas

of Frog's code base that are candidates for refactoring [Fow99]. The result of this process would inevitably lead to performance enhancement and an increased ability of the tool to handle external interaction.

The toolkit could also be enhanced by implementing some of the suggestions that we detailed within chapter 7. One of the most desirable is the ability to take advantage of Frog's modular design and allow users to switch factories (for instance, the parsing factory). This would make the implementation of other features we have specified above – for instance, the ability to support multiple Z representations, syntax notations, theorem prover logics – much easier. Providing a clear cut interface for each factory would provide a ready made solution to the extension point requirement. A user could simply replace any part of Frog – to suit their needs – by implementing the required factory's interface. As the factories cover rather large pieces of the toolkit's functionality it would probably be necessary to sub-divide each of the factories, but the principle is sound regardless. The ways in which a user could indicate which plug-ins they wished to use were discussed in section 7.2. Other desirable features that we have previously discussed include: the provision of tools to edit a specification's configuration dynamically, discrete tools for editing and testing a character dictionary, improving the text editing capabilities of the GUI and implementing graph layout algorithms for the dependency browser. As well as these suggestions there are many more GUI and small functional changes that could greatly improve the usability of the toolkit (which hitherto had not been a top priority).

Using Frog

While there are many aspects of the tools which could be improved, the truly interesting further research, and the test of any tool, is the ability to actually provide support to other projects. First amongst these is the continued development of retrenchment as a formal technique. This can be supported through the automatic examination of existing case studies and providing mechanized verification of their associated proof obligations. More significantly, we can contribute to the furthering of retrenchment's use in

formal developments by showing that new mechanisms can be automated and successfully used in practical specifications.

Two opportunities to use Frog to provide support to other research teams are immediately apparent. The first of these involves the Mondex project. We mentioned in section 2.5 how retrenchment has been used to augment the original Mondex specification, enhancing clarity whilst retaining rigour. This process has produced a number of specifications [BPJS05a, BPJS05b, BJPS06a, BJPS06b, BJPS07] which have been manually created and verified. These specifications would make ideal case studies to illustrate Frog's capabilities. This process would be one of mutual validation, with Frog able to expose any flaw in the specifications and vice versa. Having successfully used Frog to validate the existing case studies, the tool could become a vital resource for the ongoing work to incorporate retrenchment into the Mondex specification.

A second, related area in which Frog could be used to provide support is in Jeske's work [Jes05] to generate new models from existing models (related by refinement and retrenchment). Jeske shows that it is not only possible to use retrenchment and refinement together, but that they complement one another. If the relationships are combined in specific ways then it is possible to take two existing and related models and combine their required properties to directly derive a new model that is implicitly related to the existing models (through retrenchment or refinement). One example provided by Jeske describes three models. The first is a specification, the second is a code refinement of that specification and the third is a retrenchment of the specification that exhibits slightly different properties to the original. The technique 'completes the square' by directly generating (in a mechanizable fashion) a new version of the code that incorporates the changes introduced by the refinement in such a way that the new model produced can be guaranteed to be a refinement of the original retrenchment. Initially, Frog could be used to verify that the relationships automatically generated by the techniques are valid. In the longer term, we would like to implement the techniques described (culminating in a mechanization of the Tower Pattern's framework) so that Frog is able to automate the process of creating and relating the new

model.

Summary

In summary, we can conclude that whilst Frog has been able to successfully meet our objectives, there are many ways in which the tool can be enhanced. Its functionality can be extended and its usability improved. In its current state, the tool will be able to provide support to researchers using retrenchment and other formal techniques for which there was no pre-existing tool support. In the longer term, the tool can be augmented to support more notations and model-generating techniques. Our research has provided a tool that provides mechanized support for retrenchment and which can be extended to aid the whole of the formal methods community.

Appendix A

A Complete Example: Generating Proof Obligations From a Specification

In this appendix we present an additional example illustrating the generation of proof obligations for a machine, examining each of the stages that Frog passes thorough. This example is intended to demonstrate in detail the process by which Frog generates proof obligations from a construct's initial specification and we will therefore not pay particular attention to the configuration and assume the default machine configuration presented in example 5.9 on page 267.

For this example we present a simple machine, *boolean*, capable of storing a boolean value. Our machine declares a single operation that allows us to toggle the value of the boolean. We use the values zero and one to represent our boolean values, but of course, these choices are unimportant.

The machine *boolean* can be specified as follows.

```
MACHINE boolean  
TYPE machine  
SECTION standard_toolkit
```

STATE

$$a : \mathbb{N}; b : \mathbb{P}\mathbb{N}$$

$$a \in b$$

INITIALIZATION

$$a = 0 \wedge b = \{0, 1\}$$
OPERATION *toggle* $\hat{=}$
POST

$$\neg a' = a$$

$$b' = b$$
END *toggle*END *boolean*

The first action that Frog performs is to convert the construct specification into a standard Z section. This is done by taking the construct's configuration and applying the methodology outlined in section 5.3.2.

The resultant section begins with a declaration of its name and an indication that it inherits from the standard Z toolkit.

$$\text{section_machine_boolean parents standard_toolkit}$$

Three schema definition paragraphs then follow and these equate to the machine's clauses. The first specifies the state of the *boolean* machine and indicates that we have two state variables a natural number and a set of natural numbers. The first variable is also restricted, with its value required to belong to the set of natural numbers specified by the second variable.

<i>_boolean_state</i>
$a : \mathbb{N}; b : \mathbb{P}\mathbb{N}$
$(a \in b)$

The next paragraph initializes the machine. In this instance we initialize the machine so that the values zero and one can be used to represent the possible values of our boolean state. From the paragraph's specification we can see that the variables of the state are available in the predicate part of the schema construction expression as Frog uses the clause's configuration when translating to Z.

<i>_boolean_initialization</i>
<i>_boolean_state</i>
$((a = 0)) \wedge ((b = \{0, 1\}))$

The final paragraph declares the postcondition for our *toggle* operation. This indicates that *a* must not end the operation holding the same value it had at the start. With our set *b* having only two values, this ensures that the value of *a* is toggled every time the operation is called.

<i>_boolean_toggle_post</i>
<i>_boolean_state; _boolean_state'</i>
$\neg ((a' = a))$ $(b' = b)$

Once Frog has determined and parsed the Z specification for the construct, it seeks to transform the resultant abstract syntax tree to produce a new specification that satisfies the annotated grammar of [ISO02]. The process by which this transformation is performed is detailed in section 4.6.5. In this instance the application of the syntax transformation rules results in the

following specification which is semantically equivalent to the original.

```
section _machine_boolean parents standard_toolkit
```

```
AX [_boolean_state : {[[a :  $\mathbb{N}$ ]  $\wedge$  [b :  $\mathbb{P}(\mathbb{N})$ ] | (a)  $\in$  (b)]]} END
```

```
AX [_boolean_initialization :
  {[_boolean_state
  | ((a)  $\in$  ({number_literal_0})
   $\wedge$  ((b)  $\in$  ({number_literal_0, number_literal_1})))]}
END
```

```
AX [_boolean_toggle_post :
  {[_boolean_state  $\wedge$  _boolean_state'
  | ( $\neg$  ((a')  $\in$  ({a})))
   $\wedge$  ((b')  $\in$  ({b})))]}
END
```

In the type checking phase that follows, Frog seeks to ensure that the specification has been correctly typed. Success can be judged on the ability to produce a typed abstract syntax tree (AST). Frog applies typing rules through a process described fully in section 4.6.6 to generate such trees. In our example, this results in the production of an AST equivalent to the following specification.

```
section _machine_boolean parents standard_toolkit
```

```

AX ([_boolean_state :
  (({(((([a : ((N)°P(GIVENA))])°P([a : GIVENA]))
  ∧ (([b : ((P((N)°P(GIVENA)))])°P(P(GIVENA))))]
  °P([b : P(GIVENA)]))°P([a : GIVENA; b : P(GIVENA)]))
  | (((a)°GIVENA) ∈ ((b)°P(GIVENA))))]
  °P([a : GIVENA; b : P(GIVENA)]))}
  °P(P([a : GIVENA; b : P(GIVENA)])))
  °P([_boolean_state : P([a : GIVENA; b : P(GIVENA)]))])
END °[_boolean_state : P([a : GIVENA; b : P(GIVENA)]))

```

```

AX ([_boolean_initialization :
  (({(((([_boolean_state]°P([a : GIVENA; b : P(GIVENA)]))
  | (((a)°GIVENA) ∈
    (({((number_literal_0)°GIVENA})°P(GIVENA))))
  ∧ (((b)°P(GIVENA)) ∈ (({(((number_literal_0)°GIVENA,
    (number_literal_1)°GIVENA})°P(GIVENA))}
  °P(P(GIVENA))))])°P([a : GIVENA; b : P(GIVENA)]))}
  °P(P([a : GIVENA; b : P(GIVENA)])))
  °P([_boolean_initialization : P([a : GIVENA; b : P(GIVENA)]))])
END °[_boolean_initialization : P([a : GIVENA; b : P(GIVENA)]))

```

```

AX ([_boolean_toggle_post :
  (({(((([_boolean_state]°P([a : GIVENA; b : P(GIVENA)]))
  ∧ ([_boolean_state']°P([a' : GIVENA; b' : P(GIVENA)]))
  °P([a : GIVENA; b : P(GIVENA); a' : GIVENA; b' : P(GIVENA)]))
  | (¬ (((a')°GIVENA) ∈ (({((a)°GIVENA})
    °P(GIVENA))))))
  ∧ (((b')°P(GIVENA)) ∈ (({((b)°P(GIVENA))}
    °P(P(GIVENA))))))
  °P([a : GIVENA; b : P(GIVENA); a' : GIVENA; b' : P(GIVENA)]))}
  °P(P([a : GIVENA; b : P(GIVENA); a' : GIVENA; b' : P(GIVENA)])))
  °P([_boolean_toggle_post : P([a : GIVENA; b : P(GIVENA);
    a' : GIVENA; b' : P(GIVENA)]))])
END °[_boolean_toggle_post : P([a : GIVENA; b : P(GIVENA);
  a' : GIVENA; b' : P(GIVENA)]))

```

As the production of an AST was possible we can conclude that our specification has been correctly typed.

Once our specification has been typed correctly, we can progress to generating the appropriate proof obligations. As we explain in section 6.2, the first stage of this process is to calculate the semantic binding environment for the Z section representing our construct.

As the semantic binding environment for a section is simply the overridden union of the semantic binding environments of its parents and paragraphs, we will for simplicity consider the construction of the semantic binding environment at the paragraph level.

The first paragraph within our section is that which defines the state of our machine. As the semantic binding environment for a paragraph is dependent on those of its expressions, which in turn are dependent on those of its sub-expressions, it makes most sense to consider the construction of a paragraph's semantic binding environment from the bottom up. The simplest expression within the *_boolean_state* paragraph is shown below.

\mathbb{N}

This expression is a reference expression (rule 6.18) and will produce the following semantic binding environment. Note that, in this example, we will present the semantic binding environments in tabular format to improve clarity and readability. The hypotheses and goals presented in these tables should be assumed to be textual representations of the ZF equivalents.

Bound Object	Hypotheses	Goals
\mathbb{N}	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \mathbb{A}$	

For this expression, the semantic binding environment contains a single semantic binding derived by querying the semantic binding environment of a parent section.

The semantic binding environment for this expression is then used in the construction of semantic binding environments for more complex expressions. The first such expression is shown below.

$$[a : \mathbb{N}]$$

This expression is a variable construction expression (rule 6.25) resulting in a semantic binding environment containing a single binding. This semantic binding inherits the hypotheses of its sub-expression and contains a goal that binds the variable to the sole bound object in the sub-expression's semantic binding environment.

Bound Object	Hypotheses	Goals
a	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$a \in \mathbb{N}$

Our simplest expression is also used in a powerset expression which is in turn used by another variable construction expression as follows.

$$[b : \mathbb{P} \mathbb{N}]$$

The semantic binding environment produced by applying rules 6.22 and 6.25 is shown below. Again a semantic binding environment with a single binding is produced, such that the semantic binding inherits the hypotheses of its sub-expression and the goal is the binding of the variable to the powerset of the sole bound object of the simplest expression's semantic binding environment.

Bound Object	Hypotheses	Goals
b	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$b \in \mathbb{P} \ \mathbb{N}$

In our paragraph, the previous two expressions are conjoined. Therefore, our next step is to determine the semantic binding environment for that expression (rule 6.28) which is shown below.

$$[a : \mathbb{N}] \wedge [b : \mathbb{P} \mathbb{N}]$$

A schema conjunction involves what may be loosely described as a merging

of its component's semantic binding environments. The bound objects of each component's semantic binding environment are added to the expression's semantic binding environment, but the hypotheses and goals of each are augmented by the union of the hypotheses and goals from the semantic binding environment on the other side of the conjunction. In this instance, this results in the following semantic binding environment, in which the objects a and b are bound.

Bound Object	Hypotheses	Goals
a	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \mathbb{A}$	$a \in \mathbb{N}$ $b \in \mathbb{P} \mathbb{N}$
b	$\mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \mathbb{A}$	$a \in \mathbb{N}$ $b \in \mathbb{P} \mathbb{N}$

The conjoined expression is then used within a schema construction expression. However, that expression requires that the semantic information for its enclosed predicate, $a \in b$, be determined first. That predicate is a membership predicate and its semantic information is calculated using rule 6.12, resulting in the following.

$$\mathbb{S}^{\mathbb{P}}(a \in b) = (a \in b \wedge a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N}, \{a, b\})$$

This semantic information can be used alongside the semantic binding environment for our schema conjunction expression to calculate the semantic binding environment for the following schema construction expression (rule 6.26).

$$[[a : \mathbb{N}] \wedge [b : \mathbb{P} \mathbb{N}] \mid a \in b]$$

The list of references in the predicate's semantic information (in this case $\{a, b\}$) is used to determine which bound objects are included in the expression's semantic binding environment. The semantic bindings for these objects have their goals augmented by the constraining term retrieved from the semantic information for the predicate (here, $a \in b \wedge a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N}$).

This results in the following semantic binding environment. (Note that, in order to maintain simplicity, redundant terms in the goal have been removed in the following table.)

Bound Object	Hypotheses	Goals
a	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$
b	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$

The schema construction expression is wrapped in a set extension expression as follows.

$$\{[[a : \mathbb{N}] \wedge [b : \mathbb{P} \ \mathbb{N}] \mid a \in b]\}$$

Rule 6.20 can be applied to this expression resulting in the following semantic binding environment. Note that, the bound object in this instance is a set of semantic bindings, but the goals of the resultant semantic binding are equivalent to the union of those bindings' goals. This results in the following semantic binding environment.

Bound Object	Hypotheses	Goals
$([a, b])$	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$

The set extension expression is used within a variable construction expression to define our schema. This variable construction expression is shown below.

$$[_{boolean_state} : \{[[a : \mathbb{N}] \wedge [b : \mathbb{P} \ \mathbb{N}] \mid a \in b]\}]$$

As with previous variable construction expressions, the hypotheses and goals

of the sub-expression's single semantic binding are duplicated. The goals of the new semantic binding are however augmented by the binding of the new bound object to the bound object in the sub-expression's semantic binding. The following semantic binding environment is generated.

Bound Object	Hypotheses	Goals
<i>_boolean_state</i>	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$_boolean_state \in (\mid a, b \mid)$ $a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$

This expression is now equivalent to the one at the top level of our axiomatic description paragraph (below).

$$\mathbf{AX} \[_boolean_state : \{[a : \mathbb{N}] \wedge [b : \mathbb{P} \ \mathbb{N}] \mid a \in b\}\} \mathbf{END}$$

The semantic binding environment for an axiomatic description paragraph is equal to that of its top level expression's (rule 6.8). The final semantic binding environment for our state paragraph can be concluded, therefore, to be the following.

Bound Object	Hypotheses	Goals
<i>_boolean_state</i>	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	$_boolean_state \in (\mid a, b \mid)$ $a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$

The process for calculating the semantic binding environments of the remaining paragraphs can be calculated in much the same way as the state paragraph. For brevity we will not illustrate exactly how the semantic binding environments for the initialization and toggle paragraphs are generated.

The semantic binding environment for the paragraph dealing with the initialization of our machine has a similar structure to that of our state

paragraph. As such, the resultant semantic binding environment, shown below, is fairly familiar.

Bound Object	Hypotheses	Goals
<i>_boolean</i> <i>_initialization</i>	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	<i>_boolean_initialization</i> $\in (\ a, b \)$ $a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$ $a \in \{number_literal_0\}$ $\wedge b \in \{\{number_literal_0,$ $number_literal_1\}\}$

Similarly, the only significant difference with the paragraph used for the toggle operation is that both the pre and post-transitional states of the machine are used. The semantic binding environment for that paragraph is shown below.

Bound Object	Hypotheses	Goals
<i>_boolean</i> <i>_toggle_post</i>	$\mathbb{A} \in \mathbb{P}(\mathbf{GIVEN} \ \mathbb{A})$ $\mathbb{N} \in \mathbb{P} \ \mathbb{A}$	<i>_boolean_toggle_post</i> $\in (\ a, b, a', b' \)$ $a \in \mathbb{N}$ $b \in \mathbb{P} \ \mathbb{N}$ $a \in b$ $a' \in \mathbb{N}$ $b' \in \mathbb{P} \ \mathbb{N}$ $a' \in b'$ $\neg (a' \in \{a\}) \wedge b' \in \{b\}$

Once the semantic binding environments have been generated it is possible for Frog to instantiate the generic proof obligations that are part of each construct's configuration. The process by which it does this is explained fully in section 6.3.

Our construct is a machine and its first generic proof obligation allows us to show that it is possible to initialize it in a valid state. The tree for this generic proof obligation can be seen in figure 6.3 on page 370 and can be flattened to produce the following theorem.

$$\exists u \bullet u \in \textit{state} \wedge u \in \textit{initialization}$$

The first stage in instantiating this generic proof obligation is to produce relational definitions for the *state* and *initialization* clauses. This is done using the construct's configuration and the semantic binding environment for that construct.

We begin by creating a relational definition for the *state* clause. The first step in this is to retrieve the relational shape from the clause's configuration. In this instance the configuration only references itself. The expression part of our relation is therefore derived from the schema variables in that clause; that is, *a* and *b*. We then use our typed AST to determine the appropriate types for those schema variables and produces the following start to our relational definition of the clause.

$$\{ \langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{ GIVEN } \mathbb{A} \bullet \textit{predicate} \}$$

The semantic model of the construct is then used to determine the appropriate predicate to complete the relational definition. We obtain the semantic binding environment for the paragraph using the appropriate name and then use rule 6.20 to retrieve the semantic bindings for each of the schema variables that were used to create the expression part of the relational definition. The hypotheses for each of these are added to the theorem's hypotheses and the goals for each are conjoined to become the predicate part of the relational definition. We can then substitute this definition into the generic proof obligation giving the following intermediate proof obligation.

$$\begin{aligned}
& \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \\
& \mathbb{N} \in \mathbb{P} \mathbb{A} \\
& \vdash? \\
& \exists u \bullet u \in \{ \langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b \} \\
& \quad \wedge u \in \textit{initialization}
\end{aligned}$$

The relational definition for the *initialization* clause is calculated in exactly the same way. The initialization clause's configuration again states a relational shape consisting solely of the state clause's schema variables. This again gives the following beginning to the relational definition.

$$\{ \langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet \textit{predicate} \}$$

The semantic binding environment associated with the initialization paragraph is then retrieved and its hypotheses and goals used to determine the predicate part of the relational definition. This relational definition can again be substituted into the generic proof obligation. This produces the fully instantiated proof obligation shown below.

$$\begin{aligned}
& \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \\
& \mathbb{N} \in \mathbb{P} \mathbb{A} \\
& \textit{number_literal_0} \in \mathbb{N} \\
& \textit{number_literal_1} \in \mathbb{N} \\
& \vdash? \\
& \exists u \bullet \\
& u \in \{ \langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b \} \\
& \wedge u \in \{ \langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b \\
& \quad \wedge a \in \{ \textit{number_literal_0} \} \\
& \quad \wedge b \in \{ \{ \textit{number_literal_0}, \textit{number_literal_1} \} \} \}
\end{aligned}$$

Hence, we have a suitable initialization proof obligation for our construct.

Obviously, the techniques that we have used to generate the initialization proof obligation can be used to instantiate the other generic proof obligations for our machine.

The semantic binding environment of the state clause's paragraph is used to generate the applicability proof obligation for the toggle operation. This results in the following theorem.

$$\begin{aligned}
& \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \\
& \mathbb{N} \in \mathbb{P} \mathbb{A} \\
& \vdash? \\
& \exists u \bullet \\
& u \in \{\langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b\}
\end{aligned}$$

The generic correctness proof obligation (see figure 6.4) for our construct's toggle operation can be instantiated using the semantic binding environment for that operation's post clause (in this instance we have no inputs, outputs or pre clauses). This proof obligation can be shown to be the following.

$$\begin{aligned}
& \mathbb{A} \in \mathbb{P}(\text{GIVEN } \mathbb{A}) \\
& \mathbb{N} \in \mathbb{P} \mathbb{A} \\
& \text{number_literal_0} \in \mathbb{N} \\
& \text{number_literal_1} \in \mathbb{N} \\
& \vdash? \\
& \forall u \bullet \\
& u \in \{\langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b\} \\
& \Rightarrow \exists u' \bullet \\
& \quad \wedge u' \in \{\langle a, b \rangle : \text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A} \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b\} \\
& \quad \wedge \langle u, u' \rangle \in \{\{\langle a, b \rangle, \langle a', b' \rangle\} : \\
& \quad \quad (\text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A}) \times (\text{GIVEN } \mathbb{A} \times \mathbb{P} \text{GIVEN } \mathbb{A}) \\
& \quad \quad \bullet a \in \mathbb{N} \wedge b \in \mathbb{P} \mathbb{N} \wedge a \in b \wedge a' \in \mathbb{N} \wedge b' \in \mathbb{P} \mathbb{N} \wedge a' \in b' \\
& \quad \quad \wedge \neg (a \in \{a\}) \wedge b' \in \{b\}\}
\end{aligned}$$

Appendix B

Formatting The Z Notation

B.1 Paragraphs

B.1.1 Given Types

In Z, the given types paragraph is typically formatted in the manner shown below. This thesis follows this convention.

[*name*]

B.1.2 Axiomatic Description

In Z, the axiomatic description paragraph is typically formatted in the manner shown below. This thesis follows this convention.

| *schema_text*

The generic version of the paragraph has the following formatting.

┌ [*name*] ───────────────────────────────────
└ *schema_text*

B.1.3 Schema Definition

In Z, the schema definition paragraph is typically formatted in the manner shown below. This thesis follows this convention.

\textit{name} $\textit{schema_text}$
--

The generic version of the paragraph has the following formatting.

