Configurable Proof Obligations in the Frog Toolkit

Simon Fraser, Richard Banach
School of Computer Science, University of Manchester,
Manchester M13 9PL, UK.
{sfraser,banach}@cs.man.ac.uk

Abstract

In model based formal methods, incompatible tools for different techniques is the norm. However, greater applicability to industrial scale systems increasingly requires combining the strengths of different techniques, in line with the Verification Grand Challenge. The Frog tool embodies a construct-based specification syntax, and its meta-language Frog-CCL allows the generic configuration of both a construct's syntax and its proof obligations. For a specific system, Frog generates the system's verification conditions mechanically from the generic ones. Relationships between systems such as refinement and retrenchment can be configured. An example retrenchment between two simple systems illustrates the technique.

1. Introduction

In 2003 Hoare propounded a grand challenge for verification [13]. Since that time, the initial idea has evolved [16, 26] into a long-term research program with three principal objectives, one of them being the creation of a tool set that provides support for the verification process. The aim of such a tool set would be to produce compatible and comprehensive tools that are able to tackle all of the activities involved, since typically, each of the many verification tools that exist today tend to utilize a single technique, and in most cases they are unable to interact easily with tools belonging to other kits. This more dynamic approach to tools is slowly finding increasing support within the formal development community, with tools such as RODIN [10, 25] and Overture [18, 21] pursuing a more configurable approach.

The creation of the Frog toolkit, the subject of this paper, was principally motivated by the need to mechanically support retrenchment [7, 22, 23]. Retrenchment is an emerging formal technique that can fruitfully be used alongside refinement [2, 11] in the development of systems. The need for Frog to combine retrenchment and refinement meant that a highly configurable approach to the internal structure of the tool was desirable from the outset. Thus Frog 's need for flexibility neatly meets the contemporary trend for configurability and interworking, though it was conceived independently.

Frog is a toolkit supporting the rigorous development of model based specifications, generating proof obligations whose discharge would verify the associated development. Tools exist to support such proof driven model based processes, but they are typically monolithic, resisting the introduction and integration of techniques other than the one they were originally conceived to support. See [3] for an attempt to integrate retrenchment into the B-Toolkit[1] [17, 28].

So we decided to create a new toolkit that was capable of using a variety of techniques within a single development environment — and which moreover would keep the theory of those techniques distinct from the internals of the tool itself. Thus Frog was intended to be capable of supporting not just techniques like refinement and retrenchment as currently conceived, but to be fully configurable and extendable, so that it would support experimentation in the nature of the relationship between models. We sought therefore to produce a mechanizable framework where the shape of the models in a specification, and the relationships between those model, were fully manipulable. This paper describes such a framework.

2. A Construct-based Specification Syntax

Frog uses the notion of a construct to refer to an entity that specifies either a machine or a relationship between machines. We define a state transition machine to be a representation of some part of a system's behaviour that can be combined with other machines to specify a complete system (similar to that of an abstract machine in the B-Method [1]). We define a relationship to be a formal description of the way in which two machines' behaviour relates.

We use a simple syntax (which we will discuss in more detail below) to describe our constructs and leverage

[1] Developed by B-Core Ltd. (`http://www.b-core.com/`).

the Z notation [24, 27] to specify the behaviour of those constructs. An example of a machine with a single state variable and an operation to increment that variable is shown below.

MACHINE $myNumberMachine$
TYPE $simpleMachine$
SECTION $standard\_toolkit$
STATE

$a : \mathbb{N}$

INITIALIZATION

$| \; a = 0$

OPERATION $increment \mathrel{\widehat=}$
POST

$| \; a' = a + 1$

END $increment$

END $myNumberMachine$

As our system stipulates that constructs must be configurable, we need to specify that we are using a machine and also the particular configuration used in this instance. The 'TYPE' clause is used to indicate a construct's configuration; in this instance $simpleMachine$. The only other clause of notable interest is the 'SECTION' clause that allows us to take advantage of Z's section structuring, and declare one or more parent sections for our construct. As with standard Z this permits us to use the variables defined in that section; here we use the standard mathematical toolkit as described in Appendix A of [14].

3. Frog-CCL

We now present a meta-language, Frog-CCL, 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 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 its proof obligations. In the future we may desire to incorporate more intricate options in configurations. 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 and cannot be further refined. We did not feel that this level of detail would be required in our first iteration, but have attempted to make the definition as extensible as possible.

