OPTIMISING TABLEAUX DECISION PROCEDURES FOR DESCRIPTION LOGICS

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

1997

By
Ian R. Horrocks
Department of Computer Science

Contents

Abstract **12**

Declaration **13**

Acknowledgements **14**

The Author **15**

1 Introduction **16**

1.1 Description Logics 16

1.2 DL Applications 18

1.2.1 GALEN and GRAIL 18

1.3 Subsumption Reasoning in DLs 19

1.4 $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ and the FaCT System 21

1.5 Thesis Outline 21

2 Formal Foundations of Description Logics **23**

2.1 Syntax 23

2.1.1 Formal Syntax 25

2.2 Model Theoretic Semantics 26

2.2.1 Concept Expressions 28

2.2.2 Role and Attribute Expressions 30

2.2.3 Introduction Axioms 31

2.2.4 General Terminological Axioms 33

2.2.5 Functional and Transitive Role Axioms 34

2.3 Subsumption and Classification 35

2.3.1 Subsumption in General Terminologies 36

2.3.2 Subsumption in Unfoldable Terminologies 37
2.4 Theoretical and Implemented DLs 39
2.4.1 The $\mathcal{ALC}$ Family of DLs 39
2.4.2 Implemented DLKRSs 41

3 Tableaux Algorithms 45
3.1 Tableaux Subsumption Testing 45
3.2 General Method 47
3.2.1 A Tableaux Algorithm for $\mathcal{ALC}$ 47
3.3 Dealing with General Terminologies 52
3.3.1 Meta Constraints 52
3.3.2 Blocking 54
3.3.3 Semi-unfoldable Terminologies 56

4 The $\mathcal{ALCH}_{R^+}$ Description Logic 57
4.1 Transitive Extensions to $\mathcal{ALC}$ 58
4.2 A Tableaux Algorithm for $\mathcal{ALCH}_{R^+}$ 60
4.2.1 Constructing an $\mathcal{ALCH}_{R^+}$ Tableau 63
4.2.2 Soundness and Completeness 65
4.2.3 Worked Examples 67
4.3 Internalising GCIs 71
4.4 $\mathcal{ALCH}_{R^+}$ Extended with Attributes 71

5 Optimising Tableaux Algorithms 77
5.1 Reducing Storage Requirements 78
5.2 Lazy Unfolding 79
5.3 Normalisation and Encoding 80
5.4 GCI Absorption 83
5.5 Semantic Branching Search 85
5.5.1 Boolean Constraint Propagation 86
5.5.2 Heuristic Guided Search 87
5.5.3 The Optimised Search Algorithm 88
5.6 Dependency Directed Backtracking 89
5.7 Caching 91

5.7.1 Using Caching in Sub-problems 94
5.8 Interactions Between Optimisations 95

6 The FaCT Classifier 96

6.1 FaCT Overview 96
6.1.1 Concept Description Syntax 97
6.1.2 Knowledge Base Syntax 97
6.2 Pre-processing Terminological Axioms 98
6.2.1 Terminological Cycles 99
6.2.2 Pre-processing Roles and Attributes 100
6.2.3 Absorbing GCIs 101
6.2.4 Normalisation and Encoding 101
6.3 Classifying the Knowledge Base 102
6.4 Configuring FaCT 104
6.4.1 Configuring Optimisations 104
6.4.2 Configuring Reasoning Power 104

7 Test Methodology and Data 105

7.1 The GALEN Ontology 106
7.1.1 Translating GRAIL into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ 106
7.1.2 Test Knowledge Bases 109
7.2 Testing Methodology 111
7.2.1 Percentile Plots 111
7.2.2 Data Gathering 112
7.2.3 System Specification 112
7.3 Classifying the GALEN Terminology 113
7.3.1 Testing Optimisation Techniques 115
7.4 Comparing FaCT and KRIS 121
7.5 Solving Satisfiability Problems 124
7.5.1 Using the Giunchiglia and Sebastiani Generator 126
7.5.2 Using the Hustadt and Schmidt Generator 130
7.6 Testing for Correctness 133
7.6.1 Hard Problems 134

7.6.2 Classification . . . 134
7.6.3 Satisfiability Testing . . . 134
7.7 Summary . . . 135
7.7.1 GCIs and Absorption . . . 135
7.7.2 Backjumping and MOMS Heuristic . . . 136
7.7.3 Other Optimisations . . . 136

8 Discussion **137**
8.1 Thesis Overview . . . 137
8.2 Significance of Major Results . . . 137
8.2.1 Satisfiability Testing Algorithms . . . 138
8.2.2 Optimisation Techniques . . . 138
8.2.3 Empirical Evaluation . . . 139
8.2.4 The FaCT System . . . 140
8.3 Outstanding Issues . . . 141
8.3.1 The Spectre of Intractability . . . 141
8.3.2 GALEN and GRAIL . . . 142
8.3.3 Tools and Environments . . . 142
8.4 Future Work . . . 143

A FaCT Reference Manual **145**
A.1 Introduction . . . 145
A.1.1 Obtaining FaCT . . . 145
A.2 Concept Descriptions . . . 146
A.3 Function and Macro Interface . . . 147
A.3.1 Built-in Concepts . . . 147
A.3.2 Knowledge Base Management . . . 147
A.3.3 TBox Definitions . . . 148
A.3.4 TBox Inferences . . . 153
A.3.5 TBox Queries . . . 159
A.4 Controlling FaCT's Behavior . . . 163
A.5 Modal Logic Theorem Proving . . . 167

Bibliography **168**

List of Tables

2.1 The concept vegan in various DLKRSs 24
2.2 A Generalised DL Syntax in Backus-Naur form 27
2.3 Concept expressions 28
2.4 Concept expressions and equivalent FOPC formulae 29
2.5 Role and attribute expressions 30
2.6 Role expressions and equivalent FOPC formulae 30
2.7 Introduction axioms 32
2.8 Introduction axioms and equivalent FOPC axioms 32
2.9 General terminological axioms 34
2.10 Concept expressions supported by DLs 43
2.11 Role expressions supported by DLs 44

3.1 Tableaux expansion rules for $\mathcal{ALC}$ 48
3.2 Modified $\exists$-rule with meta constraint 55

4.1 Semantics of $\mathcal{ALCH}_{R^+}$ concept expressions 61
4.2 Tableaux expansion rules for $\mathcal{ALCH}_{R^+}$ 65
4.3 Semantics of $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ concept expressions 73
4.4 Extended $\exists$-rule for $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ 74

5.1 Normalisation and encoding 81

6.1 FaCT concept expressions 97
6.2 FaCT axiomatic macros 98

7.1 Grail concept statements and equivalent $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ 106
7.2 Grail role axioms and equivalent $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ 109
7.3 Test TKB characteristics 110

7.4 Total classification times (**TKB0**/FaCT) 113
7.5 Total classification times without encoding (**TKB0**/FaCT) 116
7.6 Total classification times without caching (**TKB0**/FaCT) 118
7.7 Total classification times with MOMS heuristic (**TKB0**/FaCT) . 120
7.8 Total classification times (**TKB1**) 122
7.9 Total classification times (**TKB2**) 124
7.10 The correspondence between modal $\mathbf{K_{(m)}}$ and $\mathcal{ALC}$ 125
7.11 Parameter settings **PS0**–**PS11** 127
7.12 Parameter settings **PS12** and **PS13** 131