$\textit{name} [\textit{name}]$ $\textit{schema_text}$
--

B.1.4 Horizontal Definition

In Z, the horizontal definition paragraph is typically formatted in the manner shown below. This thesis follows this convention.

$$\textit{name} == \textit{expression}$$

The generic version of the paragraph has the following formatting.

$$\textit{name}[\textit{name}] == \textit{expression}$$

B.1.5 Generic Operator Definition

In Z, the generic operator definition paragraph is typically formatted in the manner shown below. This thesis follows this convention.

$$\textit{name} == \textit{expression}$$

B.1.6 Free Type

In Z, the free type paragraph is typically formatted in the manner shown below. This thesis follows this convention.

$$TYPE ::= alt1 \mid alt2 \mid alt3$$

B.1.7 Conjecture

In Z, the conjecture paragraph is typically formatted in the manner shown below. This thesis follows this convention.

$$\vdash? \textit{predicate}$$

The generic version of the paragraph has the following formatting.

$$[name] \vdash? \textit{predicate}$$

B.1.8 Operator Template

In Z, the operator template paragraph is typically formatted in the manner shown below. This thesis follows this convention. The *operator_definition* allows the specifier to indicate the associativity, precedence and operands of the operator.

$$\textit{relation operator_definition}$$

$$\textit{function operator_definition}$$

$$\textit{generic operator_definition}$$

B.2 Schema Text

In Z, a schema text can be written in one of the two ways shown below.

$$\textit{declaration} \mid \textit{predicate}$$

declaration

predicate

In this thesis the first style will be used when the schema text can be expressed in a single line, and the second when multiple lines are required. When a schema text is used as part of an expression or predicate, however, the formatting of that expression or predicate will take precedence.

B.3 Expressions

B.3.1 Conditional

In Z, the conditional expression is typically written as follows.

if predicate then expression else expression

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

if predicate
then expression
else expression

Where conditional expressions are nested, we will indent to improve readability.

B.3.2 Schema Composition

In Z, the schema composition expression is typically written in the manner shown below. This thesis follows this convention.

expression § expression

B.3.3 Schema Piping

In Z, the schema piping expression is typically written in the manner shown below. This thesis follows this convention.

$$expression \gg expression$$

B.3.4 Schema Hiding

In Z, the schema hiding expression is typically written in the manner shown below. This thesis follows this convention.

$$expression \setminus name$$

B.3.5 Cartesian Product

In Z, the cartesian product expression is typically written in the manner shown below. This thesis follows this convention.

$$expression \times expression$$

B.3.6 Schema Projection

In Z, the schema projection expression is typically written in the manner shown below. This thesis follows this convention.

$$expression \upharpoonright expression$$

B.3.7 Schema Precondition

In Z, the schema precondition expression is typically written in the manner shown below. This thesis follows this convention.

$$\text{pre } expression$$

B.3.8 Binding Selection

In Z, the binding selection expression is typically written in the manner shown below. This thesis follows this convention.

expression.name

B.3.9 Tuple Selection

In Z, the tuple selection expression is typically written in the manner shown below. This thesis follows this convention.

expression.number

B.3.10 Binding Construction

In Z, the binding construction expression is typically written in the manner shown below. This thesis follows this convention.

θ *expression*

B.3.11 Schema Negation

In Z, the schema negation expression is typically written in the manner shown below. This thesis follows this convention.

\neg *expression*

B.3.12 Schema Conjunction

In Z, the schema conjunction expression is typically written in the manner shown below. This thesis follows this convention.

expression \wedge *expression*

B.3.13 Schema Disjunction

In Z, the schema disjunction expression is typically written in the manner shown below. This thesis follows this convention.

$$\textit{expression} \vee \textit{expression}$$

B.3.14 Schema Equivalence

In Z, the schema equivalence expression is typically written in the manner shown below. This thesis follows this convention.

$$\textit{expression} \Leftrightarrow \textit{expression}$$

B.3.15 Schema Implication

In Z, the schema implication expression is typically written in the manner shown below. This thesis follows this convention.

$$\textit{expression} \Rightarrow \textit{expression}$$

B.3.16 Schema Existential Quantification

In Z, the schema existential quantification expression, and schema unique existential expression are typically written as follows.

$$\exists \textit{schema_text} \bullet \textit{expression}$$

$$\exists_1 \textit{schema_text} \bullet \textit{expression}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \exists \textit{ schema_text_declaration} \\ | \textit{ schema_text_predicate} \\ \bullet \textit{ expression} \end{array}$$

$$\begin{array}{l} \exists_1 \textit{ schema_text_declaration} \\ | \textit{ schema_text_predicate} \\ \bullet \textit{ expression} \end{array}$$

Where existential quantification expressions are nested, we will indent to improve readability.

B.3.17 Schema Universal Quantification

In Z, the schema universal quantification expression is typically written as follows.

$$\forall \textit{ schema_text} \bullet \textit{ expression}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \forall \textit{ schema_text_declaration} \\ | \textit{ schema_text_predicate} \\ \bullet \textit{ expression} \end{array}$$

Where schema universal quantification expressions are nested, we will indent to improve readability.

B.3.18 Function Construction

In Z, the function construction expression is typically written as follows.

$$\lambda \textit{ schema_text} \bullet \textit{ expression}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will –

for ease of understanding – format the expression as follows.

$$\begin{array}{l} \lambda \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{expression} \end{array}$$

Where function construction expressions are nested, we will indent to improve readability.

B.3.19 Substitution

In Z, the substitution expression is typically written as follows.

$$\textit{let name} == \textit{expression} \bullet \textit{expression}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \textit{let name} == \textit{expression} \\ \bullet \textit{expression} \end{array}$$

Where substitution expressions are nested, we will indent to improve readability.

B.3.20 Definite Description

In Z, the definite description expression is typically written as follows.

$$\mu \textit{schema_text} \bullet \textit{expression}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \mu \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{expression} \end{array}$$

Where definite description expressions are nested, we will indent to improve readability.

B.3.21 Characteristic Definite Description

In Z, the characteristic definite description expression is typically written as follows.

$$\mu \textit{schema_text}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \mu \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \end{array}$$

Where characteristic definite description expressions are nested, we will indent to improve readability.

B.3.22 Set Extension

In Z, the set extension expression is typically written in the manner shown below. This thesis follows this convention.

$$\{\textit{expression}\}$$

B.3.23 Set Comprehension

In Z, the set comprehension expression is typically written as follows.

$$\{\textit{schema_text} \bullet \textit{expression}\}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \{ \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{expression} \} \end{array}$$

Where set comprehension expressions are nested, we will indent to improve readability.

B.3.24 Characteristic Set Comprehension

In Z , the characteristic set comprehension expression is typically written as follows.

$$\{ \textit{schema_text} \}$$

In this thesis we follow this convention except where the definition of the expression requires more than one line of text. In these instances we will – for ease of understanding – format the expression as follows.

$$\begin{array}{l} \{ \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \} \end{array}$$

Where characteristic set comprehension expressions are nested, we will indent to improve readability.

B.3.25 Tuple Extension

In Z , the tuple extension expression is typically written in the manner shown below. This thesis follows this convention.

$$(\textit{expression})$$

B.4 Predicates

B.4.1 Literals

In Z, the literals are typically written in the manner shown below. This thesis follows this convention.

`true`

`false`

B.4.2 Membership

In Z, the membership predicate is typically written in the manner shown below. This thesis follows this convention.

expression \in *expression*

B.4.3 Equality

In Z, the equality predicate is typically written in the manner shown below. This thesis follows this convention.

expression = *expression*

B.4.4 Negation

In Z, the negation predicate is typically written in the manner shown below. This thesis follows this convention.

\neg *predicate*

B.4.5 Conjunction

In Z, the conjunction predicate is typically written in the manner shown below. This thesis follows this convention.

predicate \wedge *predicate*

It is also possible to indicate the conjunction of two predicates through the use of a semicolon.

predicate; *predicate*

Finally, two predicates on adjacent lines are considered to be conjoined.

predicate
predicate

B.4.6 Disjunction

In Z, the disjunction predicate is typically written in the manner shown below. This thesis follows this convention.

predicate \vee *predicate*

B.4.7 Equivalence

In Z, the equivalence predicate is typically written in the manner shown below. This thesis follows this convention.

predicate \Leftrightarrow *predicate*

B.4.8 Implication

In Z, the implication predicate is typically written in the manner shown below. This thesis follows this convention.

predicate \Rightarrow *predicate*

B.4.9 Existential Quantification

In Z, the existential quantification predicate, and unique existential predicate are typically written as follows.

$$\exists \textit{schema_text} \bullet \textit{predicate}$$

$$\exists_1 \textit{schema_text} \bullet \textit{predicate}$$

In this thesis we follow this convention except where the definition of the predicate requires more than one line of text. In these instances we will – for ease of understanding – format the predicate as follows.

$$\begin{array}{l} \exists \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{predicate} \end{array}$$

$$\begin{array}{l} \exists_1 \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{predicate} \end{array}$$

Where existential quantification predicates are nested, we will indent to improve readability.

B.4.10 Universal Quantification

In Z, the universal quantification predicate is typically written as follows.

$$\forall \textit{schema_text} \bullet \textit{predicate}$$

In this thesis we follow this convention except where the definition of the predicate requires more than one line of text. In these instances we will – for ease of understanding – format the predicate as follows.

$$\begin{array}{l} \forall \textit{schema_text_declaration} \\ | \textit{schema_text_predicate} \\ \bullet \textit{predicate} \end{array}$$

Where universal quantification predicates are nested, we will indent to improve readability.

B.5 The Mathematical Toolkits

A number of mathematical toolkits are usually used to extend the power of the Z notation. Each toolkit defines a number of literals, operators, sets and types that may be used within this thesis. The symbols used for each literal, operator, set or type are given in this section. Note that the full definitions for the operators are not given here, any operators used in this thesis will be defined in C. A full formal definition of each operator is available in [ISO02]

B.5.1 Prelude

- \mathbb{P} — powerset
- \mathbb{A} — the given type of numbers
- \mathbb{N} — the set of natural numbers
- *number_literal_0* — the literal value of the number 0
- *number_literal_1* — the literal value of the number 1
- $+$ — arithmetic addition

B.5.2 Set Toolkit

- \leftrightarrow — relation
- \rightarrow — total function
- \neq — inequality
- \notin — non-membership
- \emptyset — emptyset
- \subseteq — subset
- \subset — proper subset
- \mathbb{P}_1 — non-empty subsets

- \cup — set union
- \cap — set intersection
- \setminus — set difference
- \oplus — symmetric difference
- \bigcup — generalized union
- \bigcap — generalized intersection
- \mathbb{F} — finite sets

B.5.3 Relation Toolkit

- *first* — first component projection
- *second* — second component projection
- \mapsto — maplet
- *dom* — domain
- *ran* — range
- *id* — identity
- \circledast — relational composition
- \circ — functional composition
- \triangleleft — domain restriction
- \triangleright — range restriction
- \triangleleft — domain subtraction
- \triangleright — range subtraction
- \sim — relational inversion

- $(\)$ — relational image
- \oplus — override
- $+$ — transitive closure
- $*$ — reflexive transitive closure

B.5.4 Function Toolkit

- \rightarrow — partial function
- \mapsto — total injection
- \rightsquigarrow — partial injection
- \twoheadrightarrow — total surjection
- \twoheadrightarrow — partial surjection
- $\xrightarrow{\sim}$ — bijection
- \mapsto — finite function
- \rightsquigarrow — finite injection
- *disjoint* — disjointness
- *partition* — partition

B.5.5 Number Toolkit

- *succ* — successor
- \mathbb{Z} — the set of integers
- $-$ — arithmetic negation
- $-$ — arithmetic subtraction
- \leq — less than or equal

- $<$ — less than
- \geq — greater than or equal
- $>$ — greater than
- \mathbb{N}_1 — strictly positive natural numbers
- \mathbb{Z}_1 — non-zero integers
- $*$ — arithmetic multiplication
- div — arithmetic division
- mod — arithmetic modulus

B.5.6 Sequence Toolkit

- $..$ — number range
- $iter$ — iteration
- $\#$ — cardinality of a set
- min — minimum
- max — maximum
- $items$ — items
- seq — finite sequences
- seq_1 — non-empty finite sequences
- $iseq$ — injective sequences
- $\langle \rangle$ — sequence enumeration
- \wedge — sequence concatenation
- rev — reverse

- *head* — head of a sequence
- *last* — last of a sequence
- *tail* — tail of a sequence
- *front* — front of a sequence
- *squash* — squash
- \uparrow — extract
- \downarrow — filter
- *prefix* — prefix relation
- *suffix* — suffix relation
- *infix* — infix relation
- $\wedge/$ — distributed concatenation

B.5.7 Standard Toolkit

The standard toolkit introduces no operators.

C.3 Domain Subtraction

The domain subtraction operator, \triangleleft , belongs to the relation toolkit. The domain subtraction of a set from a relation gives the set of ordered pairs that belong to the relation, but whose first elements do not belong to the set. The operator is defined as follows.

$$\begin{array}{l} \overline{\overline{[X, Y]}} \\ \hline \hline _ \triangleleft _ : (\mathbb{P} X \times (X \leftrightarrow Y)) \rightarrow (X \leftrightarrow Y) \\ \hline \forall a : \mathbb{P} X; r : X \leftrightarrow Y \bullet a \triangleleft r = \{p : r \mid p.1 \notin a\} \end{array}$$

C.4 Extraction

The extraction operator, \upharpoonright , belongs to the sequence toolkit. The extraction of a set of indices from a sequence is the sequence formed by discarding all members of the sequence that have indices not included in the given set of indices, and re-indexing. The extraction operator is defined as follows.

$$\begin{array}{l} \overline{\overline{[X]}} \\ \hline \hline _ \upharpoonright _ : (\mathbb{P} \mathbb{N} \times \text{seq } X) \rightarrow \text{seq } X \\ \hline \forall a : \mathbb{P} \mathbb{N}; s : \text{seq } X \bullet a \upharpoonright s = \text{squash}(a \triangleleft s) \end{array}$$

C.5 Finite Sequence

The finite sequence operator, seq , belongs to the sequence toolkit and defines the set of all finite sequences containing the elements of a given set. Each finite sequence is a finite indexed set of elements that all belong to a given

set where the index is a set of contiguous positive integers that begins at one. Formally, this is defined as follows.

$$\begin{aligned} \text{seq } X &== \{f : \mathbb{N} \\ \text{fun } X \mid \text{dom } f = 1 .. \#f\} \end{aligned}$$

C.6 Generalized Intersection

The generalized intersection operator, \bigcap , belongs to the set toolkit. The generalized intersection of a set of sets of elements is the set of elements that belong to each of the elements sets in the containing set. The operator is defined as follows.

$$\begin{array}{|l} \hline [X] \\ \hline \bigcap : \mathbb{P} \mathbb{P} X \rightarrow \mathbb{P} X \\ \hline \forall A : \mathbb{P} \mathbb{P} X \bullet \bigcap A = \{x : X \mid \forall a : A \bullet x \in a\} \end{array}$$

C.7 Identity

The identity operator, id , belongs to the relation toolkit. The identity relation of a set (produced by applying the identity operator to that set) contains a set of ordered pairs that relate every member of the set to itself. Formally, this is defined as follows.

$$\text{id } X == \{x : X \bullet x \mapsto x\}$$

C.8 Injective Sequence

The injective sequence operator, iseq , belongs to the sequence toolkit and defines the set of all injective sequences containing the elements of a given set. Each injective sequence is a finite sequence (see C.5) and also an injection (see C.9). This is defined as follows.

$$\text{iseq } X == \text{seq } X \cap (\mathbb{N} \rightsquigarrow X)$$

C.9 Partial Injection

The partial injection operator, \rightsquigarrow , belongs to the function toolkit and defines the set of all partial injections between two sets. The partial injection between two sets is the set of ordered pairs whose first element is a member of the first set, and the second element a member of the second set. This set is formed such that every element of the first set is related to no more than one element of the second set and every element of the second set is related to no more than one element of the first set. Formally, this is defined as follows.

$$X \rightsquigarrow Y == \{f : X \leftrightarrow Y \mid \forall p, q : f \bullet p.1 = q.1 \Leftrightarrow p.2 = q.2\}$$

C.10 Power Set

The power set operator, \mathbb{P} , belongs to the prelude. The power set of a set is the set of all subsets of that set, and is defined as follows.

$$\forall A; B \bullet B \in \mathbb{P} A \Leftrightarrow (\forall C \mid C \in B \bullet C \in A)$$

C.11 Range

The range operator, ran , belongs to the relation toolkit. The range of a relation is the set of elements that form the second element in each of the relation's ordered pairs. Formally it is defined as follows.

$$\begin{array}{l} \text{[X, Y]} \\ \text{ran} : (X \leftrightarrow Y) \rightarrow \mathbb{P} Y \\ \forall r : X \leftrightarrow Y \bullet \text{ran } r = \{p : r \bullet p.2\} \end{array}$$

as follows.

$$\frac{\frac{[X]}{_ \cup _ : (\mathbb{P} X \times \mathbb{P} X) \rightarrow \mathbb{P} X}}{\forall a, b : \mathbb{P} X \bullet a \cup b = \{x : X \mid x \in a \vee x \in b\}}$$

C.18 Total Function

The total function operator, \rightarrow , belongs to the set toolkit and defines the set of all total functions between two sets. The total function between two sets is the set of ordered pairs whose first element is a member of the first set, and the second element a member of the second set. This set is formed such that every element of the first set is related to exactly one element of the second set. Formally, this is defined as follows.

$$X \rightarrow Y == \{f : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f\}$$

C.19 Transitive Closure

The transitive closure operator, $^+$, belongs to the relation toolkit. The transitive closure of a relation is the smallest set that contains that relation and is closed under the action of composing the relation with its members. This can be defined formally as follows.

$$\frac{\frac{[X]}{_ ^+ : (X \leftrightarrow X) \rightarrow (X \leftrightarrow X)}}{\forall r : X \leftrightarrow X \bullet r^+ = \bigcap \{s : X \leftrightarrow X \mid r \subseteq s \wedge r \circ s \subseteq s\}}$$

Appendix D

Axioms and Lemmas

D.1 Existential Quantification One Point Rule

$$(\exists x : A \bullet p \wedge x = t) \Leftrightarrow t \in A \wedge p[t/x]$$

Where x is not free in t .

D.2 Law of Excluded Middle

$$p \vee \neg p \Leftrightarrow \text{true}$$

D.3 Laws of Simplification

\vee -simplification

$$p \vee p \Leftrightarrow p$$

$$p \vee \text{true} \Leftrightarrow \text{true}$$

$$p \vee \text{false} \Leftrightarrow p$$

\wedge -simplification

$$\begin{aligned}
p \wedge p &\Leftrightarrow p \\
p \wedge \text{true} &\Leftrightarrow p \\
p \wedge \text{false} &\Leftrightarrow \text{false}
\end{aligned}$$

 \Rightarrow -simplification

$$p \Rightarrow p$$

D.4 Set Comprehension One Point Rule

Note that set comprehension is just another form of quantification.

$$t \in \{x : A \mid p \bullet x\} \Leftrightarrow t \in A \wedge p[t/x]$$

Where x is not free in t .

D.5 Universal Quantification One Point Rule

$$(\forall x : A \bullet p \wedge x = t) \Leftrightarrow t \in A \Rightarrow p[t/x]$$

Where x is not free in t .

Appendix E

Annotated Lexical Grammar

This appendix presents the lexical grammar that is used by Frog to parse character streams produced from Z specifications in the \LaTeX representation. We have provided brief annotation to this grammar, particularly where it is necessary to explain differences in syntax between ANTLR and EBNF, and where our rules differ significantly from those described in [ISO02].

In ANTLR all rules generate a token that is added to the token stream. How then is it possible to split complicated rules into simpler, reusable components? ANTLR achieves this through the use of the ‘`protected`’ keyword (which is completely unrelated to the nominally-equivalent keyword of Java). When reading the following grammar, we must note that `protected` rules do not generate tokens. Apparent ambiguity between `protected` rules is typically resolved by a non-`protected` rule which operates directly on the input character stream.

We will begin by presenting the rules for defining a Z character that are equivalent to those specified in section 6 of [ISO02].

We do not define rules for the tokens `ZCHAR`, `SPECIAL`, or `BRACKET` as these are simply wrapper tokens without any genuine purpose.

```
protected DIGIT : '0' .. '9';
```

```
protected LETTER : LATIN | GREEK | OTHERLETTER;
```

```
protected LATIN : ('a'..'z' | 'A'..'Z');
```

```
protected GREEK : ("\\Delta" | "\\lambda" | "\\mu"
                  | "\\theta" | "\\Xi")
                (SOFT_SPACE)*;
```

```
protected OTHERLETTER : "\\arithmos"| "\\power"| "\\nat";
```

The rules for the definition of digits and letters are essentially the same as in [ISO02]. The only difference being the use of ANTLR's range operator to abbreviate the definition of DIGIT and LATIN, and the consumption of white space following a Greek letter (we have omitted the Java code that strips the white space from the resulting token)¹.

```
protected STROKECHAR : '\\'' | '?'' | '!';
```

```
protected WORDGLUE : '_'' | '^'' | "\\_";
```

The other special characters that can appear as part of a word are again described similarly to [ISO02]. At present we handle only single character subscript and superscript.

```
protected
BOXCHAR
  options { testLiterals= true; }
  :      "\\begin{"
        (      "axdef}"
          |      "schema}"
          |      "gendif}"
          |      "syntax}"
          |      "zsection}"
          |      "zed}"
          |      "machine}");
```

¹This behaviour is described in section A.2.7 of [ISO02].

```

        |      "relationship}"
        |      "openv}"
        )
;

protected
END      :      "\\end{"
        (      "axdef}"
        |      "schema}"
        |      "gendif}"
        |      "syntax}"
        |      "zsection}"
        |      "zed}"
        |      "machine}"
        |      "relationship}"
        |      "openv}"
        )
;

```

The rules for the characters that mark the beginning and ending of Z boxes highlight another ANTLR feature. The ‘testLiterals’ option allows us to create a symbol table linking literals with specific token types, that subdivides a generic token type. For instance, the character string ‘\begin{schema}’, is initially considered to be of token type BOXCHAR, but when tested against the literal table, this type is refined to SCH. Note that we use the testLiterals option for the character strings marking the start of boxes, but that all ending strings resolve to the same token of type END.

```
protected LATEX_SYMBOL : "\\\" ('a'..'z' | 'A'..'Z')*
```

```
protected
SYMBOL
    options { testLiterals= true; }
: '@' | '+' | '-' | '*' | '<' | '>' | '=' | '&' | ':'
```

```

| ';' | '/' | '.'
| symbol:LATEX_SYMBOL
  { if (!characterDictionary.lookup(LATEX,symbol))
    { throwError(); }
  }

```

In order to be able to support any user-defined symbol, we use a generic rule to match latex markups and then confirm that they are valid through the use of the character dictionary (see 4.6.3).

We have now defined our rules equivalent to those used to create Z characters, and move on to the general lexical rules defined in section 7 of [ISO02]. Again, we do not consider the rules `TOKENSTREAM` and `TOKEN` as ANTLR does not require such wrapper classes.