3.1. Syntax

In the syntax of Frog-CCL (the grammar for this language can be found in [12]), a construct's configuration has three constituents. 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 its 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. 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. An example of the configuration for a simple machine is given below.

```
DEFINE MACHINE simpleMachine
  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
```

The first section of our configuration describes the available clauses. Each clause has a number of attributes most of which are self-explanatory. The 'LEVEL' attribute indicates whether a clause is used at the construct level (in which case we use the keyword 'MACHINE') or is specific to an operation environment (in which case we use the name of the required operation environment). In our example we have clauses to store the state of a machine and to initialize that state at the construct level. We then define a number of clauses to be used to form an operation, these define the inputs and outputs of the operation and its pre and postconditions. Of interest is the 'RELATION' attribute. It is used to both indicate the variables that are within the scope of the clause and also the signature of the relation produced when the clause is used in a proof obligation. When configuring a machine we can refer to the clauses of the machine, with a relationship we can refer to the clauses of both machines involved in that relationship. If a clause name is used within the relation attribute, and is decorated with an apostrophe we also decorate all of its variables likewise[2]. We will discuss the use of the 'RELATION' attribute in generating proof obligations below.

Each construct can also have a number of operation environments, each of which again has a number of attributes. In our example we use an operation environment to perform an operation on the state of a machine. The operation environment itself needs little configuration and is used principally as a container.

Finally, each construct will also have a number of proof obligation configurations, the first part of which specifies the level of the proof obligation. Construct level proof obligations will be instantiated once for every construct. Operation level proof obligations will be instantiated once for every valid operation environment. The second part is a specification of the proof obligation itself. We provide a basic, ASCII syntax for creating theorems that can express the proof obligation. This syntax is best understood by examining how the configuration is used to create the specific proof obligations for an instance of a construct.

3.2. Generating Proof Obligations

The first stage of instantiating the generic proof obligations of a configuration is to create an instance of each 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 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 will illustrate this process through an example. Consider the machine $myNumberMachine$ alongside the $simpleMachine$ configuration that we presented in section 3.1.

Construct Level Proof Obligations. Our configuration states that we have a single construct level proof obligation which we can present in the Z notation as follows.

$$\vdash ?\ \exists\, u \bullet u \in state \land u \in initialization$$

Each clause in the generic proof obligation is substituted with its equivalent relation. Woodcock and Davies [27] 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,

$$\{state\_schema \bullet \theta state\_schema\}$$

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.

$$state\_schema == [a : \mathbb{N}]$$

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.

$$\vdash ?\ \exists\, u \bullet u \in \{a : \mathbb{N}\} \land u \in initialization$$

Consider now the generation of the relation for the initialization clause of the example machine. Again, we look first

[2] This 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.

at the relation attribute and determine that the relation requires the variables of the state clause. The relation we require is described below.

$$\{initialization\_schema \bullet \theta state\_schema\}$$

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. Therefore, the schema of the state clause is incorporated into our new schema's declaration.

$$\begin{aligned} initialization\_schema &== [state\_schema \mid a = 0] \\ &== [a : \mathbb{N} \mid a = 0] \end{aligned}$$

We can then substitute our schema definitions into our relation definition and resolve the binding construction, giving a final relation definition as follows.

$$\{a : \mathbb{N} \mid a = 0\}$$

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.

$$\vdash ?\ \exists\, u \bullet u \in \{a : \mathbb{N}\} \land u \in \{a : \mathbb{N} \mid a = 0\}$$

Operation Level Proof Obligations. We only have one operation environment in our machine specification and two operation level proof obligations in our configuration so we create an instance of each generic proof obligation. For the sake of brevity we will only consider the second of these, the functional correctnes obligation, a Z representation of which is given below.

$$\begin{aligned} &\vdash ?\ \forall\, u, i \bullet u \in state \land i \in operation.inputs \\ &\quad \land\ \langle u, i \rangle \in operation.pre \\ &\quad \Rightarrow (\exists\, u', o \bullet u' \in state \land o \in operation.outputs \\ &\qquad \land \langle u, u', i, o \rangle \in operation.post) \end{aligned}$$

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.

$$\begin{aligned} &state == \{a : \mathbb{N}\} \\ &operation.post == \{a : \mathbb{N};\ a' : \mathbb{N} \mid a' = a + 1\} \end{aligned}$$

When we come to replace the remaining clauses, we notice that they have not been defined in the $increment$ operation environment. We therefore need to eliminate the use of these clauses in our proof obligation. If we use the available relations to replace their respective clauses, our current proof obligation is as follows.

$$\begin{aligned} &\vdash ?\ \forall\, u, i \bullet u \in \{a : \mathbb{N}\} \land i \in operation.inputs \\ &\quad \land\ \langle u, i \rangle \in operation.pre \\ &\quad \Rightarrow (\exists\, u', o \bullet u' : \{a : \mathbb{N}\} \\ &\qquad \land\ o \in operation.outputs \\ &\qquad \land \langle u, u', i, o \rangle \in \{a : \mathbb{N};\ a' : \mathbb{N} \mid a' = a + 1\}) \end{aligned}$$

The first stage in the elimination of the unused clause is the replacement of references to that clause with the special term $\perp$. 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. The second stage involves the exhaustive application of the rules in the table below.