A.1 FaCT concept expressions 146

List of Figures

2.1 A hierarchy of DLs from the $\mathcal{ALC}$ family 41

4.1 A fraction of the GALEN role hierarchy 59
4.2 The role hierarchy represented by $\mathcal{T}$ 67
4.3 Expanded tree **T** for $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$ 68
4.4 Expanded tree **T** and model for $\exists R.C \sqcap \forall R.(\exists R.C)$ 69
4.5 Expanded tree **T** and model for $\exists R.(C \sqcup D) \sqcap \forall R.\neg C$ 70
4.6 Expanded tree **T** and models for $\exists A.(C \sqcup D)$ 75
4.7 The role hierarchy represented by $\mathcal{T}$ 75
4.8 Expanded tree **T** for $\exists B_1.C \sqcap \exists B_2.\neg C$ 76

5.1 Syntactic branching 86
5.2 Semantic branching with BCP 88
5.3 Thrashing in backtracking search 90
5.4 Pruning the search using backjumping 91
5.5 Merging models of C and $\neg D$ 92

7.1 Classification time per concept -v- TKB size (**TKB0**/FaCT) . . . 113
7.2 Classification with and without lexical encoding (**TKB0**/FaCT) . 117
7.3 Search space with and without lexical encoding (**TKB0**/FaCT) . 117
7.4 Classification with and without caching (**TKB0**/FaCT) 119
7.5 Satisfiability tests with and without caching (**TKB0**/FaCT) . . . 119
7.6 Classification with and without MOMS heuristic (**TKB0**/FaCT) . 120
7.7 Search space with and without MOMS heuristic (**TKB0**/FaCT) . 121
7.8 Classification time per concept -v- classified TKB size (**TKB1**) . 123
7.9 Classification time per concept -v- classified TKB size (**TKB2**) . 124
7.10 Classification without optimisations (**TKB2**) 125

7.11 Median solution times — varying N 128
7.12 Median solution times — varying M 128
7.13 Median solution times — varying D 129
7.14 Percentages of satisfiable problems generated 129
7.15 Percentile solution times — **PS10**/FaCT 130
7.16 Percentile solution times — **PS12** 132
7.17 Percentile solution times — **PS13** 132
7.18 Search space — **PS12**/FaCT 133

List of examples

2.1 Standard Infix Notation 24
2.2 Correspondence With FOPC 24
2.3 A Primitive Introduction Axiom 33
2.4 A Non-primitive Introduction Axiom 33
2.5 A General Concept Inclusion Axiom 33
2.6 Unfolding A Concept Expression 37

3.1 Negation Normal Form 47
3.2 Demonstrating Subsumption 49
3.3 Demonstrating Non-subsumption 50
3.4 Converting a GCI Into a Meta Constraint 53
3.5 Non Terminating Tableaux Expansion 54
3.6 Termination Resulting From Blocking 55

4.1 Complex Role Interactions 67
4.2 Blocking 68
4.3 Disjunctions 70
4.4 Attribute Interactions 75

5.1 Contradiction Detection Due To Lazy Unfolding 79
5.2 Detecting Unsatisfiability Via Normalisation and Encoding 80
5.3 Absorbing a GCI 83
5.4 Wasted Search Due To Syntactic Branching 85
5.5 Boolean Constraint Propagation 87
5.6 Semantic Branching Search 87
5.7 Thrashing 89
5.8 Merging Models 92

6.1 Unsatisfiability Due To Cyclical Non-primitive Axioms 99
6.2 Eliminating Cyclical Non-primitive Axioms 100
6.3 Primitive Subsumption Caused By GCIs 103

Abstract

Description Logics form a family of formalisms closely related to semantic networks but with the distinguishing characteristic that the semantics of the concept description language is formally defined, so that the subsumption relationship between two concept descriptions can be computed by a suitable algorithm. Description Logics have proved useful in a range of applications but their wider acceptance has been hindered by their limited expressiveness and the intractability of their subsumption algorithms.

This thesis investigates the practicability of providing sound, complete and empirically tractable subsumption reasoning for a Description Logic with an expressive concept description language. It suggests that, while subsumption reasoning in such languages is known to be intractable in the worst case, a suitably optimised algorithm can provide acceptable performance with a realistic knowledge base. This claim is supported by the implementation and testing of the FaCT system.

FaCT is a Description Logic classifier for an expressive concept description language which includes support for both transitive roles and a role hierarchy. A tableaux calculus style algorithm for subsumption reasoning in this language is presented along with a proof of its soundness and completeness. The wide range of novel and adapted optimisation techniques employed by FaCT is also described and their effectiveness is evaluated by extensive empirical testing using both a large realistic knowledge base (from the Galen project) and randomly generated satisfiability problems. These tests demonstrate that the optimisation techniques improve FaCT's performance by at least three orders of magnitude, and that as a result, FaCT provides acceptable performance when used with the Galen knowledge base.

The work presented in this thesis should be of value to both users of Description Logics, to whom the FaCT system has been made available, and to implementors of Description Logic systems, who will be able to incorporate some or all of the optimisation techniques in their algorithms. The optimisation techniques may also be of interest to a wider audience of Automated Deduction and Artificial Intelligence researchers.

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.

Acknowledgements

I gratefully acknowledge the help and support of my advisors, both past and present, Alan Rector, Graham Gough, Richard Banach and Carole Goble. Without their assistance the work embodied in this thesis would not have been possible.

Thanks are also due to the members of the Medical Informatics and Artificial Intelligence groups, in particular Sean Bechhofer, Ian Pratt and Dominik Schoop.

Finally, I would like to thank all the members of the wider Description Logic community who have provided inspiration, advice and encouragement.

This work was supported by a grant from the Engineering and Physical Sciences Research Council.

The Author

Ian Horrocks obtained a B.Sc. in Computer Science from Manchester University in 1981. After graduation he worked as a Research Assistant in the Computer Science Department, first in the Barclays microprocessor laboratory and later with the dataflow parallel architecture group.

In 1983 he joined Scientex Limited as Technical Director and was responsible for the development of a range of word processing and desk-top publishing products.

In 1994 he returned to Manchester University as a research student in the Medical Informatics Group. He obtained an M.Sc. in 1995 with a thesis entitled *A Comparison of Two Terminological Knowledge Representation Systems*.

His research interests are knowledge representation, automated reasoning and, in particular, description logics. He has a number of publications to his credit and is a member of the organising committee for the 1997 International Workshop on Description Logics.

Chapter 1

Introduction

Description Logics form a family of formalisms which have grown out of knowledge representation techniques using frames and semantic networks; their distinguishing characteristic is a formally defined semantics which enables the subsumption (kind-of) relationship to be computed by a suitable algorithm. Description Logic based knowledge representation systems have proved useful in a range of applications, but their wider acceptance has been hindered by their limited expressiveness and the intractability of their subsumption algorithms.

This thesis investigates the practicability of providing a sound, complete and empirically tractable subsumption algorithm for an expressive Description Logic. It suggests that, while subsumption testing in such Description Logics is known to be intractable in the worst case, a suitably optimised algorithm can provide acceptable performance with a realistic knowledge base.

This claim has been strongly supported by:

- the design of a sound and complete subsumption testing algorithm for an expressive Description Logic;
- the development of a range of optimisation techniques which dramatically improve the performance of the algorithm;
- the evaluation of the optimised algorithm through extensive empirical testing, including detailed comparisons with existing systems.