We do not need to define rules for tokens such as `AX` and `SCH` as these tokens are assigned through the use of the literals symbol table as described above.

```
protected NLCHAR : "\\\";
```

The `NLCHAR` character is considered to be the ‘hard’ version, that is a newline explicitly defined in the \LaTeX notation. This character is treated differently depending on its context, for more details see below.

```
NUMBER : (DIGIT)+;
```

Our first top-level token is `NUMBER` which is produced from strings of digits.

```
protected
NAME
  options { testLiterals= true; }
  : WORD (STROKECHAR)?;
```

```
protected
WORD : ( WORD_PART
        )+
```

```

    | (SYMBOL) =>
        SYMBOL
        (
            SYMBOL
        )*
        (
            WORD_PART
        )*
    | (LETTER) =>
        LETTER
        ( (LETTER | DIGIT) =>
            ALPHA_STR
        )?
        (
            WORD_PART
        )*
;

protected
WORD_PART
:
    WORDGLUE
    ( (LETTER | DIGIT) =>
        ALPHA_STR
    | (SYMBOL) =>
        SYMBOL_STR
    )
;

protected ALPHA_STR : (LETTER | DIGIT)+;

protected SYMBOL_STR : (SYMBOL)+;

```

The next set of rules handle the combination of characters to form words. Note that, the `NAME` rule is set as `protected`. The reasons for this will be explained below, for now it is only important to realize that this rule does not create `NAME` tokens directly. The rules presented here are essentially the same

as those presented in [ISO02]. Syntactic predicates are used to disambiguate between letters and symbols where necessary (as letters and symbols can both begin with several identical characters, the use of syntactic predicates is preferable to the use of a large lookahead).

LEFTBR : '(';

RIGHTBR : ')';

LEFTSQ : '[';

RIGHTSQ : ']';

LEFTCU : '{';

RIGHTCU : '}';

EQUIVALENCE : "==";

COMMA : ',';

DBLCOMMA : ",,";

protected LEFTCURLY : "\\{";

protected RIGHTCURLY : "\\}";

protected HALFPIPE : "\\|" | '|';

protected THEOREM : "\\thrm";

These rules handles character and strings that are required as tokens, but that are not covered by the symbol rule.

```
protected SOFT_SPACE : ' ' | '\t' | '\n' | '\r';
```

```
protected SPACE : SOFT_SPACE | '~';
```

These rules allow white space to be matched and discarded.

```
protected
```

```
KEYWORD
```

```
options { testLiterals= true; }
      : "\\\"
          (
            "ELSE"
          | "IF"
          | "LET"
          | "SECTION"
          | "THEN"
          | "clause"
          | "function"
          | "generic"
          | "leftassoc"
          | "parents"
          | "pre"
          | "relation"
          | "rightassoc"
          )
      ;
```

The KEYWORD rule specifies the reserved words that cannot be used as symbols.

```
RESOLVE
```

```
: (SPACE)+
    { $setType(Token.SKIP); }
| ("\\where" | ' | ') => HALFPIPE
    { $setType(HALFPIPE); }
| ("\\_" ) =>
```

```

    UNDERSCORE
    { $setType(UNDERSCORE); }
| ("\\{") => LEFTCURLY
    { $setType(LEFTCURLY); }
| ("\\}") => RIGHTCURLY
    { $setType(RIGHTCURLY); }
| ("\\\\") => NLCHAR
    { if (isCurrentTokenHardNLChar())
        { $setType(NLCHAR); }
      else
        { $setType(Token.SKIP); }
    }
| ("\\thrm") => THEOREM
    { $setType(THEOREM); }
| (KEYWORD) => KEYWORD
    { $setType(KEYWORD); }
| ("\\end") => END
    { $setType(END); }
| ("\\begin") => BOXCHAR
    { $setType(BOXCHAR); }
| name:NAME
    { if (currentTokenEnvironment.isDefined(name))
        { $setType(currentTokenEnvironment
                    .getTokenType(name)); }
      else
        { $setType(NAME); }
    }
;

```

The **RESOLVE** rule is responsible for producing all other tokens (although no token with type **RESOLVE** is ever produced). It is here that the inherent ambiguity in the previous definitions is resolved. The alternatives in the rule are matched by order of precedence from top to bottom. That is, a **NAME**

token will only ever be produced if none of the other rules can be matched. Syntactic predicates protect each alternative, so we will only attempt to match them if we know they can succeed. Note that, the `$setType` method is an ANTLR method that allows us to dynamically declare the type of token.

When matching the new line character we apply the rules we discussed in section 4.6.3, if the character is determined to be ‘hard’ we incorporate it in the token stream; otherwise, we ignore it.

When matching name tokens we must ensure that they are not actually tokens from user-defined operators. We perform a look-up on the current token environment, and if necessary set the appropriate token type. The reasons for this are also discussed in section 4.6.3.

Table E.1: The Literals Symbol Table

Token	String
AX	<code>\begin{axdef}</code>
SCH	<code>\begin{schema}</code>
GENAX	<code>\begin{gendef}</code>
ZED	<code>\begin{zed}</code>
SYNTAX	<code>\begin{syntax}</code>
SECTION	<code>\begin{zsection}</code>
MACHINE	<code>\begin{machine}</code>
RELATIONSHIP	<code>\begin{relationship}</code>
OPERATION_ENV	<code>\begin{openv}</code>
MINUS	<code>\-</code>
HASH	<code>\#</code>
FWDSLASH	<code>\/</code>
AT	<code>@</code>
CAT	<code>\cat</code>
CROSS	<code>\cross</code>
DCAT	<code>\dcat</code>
DISJOINT	<code>\disjoint</code>
EXISTS	<code>\exists</code>
EXISTS1	<code>\exists_1</code>
EXTRACT	<code>\extract</code>
FILTER	<code>\filter</code>
FORALL	<code>\forall</code>
HIDE	<code>\hide</code>
IFF	<code>\iff</code>
IMPLIES	<code>\implies</code>
IN	<code>\in</code>

Token	String
LAMBDA	\lambda
LAND	\land
LANGLE	\langle
LBLLOT	\lblot
LDATA	\ldata
LNOT	\lnot
LOR	\lor
MODELS	\models
MU	\mu
NUM	\num
PING	\ping
PIPE	\pipe
PROJECT	\project
RANGLE	\rangle
RBLLOT	\rblot
RDATA	\rdata
SEMI	\semi
SEQ	\seq
THETA	\theta
KW_CLAUSE	\clause
KW_ELSE	\ELSE
KW_FUNCTION	\function
KW_GENERIC	\generic
KW_IF	\IF
KW_LEFTASSOC	\leftassoc
KW_LET	\LET

Token	String
KW_PARENTS	\parents
KW_PRE	\pre
KW_RELATION	\relation
KW_RIGHTASSOC	\rightassoc
KW_SECTION	\SECTION
KW_THEN	\THEN
KW_TRUE	true
KW_FALSE	false
EQUALS	=
AMPERSAND	&
COLON	:
SEMICOLON	;
FWDSLASH	/
FULLSTOP	.

Appendix F

Annotated Parsing Grammar

In this appendix we present – for completeness and reference – the ANTLR grammar used for parsing Frog files. It will become obvious that, through the use of ANTLR’s parsing tricks – particularly syntactic and semantic predicates – we have been able to produce a machine readable grammar that is very close to the human readable grammar presented in [ISO02]. Although the use of additional lookahead in this way may produce a performance drag when using the tool, we have preferred to stick to the style of the standard ensuring not only compliance (and therefore, correctness), but also ease of readability and maintainability.

Rule F.1 - start:

```
start : z_file | modelling_file;
```

Each file contains either a specification in the Z notation or in our modelling notations that is, a construct). We consider first those files that contain specifications solely in the Z notation.

Rule F.2 - z_file:

```
z_file : (section | paragraph)+;  
        { setRoot(Z_FILE); }
```

Rule F.3 - section:

```
section : SECTION KW_SECTION NAME
        (KW_PARENTS NAME (COMMA NAME)*)? END
        (options {greedy= true;} : paragraph)*;
```

Our definition of a Z file differs slightly from the definition of a specification in [ISO02] in that we allow an anonymous section to be declared at the beginning of a sectioned specification. We have found this format more convenient whilst developing the tool, and as it only extends the number of valid file we have left this way presently. In the future, we propose a ‘strict ISO standard’ option within the tool that will allow specifications to be checked strictly to the letter of the standard (or more loosely) as required. We ensure that paragraphs are associated with their section rather than the file through the use of the greedy option when matching paragraphs at the end of a section.

Rule F.4 - paragraph:

```
paragraph
: (ZED LEFTSQ NAME (COMMA NAME)* RIGHTSQ END)
=> ZED! LEFTSQ NAME (COMMA NAME)* RIGHTSQ END
| AX schema_text END
| SCH LEFTCU! paragraphNameVar:NAME RIGHTCU!
  (LEFTSQ formals RIGHTSQ)? schema_text END
| GENAX LEFTSQ formals RIGHTSQ schema_text END
| (ZED decl_name (LEFTSQ formals RIGHTSQ)?
  EQUIVALENCE expression[NOT_PART_OF_PREDICATE])
=> ZED! decl_name (LEFTSQ formals RIGHTSQ)?
  EQUIVALENCE expression[NOT_PART_OF_PREDICATE] END
  { setRoot(paragraph,HORIZ_DEF_PARA); }
| ZED! gen_name EQUIVALENCE
  expression[NOT_PART_OF_PREDICATE] END
  { setRoot(paragraph,GEN_OP_DEF_PARA); }
| SYNTAX! free_type (AMPERSAND free_type)* END
  { setRoot(paragraph,FREE_TYPE_PARA); }
```

```

| ZED! (LEFTSQ formals RIGHTSQ)? THEOREM predicate END
    { setRoot(paragraph,CONJECT_PARA); }
| ZED! operator_template END;

```

Rule F.5 - free_type:

```
free_type : NAME DEFINITION branch (HALFPIPE branch)*;
```

Rule F.6 - branch:

```
branch : decl_name
        (LDATA expression[NOT_PART_OF_PREDICATE] RDATA)?;
```

Rule F.7 - formals:

```
formals : NAME (COMMA NAME)*;
```

The paragraphs are defined in the same order as within the grammar specified in [ISO02]. That is, from top to bottom: given type, axiomatic description, schema definition (both non-generic and generic), generic axiomatic description, horizontal definition (both non-generic and generic), generic operator definition, free type, conjecture (both non-generic and generic) and operator template. A syntactic predicate is required to distinguish given type and generic conjecture paragraphs; both can begin with a square bracket followed by a list of names. We don't see this as a problem as we do not foresee huge lists of given types, and in any case the processing of such a list is not particularly complex. More of an issue is the syntactic predicate required to disambiguate horizontal definition and generic operator definition paragraphs; as both can match identical sentences we gave precedence to the horizontal definition paragraph as we believe this is what is intended by the standard.

Note that, the '!' decoration on some of the tokens above, indicates that they are discarded when creating the abstract syntax tree.

Rule F.8 - predicate:

```
predicate : pred_quantification
           ((NLCHAR^|SEMICOLON^) pred_quantification)*;
```

Rule F.9 - pred_quantification:

```
pred_quantification
  : (FORALL) => FORALL^ schema_text AT pred_quantification
  | (EXISTS) => EXISTS^ schema_text AT pred_quantification
  | (EXISTS1) => EXISTS1^ schema_text AT pred_quantification
  | pred_equivalence
  ;
```

Rule F.10 - pred_equivalence:

```
pred_equivalence : pred_implication (IFF^ pred_implication)*;
```

Rule F.11 - pred_implication:

```
pred_implication : pred_disjunction (IMPLIES^ pred_disjunction)*;
```

Rule F.12 - pred_disjunction:

```
pred_disjunction : pred_conjunction (LOR^ pred_conjunction)*;
```

Rule F.13 - pred_conjunction:

```
pred_conjunction : pred_negation (LAND^ pred_negation)*;
```

Rule F.14 - pred_atom:

```
pred_atom : (relation) => relation
           | (expression[PART_OF_PREDICATE])
             => expression[PART_OF_PREDICATE]
           | KW_TRUE
           | KW_FALSE
```

```
| LEFTBR ^ predicate RIGHTBR!;
```

The predicates are defined in much the same way as in [ISO02], the only noticeable differences being the syntactic difference enforced in translation from human-readable grammar to ANTLR grammar. Syntactic predicates in our quantification clauses to disambiguate between predicate quantifiers and schema quantifiers, a discussion of the ways in which expression and predicates are disambiguated is given in section 4.6.4.

Note that, the `^` decorate on a token indicates that the token specified is the one to be used as the root of the tree generated by the rule (or sub-rule where applicable).

Rule F.15 - expression:

```
expression[boolean isPartOfPredicate]
  : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    FORALL ^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
  | { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    EXISTS ^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
  | { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    EXISTS1 ^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
  | expr_fn_construction[isPartOfPredicate];
```

The rule for parsing an expression takes an argument that allows us to determine whether the current expression is nested within a predicate. Throughout the grammar we use the constants `PART_OF_PREDICATE` and `NOT_PART_OF_PREDICATE` whenever we expect an expression, choosing the most appropriate constant given the current context. Again, the way in which these constants are used to aid the disambiguation of expressions and predicates is discussed in section 4.6.4. In this first rule, we use ANTLR's semantic predicates to test the value of the received parameter, and will

only match the quantification tokens if we are not currently nested within a predicate.

Rule F.16 - expr_fn_construction:

```
expr_fn_construction[boolean isPartOfPredicate]
  : LAMBDA^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
    | expr_definite_descr[isPartOfPredicate];
```

Rule F.17 - expr_definite_descr:

```
expr_definite_descr[boolean isPartOfPredicate]
  : MU^ schema_text
    AT expression[NOT_PART_OF_PREDICATE]
    | expr_substitution[isPartOfPredicate];
```

This rule allows us to parse a standard definite description expression. Characteristic definite description expression are handled later in the grammar.

Rule F.18 - expr_substitution:

```
expr_substitution[boolean isPartOfPredicate]
  : KW_LET decl_name EQUIVALENCE
    expr_equivalence[isPartOfPredicate]
    (SEMICOLON decl_name EQUIVALENCE
     expr_equivalence[isPartOfPredicate])*
    AT expr_equivalence[isPartOfPredicate]
    | expr_equivalence[isPartOfPredicate];
```

Rule F.19 - expr_equivalence:

```
expr_equivalence[boolean isPartOfPredicate]
  : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    expr_implication[isPartOfPredicate]
    (IFF^ expr_implication[isPartOfPredicate])*
```

```
| { isPartOfPredicate == PART_OF_PREDICATE }?
    expr_implication[isPartOfPredicate];
```

As with the quantifiers above all logical operators over schemas will only be parsed if the expression does not belong within a predicate.

Rule F.20 - expr_implication:

```
expr_implication[boolean isPartOfPredicate]
    : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
        expr_disjunction[isPartOfPredicate]
        (
            IMPLIES^
            expr_disjunction[isPartOfPredicate]
        )*
    | { isPartOfPredicate == PART_OF_PREDICATE }?
        expr_disjunction[isPartOfPredicate];
```

Rule F.21 - expr_disjunction:

```
expr_disjunction[boolean isPartOfPredicate]
    : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
        expr_conjunction[isPartOfPredicate]
        (
            LOR^
            expr_conjunction[isPartOfPredicate]
        )*
    | { isPartOfPredicate == PART_OF_PREDICATE }?
        expr_conjunction[isPartOfPredicate];
```

Rule F.22 - expr_conjunction:

```
expr_conjunction[boolean isPartOfPredicate]
    : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
        expr_negation[isPartOfPredicate]
        (
            LAND^
            expr_negation[isPartOfPredicate]
```

```

    )*
  | { isPartOfPredicate == PART_OF_PREDICATE }?
    expr_negation[isPartOfPredicate];

```

Rule F.23 - expr_negation:

```

expr_negation[boolean isPartOfPredicate]
  : { isPartOfPredicate == NOT_PART_OF_PREDICATE }?
    LNOT^ expr_conditional
  | expr_conditional;

```

Rule F.24 - expr_conditional:

```

expr_conditional
  : KW_IF^ predicate KW_THEN expr_composition
    KW_ELSE expr_composition
  | expr_composition;

```

Rule F.25 - expr_composition:

```

expr_composition
  : expr_piping
    (options {greedy= true;} : SEMI^ expr_piping)*;

```

Rule F.26 - expr_piping:

```

expr_piping
  : expr_hiding
    (options {greedy= true;} : PIPE^ expr_hiding)*;

```

Rule F.27 - expr_hiding:

```

expr_hiding
  : expr_projection
    (options {greedy= true;} :
    HIDE^ LEFTBR decl_name

```

```
(COMMA decl_name)* RIGHTBR)?;
```

Rule F.28 - expr_projection:

```
expr_projection
  : expr_precondition
    (options {greedy= true;} :
     PROJECT^ expr_precondition)*;
```

Rule F.29 - expr_precondition:

```
expr_precondition
  : KW_PRE^ expr_cart_prod
    | expr_cart_prod;
```

Rule F.30 - expr_cart_prod:

```
expr_cart_prod
  { boolean isCartesianProduct= false: }
  : expr_application
    (options {greedy= true;} :
     CROSS expr_application
       { isCartesianProduct= true; }
    )*
  { if (isCartesianProduct)
     { setRoot(#expr_cart_prod,CART_PROD); }
  };
```

Rule F.31 - expr_application:

```
expr_application
  : (application) => application
    | expr_decoration
     (expr_decoration
      { setRoot(#expr_application,APPLICATION); }
    );
```

```
)?;
```

We require a syntactic predicate on the rule handling applications in order to distinguish from generic instantiation expressions. This is discussed in more detail in section 4.6.4.

Rule F.32 - expr_decoration:

```
expr_decoration : expr_renaming (STROKECHAR^)?;
```

Rule F.33 - expr_renaming:

```
expr_renaming
  : expr_binding_selection
    (LEFTSQ^ decl_name FWDSLASH decl_name
      (COMMA decl_name FWDSLASH decl_name)*
      RIGHTSQ!)?;
```

Rule F.34 - expr_binding_selection:

```
expr_binding_selection
  : expr_binding_construction
    (FULLSTOP!
      (ref_name
        { setRoot(#expr_binding_selection,
                  BINDING_SELECTION); }
      | NUMBER
        { setRoot(#expr_binding_selection,
                  TUPLE_SELECTION); }
      ))?;
```

In order to make the remaining phases of syntax checking easier, we use this opportunity to distinguish binding selection and tuple selection expressions.

Rule F.35 - expr_binding_construction:

```

expr_binding_construction
  : THETA^ expr_atom
    (options {greedy= true;} : STROKECHAR^)?
  | expr_atom;

```

Rule F.36 - expr_atom:

```

expr_atom
  : (ref_name)
    => ref_name
    ((LEFTSQ expression[NOT_PART_OF_PREDICATE]
      (COMMA expression[NOT_PART_OF_PREDICATE])* RIGHTSQ)
    => (LEFTSQ expression[NOT_PART_OF_PREDICATE]
      (COMMA expression[NOT_PART_OF_PREDICATE])*
      RIGHTSQ))?
  | NUMBER
  | (LEFTCURLY schema_text AT
    expression[NOT_PART_OF_PREDICATE] RIGHTCURLY)
    => LEFTCURLY! schema_text AT
      expression[NOT_PART_OF_PREDICATE] RIGHTCURLY!
      { setRoot(#expr_atom,SET_COMP); }
  | (LEFTCURLY expression[NOT_PART_OF_PREDICATE])
    => LEFTCURLY!
      (expression[NOT_PART_OF_PREDICATE]
      (COMMA expression[NOT_PART_OF_PREDICATE])*
      )? RIGHTCURLY!
      { setRoot(#expr_atom,SET_EXT); }
  | LEFTCURLY! schema_text RIGHTCURLY!
    { setRoot(#expr_atom,CHAR_SET_COMP); }
  | LBLLOT
    (decl_name EQUIVALENCE
    expression[NOT_PART_OF_PREDICATE]

```

```

        (COMMA expression[NOT_PART_OF_PREDICATE])*
    )? RBL0T
| (LEFTBR MU schema_text RIGHTBR)
=> LEFTBR! MU^ schema_text RIGHTBR!
| (LEFTBR expression[NOT_PART_OF_PREDICATE] COMMA)
=> LEFTBR! expression[NOT_PART_OF_PREDICATE]
    (COMMA expression[NOT_PART_OF_PREDICATE])*
    RIGHTBR!
    { setRoot(#expr_atom,TUPLE_EXT); }
| LEFTBR! expression[NOT_PART_OF_PREDICATE] RIGHTBR!
    { setRoot(#expr_atom,PARENTHESSES); };

```

The last rule for parsing an expression is used to disambiguate the remaining expression types. This disambiguation is required because of the overloading of the various brackets: ‘(’, ‘{’ and ‘[’. We described in section 4.6.4 the particular need to disambiguate between set comprehension (characteristic or otherwise) and set extension. We also noted the tactics used to distinguish generic instantiation from schema application. However, we also require syntactic predicates to distinguish between tuple extension and parenthesized expressions. Furthermore, we use another syntactic predicate to distinguish between a characteristic definite description (which includes parentheses as part of its definition) and a standard definite description that coincidentally has been parenthesized.

Rule F.37 - schema_text:

```
schema_text : (decl_part)? (HALFPIPE predicate)?;
```

Rule F.38 - decl_part:

```
decl_part : declaration ((SEMICOLON | NLCHAR) declaration)*;
```

Rule F.39 - declaration:

```
declaration
```

```

: (decl_name (COMMA decl_name)* COLON
  expression[NOT_PART_OF_PREDICATE])
=> decl_name (COMMA decl_name)*
    COLON expression[NOT_PART_OF_PREDICATE]
| (decl_name EQUIVALENCE
  expression[NOT_PART_OF_PREDICATE])
=> decl_name EQUIVALENCE
    expression[NOT_PART_OF_PREDICATE]
| expression[NOT_PART_OF_PREDICATE];

```

The parsing of schema texts is mainly performed as it is specified in [ISO02]. The only disambiguation required is to ensure that newly declared names are parsed as variable declarations rather than references. Syntactic predicates are used to ensure that the beginning of a declaration can be parsed as a declaration name (or list of declaration names), then it will be.

Rule F.40 - operator_template:

```

operator_template : KW_RELATION template
                  | KW_FUNCTION category_template
                  | KW_GENERIC category_template;

```

Rule F.41 - category_template:

```

category_template : prec prefix_template
                  | prec postfix_template
                  | prec assoc infix_template
                  | nofix_template;

```

Rule F.42 - prec:

```

prec : NUMBER;

```

Rule F.43 - assoc:

```

assoc : KW_LEFTASSOC | KW_RIGHTASSOC;

```

Rule F.44 - template:

```
template : (prefix_template) => prefix_template
         | (postfix_template) => postfix_template
         | (infix_template) => infix_template
         | (nofix_template) => nofix_template;
```

Rule F.45 - prefix_template:

```
prefix_template : LEFTBR NAME ((UNDERSCORE | DBLCOMMA) NAME)*
                UNDERSCORE RIGHTBR;
```

Rule F.46 - postfix_template:

```
postfix_template : LEFTBR UNDERSCORE NAME
                 ((UNDERSCORE | DBLCOMMA) NAME)* RIGHTBR;
```

Rule F.47 - infix_template:

```
infix_template : LEFTBR UNDERSCORE NAME
                ((UNDERSCORE | DBLCOMMA) NAME)* UNDERSCORE
                RIGHTBR;
```

Rule F.48 - nofix_template:

```
nofix_template : LEFTBR NAME ((UNDERSCORE | DBLCOMMA) NAME)*
                RIGHTBR;
```

Operator templates are parsed exactly as described in [ISO02]. However, we need to use syntactic predicates (below) in order to distinguish the affix of operators with more than two operands.