Original term	Resolved term
$x \in \perp$	$\perp$
$P \land \perp$	P
$P \lor \perp$	P
$\forall\, x \bullet \perp$	$\perp$
$\exists\, x \bullet \perp$	$\perp$
$\forall\, n_1, ..., n_{i-1}, \perp, n_{i+1}, ..., n_j \bullet P$	$\forall\, n_1, ..., n_{i-1}, n_{i+1} ..., n_j \bullet P$
$\exists\, n_1, ..., n_{i-1}, \perp, n_{i+1}, ..., n_j \bullet P$	$\exists\, n_1, ..., n_{i-1}, n_{i+1} ..., n_j \bullet P$
$P \Rightarrow \perp$	P
$\perp \Rightarrow P$	P
$x = \perp$	true
$(n_1, ..., n_{i-1}, \perp, n_{i+1}, ..., n_j)$	$(n_1, ..., n_{i-1}, n_{i+1}, ..., n_j)$

The application of these rules to the proof obligation of our example gives the following result.

$$\begin{aligned} &\vdash ?\ \forall\, u \bullet u \in \{a : \mathbb{N}\} \\ &\quad \Rightarrow (\forall\, u' \bullet u' : \{a : \mathbb{N}\} \\ &\qquad \land \langle u, u' \rangle \in \{a : \mathbb{N};\ a' : \mathbb{N} \mid a' = a + 1\}) \end{aligned}$$

4. Incorporating Constructs within a Z Specification

We now describe how the Z syntax can be extended and how this augmented syntax can be syntax checked mechanically following the framework described in the ISO Z standard [14].

4.1. Extending the Syntax

In order to allow our constructs to be parsed within an existing Z framework we extend the LaTeX syntax presented in [14] to allow the specification of machines and relationships. In order to maintain the utmost flexibility we introduce just four directives that allow us to present a machine, relationship, clause and operation environment. Again the grammar for this extended syntax is presented in [12]. Below we present the syntax for our example machine.

```
\begin{machine}{myNumberMachine}{simpleMachine}
      {standard\_toolkit}
\clause{state}{a : \nat}
\clause{initialization}{ | a = 0 }
\begin{openv}{operation}{increment}
\clause{post}{ | a' = a + 1}
\end{openv}
\end{machine}
```

4.2. Parsing the Extended Grammar

When parsing a construct we need to ensure that as well as being grammatically correct, the construct obeys its configuration rules. 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.

Hence when a construct is parsed, we must first relate it to an existing configuration. We immediately know whether a construct is a machine or relationship through the LaTeX directive used; the particular configuration of a machine or relationship can be determined by taking the name declared in the construct's specification (the second argument to the directive) and searching for a match in the available configurations. If no match is found, the construct will be rejected.

Once we have a valid configuration, a Z section is 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 parent sections they will be declared as the parents of the newly created section; if no parent section is specified then the standard toolkit will be used. A section created for a relationship will have two parent sections, corresponding to the source and target machines. The declaration of these parent sections allows the tokens and variables defined to be referred to within the clauses of a child construct.

The clauses and operation environments of the construct can then be examined. The attributes of the current clause or operation environment are determined by finding a matching section from the construct's configuration. A clause possesses attributes that determine the level of the clause and its content. Should a clause be used at a level contradicting the level specified in the configuration it will not be possible to parse the construct successfully. The content attribute allows us to invoke a Z parser on the clause's contents (informing it whether we expect a predicate or schema text) and determine any syntax errors within. Finally, it is necessary to ensure that any mandatory clauses have been declared in the construct or operation environment in which they are expected. Similarly, all mandatory operation environments must be present.

[3] Whilst the use of an initial double underscore character is technically illegal in the Z notation, we use it internally within our system to prevent unintentional name clashes.

4.3. 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 semantic processing. This process transforms our construct definition into a specification that matches the grammar for the annotated Z syntax described in chapter 10 of [14].