1.1 Description Logics

Many computer applications require a symbolic model of the application world which can be updated or queried. A common methodology is to describe objects, their relationships and classes of objects with similar characteristics. A class hierarchy (partial ordering) is often maintained based on the sub-class/super-class relationship. This methodology can be seen in formalisms such as object oriented databases, semantic networks and frame systems.

Description Logics (DLs) form a family of formalisms closely related to both semantic networks and frame systems but with the distinguishing characteristic that the semantics of the concept (class) description language is formally defined, so that the resulting structured objects can be reasoned with [WS92]. In particular the subsumption (sub-class/super-class) relationship between two concept descriptions can be computed by a suitable algorithm [Neb90a].

The use of DLs in knowledge representation systems originated with Kl-One [BS85] and has led to the development of a wide range of related systems including Nikl [Mos83], Krypton [BFL83], Kris [BH91c], Sb-One [Kob91], CycL [LG91], loom [Mac91b], K-Rep [MDW91], Classic [PS91], Back [Pel91], Candide [NSA+94] and Grail [RBG+97]. An excellent historical account and overview can be found in [WS92].

DL Knowledge Representation Systems (DLKRSs) support the logical description of concepts and roles (relationships) and their combination, using a variety of operators, to form more complex descriptions. For example, if person is an existing concept and *has-child* is an existing role, then the concept parent might be described as a kind of person who is related to another person via the *has-child* role. DLKRSs may also allow concept and role descriptions to be used in assertions of facts about individuals. For example, it might be asserted that John and Jane are instances of person and that the relationship between John and Jane is an instance of *has-child*.

A typical DLKRS will provide a range of reasoning services, in particular inferring subsumption and instantiation (instance-of) relationships. For example, it might be inferred that person subsumes parent and that John is an instance of (instantiates) parent. Most DLKRSs have adopted a hybrid architecture, pioneered by the Krypton system, in which the reasoning services are divided into a terminological component (the TBox) and an assertional component (the ABox) [BGL85]. The TBox, or term classifier, reasons about concept and role descriptions: the basic reasoning service provided is subsumption, which the TBox uses to maintain a concept hierarchy—a partial ordering of concepts based on the subsumption relation. The ABox reasons about individuals and their relationships: the basic reasoning service provided is instantiation, which the ABox can use to perform *realisation* and *retrieval*—computing the most specific concepts which an individual instantiates (realisation) and the individuals which are instances of a given concept (retrieval)[1].

The expressiveness of the concept description language is often much more restricted than that of the assertional language, with the objective of providing subsumption reasoning which is *sound* (only valid subsumption relationships

[1] In practice, ABox reasoning may only be performed in response to queries [MB92].

are identified), *complete* (all valid subsumption relationships are identified) and *tractable* (subsumption relationships can be computed in a “reasonable” time). However, in spite of their restricted expressiveness, most implemented systems do not provide complete subsumption reasoning and have been shown to perform poorly when reasoning with large numbers of concepts [HKNP94, SvRvdVM95].

This thesis addresses the problem of providing a sound, complete and empirically tractable subsumption reasoning algorithm for a DL with an expressive concept description language, and of testing the algorithm with a realistic knowledge base.

1.2 DL Applications

DLs are typically used to provide high level terminologies (sometimes called conceptual schemas or ontologies), to perform semantic indexing and to mediate between other representations. DLs have proved useful in a variety of applications including:

- schema design, query optimisation and reasoning about views in object oriented databases [BBLS94, BJNS94, Bre95, LR96];
- federated databases and cooperative information systems [BIG94, CL94, GBBI96];
- database interoperability [GL94];
- software information systems [DBBS91];
- describing and classifying multimedia data [GHB96];
- the development of a very large knowledge base representing “human consensus knowledge” (the Cyc project) [GL90, GL94];
- process planning and control in manufacturing [N.R96];
- the configuration of complex telecommunications systems [BIW94];
- reasoning about actions and plan generation in robotics [GINR96];

1.2.1 GALEN and GRAIL

A particularly promising application domain for DLs is in the growth area of ontological engineering—the design, construction and maintenance of large conceptual schemas or ontologies [MWD+96, HRG96, RH97]. An example of such an application is the European Galen project. The Galen project aims to promote the sharing and re-use of medical data by building a large medical terminology ontology which can be used by application designers as a flexible and extensible classification schema [RNG93]. By using a DL to build the ontology it is hoped to avoid many of the problems associated with existing hand-crafted coding schemes [Now93], as well as providing additional benefits to applications:

- more detailed descriptions with clear semantics can be constructed systematically to provide principled extensions to the schema where required;
- the DL classifier can be used to check the coherence of new descriptions and to enrich the schema by the discovery of implicit subsumption relationships;
- the DL can be used as a powerful database query language supporting intensional as well as extensional queries [Bre95] and query optimisation [BBLS94];
- data can be shared between existing applications by using the concept model as an interlingua and providing mappings to a variety of coding schemes.

However, the usefulness of existing DL systems in this and other applications is limited by the restricted expressiveness of the TBox [DP91, Mac91a, SvRvdVM95]. Design studies for the GALEN project identified expressive requirements which were satisfied by few if any implemented systems [Now93]:

- The ability to reason about transitive part-whole, causal and compositional relations. A similar requirement has been identified in other application domains, particularly those concerned with reasoning about physically composed objects, such as engineering [Hor94, PL94, Sat95].
- The ability to reason with a more general form of axiom, called *general concept inclusions* (see Section 2.2.4 on page 33), which allow subsumption relations to be asserted between arbitrary pairs of concept descriptions. This requirement has also been identified in a number of other applications [DP91, LR96].

To satisfy these requirements a new DL system, GRAIL, was designed and implemented for the GALEN project [GBS+94]. GRAIL supports reasoning about transitive relations and general concept inclusions (GCIs) but, like most other DL systems which use *structural subsumption algorithms*, GRAIL's subsumption reasoning is known to be incomplete [Hor95].

1.3 Subsumption Reasoning in DLs

Most implemented DL systems have used structural algorithms for computing subsumption relationships. These algorithms determine subsumption between two concept descriptions by using recursive structural decomposition; e.g., it would be determined that the concept "doctors all of whose *children* are doctors" subsumes the concept "pediatricians, all of whose *daughters* are pediatricians" because the concept doctors subsumes the concept pediatricians and the role *children* subsumes the role *daughters*.

The advantage with these algorithms is that by pre-processing concept descriptions and storing them in a normalised form, subsequent subsumption tests, of which many will be required to maintain the concept hierarchy, can be performed relatively efficiently [BPS94]; the disadvantages are that the structural comparisons required to deal with more expressive description languages become extremely complex, and while it is relatively easy to demonstrate that the structural comparisons are sound, they are almost invariably incomplete[2]. Of the systems mentioned in Section 1.1, all but Kris use a structural algorithm; of these only Classic claims to provide sound and complete subsumption reasoning [BPS94], and that only for a very restricted concept description language.