Rule F.49 - decl_name:

```
decl_name : NAME | op_name;
```

Rule F.50 - ref_name:

```
ref_name : NAME | LEFTBR! op_name RIGHTBR!;
```

Rule F.51 - op_name:

```
op_name : (prefix_name) => prefix_name
        | (postfix_name) => postfix_name
        | (infix_name) => infix_name
        | (nofix_name) => nofix_name;
```

Rule F.52 - prefix_name:

```
prefix_name : PRE UNDERSCORE
            | PREP UNDERSCORE
            | L (UNDERSCORE (ES | SS))* UNDERSCORE
              (ERE | SRE) UNDERSCORE
            | LP (UNDERSCORE (ES | SS))* UNDERSCORE
              (EREP | SREP) UNDERSCORE;
```

Rule F.53 - postfix_name:

```
postfix_name : UNDERSCORE POST
            | UNDERSCORE POSTP
            | UNDERSCORE EL (UNDERSCORE (ES | SS))*
              UNDERSCORE (ER | SR)
            | UNDERSCORE ELP (UNDERSCORE (ES | SS))*
              UNDERSCORE (ERP | SRP);
```

Rule F.54 - infix_name:

```
infix_name : UNDERSCORE I UNDERSCORE
           | UNDERSCORE IP UNDERSCORE
           | UNDERSCORE EL (UNDERSCORE (ES | SS))*
             UNDERSCORE (ERE | SRE) UNDERSCORE
           | UNDERSCORE ELP (UNDERSCORE (ES | SS))*
```

UNDERSCORE (EREP | SREP) UNDERSCORE;

Rule F.55 - nofix_name:

```
nofix_name : L (UNDERSCORE (ES | SS))* UNDERSCORE
            (ER | SR)
            | LP (UNDERSCORE (ES | SS))* UNDERSCORE
              (ERP | SRP);
```

Rule F.56 - gen_name:

```
gen_name : (prefix_gen_name) => prefix_gen_name
          | (postfix_gen_name) => postfix_gen_name
          | (infix_gen_name) => infix_gen_name
          | (nofix_gen_name) => nofix_gen_name;
```

Rule F.57 - prefix_gen_name:

```
prefix_gen_name : PRE NAME
                 | L (NAME (ES | SS))*
                   NAME (ERE | SRE) NAME;
```

Rule F.58 - postfix_gen_name:

```
postfix_gen_name : NAME POST
                  | NAME EL (NAME (ES | SS))*
                    NAME (ER | SR);
```

Rule F.59 - infix_gen_name:

```
infix_gen_name : NAME I NAME
                | NAME EL (NAME (ES | SS))*
                  NAME (ERE | SRE) NAME;
```

Rule F.60 - nofix_gen_name:

```
nofix_gen_name : L (NAME (ES | SS))* NAME (ER | SR);
```

The parsing of names proceeds in much the same way as in [ISO02]. The only difference again being the need to use syntactic predicates to determine whether certain name are prefix, postfix, infix or nofix.

Rule F.61 - relation:

```
relation : (prefix_relation) => prefix_relation
          | (postfix_relation) => postfix_relation
          | (infix_relation) => infix_relation
          | (nofix_relation) => nofix_relation;
```

Rule F.62 - prefix_relation:

```
prefix_relation
: PREP expression[PART_OF_PREDICATE]
| LP rel_exp_sep
  ((expression[PART_OF_PREDICATE] EREP)
   => expression[PART_OF_PREDICATE] EREP
  | (expression[PART_OF_PREDICATE] (COMMA | SREP))
   => rel_expression_list SREP)
expression[PART_OF_PREDICATE]
  { resolvePrecedence(); };
```

Rule F.63 - postfix_relation:

```
postfix_relation
: expression[PART_OF_PREDICATE]
  (POSTP
  | (ELP rel_exp_sep
    ((expression[PART_OF_PREDICATE] ERP)
     => expression[PART_OF_PREDICATE] ERP
    | (expression[PART_OF_PREDICATE] (COMMA | SRP))
     => rel_expression_list SRP)));
```

Rule F.64 - infix_relation:

```

infix_relation
    : expression[PART_OF_PREDICATE]
      (((IP | IN | EQUALS) expression[PART_OF_PREDICATE])+
      | (ELP rel_exp_sep
        ((expression[PART_OF_PREDICATE] EREP)
         => expression[PART_OF_PREDICATE] EREP
        | (expression[PART_OF_PREDICATE] (COMMA | SREP))
         => rel_expression_list SREP)
        expression[PART_OF_PREDICATE]))
      { resolvePrecedence(); };

```

Rule F.65 - nofix_relation:

```

nofix_relation
    : LP rel_exp_sep
      ((expression[PART_OF_PREDICATE] ERP)
       => expression[PART_OF_PREDICATE] ERP
      | (expression[PART_OF_PREDICATE] (COMMA | SRP))
       => rel_expression_list SRP);

```

Rule F.66 - rel_exp_sep:

```

rel_exp_sep
    : ( options {greedy= true;} :
      ((expression[PART_OF_PREDICATE] ES)
       => expression[PART_OF_PREDICATE] ES
      | (expression[PART_OF_PREDICATE] (COMMA | SS))
       => rel_expression_list SS)
      )*;

```

Rule F.67 - rel_expression_list:

```

rel_expression_list

```

```

: expression[PART_OF_PREDICATE]
  (COMMA expression[PART_OF_PREDICATE])*;

```

Rule F.68 - application:

```

application : (prefix_application) => prefix_application
             | (postfix_application) => postfix_application
             | (infix_application) => infix_application
             | (nofix_application) => nofix_application;

```

Rule F.69 - prefix_application:

```

prefix_application
: (PRE
  | (L app_exp_sep
    ((expr_application ERE) => expr_application ERE)
    |(expr_application (COMMA | SRE))
    => expr_application SRE))
) expr_application
{ resolvePrecedence(); };

```

Rule F.70 - postfix_application:

```

postfix_application
: expr_decoration
  (POST
  | (EL app_exp_sep
    ((expr_application ER) => expr_application ER)
    |(expr_application (COMMA | SR)))
    => expr_application SR));

```

Rule F.71 - infix_application:

```

infix_application
: expr_decoration

```

```

(I
 | (EL app_exp_sep
   ((expr_application ERE) => expr_application ERE)
   |(expr_application (COMMA | SRE))
   => expr_application SRE))
) expr_application
{ resolvePrecedence(); };

```

Rule F.72 - nofix_application:

```

nofix_application
  : L app_exp_sep
    ((expr_application ER) => expr_application ER
 | (expr_application (COMMA | SR))
 => app_expression_list sr);

```

Rule F.73 - app_exp_sep:

```

app_exp_sep
  : ( options {greedy= true;} :
    ((expr_application ES) => expr_application ES
 | (expr_application (COMMA | SS))
 => app_expression_list SS)*;

```

Rule F.74 - app_expression_list:

```

app_expression_list
  : expr_application (COMMA expr_application)*;

```

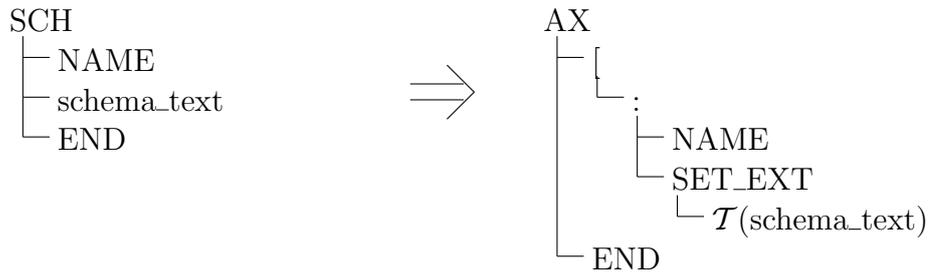
Applications and relations are parsed in a similar manner. Both conform to their definitions in [ISO02]. Again, it is necessary to use syntactic predicates to determine the affix of the application or expression. We also described in section 4.6.4 the need to resolve precedence issues when applications or relations are chained. The rules described in that section define this intricately and we will not present further detail here.

Appendix G

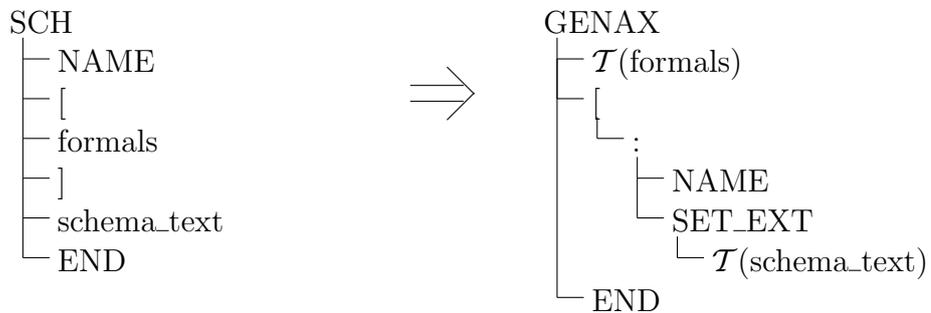
Tree Transformation Rules

G.1 Paragraph Transformation Rules

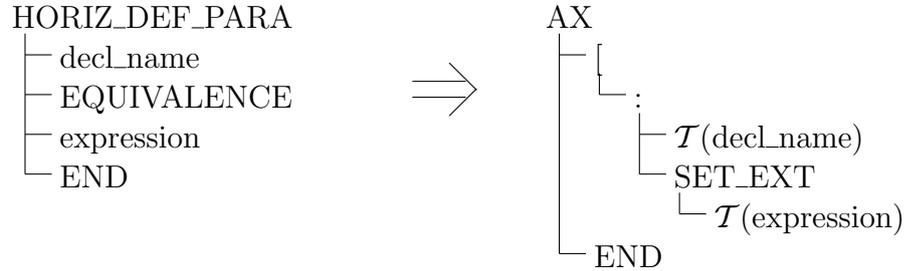
Rule G.1 - Schema Definition Paragraph:



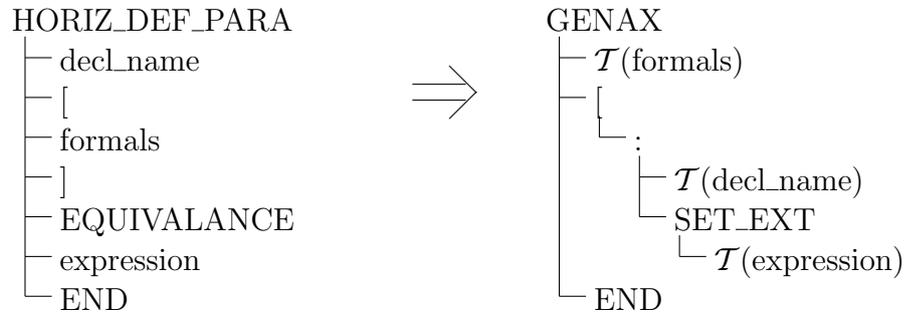
Rule G.2 - Generic Schema Definition Paragraph:



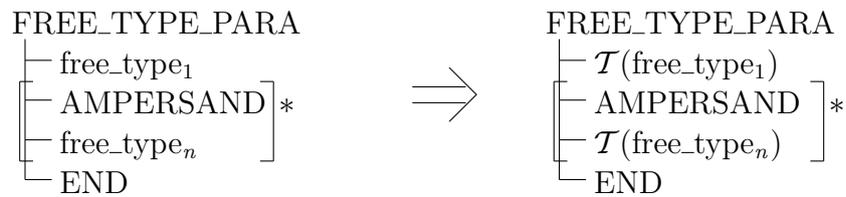
Rule G.3 - Horizontal Definition Paragraph:



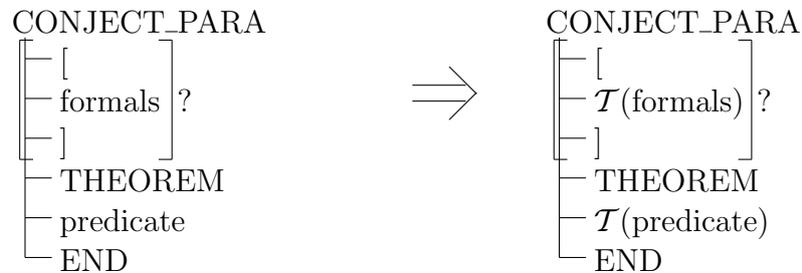
Rule G.4 - Generic Horizontal Definition Paragraph:



Rule G.5 - Free Type Paragraph:



Rule G.6 - Conjecture Paragraph:



Rule G.7 - Axiomatic Description Paragraph:

$$\begin{array}{l}
 \text{AX} \\
 \left| \begin{array}{l} \text{schema_text} \\ \text{END} \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 \text{AX} \\
 \left| \begin{array}{l} \mathcal{T}(\text{schema_text}) \\ \text{END} \end{array} \right.
 \end{array}$$

Rule G.8 - Generic Axiomatic Description Paragraph:

$$\begin{array}{l}
 \text{GENAX} \\
 \left| \begin{array}{l} [\\ \text{formals} \\] \\ \text{schema_text} \\ \text{END} \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 \text{GENAX} \\
 \left| \begin{array}{l} \mathcal{T}(\text{formals}) \\ \mathcal{T}(\text{schema_text}) \\ \text{END} \end{array} \right.
 \end{array}$$

Rule G.9 - Generic Operator Definition Paragraph:

$$\begin{array}{l}
 \text{GEN_OP_DEF_PARA} \\
 \left| \begin{array}{l} \text{gen_name} \\ \text{EQUIVALANCE} \\ \text{expression} \\ \text{END} \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 \text{GENAX} \\
 \left| \begin{array}{l} \mathcal{T}(\text{gen_name}).\text{getFormals}() \\ \left\{ \begin{array}{l} : \\ \mathcal{T}(\text{gen_name}).\text{getName}() \\ \text{SET_EXT} \\ \quad \mathcal{T}(\text{expression}) \end{array} \right. \\ \text{END} \end{array} \right.
 \end{array}$$

G.2 Predicate Transformation Rules

Rule G.10 - Newline Conjunction Predicate:

$$\begin{array}{c} \text{NLCHAR} \\ \left[\begin{array}{l} \text{predicate}_a \\ \text{predicate}_b \end{array} \right. \end{array} \quad \Longrightarrow \quad \begin{array}{c} \text{LAND} \\ \left[\begin{array}{l} \mathcal{T}(\text{predicate}_a) \\ \mathcal{T}(\text{predicate}_b) \end{array} \right. \end{array}$$

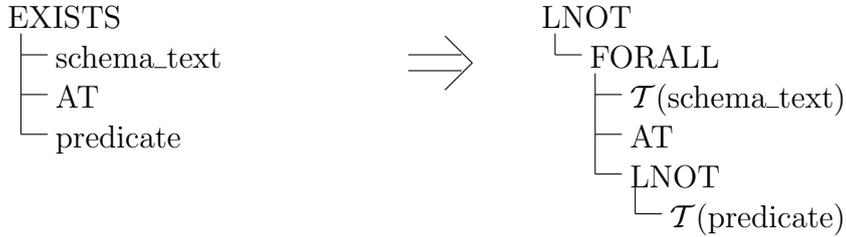
Rule G.11 - Semicolon Conjunction Predicate:

$$\begin{array}{c} \text{SEMICOLON} \\ \left[\begin{array}{l} \text{predicate}_a \\ \text{predicate}_b \end{array} \right. \end{array} \quad \Longrightarrow \quad \begin{array}{c} \text{LAND} \\ \left[\begin{array}{l} \mathcal{T}(\text{predicate}_a) \\ \mathcal{T}(\text{predicate}_b) \end{array} \right. \end{array}$$

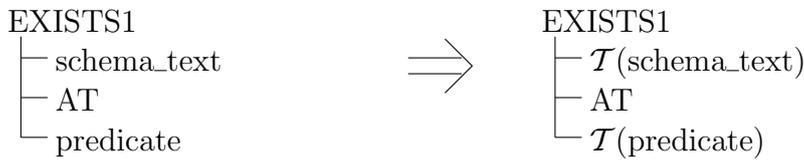
Rule G.12 - Universal Quantification Predicate:

$$\begin{array}{c} \text{FORALL} \\ \left[\begin{array}{l} \text{schema_text} \\ \text{AT} \\ \text{predicate} \end{array} \right. \end{array} \quad \Longrightarrow \quad \begin{array}{c} \text{FORALL} \\ \left[\begin{array}{l} \mathcal{T}(\text{schema_text}) \\ \text{AT} \\ \mathcal{T}(\text{predicate}) \end{array} \right. \end{array}$$

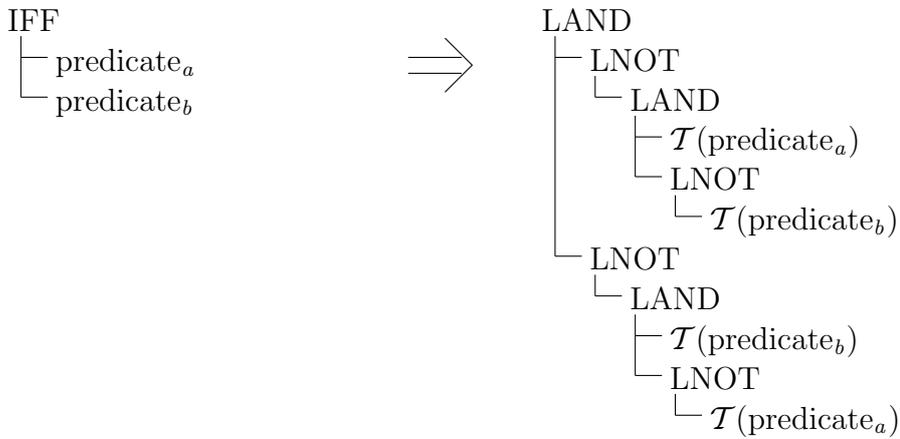
Rule G.13 - Existential Quantification Predicate:



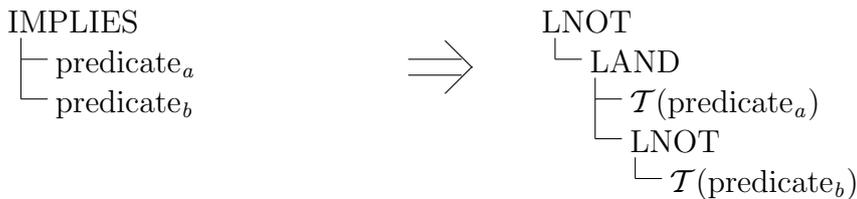
Rule G.14 - Unique Existential Quantification Predicate:



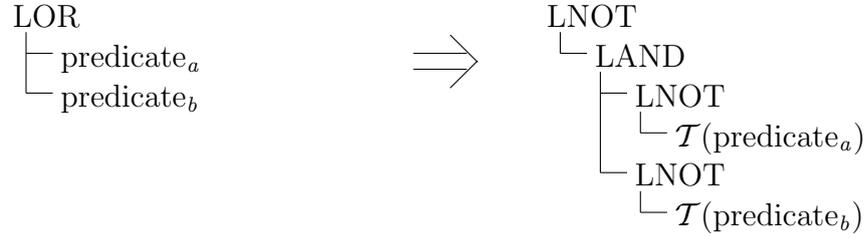
Rule G.15 - Equivalence Predicate:



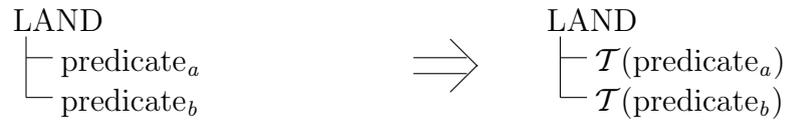
Rule G.16 - Implication Predicate:



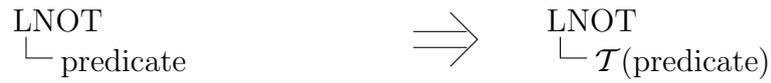
Rule G.17 - Disjunction Predicate:



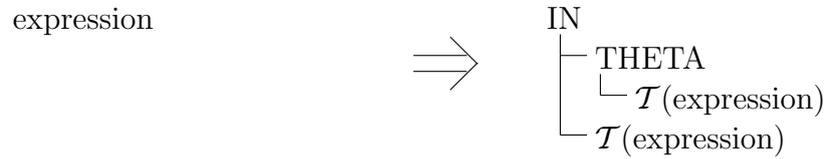
Rule G.18 - Conjunction Predicate:



Rule G.19 - Negation Predicate:



Rule G.20 - Schema Predicate:



Rule G.21 - Truth Predicate:



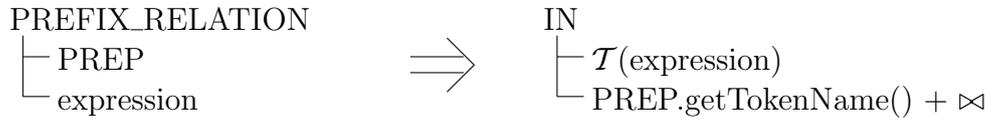
Rule G.22 - False Predicate:



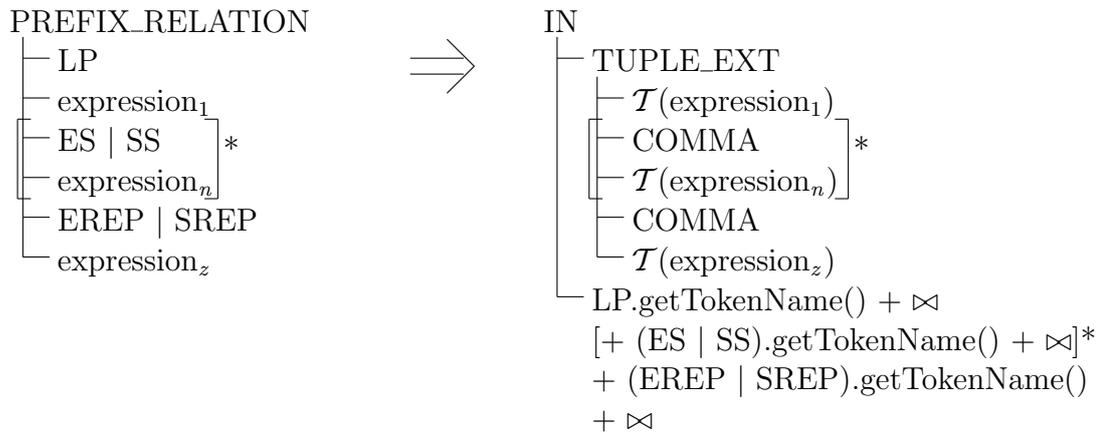
Rule G.23 - Parenthesized Predicate:



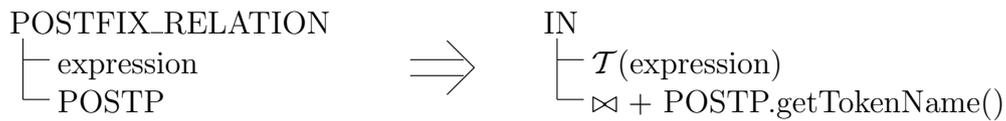
Rule G.24 - Unary Prefix Relation:



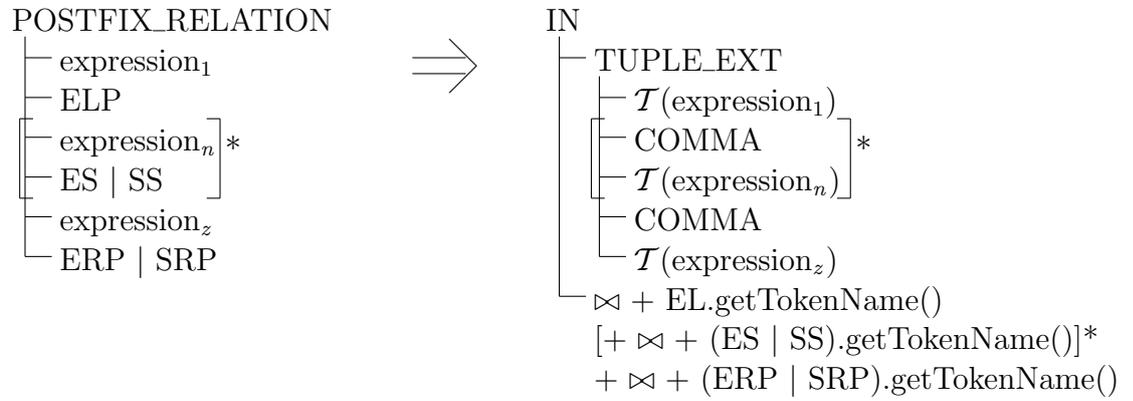
Rule G.25 - Prefix Relation:



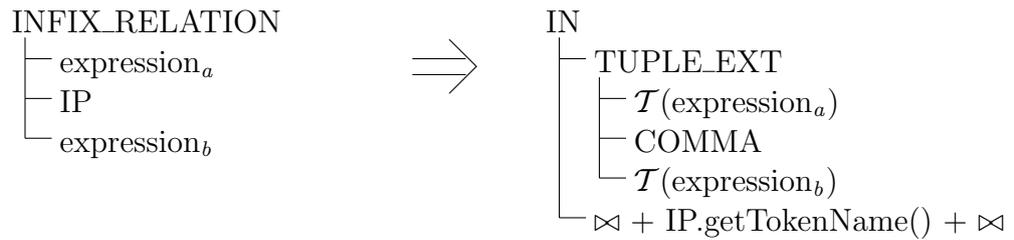
Rule G.26 - Unary Postfix Relation:



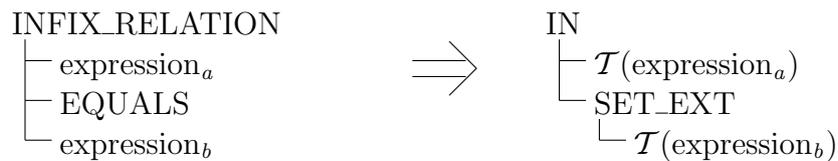
Rule G.27 - Postfix Relation:



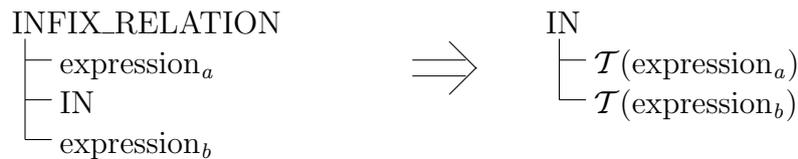
Rule G.28 - Binary Infix Relation:



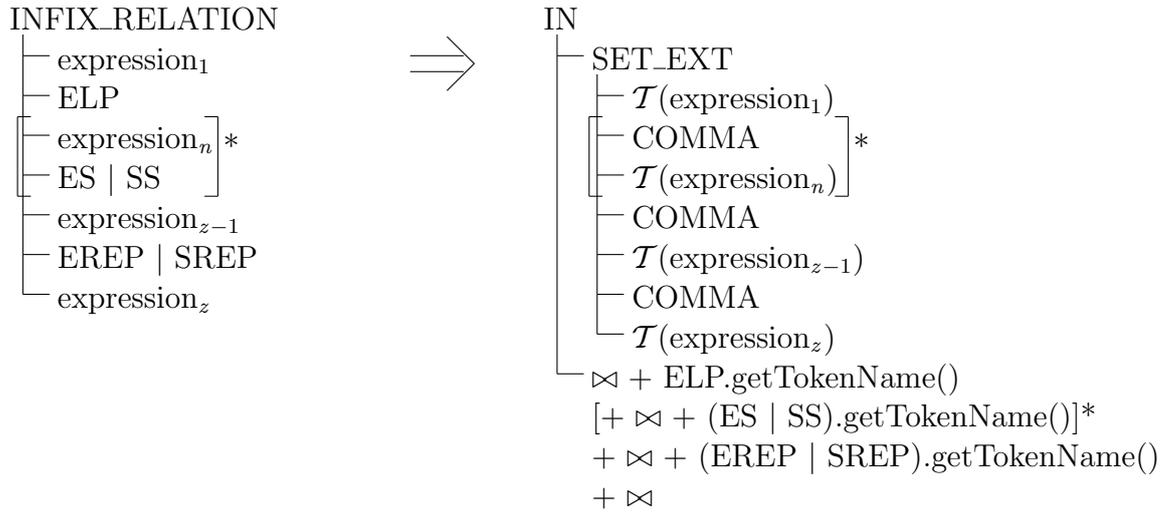
Rule G.29 - Equality Relation:



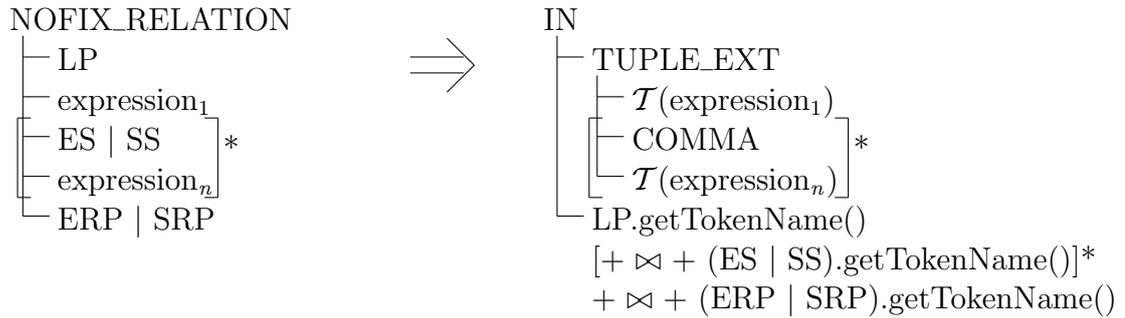
Rule G.30 - Membership Relation:



Rule G.31 - Infix Relation:

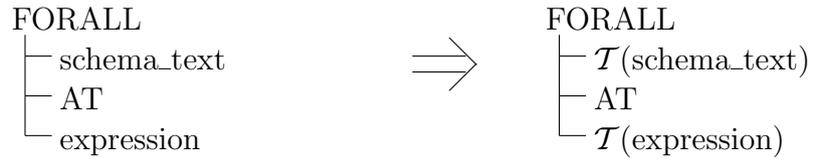


Rule G.32 - Nofix Relation:

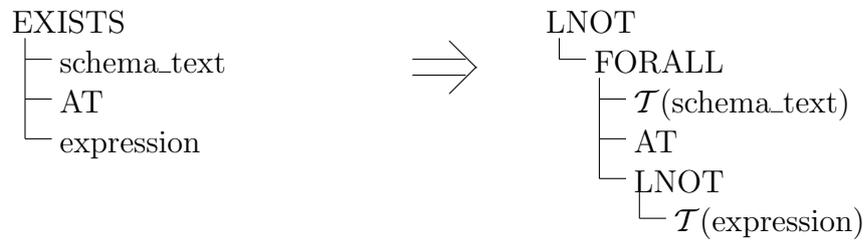


G.3 Expression Transformation Rules

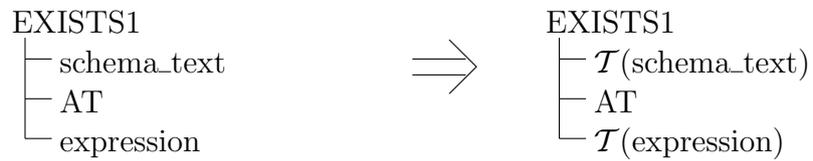
Rule G.33 - Schema Universal Quantification:



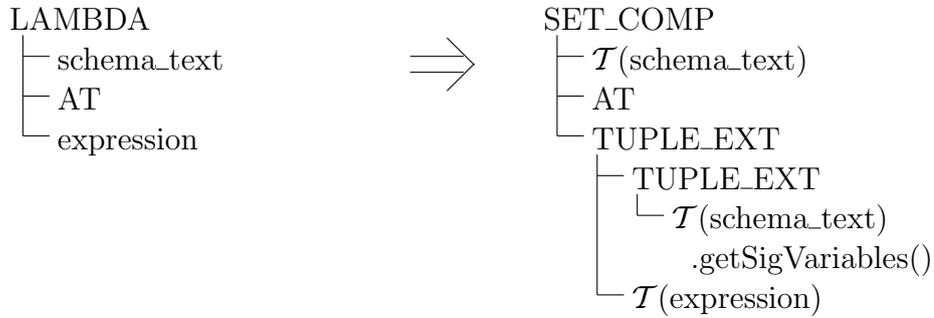
Rule G.34 - Schema Existential Quantification:



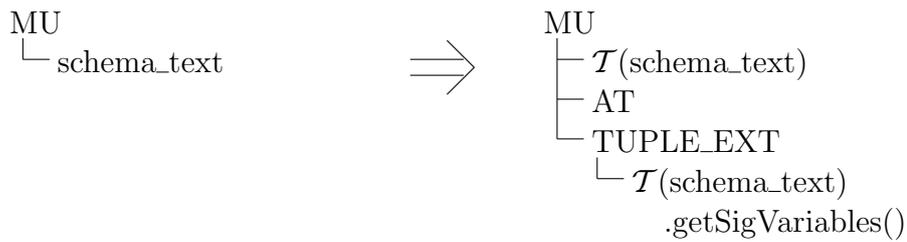
Rule G.35 - Schema Unique Existential Quantification:



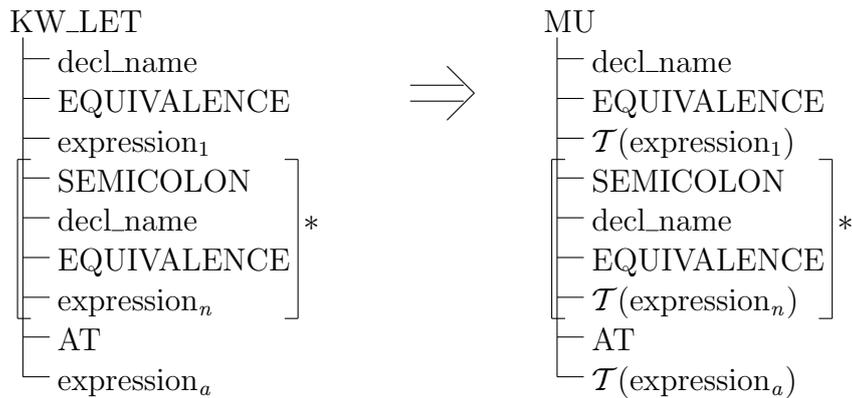
Rule G.36 - Lambda Function Construction:



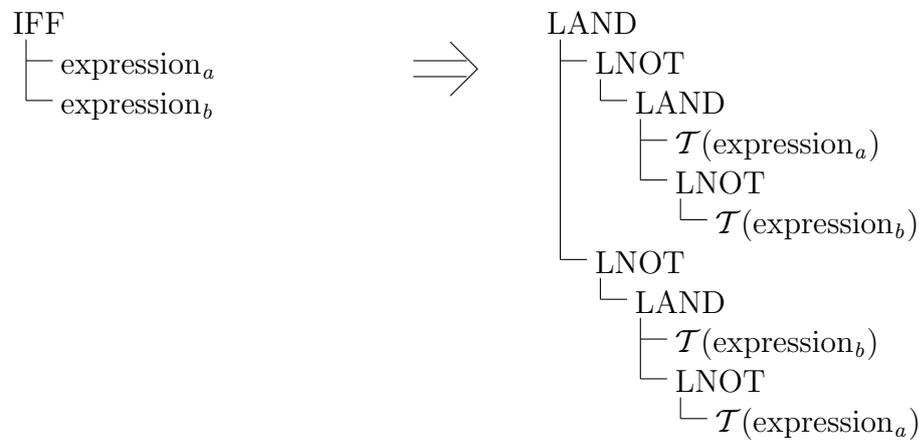
Rule G.37 - Characteristic Definite Description:



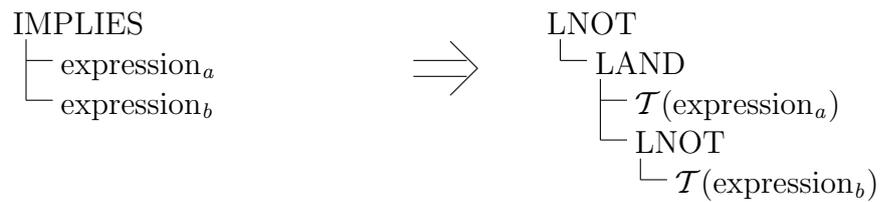
Rule G.38 - Function Construction:



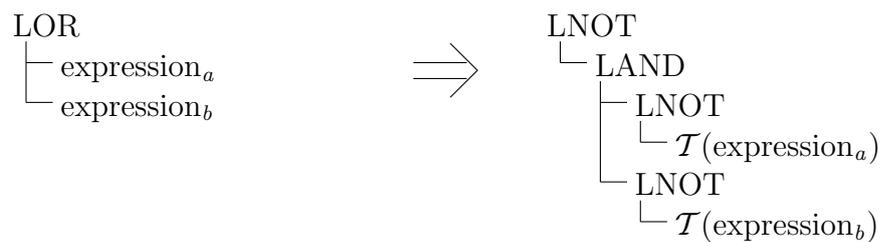
Rule G.39 - Schema Equivalence:



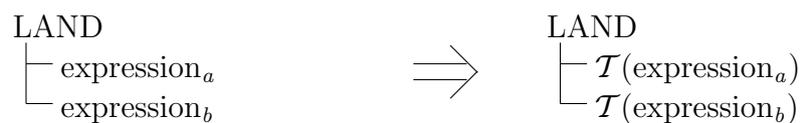
Rule G.40 - Schema Implication:



Rule G.41 - Schema Disjunction:



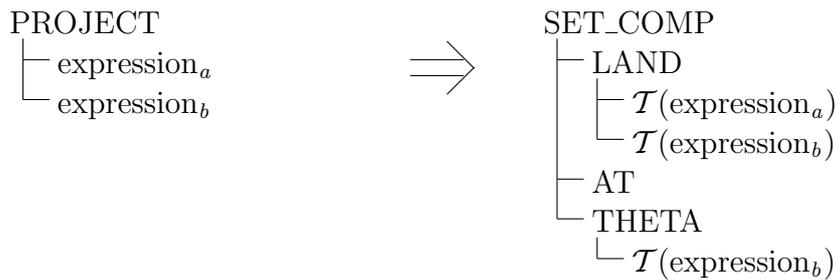
Rule G.42 - Schema Conjunction:



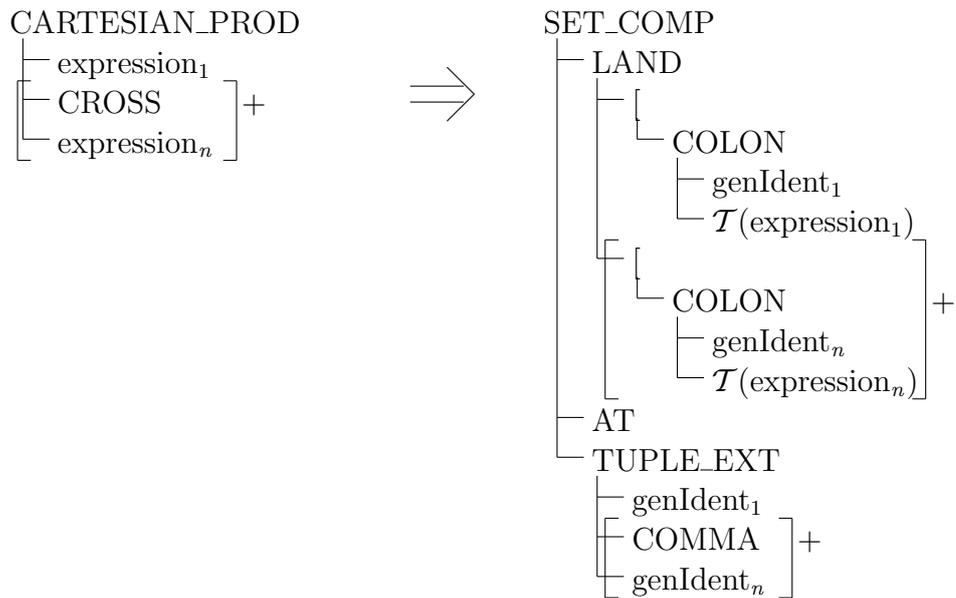
Rule G.43 - Schema Negation:



Rule G.44 - Schema Projection:



Rule G.45 - Cartesian Product:



Rule G.46 - Reference Expression:



Rule G.47 - Parenthesized Expression:

$$\begin{array}{l} \text{LEFTBR} \\ \lrcorner \\ \text{expression} \end{array} \quad \Rightarrow \quad \mathcal{T}(\text{expression})$$

Rule G.48 - Tuple Extension:

$$\begin{array}{c}
 \text{TUPLE_EXT} \\
 \left[\begin{array}{l} \text{expression}_1 \\ \text{COMMA} \\ \text{expression}_n \end{array} \right]^*
 \end{array}
 \Rightarrow
 \begin{array}{c}
 \text{TUPLE_EXT} \\
 \left[\begin{array}{l} \mathcal{T}(\text{expression}_1) \\ \text{COMMA} \\ \mathcal{T}(\text{expression}_n) \end{array} \right]^*
 \end{array}$$

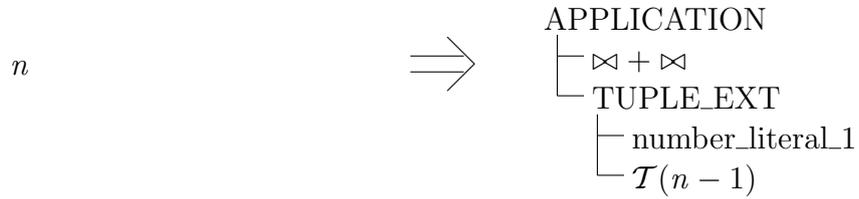
Rule G.49 - Number Literal (0):

$$0 \Rightarrow \text{number_literal}_0$$

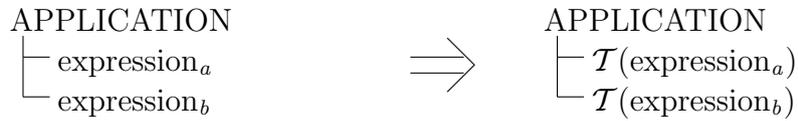
Rule G.50 - Number Literal (1):

$$1 \Rightarrow \text{number_literal}_1$$

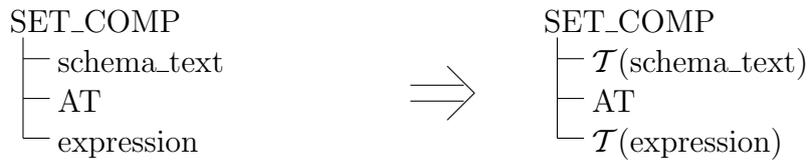
Rule G.51 - Number Literal:



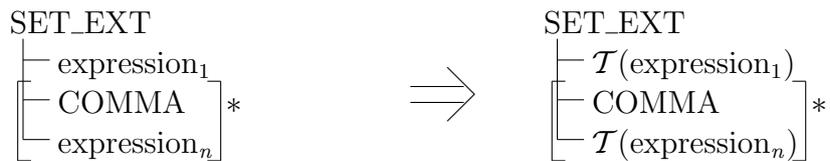
Rule G.52 - Application:



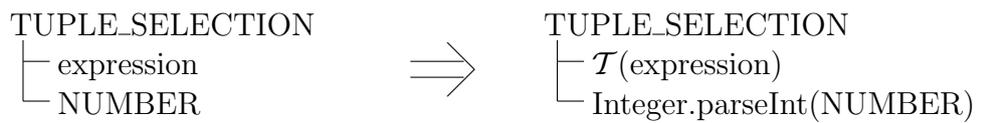
Rule G.53 - Set Comprehension:



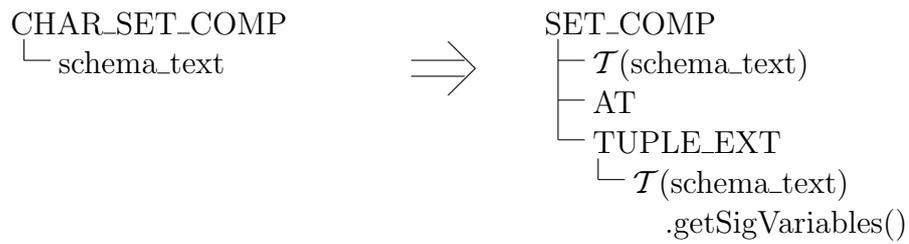
Rule G.54 - Set Extension:



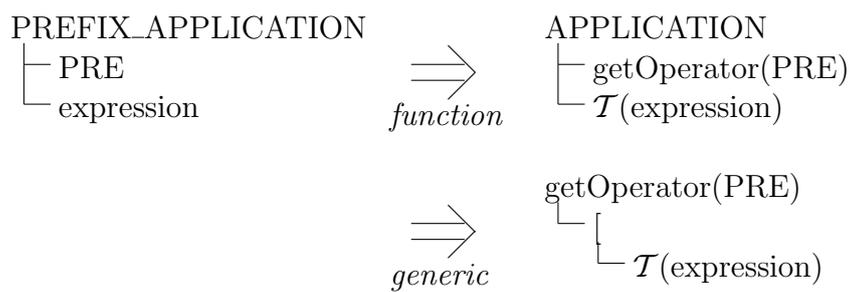
Rule G.55 - Tuple Selection:



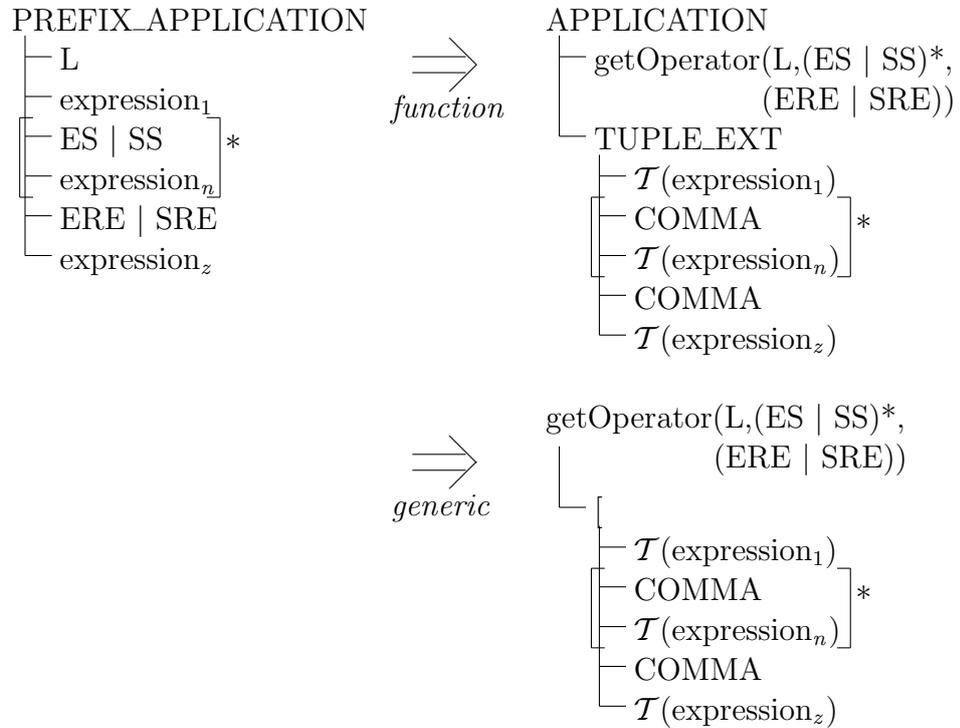
Rule G.56 - Characteristic Set Comprehension:



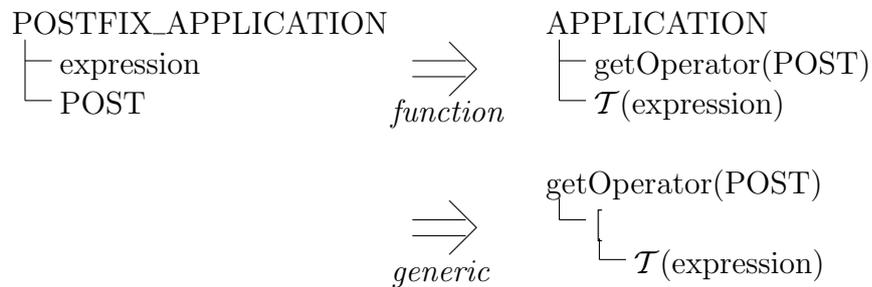
Rule G.57 - Unary, Prefix Function Application:



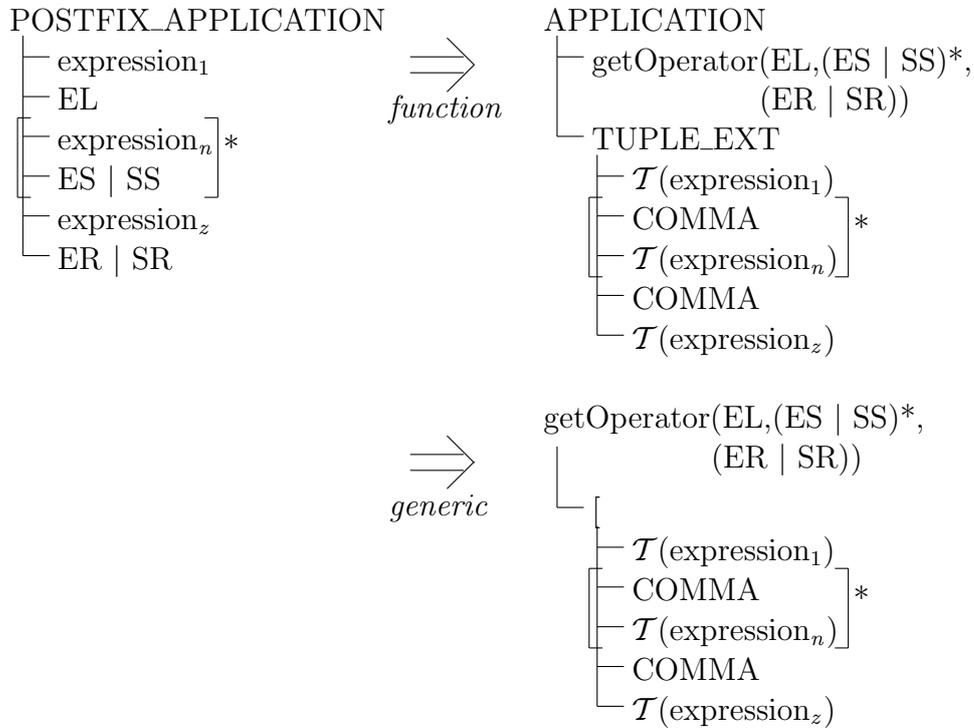
Rule G.58 - Prefix Function Application:



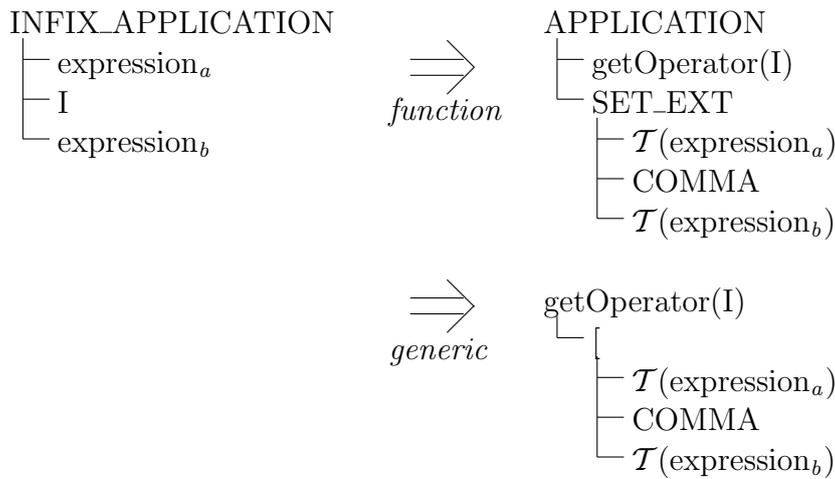
Rule G.59 - Unary, Postfix Function Application:



Rule G.60 - Postfix Application:



Rule G.61 - Binary, Infix Function Application:



G.4 Schema Text Transformation Rules

Rule G.64 - Schema Text (No declaration or predicate):

$$\text{SCHEMA_TEXT} \quad \Rightarrow \quad \begin{array}{l} \text{SET_EXT} \\ \lrcorner \text{LBLOT} \end{array}$$

Rule G.65 - Schema Text (No declaration):

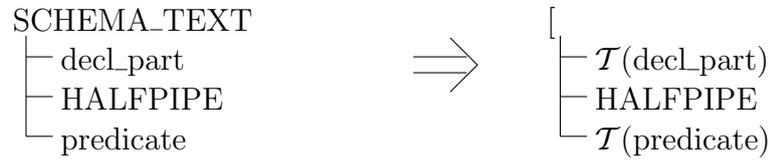
$$\begin{array}{l} \text{SCHEMA_TEXT} \\ \lrcorner \text{HALFPIPE} \\ \lrcorner \text{predicate} \end{array} \quad \Rightarrow \quad \begin{array}{l} \lrcorner \begin{array}{l} \text{SET_EXT} \\ \lrcorner \text{LBLOT} \end{array} \\ \lrcorner \text{HALFPIPE} \\ \lrcorner \mathcal{T}(\text{predicate}) \end{array}$$

Rule G.66 - Schema Text (No predicate):

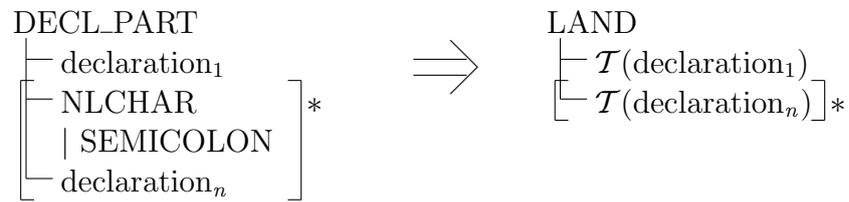
$$\begin{array}{l} \text{SCHEMA_TEXT} \\ \lrcorner \text{decl_part} \\ \lrcorner \text{HALFPIPE} \end{array} \quad \Rightarrow \quad \mathcal{T}(\text{decl_part})$$

comment about typing

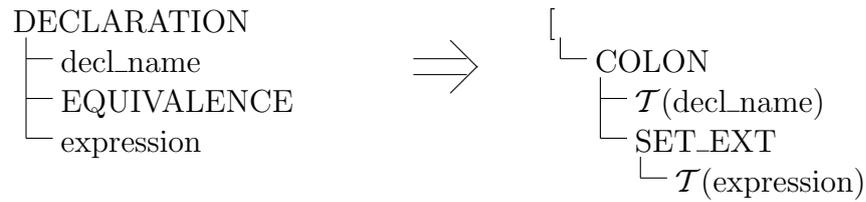
Rule G.67 - Schema Text:



Rule G.68 - Schema Text Declaration:



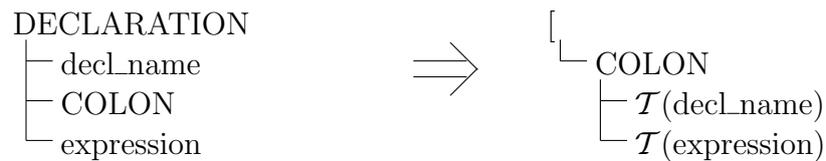
Rule G.69 - Constant Declaration:



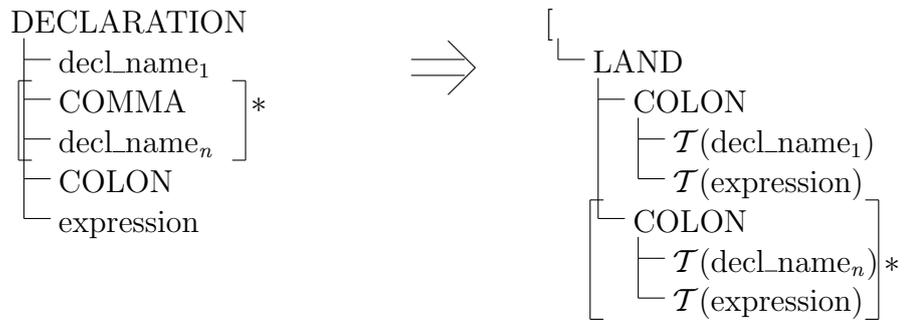
Rule G.70 - Non-Specific Declaration:



Rule G.71 - Variable Declaration:



Rule G.72 - List of Variable Declarations:



Rule G.73 - Variable Declaration Name:



Rule G.74 - Operation Declaration Name:



G.5 Operator Name Transformation Rules

Rule G.75 - Unary, Prefix Operator Name:

$$\begin{array}{l}
 \text{PREFIX_OP_NAME} \\
 \left[\begin{array}{l}
 \text{PRE | PREP} \\
 \text{UNDERSCORE}
 \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 (\text{PRE | PREP}).\text{getTokenName}() \\
 + \infty
 \end{array}$$

Rule G.76 - Prefix Operator Name:

$$\begin{array}{l}
 \text{PREFIX_OP_NAME} \\
 \left[\begin{array}{l}
 \text{L | LP} \\
 \text{UNDERSCORE} \\
 \left[\begin{array}{l}
 \text{ES | SS} \\
 \text{UNDERSCORE}
 \end{array} \right]^* \\
 \text{ERE | SRE} \\
 \quad | \text{EREP | SREP} \\
 \text{UNDERSCORE}
 \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 (\text{L | LP}).\text{getTokenName}() + \infty \\
 [+ (\text{ES | SS}).\text{getTokenName}() + \infty] \\
 + (\text{ERE | SRE | EREP} \\
 \quad | \text{SREP}).\text{getTokenName}() + \infty
 \end{array}$$

Rule G.77 - Unary, Postfix Operator Name:

$$\begin{array}{l}
 \text{POSTFIX_OP_NAME} \\
 \left[\begin{array}{l}
 \text{UNDERSCORE} \\
 \text{POST | POSTP}
 \end{array} \right.
 \end{array}
 \Rightarrow
 \begin{array}{l}
 \infty \\
 + (\text{POST | POSTP}).\text{getTokenName}()
 \end{array}$$

Rule G.78 - Postfix Operator Name:

<pre> POSTFIX_OP_NAME ├── UNDERSCORE ├── EL ELP ├── [UNDERSCORE] * ├── ES SS ├── UNDERSCORE ├── ER SR └── ERP SRP </pre>	\Rightarrow	<pre> ∞ + (EL ELP).getTokenName() [+ ∞ + (ES SS).getTokenName()]* + ∞ + (ER SR ERP SRP).getTokenName() </pre>
--	---------------	---

Rule G.79 - Binary, Infix Operator Name:

<pre> INFIX_OP_NAME ├── UNDERSCORE ├── I IP └── UNDERSCORE </pre>	\Rightarrow	<pre> ∞ + (I IP).getTokenName() + ∞ </pre>
---	---------------	--

Rule G.80 - Infix Operator Name:

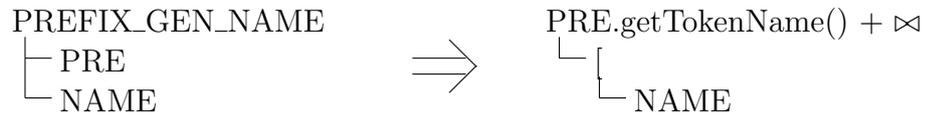
<pre> INFIX_OP_NAME ├── UNDERSCORE ├── EL ELP ├── [UNDERSCORE] * ├── ES SS ├── UNDERSCORE ├── ERE SRE ├── EREP SREP └── UNDERSCORE </pre>	\Rightarrow	<pre> ∞ + (EL ELP).getTokenName() [+ ∞ + (ES SS).getTokenName()]* + ∞ + (ERE SRE EREP SREP).getTokenName() + ∞ </pre>
---	---------------	---

Rule G.81 - Nofix Operator Name:

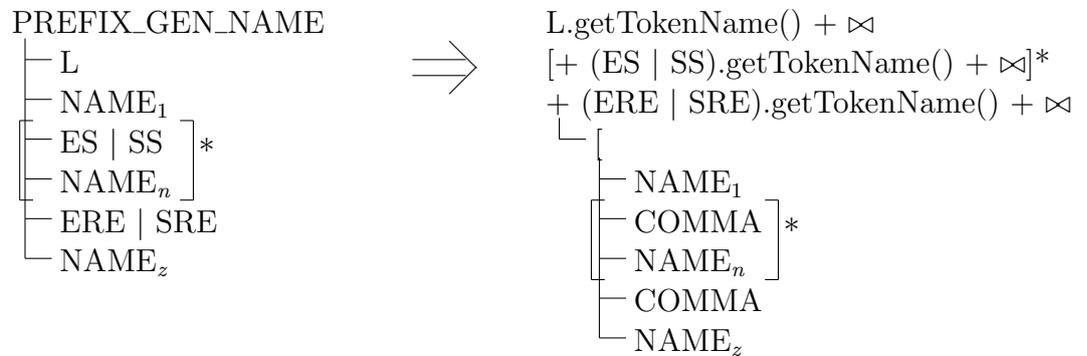
<pre> NOFIX_OP_NAME ├── L LP ├── UNDERSCORE ├── [ES SS ├── UNDERSCORE] * ├── ER SR └── ERP SRP </pre>	\Rightarrow	<pre> (L LP).getTokenName() [+ ∞ + (ES SS).getTokenName()]* + ∞ + (ER SR ERP SRP).getTokenName() </pre>
---	---------------	---

G.6 Generic Name Transformation Rules

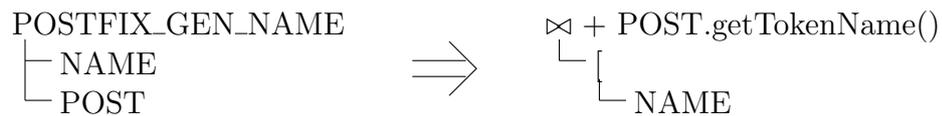
Rule G.82 - Unary, Prefix Generic Name:



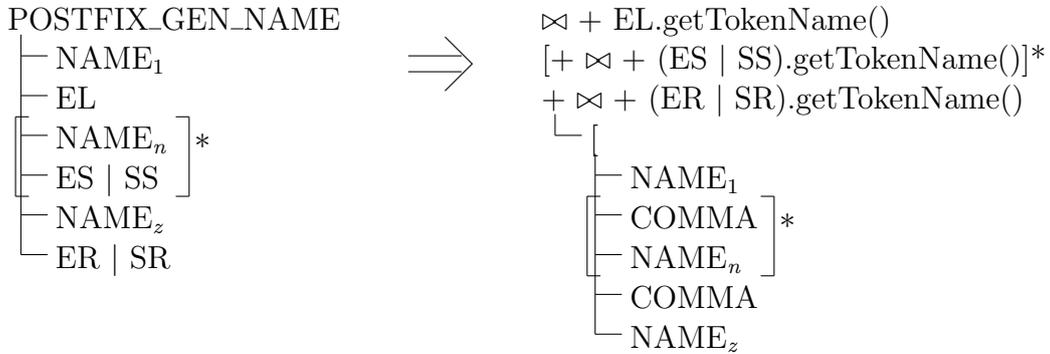
Rule G.83 - Prefix Generic Name:



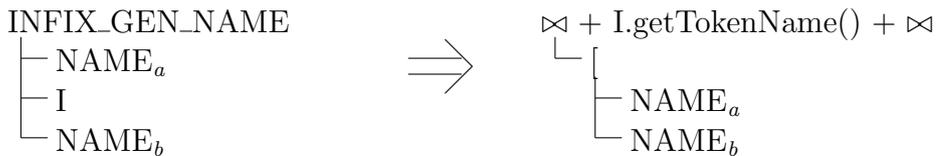
Rule G.84 - Unary, Postfix Generic Name:



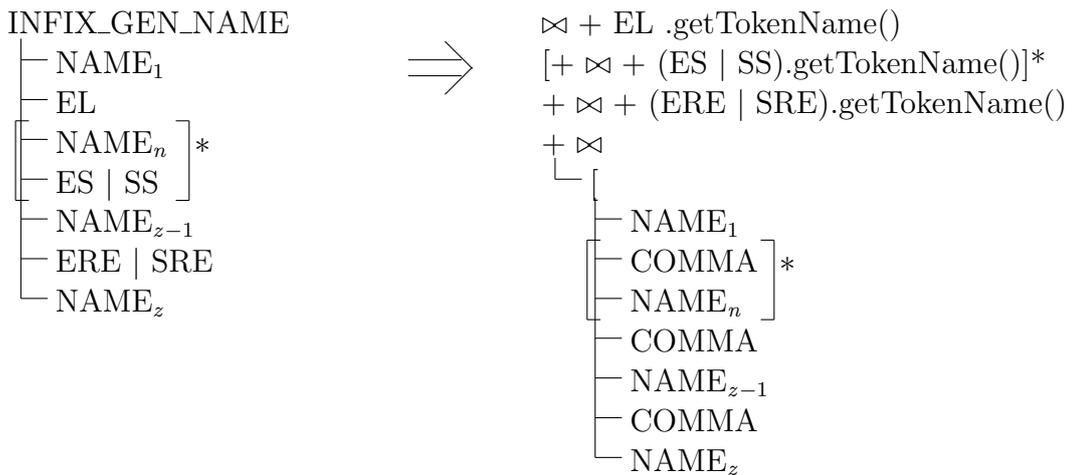
Rule G.85 - Postfix Generic Name:



Rule G.86 - Binary, Infix Generic Name:



Rule G.87 - Infix Generic Name:



Rule G.88 - Nofix Generic Name:

NOFIX_GEN_NAME
 └─ L
 └─ NAME₁
 ┌─ ES | SS ─┘*
 └─ NAME_n ─┘
 └─ ER | SR



L.getTokenName()
 [+ ∞ + (ES | SS).getTokenName()*
 + ∞ + (ER | SR).getTokenName()
 └─ {
 └─ NAME₁
 ┌─ COMMA ─┘*
 └─ NAME_n ─┘

Appendix H

Isabelle/ZF Proof Obligation

The following is the result of applying Isabelle/ZF's simplification tactic to the initialization proof obligation of the retrenchment relationship between *POTS* and *POTSHold*, a \LaTeX representation of which is presented in section 8.4.4.

```
goal (lemma, 1 subgoal):
  1. [| ZFName1198 =
      Pow({ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
          ALL ZFName66.
            ZFName66 : ZFName1187 &
            (ALL ZFName67.
              ZFName67 : ZFName1187 &
              ZFName65 = <ZFName66, ZFName67>)});
      ZFName853 : ZFCarrierSet1188; ZFName857 : ZFCarrierSet1188;
      ZFName861 : ZFCarrierSet1188; ZFName865 : ZFCarrierSet1188;
      ZFName869 : ZFCarrierSet1188; ZFName873 : ZFCarrierSet1188;
      ZFName877 : ZFCarrierSet1188; ZFName881 : ZFCarrierSet1188;
      ZFCarrierSet1188 <= Inf; ZFName1187 <= ZFCarrierSet1188;
      ZFName1248 =
      {ZFName846: Pow(ZFCarrierSet1188 * ZFCarrierSet1188) .
        ALL ZFName847.
          ZFName847 <=
          {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
            ALL ZFName66.
              ZFName66 : ZFName1187 &
              (ALL ZFName67.
```

```

ZFName67 : ZFName1187 &
ZFName65 = <ZFName66, ZFName67>)} &
(ALL ZFName850 ZFName849.
ZFName849 : ZFName847 & ZFName850 : ZFName847 -->
((ALL ZFName854.
  (EX ZFName851 ZFName852.
    ~ (<ZFName851, ZFName852> : ZFName849 <->
      ZFName851 : ZFName854)) |
  ZFName853 ~: ZFName854) |
(ALL ZFName858.
  (EX ZFName855 ZFName856.
    ~ (<ZFName855, ZFName856> : ZFName850 <->
      ZFName855 : ZFName858)) |
  ZFName857 ~: ZFName858) |
ZFName853 ~= ZFName857 |
(EX ZFName862.
  (ALL ZFName859 ZFName860.
    <ZFName859, ZFName860> : ZFName849 <->
    ZFName860 : ZFName862) &
  ZFName861 : ZFName862) &
(EX ZFName866.
  (ALL ZFName863 ZFName864.
    <ZFName863, ZFName864> : ZFName850 <->
    ZFName864 : ZFName866) &
  ZFName865 : ZFName866) &
ZFName861 = ZFName865) &
(ALL ZFName870.
  (EX ZFName867 ZFName868.
    ~ (<ZFName867, ZFName868> : ZFName849 <->
      ZFName868 : ZFName870)) |
  ZFName869 ~: ZFName870) |
(ALL ZFName874.
  (EX ZFName871 ZFName872.
    ~ (<ZFName871, ZFName872> : ZFName850 <->
      ZFName872 : ZFName874)) |
  ZFName873 ~: ZFName874) |
ZFName869 ~= ZFName873 |
(EX ZFName878.
  (ALL ZFName875 ZFName876.