As described above, every construct is translated into an equivalent Z section and through the transformation process every clause becomes a Z paragraph. We illustrate the process using our running example. The section header is created as described above.

section $\_\_machine\_myNumberMachine$
parents $standard\_toolkit$

We then consider each of the clauses within our machine. The first is the state clause which will be transformed into an axiomatic definition paragraph. The name of the schemas produced from context level clauses are derived from the names of the machine and the clause. The content of the clause is transformed using the usual rules for schema texts and predicates (as appropriate).

$$[\_myNumberMachine\_state : \{[a : \mathbb{N}]\}]$$

The initialization clause is transformed in the same way as the state clause. However, note that in this instance we will use the 'RELATION' attribute of the machine's configuration to incorporate the state variables and we include a reference to the appropriate schema.

$$\begin{array}{l} [\_myNumberMachine\_initialization : \{ \\ [\_myNumberMachine\_state \\ \mid a \in \{number\_literal\_0\}]\}] \end{array}$$

The only difference between a clause defined at the construct level and one at the operation level is the way in which the produced schema's name is derived. Here we incorporate the operation environment name to ensure unicity within the section.

$$\begin{array}{l} [\_myNumberMachine\_increment\_post : \{ \\ [\_myNumberMachine\_state; \\ \_myNumberMachine\_state' \\ \mid a' \in \{\bowtie + \bowtie (a, number\_literal\_1\}]\}] \end{array}$$

The process of producing a Z section from a construct specification is fully automatable and again is described in more detail in [12]. Once the transformation process is complete the construct is entirely in the annotated Z syntax. Therefore, there is no necessity to alter the standard type checking procedure described in chapter 13 of [14].

5. Configuring the Relationship

Of course the aim of our work was to be able to produce a framework that would allow the specification and verification of configurable relationships between machines. The dynamic nature of these relationships allows us to use a variety of formal techniques to relate the models represented by our machines.

Refinement is currently one of the most widely used techniques. It involves the incremental addition of procedural detail to an abstract specification, until we reach a model of the system that can be readily converted into programming language instructions. Refinement's key property is that the behaviours of the model produced at each step can be formally related, and can be shown to meet the requirements of the most abstract. However, there are circumstances where refinement can require us to include unnecessary complexity at the abstract level. For example, consider the common refinement of an infinite set to a finite sequence. Clearly, the specified and practical versions of the model will not exhibit identical behaviour. In order to make one a refinement of the other, the abstract model must be extended to incorporate the procedural detail or modelled using $skip$s. Neither of these solutions is particularly satisfactory. The first adds complexity to the abstract model and surely the point of refinement is to add this detail gradually. The second confuses the abstract specification and could be considered misleading [5].

Retrenchment can be seen as a more liberal version of refinement. A retrenchment step is not just a transition from an abstract model to a more concrete but faithful version of that same model, but a transparent description of the relationship between the two models. Retrenchment allows abstract specifications, which provide only high-level stipulations that result from the system's high-level requirements, to be used by those needing only such a high-level view of the application. Meanwhile more concrete specifications, incorporating incompatible implementation requirements, can be used by developers. Users of both can be safe in the knowledge that they are working with well-understood, retrenchment-related, models that meet the system's needs.

As there are many tools that support refinement, and the mechnical support for retrenchment was one of the motivators for the creation of our tool, we will demonstrate how Frog-CCL can be used to configure the proof obligations for retrenchment. There are a number of forms of retrenchment [4, 6]. Here we will describe the most used one, output retrenchment (nowadays usually refered to as just retrenchment [4]). In our example we will configure a relationship between abstract and concrete machines possessing the configuration that we introduced in section 3.1. The retrenchment relationship allows us to augment the retrieve relation of the refinement relationship, with within, concedes and output relations. We will use the clauses that define these relations to construct appropriate proof obligations for the retrenchment relationship.

It is in the construction of these proof obligations that the power of the configuration language can be seen. We use the first proof obligation to show that wherever the more concrete machine can be initialized, there is an equivalent initialization of the more abstract machine. This obligation simply describes the relationship between the construct level, initialization clauses of the machines and the retrieve clause of the retrenchment.

The second, applicability proof obligation is more interesting. In the simple system we have described in this paper we have considered our machine's operations to be partial relations (in the same way as Z's operations). We have also named the domain of those operations as preconditions indicating that we do not guarantee behaviour when those preconditions are not met. However, the advantage of our configuration-based framework is that we are not constrained to such an approach. Should a theory require it, Frog-CCL is perfectly capable of handling a guarded approach or even a non-standard, three valued interpretation [20]. As our machines have used preconditions to restrict the domain, we use the proof obligation described in [15]. That is, if there is a valid initial relationship between the machines and the within relation holds, then the operation of both abstract and concrete machines is applicable.