An alternative approach is to transpose the subsumption problem into an equivalent satisfiability problem: concept C subsumes concept D if and only if the concept description "D and not C" is not satisfiable. The satisfiability problem can then be solved using a provably sound and complete algorithm based on the tableaux calculus [Smu68]. This approach was first described for the $\mathcal{ALC}$ Description Logic [SSS91] and its practical application has been demonstrated in the Kris system; it has many advantages (see Chapter 3) and has led to the development of sound and complete algorithms for a wide range of DLs, including those which support the transitive closure of roles [Baa90a] and GCIs [BDS93].

Use of the satisfiability testing approach is restricted to DLs which support general negation, as this is required for the transformation from subsumption test to satisfiability test. Unfortunately, the worst case complexity of the satisfiability problem for these DLs is at least NP-hard [BFT95], and for more expressive DLs may be Exptime-complete [Sch91, Cal96]. These complexity results have led to the conjecture that expressive DLs might be of limited practical applicability [BDS93]. However, although the theoretical complexity results are discouraging, empirical analyses of real applications have shown that the kinds of construct which lead to worst case intractability rarely occur in practice [Neb90b, HKNP94, SvRvdVM95, Hor95]. Moreover, it has been demonstrated that for a relatively expressive DL (Kris), applying some simple optimisation techniques to a tableaux algorithm can result in performance which is comparable with that of incomplete structural algorithms [BHNP92]. There are many other optimisation techniques which could be applied to tableaux algorithms, and it therefore seems worthwhile to investigate empirically the practicability of using a highly optimised tableaux algorithm for subsumption testing in a more expressive DL, in particular one which supports transitive roles and GCIs.

[2]Although some structural algorithms may be complete with respect to a weaker four-valued semantics [PS89].

1.4 $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ and the FaCT System

The logic chosen for this investigation was $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, an extension of $\mathcal{ALC}$ (see Section 2.1.1 on page 25) which supports functional roles, transitively closed roles and a role hierarchy. The reasons for choosing this logic were:

- it provides the kind of expressive possibilities required by many applications, in particular the GALEN project;
- the satisfiability testing algorithm required only a relatively minor extension to an existing algorithm for the $\mathcal{ALC}_{R\oplus}$ DL [Sat96];
- the satisfiability testing algorithm is relatively simple, facilitating experimentation with a range of optimisation techniques.

The FaCT system is a terminological classifier (TBox), based on the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ Description Logic, which has been developed for this thesis and used as a testbed for a highly optimised implementation of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ subsumption testing algorithm.

Optimisation techniques investigated in the FaCT system include the normalisation and encoding of concept descriptions, eliminating GCIs by absorbing them into concept descriptions, an improved search algorithm adapted from the Davis-Putnam-Logemann-Loveland procedure, heuristic guided search, constraint propagation, dependency directed backtracking and the caching and re-use of partial results. Many of these techniques are novel or have not previously been applied in a DL subsumption algorithm.

1.5 Thesis Outline

The remainder of the thesis is organised as follows:

Chapter 2 Presents the standard DL syntax and Tarski style model theoretic semantics which is used to interpret both concept descriptions and subsumption relationships.

Chapter 3 Explains the basic principles of tableaux subsumption algorithms and how they can be extended to deal with more expressive DLs.

Chapter 4 Introduces the $\mathcal{ALCH}_{R^+}$ DL, describes an algorithm for deciding the satisfiability of $\mathcal{ALCH}_{R^+}$ concept expressions and presents a proof of its soundness and completeness. $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, an extension to $\mathcal{ALCH}_{R^+}$ which supports reasoning about functional relations, is also described.

Chapter 5 Describes a range of optimisation techniques which are designed to improve the empirical performance of tableaux algorithms in general and of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm in particular.

Chapter 6 Describes the FaCT system, explains how FaCT uses the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm to compute the concept hierarchy, and describes additional optimisation techniques which are applicable to this process.

Chapter 7 Explains the empirical testing procedures used to evaluate FaCT's performance, describes how the test data was generated, and presents the results of the evaluation.

Chapter 8 Reviews the work presented and the extent to which the stated objectives have been met. The significance of the major results is summarised, outstanding issues are discussed and directions for future work are suggested.

Chapter 2

Formal Foundations of Description Logics

A key characteristic of Description Logics (DLs) is that they support the composition of structured concept descriptions, with which they can then reason [WS92]. Reasoning is made possible by the rigorous formal specification of the syntax and semantics of their concept description languages. This chapter presents a standard syntax and semantics for concept description languages which is widely used in the DL literature and which will be used in the remainder of this thesis. A number of useful theoretical results and equivalences will also be presented.

The chapter is organised as follows: Section 2.1 describes the syntax of DLs using a standard infix notation; Section 2.2 describes how DLs are interpreted using the standard Tarski style model theoretic semantics and also shows how DLs can be interpreted using the First Order Predicate Calculus; Section 2.3 shows how the semantics are used to define the relation of *subsumption* and how this definition can be simplified by restricting the syntax of a DL; and finally, Section 2.4 describes some specific DLs, both theoretical and implemented, with particular emphasis on the $\mathcal{ALC}$ family of DLs.

2.1 Syntax

DLs support the logical description of concepts, roles (relationships) and attributes (single-valued or functional roles). Concepts, roles and attributes can be combined, using a variety of operators, to form more complex expressions which can be used in terminological axioms to add information to a knowledge base [WS92]. The operators supported by DLs usually include some or all of the standard logical connectives along with one or both of the universally and existentially quantified relational operators (called value restrictions and exists restrictions).

Various forms of syntax have been used in implemented Description Logic Knowledge Representation Systems (DLKRSs), often reflecting the programming language used in the implementation; Table 2.1 shows the syntax of an axiom which introduces the concept vegan, a person who only eats plants, as it would appear in several implemented DLKRSs. Classic, Kris and Loom have a similar list based syntax, which is convenient for their Lisp implementations, whereas Back uses a syntax which is more convenient for its Prolog implementation.

DLKRS	Concept introduction axiom
Back	vegan := person and all(*eats* plant)
Classic	(cl-define-concept 'vegan '(AND person (ALL *eats* plant)))
Kris	(defconcept vegan (AND person (ALL *eats* plant)))
Loom	(defconcept vegan :is (:AND person (:ALL *eats* plant)))

Table 2.1: The concept vegan in various DLKRSs

A standard infix notation, commonly known as German syntax due to the nationality of its originators, has been widely adopted for the theoretical discussion of DLs [BHH+91]. This notation uses the symbols $\sqcap$ and $\sqcup$ for conjunction and disjunction operators, reflecting their model theoretic interpretation as set intersection and union (see Section 2.2.1), the standard logical quantifier symbols $\forall$ and $\exists$ for value and exists restrictions and the $\neg$ symbol for complementation. A variety of other symbols may also by used to represent additional operators, the most commonly used of which will be described in the following sections. The relation symbols $\doteq$ and $\sqsubseteq$ are used in axioms and reflect their model theoretic interpretations as set equality and set inclusion (see Section 2.2.3).

Example 2.1 Standard Infix Notation

Using the standard infix notation, the vegan concept introduction axiom would be written as:

$$\mathsf{vegan} \doteq \mathsf{person} \sqcap \forall \mathit{eats}.\mathsf{plant}$$

For readers more familiar with First Order Predicate Calculus (FOPC) than DLs, it might be useful to note that concepts correspond to unary predicates in FOPC and that roles correspond to binary predicates [DLNN95].