```

```

        <ZName875, ZName876> : ZName849 <->
        ZName875 : ZName878) &
    ZName877 : ZName878) &
    (EX ZName882.
        (ALL ZName879 ZName880.
            <ZName879, ZName880> : ZName850 <->
            ZName879 : ZName882) &
            ZName881 : ZName882) &
        ZName877 = ZName881)) &
    ZName846 = ZName847};
ZName1252 =
Pow({ZName65: (Pow(Pow(ZFCarrierSet1188)) *
    Pow(Pow(ZFCarrierSet1188))) *
    Pow(Pow(ZFCarrierSet1188)) .
    ALL ZName66.
        ZName66 :
        Pow(Pow(ZFCarrierSet1188)) * Pow(Pow(ZFCarrierSet1188)) &
        (ALL ZName144.
            ZName144 <= Pow(ZFCarrierSet1188) &
            (ALL ZName145.
                ZName145 <= Pow(ZFCarrierSet1188) &
                ZName66 = <ZName144, ZName145>)) &
            ZName65 = ZName66});
ZName1253 =
{ZName73: Pow((Pow(Pow(ZFCarrierSet1188)) *
    Pow(Pow(ZFCarrierSet1188))) *
    Pow(Pow(ZFCarrierSet1188))) .
    ALL ZName74. ZName73 = ZName74};
ZName152 <= Pow(ZFCarrierSet1188);
ZName1249 <=
(Pow(Pow(ZFCarrierSet1188)) * Pow(Pow(ZFCarrierSet1188))) *
Pow(Pow(ZFCarrierSet1188)) &
(ALL ZName74.
    ZName74 <=
    {ZName65: (Pow(Pow(ZFCarrierSet1188)) *
        Pow(Pow(ZFCarrierSet1188))) *
        Pow(Pow(ZFCarrierSet1188)) .
    ALL ZName66.
        ZName66 :

```

```

Pow(Pow(ZFCarrierSet1188)) * Pow(Pow(ZFCarrierSet1188)) &
(ALL ZFName144.
  ZFName144 <= Pow(ZFCarrierSet1188) &
  (ALL ZFName145.
    ZFName145 <= Pow(ZFCarrierSet1188) &
    ZFName66 = <ZFName144, ZFName145>)) &
ZFName65 = ZFName66} &
(ALL ZFName76.
  ZFName76 :
  Pow(Pow(ZFCarrierSet1188)) * Pow(Pow(ZFCarrierSet1188)) &
  (ALL ZFName144.
    ZFName144 <= Pow(ZFCarrierSet1188) &
    (ALL ZFName145.
      ZFName145 <= Pow(ZFCarrierSet1188) &
      ZFName76 = <ZFName144, ZFName145>)) -->
  (EX! ZFName77. ZFName76 : ZFName74)) &
ZFName1249 = ZFName74);
ALL ZFName150 ZFName149.
ZFName149 <= Pow(ZFCarrierSet1188) &
ZFName150 <= Pow(ZFCarrierSet1188) -->
(EX ZFNameFunction153.
  (ALL ZFName151.
    <<ZFName149, ZFName150>, ZFName151> : ZFName1249 <->
    ZFName151 : ZFNameFunction153) &
    ZFName152 : ZFNameFunction153) &
ZFName152 =
{ZFName154: Pow(ZFCarrierSet1188) .
  ALL ZFName155.
    ZFName155 <= ZFCarrierSet1188 &
    ZFName155 : ZFName149 &
    ZFName155 : ZFName150 & ZFName154 = ZFName155};
ZFName1209 =
Pow({ZFName65: Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
  Pow(ZFCarrierSet1188) .
  ALL ZFName66.
    ZFName66 <=
  {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
  ALL ZFName66.
    ZFName66 : ZFCarrierSet1188 &

```

```

        (ALL ZFName67.
          ZFName67 : ZFCarrierSet1188 &
          ZFName65 = <ZFName66, ZFName67>)} &
      (ALL ZFName67.
        ZFName67 <= ZFCarrierSet1188 &
        ZFName65 = <ZFName66, ZFName67>)});
ZFName1208 =
Pow({ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
  ALL ZFName66.
    ZFName66 : ZFCarrierSet1188 &
    (ALL ZFName67.
      ZFName67 : ZFCarrierSet1188 &
      ZFName65 = <ZFName66, ZFName67>)});
ZFName1210 =
{ZFName73: Pow(Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
  Pow(ZFCarrierSet1188)) .
  ALL ZFName74. ZFName73 = ZFName74};
ZFName324 <= ZFCarrierSet1188; ZFName330 : ZFCarrierSet1188;
ZFName1254 <=
Pow(ZFCarrierSet1188 * ZFCarrierSet1188) * Pow(ZFCarrierSet1188) &
(ALL ZFName74.
  ZFName74 <=
  {ZFName65: Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
    Pow(ZFCarrierSet1188)) .
  ALL ZFName66.
    ZFName66 <=
    {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
      ALL ZFName66.
        ZFName66 : ZFCarrierSet1188 &
        (ALL ZFName67.
          ZFName67 : ZFCarrierSet1188 &
          ZFName65 = <ZFName66, ZFName67>)} &
      (ALL ZFName67.
        ZFName67 <= ZFCarrierSet1188 &
        ZFName65 = <ZFName66, ZFName67>)} &
    (ALL ZFName76.
      ZFName76 <=
      {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
        ALL ZFName66.

```

```

ZFName66 : ZFCarrierSet1188 &
  (ALL ZFName67.
    ZFName67 : ZFCarrierSet1188 &
    ZFName65 = <ZFName66, ZFName67>)} -->
(EX! ZFName77.
  ZFName77 <= ZFCarrierSet1188 &
  <ZFName76, ZFName77> : ZFName74)) &
ZFName1254 = ZFName74);
ALL ZFName321.
ZFName321 : ZFName1211 -->
(EX ZFNameFunction325.
  (ALL ZFName323.
    <ZFName321, ZFName323> : ZFName1254 <->
    ZFName323 : ZFNameFunction325) &
  ZFName324 : ZFNameFunction325) &
ZFName324 =
{ZFName326: ZFCarrierSet1188 .
  ALL ZFName327.
  ZFName327 : ZFName321 &
  (EX ZFName331.
    (ALL ZFName328 ZFName329.
      <ZFName328, ZFName329> : ZFName327 <->
      ZFName328 : ZFName331) &
      ZFName330 : ZFName331) &
    ZFName326 = ZFName330};
ZFName1256 <= ZFCarrierSet1188;
ZFName1223 =
Pow({ZFName65: Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
  Pow(ZFCarrierSet1188) .
  ALL ZFName66.
  ZFName66 <=
  {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
  ALL ZFName66.
  ZFName66 : ZFCarrierSet1188 &
  (ALL ZFName67.
    ZFName67 : ZFCarrierSet1188 &
    ZFName65 = <ZFName66, ZFName67>)} &
  (ALL ZFName67.
    ZFName67 <= ZFCarrierSet1188 &

```

```

        ZFName65 = <ZFName66, ZFName67>});
ZFName1222 =
Pow({ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
    ALL ZFName66.
        ZFName66 : ZFCarrierSet1188 &
        (ALL ZFName67.
            ZFName67 : ZFCarrierSet1188 &
            ZFName65 = <ZFName66, ZFName67>});
ZFName1224 =
{ZFName73: Pow(Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
    Pow(ZFCarrierSet1188)) .
    All(op =(ZFName73))};
ZFName344 <= ZFCarrierSet1188; ZFName350 : ZFCarrierSet1188;
ZFName1258 <=
Pow(ZFCarrierSet1188 * ZFCarrierSet1188) * Pow(ZFCarrierSet1188) &
All(op =(ZFName1258));
ALL ZFName341.
    ZFName341 : ZFName1225 -->
    (EX ZFNameFunction345.
        (ALL ZFName343.
            <ZFName341, ZFName343> : ZFName1258 <->
            ZFName343 : ZFNameFunction345) &
            ZFName344 : ZFNameFunction345) &
ZFName344 =
{ZFName346: ZFCarrierSet1188 .
    ALL ZFName347.
        ZFName347 : ZFName341 &
        (EX ZFName351.
            (ALL ZFName348 ZFName349.
                <ZFName348, ZFName349> : ZFName347 <->
                ZFName349 : ZFName351) &
                ZFName350 : ZFName351) &
            ZFName346 = ZFName350};
ZFName1260 <= ZFCarrierSet1188; ZFName1263 <= ZFCarrierSet1188;
ZFName1265 = 0; ZFName1267 = 0; ZFName1348 = 0 []
==> ALL ZFName1424.
    ZFName1424 :
    Pow(ZFCarrierSet1188 * ZFCarrierSet1188) *
    Pow(ZFCarrierSet1188) &

```

```

(%<ZFName1247,ZFName1346>.
  ZFName1247 <= ZFCarrierSet1188 * ZFCarrierSet1188 &
  (ALL ZFName847.
    ZFName847 <=
    {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
    ALL ZFName66.
      ZFName66 : ZFName1187 &
      (ALL ZFName67.
        ZFName67 : ZFName1187 &
        ZFName65 = <ZFName66, ZFName67>)} &
    (ALL ZFName850 ZFName849.
      ZFName849 : ZFName847 & ZFName850 : ZFName847 -->
      ((ALL ZFName854.
        (EX ZFName851 ZFName852.
          ~ (<ZFName851, ZFName852> : ZFName849 <->
            ZFName851 : ZFName854)) |
          ZFName853 ~: ZFName854) |
        (ALL ZFName858.
          (EX ZFName855 ZFName856.
            ~ (<ZFName855, ZFName856> : ZFName850 <->
              ZFName855 : ZFName858)) |
            ZFName857 ~: ZFName858) |
          ZFName853 ~= ZFName857 |
          (EX ZFName862.
            (ALL ZFName859 ZFName860.
              <ZFName859, ZFName860> : ZFName849 <->
              ZFName860 : ZFName862) &
              ZFName861 : ZFName862) &
            (EX ZFName866.
              (ALL ZFName863 ZFName864.
                <ZFName863, ZFName864> : ZFName850 <->
                ZFName864 : ZFName866) &
                ZFName865 : ZFName866) &
              ZFName861 = ZFName865) &
            (ALL ZFName870.
              (EX ZFName867 ZFName868.
                ~ (<ZFName867, ZFName868> : ZFName849 <->
                  ZFName868 : ZFName870)) |
                ZFName869 ~: ZFName870) |

```

```

(ALL ZFName874.
  (EX ZFName871 ZFName872.
    ~ (<ZFName871, ZFName872> : ZFName850 <->
      ZFName872 : ZFName874)) |
    ZFName873 ~: ZFName874) |
ZFName869 ~= ZFName873 |
(EX ZFName878.
  (ALL ZFName875 ZFName876.
    <ZFName875, ZFName876> : ZFName849 <->
    ZFName875 : ZFName878) &
    ZFName877 : ZFName878) &
(EX ZFName882.
  (ALL ZFName879 ZFName880.
    <ZFName879, ZFName880> : ZFName850 <->
    ZFName879 : ZFName882) &
    ZFName881 : ZFName882) &
    ZFName877 = ZFName881)) &
ZFName1247 = ZFName847) &
(EX ZFNameFunction1264.
  (ALL ZFName1262.
    <<ZFName1256, ZFName1260>, ZFName1262> :
    ZFName1249 <->
    ZFName1262 : ZFNameFunction1264) &
    ZFName1263 : ZFNameFunction1264) &
ZFName1263 = 0 &
ZFName1247 = 0 & ZFName1346 <= ZFName1187 & ZFName1346 = 0)
(ZFName1424) -->
(ALL ZFName847.
  ZFName847 <=
  {ZFName65: ZFCarrierSet1188 * ZFCarrierSet1188 .
  ALL ZFName66.
    ZFName66 : ZFName1187 &
  (ALL ZFName67.
    ZFName67 : ZFName1187 &
    ZFName65 = <ZFName66, ZFName67>)} &
(ALL ZFName850 ZFName849.
  ZFName849 : ZFName847 & ZFName850 : ZFName847 -->
  ((ALL ZFName854.
    (EX ZFName851 ZFName852.

```

$$\begin{aligned}
& \sim (\langle \text{Z FName851}, \text{Z FName852} \rangle : \text{Z FName849} \leftrightarrow \\
& \quad \text{Z FName851} : \text{Z FName854}) \mid \\
& \text{Z FName853} \sim : \text{Z FName854} \mid \\
& (\text{ALL Z FName858.} \\
& \quad (\text{EX Z FName855 Z FName856.} \\
& \quad \quad \sim (\langle \text{Z FName855}, \text{Z FName856} \rangle : \text{Z FName850} \leftrightarrow \\
& \quad \quad \quad \text{Z FName855} : \text{Z FName858})) \mid \\
& \quad \text{Z FName857} \sim : \text{Z FName858} \mid \\
& \text{Z FName853} \sim = \text{Z FName857} \mid \\
& (\text{EX Z FName862.} \\
& \quad (\text{ALL Z FName859 Z FName860.} \\
& \quad \quad \langle \text{Z FName859}, \text{Z FName860} \rangle : \text{Z FName849} \leftrightarrow \\
& \quad \quad \quad \text{Z FName860} : \text{Z FName862}) \& \\
& \quad \text{Z FName861} : \text{Z FName862}) \& \\
& (\text{EX Z FName866.} \\
& \quad (\text{ALL Z FName863 Z FName864.} \\
& \quad \quad \langle \text{Z FName863}, \text{Z FName864} \rangle : \text{Z FName850} \leftrightarrow \\
& \quad \quad \quad \text{Z FName864} : \text{Z FName866}) \& \\
& \quad \text{Z FName865} : \text{Z FName866}) \& \\
& \text{Z FName861} = \text{Z FName865}) \& \\
& ((\text{ALL Z FName870.} \\
& \quad (\text{EX Z FName867 Z FName868.} \\
& \quad \quad \sim (\langle \text{Z FName867}, \text{Z FName868} \rangle : \text{Z FName849} \leftrightarrow \\
& \quad \quad \quad \text{Z FName868} : \text{Z FName870})) \mid \\
& \quad \text{Z FName869} \sim : \text{Z FName870} \mid \\
& (\text{ALL Z FName874.} \\
& \quad (\text{EX Z FName871 Z FName872.} \\
& \quad \quad \sim (\langle \text{Z FName871}, \text{Z FName872} \rangle : \text{Z FName850} \leftrightarrow \\
& \quad \quad \quad \text{Z FName872} : \text{Z FName874})) \mid \\
& \quad \text{Z FName873} \sim : \text{Z FName874} \mid \\
& \text{Z FName869} \sim = \text{Z FName873} \mid \\
& (\text{EX Z FName878.} \\
& \quad (\text{ALL Z FName875 Z FName876.} \\
& \quad \quad \langle \text{Z FName875}, \text{Z FName876} \rangle : \text{Z FName849} \leftrightarrow \\
& \quad \quad \quad \text{Z FName875} : \text{Z FName878}) \& \\
& \quad \text{Z FName877} : \text{Z FName878}) \& \\
& (\text{EX Z FName882.} \\
& \quad (\text{ALL Z FName879 Z FName880.} \\
& \quad \quad \langle \text{Z FName879}, \text{Z FName880} \rangle : \text{Z FName850} \leftrightarrow
\end{aligned}$$

```

        ZFName879 : ZFName882) &
        ZFName881 : ZFName882) &
        ZFName877 = ZFName881)) &
    0 = ZFName847) &
(EX ZFNameFunction1264.
    (ALL ZFName1262.
        <<ZFName1256, ZFName1260>, ZFName1262> : ZFName1249 <->
        ZFName1262 : ZFNameFunction1264) &
        ZFName1263 : ZFNameFunction1264) &
ZFName1263 = 0 &
(%<ZFName1247,ZFName1346>.
    ZFName1247 <= ZFCarrierSet1188 * ZFCarrierSet1188 &
    (ALL ZFName847.
        (ALL ZFName850 ZFName849.
            ZFName849 : ZFName847 & ZFName850 : ZFName847 -->
            ((ALL ZFName854.
                (EX ZFName851 ZFName852.
                    ~ (<ZFName851, ZFName852> : ZFName849 <->
                    ZFName851 : ZFName854)) |
                ZFName853 ~: ZFName854) |
            (ALL ZFName858.
                (EX ZFName855 ZFName856.
                    ~ (<ZFName855, ZFName856> : ZFName850 <->
                    ZFName855 : ZFName858)) |
                ZFName857 ~: ZFName858) |
            ZFName853 ~= ZFName857 |
            (EX ZFName862.
                (ALL ZFName859 ZFName860.
                    <ZFName859, ZFName860> : ZFName849 <->
                    ZFName860 : ZFName862) &
                ZFName861 : ZFName862) &
            (EX ZFName866.
                (ALL ZFName863 ZFName864.
                    <ZFName863, ZFName864> : ZFName850 <->
                    ZFName864 : ZFName866) &
                ZFName865 : ZFName866) &
            ZFName861 = ZFName865) &
            (ALL ZFName870.
                (EX ZFName867 ZFName868.

```

```

~ (<ZFName867, ZFName868> : ZFName849 <->
  ZFName868 : ZFName870)) |
ZFName869 ~: ZFName870) |
(ALL ZFName874.
  (EX ZFName871 ZFName872.
    ~ (<ZFName871, ZFName872> : ZFName850 <->
      ZFName872 : ZFName874)) |
    ZFName873 ~: ZFName874) |
  ZFName869 ~= ZFName873 |
  (EX ZFName878.
    (ALL ZFName875 ZFName876.
      <ZFName875, ZFName876> : ZFName849 <->
      ZFName875 : ZFName878) &
      ZFName877 : ZFName878) &
    (EX ZFName882.
      (ALL ZFName879 ZFName880.
        <ZFName879, ZFName880> : ZFName850 <->
        ZFName879 : ZFName882) &
        ZFName881 : ZFName882) &
      ZFName877 = ZFName881)) &
    ZFName1247 = ZFName847) &
  (EX ZFNameFunction1264.
    (ALL ZFName1262.
      <<ZFName1256, ZFName1260>, ZFName1262> :
      ZFName1249 <->
      ZFName1262 : ZFNameFunction1264) &
      0 : ZFNameFunction1264) &
    ZFName1346 <= ZFName1187)
  (ZFName1424)

```

Bibliography

- [Abr83] J.R. Abrial. The mathematical construction of a program. *Science of Computer Programming 4*, pages 45–86, 1983.
- [Abr91] J.R. Abrial. *The B-Tool Reference Manual, Version 1.1*. Edinburgh Portable Compiler, 1991.
- [Abr96] J.R. Abrial. *The B Book*. Cambridge University Press, 1996.
- [AC03] J. R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
- [AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM*. McGraw-Hill, 1991.
- [AL97] B. K. Aichernig and P. G. Larsen. A proof obligation generator for VDM-SL. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 338–357. Springer-Verlag, September 1997.
- [AL99] S. Agerholm and P. G. Larsen. The IFAD VDM tools: Lightweight formal methods. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 326–329, London, UK, 1999. Springer-Verlag.

- [Ald03] G. Alder. *Design and Implementation of the JGraph Swing Component*, 1.0.6 edition, February 2003. Available online at <http://sourceforge.net/projects/jgraph/>.
- [Alh98] S. S. Alhir. *UML in a Nutshell*. O'Reilly, 1998.
- [Art91] R. D. Arthan. Formal specification of a proof tool. In S. Prehn and W. J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551, pages 356–370. Springer-Verlag, 1991.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
- [B4F] B4Free. Available online at <http://www.b4free.com/>.
- [Ban] R. Banach. *Retrenchment: An Overview*. School of Computer Science, University of Manchester. Available online at <http://www.cs.man.ac.uk/retrenchment/>.
- [Ban00] R. Banach. Maximally abstract retrenchments. In *ICFEM '00: Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods*, page 133, Washington, DC, USA, 2000. IEEE Computer Society.
- [Ban03] R. Banach. Retrenchment and system properties. 2003. Available online at http://www.cs.man.ac.uk/~banach/Recent_publications.html.
- [BBFM99] P. Behm, P. Benoit, A. Faivre, and J. Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 369–387, 1999.
- [BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

- [BD98] E.A. Boiten and J. Derrick. IO - refinement in Z. In A Evans, D Duke, and T Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, September 1998.
- [BD05] E. A. Boiten and J. Derrick. Formal program development with approximations. In Treharne et al. [TKHS05], pages 375–393.
- [BDM98] P. Behm, P. Desforages, and J-M. Meynadier. Meteor: An industrial success in formal development. In Bert, editor, *Proceedings of B-98*, volume 1393 of *Lecture Notes In Computer Science*, page 26. Springer, 1998.
- [BDMW97] J. Bicarregui, J. Dick, B. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29(1–2):53–78, July 1997.
- [Bel96] J. Belifante. On a modification of Goedel’s algorithm for class formation. *Association for Automated Reasoning News Letter*, (34):10–15, October 1996.
- [BF98] J. M. Bruel and R. B. France. Transforming UML models to formal specifications. In Pierre-Alain Muller and Jean Bézivin, editors, *Proc. International Conference on the Unified Modelling Language (UML): Beyond the Notation*, number 1618. Springer-Verlag, 1998.
- [BF05] R. Banach and S. Fraser. Retrenchment and the B-Toolkit. In Treharne et al. [TKHS05], pages 203–221.
- [BFM89] R. Bloomeld, P. Froome, and B. Monahan. SpecBox: A toolkit for BSI-VDM. *SafetyNet*, 5:4–7, 1989.
- [BG94] J. P. Bowen and M. J. C. Gordon. Z and HOL. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

- [BG95] J. P. Bowen and M. J. C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5-6):269–276, May–June 1995.
- [BG00a] K. Beck and E. Gamma. Test-infected: programmers love writing tests. In *More Java gems*, pages 357–376. Cambridge University Press, New York, NY, USA, 2000.
- [BG00b] R. Bussow and W. Grieskamp. *The Z of ZETA*. Technische Universitat Berlin, 2000. Available online at <http://uebb.cs.tu-berlin.de/zeta/>.
- [BH95a] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
- [BH95b] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.
- [BH97] J. P. Bowen and M. G. Hinchey. The use of industrial-strength formal methods. In *Proc. 21st International Computer Software and Application Conference (COMPSAC'97)*, pages 332–337, Washington D.C., USA, 13–15 August 1997. IEEE Computer Society Press.
- [BH06] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods ...ten years later. *Computer*, 39(1):40–48, 2006.
- [BHT97] J. P. Bowen, M. G. Hinchey, and D. Till, editors. *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, volume 1212 of *Lecture Notes in Computer Science*. Springer, 1997.
- [BJF⁺04] R. Banach, C. Jeske, S. Fraser, R. Cross, M. Poppleton, S. Stepney, and S. King. Approaching the formal design and development of complex systems: The retrenchment position. In *Workshop on Software and Complex Systems, 9th IEEE International*

Conference on Engineering of Complex Computer Systems, Florence, Italy, 14-16 April 2004, 2004.