Similarly we obtain our final proof obligation from the same source and use it to show, that for each corresponding pair of operations in the machines involved, and 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 a corresponding state transition in the more abstract machine, that will give a pair of post-transition states that belong, either to both the retrieve relation and output relation, or to the concedes relation.

```
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,v> : retrieve &
            u : FROM_MACHINE(initialization) )),
 (  OPERATION_LEVEL,
    (  ! u, v, i, j
       @ <u,v> : retrieve
       & <u,v,i,j> : ramifications.within
       =>
       <u,i> : FROM_MACHINE(operation.pre)
       & <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'> : ramifications.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 relationship-type 'retrenchment'; it should be noted, however, that we do not (in our configuration) indicate the types of the machines that will be involved in the retrenchment. 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 $simpleMachine$, an as yet undefined configuration $complexMachine$ or a mix of the two. Obviously the validity of relating two machines is theory-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 a machine specified as a source. The semantic compatibility of the machines is left to the specifier of the relationship to ensure[4].

[4] 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 now consider an example specification. We will model the set of known stars in the sky. We shall assume that we have previously specified a section $theStarsInTheSky$ that defines a carrier set $stars$ which represents all the stars in the universe (known or not). The first step then is to create a simple abstract machine that maintains a set of known stars and defines an operation that models the discovery of a star.

MACHINE $starsInTheSky$
TYPE $simpleMachine$
SECTION $theStarsInTheSky$
STATE

$starsInSky : \mathbb{P}\, stars$

INITIALIZATION

$| \; starsInSky = \varnothing$

OPERATION $discoverStar \mathrel{\widehat=}$
INPUT

$newStar? : stars$

PRE

$| \; newStar? \notin starsInSky$

POST

$| \; starsInSky' = starsInSky \cup \{newStar?\}$

END $discoverStar$

END $starsInTheSky$

We then define a concrete machine that must handle the situation in which the space holding our list of stars is full. Note that, retrenchment allows us to have different sets of inputs and outputs in the operations of abstract and concrete machine. (Assume also that we defined the free type $MESSAGES$ in our parent section $theStarsInTheSky$.)

MACHINE $starsInTheSkyC$
TYPE $simpleMachine$
SECTION $theStarsInTheSky$
STATE

$starsInSky : \mathbb{P}\, stars$

INITIALIZATION

$| \; starsInSky = \varnothing$

OPERATION $discoverStar \mathrel{\widehat=}$
INPUT

$newStar? : stars$

OUTPUT

$message! : MESSAGES$

PRE

$| \ newStar? \notin starsInSky$

POST

$$
\begin{array}{l}
\hline
\#starsInSky < upperlimit \\
\qquad \Rightarrow starsInSky' = starsInSky \cup \{newStar?\} \\
\qquad\quad \wedge\ message! = starAdded \\
\#starsInSky \geq upperlimit \\
\qquad \Rightarrow starsInSky' = starsInSky \\
\qquad\quad \wedge\ message! = starArrayFull
\end{array}
$$

END $discoverStar$

END $starsInTheSkyC$

Finally we consider the specification of the retrenchment relationship between the two machines. Note that, we can abbreviate our machine names (so that $starsInTheSky$ becomes s) and that we can differentiate between identically named variables belonging to different machines using the $\gg$ syntax (for example, $s{\gg}starsInSky$ refers to the set of known stars in our abstract model). The relationship below describes an appropriate retrenchment.

RELATIONSHIP $starsInTheSky\_to\_starsInTheSkyC$
TYPE $retrenchment$
FROM $starsInTheSky$ AS s
TO $starsInTheSkyC$ AS sC
RETRIEVE

$s{\gg}starsInSky = sC{\gg}starsInSky$

RAMIFICATIONS $discoverStar \mathrel{\widehat=}$
WITHIN

$s{\gg}newStar? = sC{\gg}newStar?$

OUTPUT

$sC{\gg}message! = starAdded$

CONCEDES

$$
\begin{array}{l}
\#sC{\gg}starsInSky \geq upperlimit \\
sC{\gg}starsInSky' \\
\qquad = s{\gg}starsInSky' \setminus \{sC{\gg}newStar?\} \\
sC{\gg}message! = starArrayFull
\end{array}
$$

END $discoverStar$

END $starsInTheSky\_to\_starsInTheSkyC$