Example 2.2 Correspondence With FOPC

The vegan concept introduction from Example 2.1 could be written in FOPC as:

$$\forall x.(\mathsf{vegan}(x) \leftrightarrow \mathsf{person}(x) \wedge \forall y.(\mathit{eats}(x,y) \rightarrow \mathsf{plant}(y)))$$

The correspondence between DLs and FOPC will be described in detail in the following sections.

2.1.1 Formal Syntax

A DL *terminology* (or terminological knowledge base) consists of a finite set of *axioms* which can introduce new concept and role names, assert subsumption relationships and assert that roles are either functional or transitive. Concept and role names are treated as symbols in the logic.

A DL is characterised by the kinds of concept and role expression allowed in its description language and the kinds of axiom allowed in its terminologies. Of particular interest is the $\mathcal{ALC}$ DL [SSS91] as its properties, and those of its family of extensions, have been the subject of detailed study (see Section 2.4.1 on page 39). An $\mathcal{ALC}$ terminology is defined by the following formation rules:

- Axioms are of the form:

  $$C \sqsubseteq D \mid C \doteq D$$

  where C and D are concept expressions.

- Concept expressions are of the form:

  $$\mathsf{CN} \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

  where CN is a concept name, C and D are concept expressions and R is a role expression.

- Role expressions are of the form RN, where RN is a role name.

The special concept names $\top$ (*top*) and $\bot$ (*bottom*) represent the most general and least general concepts respectively: their formal semantics will be given in Section 2.2 on the following page. Nested conjunctive expressions $(C_1 \sqcap (C_2 \sqcap \ldots \sqcap (C_{n-1} \sqcap C_n) \ldots))$ and disjunctive expressions $(C_1 \sqcup (C_2 \sqcup \ldots \sqcup (C_{n-1} \sqcup C_n) \ldots))$ will often be written as $(C_1 \sqcap \ldots \sqcap C_n)$ and $(C_1 \sqcup \ldots \sqcup C_n)$ respectively.

The set of concept and role names which appear in a terminology will be denoted $\mathbf{C}$ and $\mathbf{R}$ respectively. There is no requirement for $\mathbf{C}$ and $\mathbf{R}$ to be disjoint: whether a name refers to a concept or a role can be determined by its context.

More expressive DLs can extend this syntax in a number of ways:

1. In DLs which support attributes or transitive roles, the sets of attribute (functional role) names $\mathbf{F} \subseteq \mathbf{R}$ and transitive role names $\mathbf{R}_+ \subseteq \mathbf{R}$ are also distinguished. Terminologies of such a DL may contain axioms which assert that a role name is a member of one of these sets. These axioms are of the form:

   $$\mathsf{AN} \in \mathbf{F} \mid \mathsf{RN} \in \mathbf{R}_+$$

   where AN is an attribute name and RN is a role name.

2. A wider range of concept expressions may be supported. The most common of these are of the form:

$$\geqslant nR \mid \leqslant nR \mid \geqslant nR.C \mid \leqslant nR.C \mid \exists A.C \mid \forall A.C \mid A = B \mid A \neq B$$

   where n is a nonnegative integer, C is a concept expression, R is a role expression and A and B are attribute expressions.

3. A range of role expressions may be supported. The most common of these are of the form:

$$\begin{gathered} \top \times \top \mid R \sqcap S \mid R \sqcup S \mid R \circ S \mid id(C) \mid R^{-1} \mid \\ R^{+} \mid R^{*} \mid A \sqcup B \mid A^{-1} \mid A^{+} \mid A^{*} \end{gathered}$$

   where R and S are role expressions and A and B are attribute expressions. The expression $\top \times \top$ represents the most general role: its formal semantics will be given in Section 2.2. The reason for $A \sqcup B$, A^{-1}, A^{+} and A^{*} being role expressions and not attribute expressions will also be explained in Section 2.2.

4. A range of attribute expressions may be supported. The most common of these are of the form:

$$A \sqcap B \mid A \circ B$$

   where A and B are attribute expressions.

5. The terminology may include axioms of the form:

$$\mathsf{RN} \sqsubseteq R \mid \mathsf{RN} \doteq R \mid \mathsf{AN} \sqsubseteq A \mid \mathsf{AN} \doteq A \mid \mathsf{AN} \sqsubseteq R$$

   where RN is a role name, AN is an attribute name, R is a role expressions and A is an attribute expressions.

The syntax of a generalised DL supporting all the above axioms and expressions is presented in Backus-Naur form (BNF) in Table 2.2 on the following page.

2.2 Model Theoretic Semantics

A Tarski style model theoretic semantics [Tar56] is used to interpret expressions and to justify subsumption inferences. Concepts, roles and attributes are taken to refer to sets of objects in the domain of interest and the relationships between them. A terminology consists of a finite set of axioms which introduce new

$\langle terminology \rangle$	$::=$	$\{\langle axiom \rangle^*\}$
$\langle axiom \rangle$	$::=$	$\langle C \rangle \sqsubseteq \langle C \rangle \mid \langle C \rangle \doteq \langle C \rangle \mid$
		$\mathit{RN} \sqsubseteq \langle R \rangle \mid \mathit{RN} \doteq \langle R \rangle \mid$
		$\mathit{AN} \sqsubseteq \langle A \rangle \mid \mathit{AN} \doteq \langle A \rangle \mid \mathit{AN} \sqsubseteq \langle R \rangle \mid$
		$\mathit{AN} \in \mathbf{F} \mid \mathit{RN} \in \mathbf{R}_+$
$\langle C \rangle$	$::=$	$\mathsf{CN} \mid \top \mid \bot \mid \neg\langle C \rangle \mid$
		$\langle C \rangle \sqcap \langle C \rangle \mid \langle C \rangle \sqcup \langle C \rangle \mid \exists\langle R \rangle.\langle C \rangle \mid \forall\langle R \rangle.\langle C \rangle \mid$
		$\geqslant\langle n \rangle\langle R \rangle \mid \leqslant\langle n \rangle\langle R \rangle \mid \geqslant\langle n \rangle\langle R \rangle.\langle C \rangle \mid \leqslant\langle n \rangle\langle R \rangle.\langle C \rangle \mid$
		$\exists\langle A \rangle.\langle C \rangle \mid \forall\langle A \rangle.\langle C \rangle \mid \langle A \rangle = \langle A \rangle \mid \langle A \rangle \neq \langle A \rangle$
$\langle R \rangle$	$::=$	$\mathit{RN} \mid \langle R \rangle \sqcap \langle R \rangle \mid \langle R \rangle \sqcup \langle R \rangle \mid \langle R \rangle \circ \langle R \rangle \mid$
		$id(\langle C \rangle) \mid \langle R \rangle^{-1} \mid \langle R \rangle^{+} \mid \langle R \rangle^{*} \mid$
		$\langle A \rangle \sqcup \langle A \rangle \mid \langle A \rangle^{-1} \mid \langle A \rangle^{+} \mid \langle A \rangle^{*}$
$\langle A \rangle$	$::=$	$\mathit{AN} \mid \langle A \rangle \sqcap \langle A \rangle \mid \langle A \rangle \circ \langle A \rangle$
CN, RN and AN are concept, role and attribute names respectively		
n is a nonnegative integer		

Table 2.2: A Generalised DL Syntax in Backus-Naur form

concept, role and attribute names and assert subsumption relationships. In the following discussion CN will be used to denote a concept name, RN a role name, AN an attribute name C and D concept expressions, R and S role expressions and A and B attribute expressions.