- [BJPS06a] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the purse: Finite exception logs, and validating the small. In *IEEE SEW-30: 30th Annual Software Engineering Workshop*. IEEE Computer Society Press, 2006.
- [BJPS06b] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the purse: Hashing injective clear codes, and security properties. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *IEEE ISOLA 2006: 2nd Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation*, pages 76–89. IEEE Computer Society, 2006.
- [BJPS07] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the purse: The balance enquiry quandary, and generalised and $(1, 1)$ forward refinements. *Fundamenta Informaticae*, 77(1-2):29–69, 2007.
- [BKS88] R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, 1988.
- [BLM⁺87] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel, and L. Wos. Set theory in first-order logic: clauses for Godel’s axioms. *Journal of Automated Reasoning*, 2(3):287–327, 1987.
- [Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [BP98] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. In D. Bert, editor, *The 2nd International B Conference, Recent Advances in the Development and Use of the B Method, Montpellier*, volume 1393 of *Lecture Notes in Computer Science*, pages 129–147. Springer, 1998.

- [BP99] R. Banach and M. Poppleton. Sharp retrenchment, modulated refinement and simulation. *Formal Aspects of Computer Science*, 11(5):498–540, 1999.
- [BP02] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Requirements Engineering Journal*, 8:266–288, 2002.
- [BPJS05a] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the purse: Finite sequence numbers and the tower pattern. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Proc. FM-05, Newcastle, UK, July 2005*, volume 3582 of *LNCS*, pages 382–398. Springer, 2005.
- [BPJS05b] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenchment and the mondex electronic purse (extended abstract). In D. Beauquier, E. Boerger, and A. Slissenko, editors, *Proc. 12th International Workshop on Abstract State Machines (ASM'05), Paris, France, March 2005*, 2005.
- [BPJS07] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Sci. Comput. Program.*, 67(2-3):301–329, 2007.
- [BRS⁺00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *FASE '00: Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering*, pages 363–366, London, UK, 2000. Springer-Verlag.
- [BRSS99] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for provably correct systems. In *Current Trends in Applied Formal Methods*, Springer LNCS 1641. Boppard, Germany, 1999.
- [BRW03] A. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.

- [BSC94] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. BCS Practitioner Series. Prentice-Hall, 1994.
- [Buc98] M. Buchi. The B bank: A complete case study. In *Proceedings of the 2nd International Conference on Formal Engineering Methods*, page 190, Los Alamitos, CA, USA, 1998. IEEE Computer Society.
- [BvW98] R.J.R. Back and J. von Wright. *Refinement Calculus, A Systematic Introduction*. Springer, 1998.
- [CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
- [CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [CJO⁺05] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. Rodin (rigorous open development environment for complex systems). Project number IST 2004-511599. In *Fifth European Dependable Computing Conference: EDCC-5 supplementary volume*, pages 23–26, Budapest, Hungary, Apr 2005.
- [Cle] ClearSy. *Manuel de Reference du Language B, Version 1.8.5*.
- [CMM05] G. Carter, R. Monahan, and J. M. Morris. Software refinement with Perfect Developer. In B. K. Aichernig and B. Beckert, editors, *SEFM*, pages 363–373. IEEE Computer Society, 2005.
- [Coq05] The LogiCal Project, INRIA. *The Coq Proof Assistant Reference Manual 8.0*, January 2005. Available online at <http://coq.inria.fr/doc-eng.html>.
- [Cro03a] D. Crocker. Developing reliable software using object-oriented formal specification and refinement, 2003. Available online at <http://www.eschertech.com/papers/index.php>.

- [Cro03b] D. Crocker. Perfect Developer: A tool for object-oriented formal specification and refinement. In *Tools Exhibition Notes at Formal Methods Europe*, 2003.
- [CS01] E. M. Clarke and B. H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 21, pages 1367—1522. Elsevier Science, 2001.
- [CSW00] D. Cooper, S. Stepney, and J. Woodcock. Derivation of Refinement Proof Rules for Z: forwards and backwards rules incorporating input/output refinement. Technical Report PRG-127, Oxford University Computing Laboratory, October 2000.
- [Cur99] E. Currie. *The Essence of Z*. Prentice Hall, 1999.
- [CWA⁺96] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
- [Dar01] I. F. Darwin. *Java Cookbook*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2001.
- [Daw91] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.
- [DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.
- [Des98] P. Desforges. Using the B-method to design safety-critical software for railway systems. *Recherche et Developpements - Fatis Marquant 97*, 1998.
- [Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

- [dRE98] W-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- [Dro99] G. Droschl. Design and Application of a Test Case Generator for VDM-SL. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 47–55, 1999.
- [EC98] A. Evans and A. Clark. Foundations of the Unified Modeling Language. In *Proc. 2nd Northern Formal Methods Workshop, Electronic Workshops in Computing*. Springer-Verlag, 1998.
- [Ecla] Eclipse. Platform plug-in developer guide. Available online at <http://www.eclipse.org/documentation/>.
- [Eclb] Eclipse. Workbench user guide. Available online at <http://www.eclipse.org/documentation/>.
- [ELL94] R. Elmstrom, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *SIG-PLAN Not.*, 29(9):77–80, 1994.
- [End77] H. B. Enderton. *Elements of Set Theory*. Academic Press, 1977.
- [FB07] S. Fraser and R. Banach. Configurable proof obligations in the Frog toolkit. In M. Hinchey and T. Margaria, editors, *SEFM*, pages 361–370. IEEE Computer Society, 2007.
- [FEG92] B. Fields and M. Elvang-Goransson. A VDM case study in Mural. *IEEE Transactions on Software Engineering*, 18(4):279–295, 1992.
- [FEL97] R. B. France, A Evans, and K. Lano. The UML as a formal modeling notation. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA '97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

- [FL98] J. Fitzgerald and P. G. Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, New York, NY, USA, 1998.
- [Fla02] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 4th edition, 2002.
- [Fla04] D. Flanagan. *Java Examples in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2004.
- [FLM⁺05] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-Oriented Systems*. Springer, New York, 2005.
- [FMB89] P. K. D. Froome, B. Q. Monahan, and R. E. Bloomfield. SpecBox - a checker for VDM specifications. In *Proceedings of Second International Conference on Software Engineering for Real Time Systems*, Cirencester, UK, 1989. IEE.
- [Fow99] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Fra05] S. Fraser. Mechanised support for retrenchment in the B-Toolkit. Master's thesis, School of Computer Science, University of Manchester, 2005.
- [FS00] M. Fowler and K. Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [GL00] W. Grieskamp and M. Lepper. Encoding temporal logics in executable Z: A case study for the ZETA system. In *Logic Programming and Automated Reasoning*, pages 43–53, 2000.

- [GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [Gor96] M. Gordon. Set theory, higher order logic or both? In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 191–201, Turku, Finland, August 1996. Springer.
- [Gri00a] W. Grieskamp. *Notes on ZAP*. Technische Universitat Berlin, 2000. Available online at <http://uebb.cs.tu-berlin.de/zeta/>.
- [Gri00b] W. Grieskamp. *The Operation of ZETA*. Technische Universitat Berlin, 2000. Available online at <http://uebb.cs.tu-berlin.de/zeta/>.
- [Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
- [Hay92] I. J. Hayes. VDM and Z: A comparative case study. *Formal Asp. Comput.*, 4(1):76–99, 1992.
- [Hei97] C. Heitmeyer. Formal methods: A panacea or academic poppycock? In Bowen et al. [BHT97], pages 3–9.
- [Hei05] C. Heitmeyer. A panacea or academic poppycock: Formal methods revisited. *HASE*, 0:3–7, 2005.
- [HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.
- [HHS87] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Inf. Process. Lett.*, 25(2):71–76, 1987.

- [HJJ⁺04] A. Hilton, I. Johnson, C. Jones, S. Leppanen, I. Oliver, A. Romanovsky, and E. Troubitsyna. Definitions of case studies and evaluation criteria for case studies. RODIN Project (IST 2004-511599) Deliverable D2, 2004.
- [HNS97] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with Isabelle. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1997.
- [HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.
- [HOS97] M. A. Hewitt, C. O'Halloran, and C. T. Sennett. Experiences with PiZA, an animator for Z. In Bowen et al. [BHT97], pages 37–51.
- [Hud] S. Hudson. *CUP User Manual*. Graphics Visualization and Usability Center, Georgia Institute of Technology. Available online at <http://www2.cs.tum.edu/projects/cup/manual.html>.
- [Inf92] Information Management and Technology Division. Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992.
- [Inf02] Information Management and Technology Division. Review of results and limitations of an early national missile defense flight test. Report GAO-02-124, United States General Accounting Office, 2002.
- [ISO99] Project number JTC1.22.45—Z Notation, 1999. Final Committee Draft, CD 13568.2, August 24th 1999 – International Standard.

- [ISO02] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics, 2002. International Standard.
- [Jac94] D. Jackson. Abstract model checking of infinite specifications. In M. Naftalin, B. Tim Denvir, and M. Bertran, editors, *FME*, volume 873 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 1994.
- [Jac96] D. Jackson. Nitpick: A checkable specification language, 1996.
- [Jes05] C. Jeske. *Algebraic Integration of Retrenchment and Refinement*. PhD thesis, University of Manchester, 2005.
- [Jiaa] X. Jia. *ZANS: A Z Animation System*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University. Available online at <http://venus.cs.depaul.edu/fm/zans.html>.
- [Jiab] X. Jia. *ZTC: A Type Checker for Z Notation*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University. Available online at <http://venus.cs.depaul.edu/fm/ztc.html>.
- [Jia95] X. Jia. An approach to animating Z specifications. In *19th Annual International Computer Software and Applications Conference, Dallas, USA, August 1995*, 1995.
- [JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *MURAL: A Formal Development Support System*. Springer Verlag, Berlin, 1991.
- [JMT91] D. Jordan, J. A. McDermid, and I. Toyn. CADiZ - computer aided design in Z. In *Proceedings of the Fifth Annual Z User Meeting*, pages 93–104. Springer-Verlag, 1991.
- [JNW99] D. Jackson, Y. Ng, and J. M. Wing. A Nitpick analysis of mobile IPv6. *Formal Aspects of Computing*, 11(6):591–615, 1999.

- [Jon80] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.
- [Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.
- [Jon96] C. Jones. Formal methods light. *ACM Comput. Surv.*, 28(4es):121, 1996.
- [JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *ICSE*, pages 730–733, 2000.
- [KA96] D. J. King and R. D. Arthan. Development of practical verification tools. *ICL Systems Journal*, 11(1), May 1996.
- [Kal01] J. A. Kalman. *Automated Reasoning with Otter*. Rinton Press, Incorporated, 2001.
- [KD99] H. Kopka and P. W. Daly. *A Guide to L^AT_EX: Document Preparation for Beginners and Advanced Users*. Addison Wesley Professional, 3rd edition, 1999.
- [Ker78] B. Kernighan. *A TROFF Tutorial*. Bell Laboratories, 1978.
- [Kin90a] P. King. Printing Z and Object-Z L^AT_EX documents. Department of Computer Science, University of Queensland, 1990.
- [Kin90b] S. King. Z and the refinement calculus. In *VDM '90: Proceedings of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development*, pages 164–188, London, UK, 1990. Springer-Verlag.
- [K.L92] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.
- [Kna96] J. Knappmann. A PVS based tool for developing programs in the Refinement Calculus. Master's thesis, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel, Kiel, Germany, October 1996.

- [Lam94] L. Lamport. *LaTeX: A Document Preparation System*. Addison Wesley Professional, 2nd edition, 1994.
- [Lar01] P. G. Larsen. Ten years of historical development ‘bootstrapping’ VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.
- [LB03] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
- [LBP05] M. A. Leuschel, M. Butler, and S. Lo Presti. *The ProB User Manual, Version 1.1.4*, 2005.
- [LdR81] W. R. LaLonde and J. des Rivieres. Handling operator precedence in arithmetic expressions with tree transformations. *ACM Trans. Program. Lang. Syst.*, 3(1):83–103, 1981.
- [Lec02] T. Lecomte. Event driven B : methodology, language, tool support, and experiments. In *International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, 2002.
- [Led95] Y. Ledru. Specification and animation of a bank transfer. In *KBSE*, pages 192–199, 1995.
- [Led06] Y. Ledru. Using Jaza to animate RoZ specifications of UML class diagrams. In *ZUM 2006*, 2006.
- [Leu01] M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2001.
- [LEW⁺02] M. Loy, R. Eckstein, D. Wood, J. E. Iliott, and B. Cole. *Java Swing*. Second edition, 2002.

- [LG97] Luqi and J. A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, 1997.
- [LH96] K. Lano and H. Haughton. *Specification in B: An Introduction Using the B-Toolkit*. Imperial College Press, 1996.
- [Lio96] J. Lions. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.
- [Liu97] S. Liu. Evolution: A more practical approach than refinement for software development. In *ICECCS '97: Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97)*, page 142, Washington, DC, USA, 1997. IEEE Computer Society.
- [Liu99] Shaoying Liu. Software development by evolution, 1999. Available online at <http://www.sel.cs.hiroshima-cu.ac.jp/~liu>.
- [LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.
- [Lou93] K. C. Loudon. *Programming Languages: Principles and Practice*. Wadsworth Publ. Co., Belmont, CA, USA, 1993.
- [LP99] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.
- [LP05] P. G. Larsen and N. Plat. Introduction to Overture. In *Overture Workshop at Formal Methods Symposium FM'05 in Newcastle upon Tyne, UK*, July 2005.
- [LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.
- [LT05] M. Leuschel and E. Turner. Visualising larger state spaces in ProB. In Treharne et al. [TKHS05], pages 6–23.

- [Mar] A. Martin. Proposal: Community Z tools project (CZT). Available online at <http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/proposal.html>.
- [Mar02] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms with Poster*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [MAV05] C. Metayer, J. R. Abrial, and L. Voisin. Event-B language. RODIN Project (IST 2004-511599) Deliverable D7, May 2005.
- [McC01] W. McCune. Mace 2.0 reference manual and guide. Technical Report ANL/MCS-TM-249, Argonne National Laboratory 9700 South Cass Avenue Argonne, IL 60439, May 2001.
- [McC03] W. McCune. Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory 9700 South Cass Avenue Argonne, IL 60439, August 2003.
- [MF91] R. Moore and P. K. D. Froome. Mural and specbox. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 672–674, London, UK, 1991. Springer-Verlag.
- [MFMU05] T. Miller, L. Freitas, P. Malik, and M. Utting. CZT support for Z extensions. In J. Romijn, G. Smith, and J. van de Pol, editors, *IFM*, volume 3771 of *Lecture Notes in Computer Science*, pages 227–245. Springer, 2005.
- [MH92] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Trans. Softw. Eng.*, 18(9):817–826, 1992.
- [Mil05] A. Mills. *ANTLR*. <http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/antlr/antlrhome.html>, 2005.

- [MJG79] C. P. Wadsworth M. J. Gordon, A. J. Milner. *Edinburgh LCF, A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [ML02] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In Ludwik Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002:06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials*, pages 91–105. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002.
- [Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.
- [Mor94] C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1994. Available online at <http://web.comlab.ox.ac.uk/oucl/publications/books/PfS/>.
- [MP04] J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 372–384. Springer, 2004.
- [MQPss] J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, in press.
- [MTM97] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

- [MU05] P. Malik and M. Utting. CZT: A framework for Z tools. In Treharne et al. [TKHS05], pages 65–84.
- [NH05] J. P. Nielsen and J. K. Hansen. Designing a flexible kernel providing VDM++ support for Eclipse. In *Overture Workshop at Formal Methods Symposium FM'05 in Newcastle upon Tyne, UK*, July 2005.
- [Noe93] P. A. J. Noel. Experimenting with Isabelle in ZF set theory. *Journal of Automated Reasoning*, 10(1):15–58, 1993.
- [NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.
- [NSS57] A. Newell, J.C. Shaw, and H. Simon. Empirical explorations with the logic theory machine. *Computers and Thought*, 1957.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.
- [OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
- [OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.
- [Par93] T. Parr. *Obtaining Practical Variants of $LL(k)$ and $LR(k)$ for $k > 1$ by Splitting the Atomic k -tuple*. PhD thesis, Purdue University, 1993.
- [Par96] T. Parr. *Language translation using PCCTS and C++*. Automata Publishing Co., 1996.

- [Par05] T. Parr. *ANTLR Reference Manual (2.7.5)*. <http://www.antlr.org/doc/index.html>, January 28, 2005.
- [Pau92] L. C. Paulson. Set theory as a computational logic: I. From foundations to functions. Technical Report 271, Computer Laboratory, University of Cambridge, 1992.
- [Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
- [Pau04] L. C. Paulson. *Isabelle's Logics: FOL and ZF (Isabelle2004)*. Computer Laboratory, University of Cambridge, June 2004. Available online at <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/documentation.html>.
- [PLT00] L. Py, B. Legeard, and B. Tatibout. valuation de specifications formelles en programmation logique avec contraintes ensemblistes – application l’animation de specifications formelles B. In *AFADL'2000*, pages 21–35, LSR/IMAG – BP 72 38402 Saint-Martin d’Heres Cedex – Grenoble – France, January 2000. AFADL2000, LSR/IMAG, LSR/IMAG.
- [PQ95] T. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.
- [Qua92] A. Quaife. Automated deduction in von Neumann-Bernays-Godel set theory. *Journal of Automated Reasoning*, 8(1):91–147, 1992.
- [RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (system description)’. *Lecture Notes in Artificial Intelligence*, (2083):376–380, 2001.
- [RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Commun.*, 15(2):91–110, 2002.
- [Saa97] M. Saaltink. The Z/EVES system. In Bowen et al. [BHT97], pages 72–85.

- [SC97] R. L. Schwartz and T. Christiansen. *Learning Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.
- [Sch01] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.
- [Sch04] C. Schroder. *Linux Cookbook*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2004.
- [SCR96] D. Stringer-Calvert and J. Rushby. A less elementary tutorial for the PVS specification and verification system. *CSL Technical Report, CSL-95-10.*, 1996.
- [SCW98] S. Stepney, D. Cooper, and J. Woodcock. More powerful z data refinement: Pushing the state of the art in industrial refinement. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer, 1998.
- [SCW00] S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory, July 2000.
- [SDLW02] J. Sun, J. S. Dong, J. Liu, and H. Wang. A formal object approach to the design of ZML. *Ann. Softw. Eng.*, 13(1-4):329–356, 2002.
- [SFW05] E. Siever, S. Figgins, and A. Weber. *Linux in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 5th edition, 2005.
- [She95] D. Sheppard. *An Introduction to Formal Specifications with Z and VDM*. McGraw-Hill, 1995.
- [Smi00a] G. Smith. *The Object Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.
- [Smi00b] G. Smith. Stepwise development from ideal specifications. In *ACSC*, pages 227–233. IEEE Computer Society, 2000.

- [Spi90] J. M. Spivey. A guide to the zed style option. December 1990.
- [Spi92a] J. M. Spivey. *The fuzz Manual*, 1992. Available online at <http://spivey.orient.ox.ac.uk/mike/fuzz/>.
- [Spi92b] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [TKHS05] H. Treharne, S. King, M. C. Henson, and S. Schneider, editors. *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*. Springer, 2005.
- [TM95] I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Softw., Pract. Exper.*, 25(3):305–330, 1995.
- [Toy] I. Toyn. *CADiZ Reference Manual*. University of York. Available online at <http://www-users.cs.york.ac.uk/~ian/cadiz/>.
- [Toy96] I. Toyn. Formal reasoning in the Z notation using CADiZ. In *Proc. 2nd Workshop on User Interfaces to Theorem Provers, York, July 1996*.
- [Tra02] G. M. Travis. *The JDK 1.4 Tutorial*. Manning Publications, 2002.
- [TS02] I. Toyn and S. Stepney. Characters + mark-up = Z lexis. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB2002: Second International Conference of B and Z Users, Grenoble, France, January 2002*, volume 2272 of *LNCS*, pages 100–119. Springer, 2002.
- [UTS⁺03] M. Utting, I. Toyn, J. Sun, A. Martin, J. Song Dong, N. Daley, and D. W. Currie. ZML: XML support for standard Z. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB*,

- volume 2651 of *Lecture Notes in Computer Science*, pages 437–456. Springer, 2003.
- [Utt00] M. Utting. Data structures for Z testing tools, 2000. Presented at FM-Tools 2000. The 4th Workshop on Tools for System Design and Verification. Available online at <http://www.cs.waikato.ac.nz/~marku/papers/jaza.pdf>.
- [Utt05] M. Utting. *Jaza Manual and Tutorial*. The University of Waikato, 2005. Available online at <http://www.cs.waikato.ac.nz/~marku/jaza/>.
- [VDM] ISO/IEC 13817-1:1996. Information technology—Programming languages, their environments and system software interfaces – Vienna Development Method – Specification language – Part 1: Base language. International Standard.
- [vdS04] P. van der Spek. The Overture project: Towards an open source tool set. Master’s thesis, Department of Computer Science, Delft University of Technology, 2004.
- [vL00] A. van Lamsweerde. Formal specification: a roadmap. In *ICSE ’00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.
- [Voi06] L. Voisin. Description of the RODIN prototype. RODIN Project (IST 2004-511599) Deliverable D15, 2006.
- [Wal00] L. Wall. *Programming Perl*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000.
- [WB98] H. Waeslynck and S. Behnia. B model animation for external verification. In S. Liu J. Staples, M. Hinchey, editor, *Second International Conference on Formal Engineering Methods*, pages 36–45. IEEE Computer Society Press, December 1998.

- [WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.
- [Wei99] C. Weidenbach. System description: Spass version 1.0.0. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 378–382, London, UK, 1999. Springer-Verlag.
- [Wei01] C. Weidenbach. Spass: Combining superposition, sorts and splitting. In *Handbook of automated reasoning*, pages 1965–2013. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
- [Wer94] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Universit Paris VII, May 1994.
- [Wir71] N. Wirth. The development of programs by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.
- [Wor96] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.