The ramifications for the $discoverStar$ operation allow us to see retrenchment in action. The within clause is used to tighten the precondition by stating that not only must the retrieve relation hold prior to a transition, but the new star discovered in each case must be the same. We then use the output clause to handle the operation-specific augmentation of the retrieve clause, here clarifying the correct output when the retrieve clause holds. Finally, we define the concedes relation that allows us to loosen the postcondition. This states that the retrieve clause (and output clause) will not be true when the maximum number of stars that can be stored has been exceeded, but that the set of stars in the concrete model will still be equal to that in the abstract model minus the new star (and also that an error message will be generated).

We can now show how our proof obligations can be extrapolated from the above specification and configuration. Consider first the generic initialization obligation contained in our configuration.

$$
\begin{array}{l}
\vdash ?\ \forall\, v \bullet v \in \text{TO\_MACHINE}(initialization) \\
\quad \Rightarrow (\exists\, u \bullet u \in \text{FROM\_MACHINE}(initialization) \\
\qquad \wedge\ (u, v) \in retrieve)
\end{array}
$$

The details of our specifications can be used to substitute the clauses with their corresponding relations, resulting in the following theorem.

$$
\begin{array}{l}
\vdash ?\ \forall\, v \bullet v \in \{starsInSky \in \mathbb{P}\, stars \mid starsInSky = \varnothing\} \\
\quad \Rightarrow (\exists\, u \bullet \\
\qquad u \in \{starsInSky \in \mathbb{P}\, sky \mid starsInSky = \varnothing\} \\
\qquad \wedge\ \ (u, v) \in \{s{\gg}starsInSky \in \mathbb{P}\, stars; \\
\qquad\quad sC{\gg}starsInSky \in \mathbb{P}\, stars \\
\qquad\quad \mid s{\gg}starsInSky = sC{\gg}starsInSky\})
\end{array}
$$

The proof obligation can be discharged easily enough, but the point here is to illustrate that it is possible to take a retrenchment specification, alongside a Frog-CCL configuration and use the framework we have described to generate the proof obligation mechanically without any notion of retrenchment hardwired in the underlying system.

Similarly, we consider the second, applicability proof obligation, which is the first to examine the relationship between our machine's operations. This proof obligation establishes that whenever we are in a state in which the retrieve and within clauses hold, the precondition of both abstract and concrete machines' operation will be met. The generic obligation from our configuration is shown below.

$$
\begin{array}{l}
\vdash ?\ \forall\, u, v, i, j \bullet \\
\quad (u, v) \in retrieve \wedge (u, v, i, j) \in within \\
\quad \Rightarrow (u, i) \in \text{FROM\_MACHINE}(operation.pre) \\
\quad \wedge\ \ (v, j) \in \text{TO\_MACHINE}(operation.pre)
\end{array}
$$

Once again we use the rules that we have defined in previous sections, along with the specifications of the machines and relationship, to successfully instantiate the proof obligation as follows.

$$
\begin{array}{l}
\vdash ?\ \forall u, v, i, j \bullet \\
\quad (u, v) \in \{s{\gg}starsInSky : \mathbb{P}\, stars; \\
\qquad sC{\gg}starsInSky : \mathbb{P}\, stars \\
\qquad \mid s{\gg}starsInSky = sC{\gg}starsInSkyC\} \\
\quad \land\ \ (u, v, i, j) \in \{s{\gg}starsInSky : \mathbb{P}\, stars; \\
\qquad sC{\gg}starsInSky : \mathbb{P}\, stars; \\
\qquad s{\gg}newStar? : stars;\ sC{\gg}newStar? : stars \\
\qquad \mid s{\gg}newStar? = sC{\gg}newStar?\} \\
\quad \Rightarrow (u, i) \in \{starsInSky : \mathbb{P}\, stars; \\
\qquad newStar? : stars \mid newStar? \notin starsInSky\} \\
\quad \land\ \ (v, j) \in \{starsInSky : \mathbb{P}\, stars; \\
\qquad newStar? : stars \mid newStar? \notin starsInSky\}
\end{array}
$$

Again, the proof obligation is generated easily enough and could be discharged fairly easily. It is when we examine the functional correctness proof obligation that we begin to see the complexity of a proof obligation for even a toy example such as this. With this complexity the benefits of a mechanized, and configurable framework are highlighted. As before, we begin with the generic proof obligation.

$$
\begin{array}{l}
\vdash ?\ \forall u, v, v', i, j, p \bullet \\
\quad (v, v', j, p) \in \text{TO\_MACHINE}(operation.post) \\