The meaning of concepts, roles and attributes is given by an interpretation $\mathcal{I}$ which is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the domain (a set) and $\cdot^{\mathcal{I}}$ is an interpretation function [BHH+91]. The interpretation function maps each concept name CN to a subset of the domain:

$$\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$$

each role name RN to a set valued function (or equivalently a binary relation):

$$\mathit{RN}^{\mathcal{I}} : \Delta^{\mathcal{I}} \longrightarrow 2^{\Delta^{\mathcal{I}}} \qquad (\mathit{RN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}})$$

and each attribute name AN to a single valued partial function:

$$\mathit{AN}^{\mathcal{I}} : dom\ \mathit{AN}^{\mathcal{I}} \longrightarrow \Delta^{\mathcal{I}}$$

where $dom\ \mathit{AN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$.

It may sometimes also be convenient to treat the interpretation of an attribute as a binary relation:

$$\mathit{AN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$$

satisfying the additional condition:

if $\langle d, e \rangle \in \mathit{AN}^{\mathcal{I}}$, then there is no $f \in \Delta^{\mathcal{I}}$ such that $\langle d, f \rangle \in \mathit{AN}^{\mathcal{I}}$ and $e \neq f$

The interpretation of a concept, role or attribute expression is derived from the interpretations of its components as described in the following sections.

2.2.1 Concept Expressions

DLs can support a wide variety of operators for building and combining concept expressions. Table 2.3 presents the formal semantics of the concept expressions introduced in Section 2.1.1.

Description	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$\Delta^{\mathcal{I}} - C^{\mathcal{I}}$
exists restriction	$\exists R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$
value restriction	$\forall R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$
number restriction	$\geqslant nR$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d)\rvert \geqslant n\}$
number restriction	$\leqslant nR$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d)\rvert \leqslant n\}$
qualified number restriction	$\geqslant nR.C$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\rvert \geqslant n\}$
qualified number restriction	$\leqslant nR.C$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\rvert \leqslant n\}$
exists restriction	$\exists A.C$	$\{d \in dom\ A^{\mathcal{I}} \mid A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$
value restriction	$\forall A.C$	$\{d \in \Delta^{\mathcal{I}} \mid d \in dom\ A^{\mathcal{I}} \Rightarrow A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$
attribute value map	$A = B$	$\{d \in \Delta^{\mathcal{I}} \mid A^{\mathcal{I}}(d) = B^{\mathcal{I}}(d)\}$
attribute value map	$A \neq B$	$\{d \in \Delta^{\mathcal{I}} \mid A^{\mathcal{I}}(d) \neq B^{\mathcal{I}}(d)\}$

Table 2.3: Concept expressions

It has already been mentioned (in Section 2.1 on page 23) that concepts correspond to unary predicates in FOPC. A concept expression therefore corresponds to an FOPC formula with a single free variable [DLNN95]: Table 2.4 on the following page shows how the concept expressions described in Table 2.3 correspond to FOPC formulae with x as the free variable.

Note that:

1. There are two possible semantics for $A = B$ and $A \neq B$ (attribute value map) concept expressions. The semantics given in Table 2.3 includes those cases where one or both of $A(d)$ and $B(d)$ are undefined. The alternative semantics, which specifies that both $A(d)$ and $B(d)$ must be defined, can be captured by the concept expressions:

$$(A = B) \sqcap (\exists A.\top) \sqcap (\exists B.\top)$$
$$(A \neq B) \sqcap (\exists A.\top) \sqcap (\exists B.\top)$$

Expression	FOPC formula
$\top$	True
$\bot$	False
$C \sqcap D$	$C(x) \wedge D(x)$
$C \sqcup D$	$C(x) \vee D(x)$
$\neg C$	$\neg C(x)$
$\exists R.C$	$\exists y.(R(x,y) \wedge C(y))$
$\forall R.C$	$\forall y.(R(x,y) \rightarrow C(y))$
$\geqslant nR$	$\exists y_1,\ldots,y_n.\left(\bigwedge_{1\leqslant i\leqslant n}\left(R(x,y_i) \wedge \bigwedge_{i<j\leqslant n} y_i \neq y_j\right)\right)$
$\leqslant nR$	$\forall y_1,\ldots,y_{n+1}.\left(\left(\bigwedge_{1\leqslant i\leqslant n+1} R(x,y_i)\right) \rightarrow \bigvee_{1\leqslant i<j\leqslant n+1} y_i = y_j\right)$
$\geqslant nR.C$	$\exists y_1,\ldots,y_n.\left(\bigwedge_{1\leqslant i\leqslant n}\left(R(x,y_i) \wedge C(y_i) \wedge \bigwedge_{i<j\leqslant n} y_i \neq y_j\right)\right)$
$\leqslant nR.C$	$\forall y_1,\ldots,y_{n+1}.\left(\left(\bigwedge_{1\leqslant i\leqslant n+1} (R(x,y_i) \wedge C(y_i))\right) \rightarrow \bigvee_{1\leqslant i<j\leqslant n+1} y_i = y_j\right)$
$\exists A.C$	$\exists y.((A(x) = y) \wedge C(y))$
$\forall A.C$	$\forall y.((A(x) = y) \rightarrow C(y))$
$A = B$	$\forall y.((A(x) = y) \leftrightarrow (B(x) = y))$
$A \neq B$	$\exists y.\neg((A(x) = y) \leftrightarrow (B(x) = y))$

Table 2.4: Concept expressions and equivalent FOPC formulae

An abbreviated notation $A \overset{\downarrow}{=} B$ and $A \overset{\downarrow}{\neq} B$ has been proposed for these expressions [BHH+91].

2. Allowing roles in value map expressions (e.g., $R = S$ or $R \neq S$) is known to lead to undecidability [SS89].

3. There is considerable redundancy in the set of operators described in Table 2.3, as demonstrated by the following identities:

$$\begin{array}{rcl}
\exists R.C & = & \neg\forall R.\neg C \\
\forall R.C & = & \neg\exists R.\neg C \\
\geqslant nR & = & \begin{cases} \neg\leqslant(n-1)R & \text{if } n \geqslant 1 \\ \top & \text{if } n = 0 \end{cases} \\
\leqslant nR & = & \neg\geqslant(n+1)R \\
\geqslant nR.C & = & \begin{cases} \neg\leqslant(n-1)R.C & \text{if } n \geqslant 1 \\ \top & \text{if } n = 0 \end{cases} \\
\leqslant nR.C & = & \neg\geqslant(n+1)R.C
\end{array} \tag{2.1}$$

4. The following identities relating to the top and bottom concepts will also be useful:

$$\begin{array}{rclcrcl} \top \sqcap C & = & C & \quad & \top \sqcup C & = & \top \\ \bot \sqcap C & = & \bot & & \bot \sqcup C & = & C \\ \exists R.\bot & = & \bot & & \forall R.\top & = & \top \end{array} \tag{2.2}$$

2.2.2 Role and Attribute Expressions

DLs can also support a range of operators for building and combining role and attribute expressions, although this feature is not common in either implementations or theoretical descriptions. Table 2.5 presents the formal semantics of the role and attribute expressions introduced in Section 2.1.1: in this context R and S are taken to refer to either role or attribute expressions.