\quad \land\ \ (u, v) \in retrieve \land (u, v, i, j) \in within \\
\quad \land\ \ (u, i) \in \text{FROM\_MACHINE}(operation.pre)) \\
\quad \Rightarrow (\exists\, u', o \bullet \\
\qquad (u, u', i, o) \\
\qquad\quad \in \text{FROM\_MACHINE}(operation.post) \\
\qquad \land (((u', v') \in retrieve \\
\qquad\quad \land (u, v, u', v', i, j, o, p) \in output) \\
\qquad\quad \lor (u, v, u', v', i, j, o, p) \in concedes)
\end{array}
$$

The clauses are then substituted with their relations in accordance with our rules and specifications.

$$
\begin{array}{l}
\vdash ?\ \forall u, v, v', i, j, p \bullet \\
\quad (u, i) \in \{starsInSky : \mathbb{P}\, stars;\ newStar? : stars \\
\qquad \mid newStar? \notin starsInSky\} \\
\quad \land\ \ (u, v) \in \{s{\gg}starsInSky, sC{\gg}starsInSky \\
\qquad : \mathbb{P}\, stars \mid s{\gg}starsInSky = sC{\gg}starsInSky\} \\
\quad \land\ \ (v, v', j, p) \in \{starsInSky, starsInSky' : \mathbb{P}\, stars; \\
\qquad newStar? : stars \mid (\#starsInSky < upperlimit \\
\qquad \Rightarrow starsInSky' = starsInSky \cup \{newStar?\} \\
\qquad \land\ message! = starAdded) \land (\#starsInSky \geq \\
\qquad upperlimit \Rightarrow starsInSky' = starsInSky \\
\qquad \land\ message! = starArrayFull)\} \\
\quad \land\ \ (u, v, i, j) \in \{s{\gg}starsInSky : \mathbb{P}\, stars; \\
\qquad sC{\gg}starsInSky : \mathbb{P}\, stars;\ s{\gg}newStar? \\
\qquad : stars;\ sC{\gg}newStar? : stars \\
\qquad \mid s{\gg}newStar? = sC{\gg}newStar?\} \\
\quad \Rightarrow (\exists\, u' \bullet \\
\qquad (u, u', i) \in \{starsInSky, starsInSky' : \mathbb{P}\, stars; \\
\qquad\quad newStar? : stars \\
\qquad\quad \mid starsInSky' = starsInSky \cup \{newStar?\}\} \\
\qquad \land\ \ (((u', v') \in \{s{\gg}starsInSky, sC{\gg}starsInSky \\
\qquad\quad : \mathbb{P}\, stars \mid s{\gg}starsInSky = sC{\gg}starsInSky\} \\
\qquad\quad \land (u, u', v, v', i, j, p) \in \\
\qquad\quad \{s{\gg}starsInSky, s{\gg}starsInSky' : \mathbb{P}\, stars; \\
\qquad\quad sC{\gg}starsInSky, sC{\gg}starsInSky' : \mathbb{P}\, stars; \\
\qquad\quad s{\gg}newStar? : stars;\ sC{\gg}newStar? : stars; \\
\qquad\quad sC{\gg}message! : MESSAGES \\
\qquad\quad \mid message! = starArrayFull\}) \\
\qquad\quad \lor (u, u', v, v', i, j, p) \in \\
\qquad\quad \{s{\gg}starsInSky, s{\gg}starsInSky' : \mathbb{P}\, stars; \\
\qquad\quad sC{\gg}starsInSky, sC{\gg}starsInSky' : \mathbb{P}\, stars; \\
\qquad\quad s{\gg}newStar? : stars;\ sC{\gg}newStar? : stars; \\
\qquad\quad sc{\gg}message! : MESSAGES \\
\qquad\quad \mid \#sC{\gg}starsInSky \geq upperlimit \\
\qquad\quad \land\ sC{\gg}starsInSky' = s{\gg}starsInSky' \\
\qquad\qquad \setminus \{sC{\gg}newStar?\} \\
\qquad\quad \land\ sC{\gg}message! = starArrayFull\}))
\end{array}
$$

6. Conclusion

In this paper 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 those constructs. 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 and retrenchment relationships.

This fully configurable approach to constructs allows users to experiment with the nature of models and the relationships between them, and we believe it would be a major asset to any tool. Not only does it allow the specifier to use any existing or future formal relationship, it also permits the optimization of the relationship in a particular instance and all without any change to the underlying system.

The framework described in this tool has already been implemented as part of the Frog tool described in [12] and has been used to describe not only some of the many flavours of retrenchment, but also to dynamically create relationships that are able to provide additional rigour in specific examples.

In the future there are many possibilities for extending the power of the configuration language. Some interesting examples would be: introducing the ability to specifically deny or allow interaction between specific machine and relationship types, or introducing inheritance between configurations (for example, having a retrenchment 'super-configuration' from which the configurations of other flavours of retrenchment could inherit behaviour).