Description	Syntax	Semantics
top role	$\top \times \top$	$\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
conjunction	$R \sqcap S$	$R^{\mathcal{I}} \cap S^{\mathcal{I}}$
disjunction	$R \sqcup S$	$R^{\mathcal{I}} \cup S^{\mathcal{I}}$
composition	$R \circ S$	$R^{\mathcal{I}} \circ S^{\mathcal{I}}$
identity	$id(C)$	$\{\langle d, d\rangle \mid d \in C^{\mathcal{I}}\}$
inverse	R^{-1}	$\{\langle d, d'\rangle \mid \langle d', d\rangle \in R^{\mathcal{I}}\}$
transitive closure	R^+	$\bigcup_{1 \leqslant n} (R^{\mathcal{I}})^n$
transitive reflexive closure	R^*	$\bigcup_{0 \leqslant n} (R^{\mathcal{I}})^n$

Table 2.5: Role and attribute expressions

A role expression corresponds to an FOPC formula with two free variables [DLNN95]: Table 2.6 shows how some of the role expressions described in Table 2.5 correspond to FOPC formulae with x and y as the free variables.

Expression	FOPC formula
$R \sqcap S$	$R(x, y) \wedge S(x, y)$
$R \sqcup S$	$R(x, y) \vee S(x, y)$
$R \circ S$	$\exists z.(R(x, z) \wedge S(z, y))$
$id(C)$	$C(x) \wedge x = y$
R^{-1}	$R(y, x)$

Table 2.6: Role expressions and equivalent FOPC formulae

Note that:

1. Expressions of the form $A \sqcup B$, A^{-1}, A^+ and A^*, where A and B are attributes, are in general role expressions and not attribute expressions. For example if $A^{\mathcal{I}} = \{\langle d, f\rangle, \langle e, f\rangle, \langle f, g\rangle\}$ and $B^{\mathcal{I}} = \{\langle d, e\rangle\}$, then:

$$\begin{array}{rcl} \{\langle d,e\rangle, \langle d,f\rangle\} & \subset & (A \sqcup B)^{\mathcal{I}} \\ \{\langle f,d\rangle, \langle f,e\rangle\} & \subset & (A^{-1})^{\mathcal{I}} \\ \{\langle d,f\rangle, \langle d,g\rangle\} & \subset & (A^{+})^{\mathcal{I}} \\ \{\langle d,d\rangle, \langle d,f\rangle\} & \subset & (A^{*})^{\mathcal{I}} \end{array}$$

2. Transitive closure and transitive reflexive closure role forming operators cannot be expressed in FOPC [Baa90a] but transitively closed roles [Sat96] can be described, e.g., for a role $\textsf{\textit{R-trans}} \sqsubseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ by adding an axiom:

$$\forall x, y.(\exists z.(\textsf{\textit{R-trans}}(x,z) \wedge \textsf{\textit{R-trans}}(z,y)) \rightarrow \textsf{\textit{R-trans}}(x,y))$$

   to the terminology.

3. Using role and attribute expressions in number restriction concept expressions (e.g., $\geqslant nR^+$) leads to undecidability for most combinations of role forming operators [BS96b].

4. Transitive reflexive closure role expressions in exists and value restriction concept expressions can be transposed into transitive closure role expressions, and vice versa, using the following identities:

$$\begin{array}{rcl} \exists R^+.C & = & \exists R.(\exists R^*.C) \\ \forall R^+.C & = & \forall R.(\forall R^*.C) \\ \exists R^*.C & = & C \sqcup \exists R^+.C \\ \forall R^*.C & = & C \sqcap \forall R^+.C \end{array} \tag{2.3}$$

5. Disjunction, composition and identity role expressions can be eliminated from exists and value restriction concept expressions by applying the following identities:

$$\begin{array}{rcl} \exists(R \sqcup S).C & = & \exists R.C \sqcup \exists S.C \\ \forall(R \sqcup S).C & = & \forall R.C \sqcap \forall S.C \\ \exists(R \circ S).C & = & \exists R.(\exists S.C) \\ \forall(R \circ S).C & = & \forall R.(\forall S.C) \\ \exists id(C).D & = & C \sqcap D \\ \forall id(C).D & = & \neg C \sqcup D \end{array} \tag{2.4}$$

2.2.3 Introduction Axioms

A DL terminology consists of a finite set of axioms. The kinds of axiom which can occur in a terminology are often restricted to those which introduce new concept role and attribute names and associate them with an expression through either an equality or a subsumption relationship: these kinds of axiom will be called *introduction axioms*. Restricting a terminology to introduction axioms which are both *unique* (each name appears only once on the left hand side of an axiom) and *acyclic* (the right hand side of an axiom cannot refer, either directly or indirectly, to the name on its left hand side) greatly simplifies subsumption reasoning (see Section 2.3 on page 35). The semantics of introduction axioms is described in Table 2.7.

Description	Syntax	Semantics
concept introduction	$\mathsf{CN} \doteq C$	$\mathsf{CN}^{\mathcal{I}} = C^{\mathcal{I}}$
role introduction	$\mathit{RN} \doteq R$	$\mathit{RN}^{\mathcal{I}} = R^{\mathcal{I}}$
attribute introduction	$\mathit{AN} \doteq A$	$\mathit{AN}^{\mathcal{I}} = A^{\mathcal{I}}$
primitive concept introduction	$\mathsf{CN} \sqsubseteq C$	$\mathsf{CN}^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
primitive role introduction	$\mathit{RN} \sqsubseteq R$	$\mathit{RN}^{\mathcal{I}} \subseteq R^{\mathcal{I}}$
primitive attribute introduction	$\mathit{AN} \sqsubseteq A$	$\mathit{AN}^{\mathcal{I}} = A^{\mathcal{I}}$

Table 2.7: Introduction axioms

Introduction axioms which use the subsumption relation ($\sqsubseteq$) correspond to implications in FOPC while those which use the equality relation ($\doteq$) correspond to a pair of implications or, equivalently, a bi-directional implication in FOPC. Table 2.8 shows FOPC axioms corresponding to the introduction axioms in Table 2.7.

DL axiom	FOPC axiom
$\mathsf{CN} \doteq C$	$\forall x.(\mathsf{CN}(x) \leftrightarrow C(x))$
$\mathit{RN} \doteq R$	$\forall x, y.(\mathit{RN}(x, y) \leftrightarrow R(x, y))$
$\mathit{AN} \doteq A$	$\forall x, y.(\mathit{AN}(x, y) \leftrightarrow A(x, y))$
$\mathsf{CN} \sqsubseteq C$	$\forall x.(\mathsf{CN}(x) \rightarrow C(x))$
$\mathit{RN} \sqsubseteq R$	$\forall x, y.(\mathit{RN}(x, y) \rightarrow R(x, y))$
$\mathit{AN} \sqsubseteq A$	$\forall x, y.(\mathit{AN}(x, y) \rightarrow A(x, y))$

Table 2.8: Introduction axioms and equivalent FOPC axioms

Names which are associated with an expression via a subsumption relation are known as *primitives* while names which are associated with an expression via an equality relation are known as *non-primitives*. Primitives are not fully defined by their characteristics—they are said to have only necessary characteristics. Non-primitives on the other hand are fully defined by their characteristics—their characteristics are said to be both necessary and sufficient. The expression associated with a non-primitive name in a unique introduction axiom will often be called its *definition*.