Other tool developers have taken a different approach to allowing configuration within their tools. An increasingly popular approach is to extend the Eclipse [8, 19] platform. Eclipse relies extensively on plug-ins [9] to augment the core functionality with support for many languages and notations. RODIN and Overture are projects involving the creation of toolkits supporting formal methods in this way (B and VDM respectively). For example, RODIN uses separate plug-ins for type checking, model checking and verification. Hence it is possible to create support for techniques such as retrenchment through the development of custom plug-ins. Of course, this approach still requires effort to implement these custom plug-ins. While these toolkits are a step in the right direction, we feel that Frog's configuration framework offers much the same flexibility, but with a significantly greater ease-of-use.

In summary, it is hard to argue that hard coding the generation of proof obligations can be favoured over a more generic framework such as that demonstrated here. We hope that other tool developers will take note and incorporate a more flexible approach in the implementation of future tools.

References

[1] J. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] R. Back and J. von Wright. *Refinement Calculus, A Systematic Introduction*. Springer, 1998.

[3] R. Banach and S. Fraser. Retrenchment and the B-Toolkit. In H. Treharne, S. King, M. C. Henson, and S. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 203–221. Springer, 2005.

[4] R. Banach and C. Jeske. Stronger compositions for retrenchments, and feature engineering. 2002. Available online at `http://www.cs.man.ac.uk/~banach/Recent.publications.html`.

[5] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. *Lecture Notes In Computer Science*, 1393:129–147, 1998.

[6] R. Banach and M. Poppleton. Sharp retrenchment, modulated refinement and simulation. *Formal Aspects of Computer Science*, 11(5):498–540, 1999.

[7] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. 2005. Available online at `http://www.cs.man.ac.uk/~banach/Recent.publications.html`.

[8] D. Carlson. *Eclipse Distilled*. Addison Wesley, 2005.

[9] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. Addison Wesley, 2006.

[10] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (rigorous open development environment for complex systems). In *Fifth European Dependable Computing Conference: EDCC-5 supplementary volume*, pages 23–26, Budapest, Hungary, Apr 2005. RODIN Project (IST 2004-511599).

[11] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[12] S. Fraser. *Mechanized Support for Retrenchment*. PhD thesis, School of Computer Science, University of Manchester, 2007.

[13] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[14] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.

[15] C. Jeske. *Algebraic Integration of Retrenchment and Refinement*. PhD thesis, School of Computer Science, University of Manchester, 2005.

[16] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, 2006.

[17] K. Lano and H. Haughton. *Specification in B: An Introduction Using the B-Toolkit*. Imperial College Press, 1996.

[18] P. G. Larsen and N. Plat. Introduction to Overture. In *Proc. Overture Workshop at Formal Methods Symposium FM'05 in Newcastle upon Tyne, UK*, July 2005.

[19] J. McAffer and J. M. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison Wesley, 2005.

[20] R. Miarka, E. A. Boiten, and J. Derrick. Guards, preconditions, and refinement in z. In *ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 286–303, London, UK, 2000. Springer-Verlag.

[21] J. P. Nielsen and J. K. Hansen. Designing a flexible kernel providing VDM++ support for Eclipse. In *Proc. Overture Workshop at Formal Methods Symposium FM'05 in Newcastle upon Tyne, UK*, July 2005.

[22] M. Poppleton and R. Banach. Retrenchment: Extending the reach of refinement. In *Proc. ASE-99*, IEEE, pages 158–165, 1999.

[23] M. Poppleton and R. Banach. Controlling control systems: An application of evolving retrenchment. In Bert, Bowen, Henson, and Robinson, editors, *Proc. ZB-02*, volume 2272 of *Lecture Notes In Computer Science*, pages 42–61. Springer, 2002.

[24] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[25] L. Voisin. Description of the RODIN prototype. RODIN Project (IST 2004-511599) Deliverable D15, 2006.

[26] J. Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, 2006.

[27] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.

[28] J. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.