Example 2.3 A Primitive Introduction Axiom

The axiom:

$$\mathsf{human} \sqsubseteq \mathsf{animal} \sqcap \mathsf{biped}$$

introduces the primitive concept human and states that a human is necessarily both an animal and a biped but that the conjunction of $\mathsf{animal}(x)$ and $\mathsf{biped}(x)$ is not sufficient to infer $\mathsf{human}(x)$. Natural kinds, such as human, are often primitive because it is difficult or impossible to describe them definitively.

Example 2.4 A Non-primitive Introduction Axiom

The axiom:

$$\mathsf{woman} \doteq \mathsf{human} \sqcap \mathsf{female}$$

introduces the non-primitive concept woman, states that a woman is necessarily both a human and a female and also states that the conjunction of $\mathsf{human}(x)$ and $\mathsf{female}(x)$ is sufficient to infer $\mathsf{woman}(x)$: $\mathsf{human} \sqcap \mathsf{female}$ is said to be the definition of woman.

Primitives which have no necessary conditions will be called *atomic primitives*. e.g., in a terminology:

$$\{\mathsf{vegan} \sqsubseteq \forall \mathit{eats}.\mathsf{plant}\}$$

the concept plant and the role *eats* are atomic primitives: nothing is known about plant and *eats* other than the fact that their interpretations are subsets of $\Delta^{\mathcal{I}}$ and $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ respectively. Atomic primitives are sometimes introduced explicitly with axioms of the form:

$$\begin{array}{rcl} \mathsf{plant} & \sqsubseteq & \top \\ \mathit{eats} & \sqsubseteq & \top \times \top \end{array}$$

2.2.4 General Terminological Axioms

DLs can support more general terminological axioms of the form $C \doteq D$ or $C \sqsubseteq D$, where C and D are not restricted to be a concept names but can be arbitrary concept expressions. Axioms of the form $C \doteq D$ are known as concept equations [Baa90a] or sort equations [BBN$^+$91] while those of the form $C \sqsubseteq D$ are known as general concept inclusion axioms (GCIs), universal terminological axioms or universal implications [Sch91]. The semantics of these axioms is described in Table 2.9 on the following page.

Description	Syntax	Semantics
concept equation	$C \doteq D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$
GCI	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

Table 2.9: General terminological axioms

Example 2.5 A General Concept Inclusion Axiom

The knowledge that all three-angled geometric figures are also three-sided might be expressed using the GCI:

$$\textsf{geometric-figure} \sqcap \exists \textsf{\textit{angles}}.\textsf{three} \sqsubseteq \exists \textsf{\textit{sides}}.\textsf{three}$$

A concept equation, like a non-primitive introduction axiom, corresponds to a bi-directional implication in FOPC:

$$C \doteq D \;\equiv\; \forall x.(C(x) \leftrightarrow D(x))$$

while a GCI, like a primitive introduction axiom, corresponds to an ordinary implication in FOPC:

$$C \sqsubseteq D \;\equiv\; \forall x.(C(x) \rightarrow D(x))$$

The left hand side of a GCI will therefore be called its *antecedent* and the right hand side its *consequent*—in this case C is the antecedent and D is the consequent.

Note that:

1. Concept equations are interchangeable with GCIs as shown by the following identities:

$$\begin{aligned} C \doteq D &= \left\{ \begin{array}{ccc} C & \sqsubseteq & D \\ D & \sqsubseteq & C \end{array} \right. \\ C \sqsubseteq D &= C \sqcap \neg D \doteq \bot \end{aligned} \tag{2.5}$$

2. The concept introduction axioms described in Section 2.2.3 are simply special cases of GCIs and concept equations where the expression on the left hand side is a concept name.

2.2.5 Functional and Transitive Role Axioms

In DLs which support attributes (functional roles) the terminology may include axioms of the form $\textsf{\textit{AN}} \in \mathbf{F}$, which assert that $\textsf{\textit{AN}}$ is an attribute. If $\mathbf{R}$ is the set of role names in a terminology $\mathcal{T}$, then the set of attribute names $\mathbf{F} \subseteq \mathbf{R}$ in $\mathcal{T}$ is defined by:

$$\begin{array}{rcll} \mathbf{F} & = & \{r \in \mathbf{R} \mid & 1.\ \ r \in \mathbf{F} \text{ is an axiom in } \mathcal{T} \ \mathit{or} \\ & & & 2.\ \ r \doteq A \text{ is an axiom in } \mathcal{T} \\ & & & \quad\ \text{and } A \text{ is an attribute expression } \mathit{or} \\ & & & 3.\ \ r \sqsubseteq A \text{ is an axiom in } \mathcal{T} \\ & & & \quad\ \text{and } A \text{ is an attribute expression}\} \end{array}$$

An interpretation $\mathcal{I}$ of $\mathcal{T}$ must satisfy the additional condition that, for all $\mathsf{AN} \in \mathbf{F}$, $\mathsf{AN}^{\mathcal{I}}$ is a single valued partial function:

$$\mathsf{AN}^{\mathcal{I}} : \mathit{dom}\ \mathsf{AN}^{\mathcal{I}} \longrightarrow \Delta^{\mathcal{I}}$$

In DLs which support transitive roles the terminology may include axioms of the form $\mathsf{RN} \in \mathbf{R}_+$, which assert that RN is a transitive role. If $\mathbf{R}$ is the set of roles in a terminology $\mathcal{T}$, then the set of transitive roles $\mathbf{R}_+ \subseteq \mathbf{R}$ in $\mathcal{T}$ is defined by:

$$\begin{array}{rcll} \mathbf{R}_+ & = & \{r \in \mathbf{R} \mid & 1.\ \ r \in \mathbf{R}_+ \text{ is an axiom in } \mathcal{T} \ \mathit{or} \\ & & & 2.\ \ r \doteq \mathsf{RN} \text{ is an axiom in } \mathcal{T} \\ & & & \quad\ \text{and } \mathsf{RN} \in \mathbf{R}_+\} \end{array}$$

An interpretation $\mathcal{I}$ of $\mathcal{T}$ must satisfy the additional condition that, for all $\mathsf{RN} \in \mathbf{R}_+$, $\mathsf{RN}^{\mathcal{I}}$ is transitively closed:

$$\text{if } \langle d, e\rangle \in \mathsf{RN}^{\mathcal{I}} \text{ and } \langle e, f\rangle \in \mathsf{RN}^{\mathcal{I}}, \text{ then } \langle d, f\rangle \in \mathsf{RN}^{\mathcal{I}}$$

2.3 Subsumption and Classification

Subsumption relationships between pairs of concept, role and attribute expressions are defined in terms of subset relationships between their interpretations. Like the subset relation, subsumption is transitive, reflexive and antisymmetric, and is therefore a partial ordering relation. *Classification* is the computation of the partial ordering based on the subsumption relation.

In practice, subsumption between and classification of concept expressions is the usual focus of interest in a Description Logic Knowledge Representation System (DLKRS). A DL classifier will cache the computed partial ordering of the set of named concepts in the form of a concept hierarchy, a directed acyclic graph in which each concept is linked to its *direct* subsumers and subsumees. A concept CN_1 is a direct subsumer of concept CN_2 iff $\mathsf{CN}_2 \sqsubseteq \mathsf{CN}_1$ and there is no concept CN_3 such that $\mathsf{CN}_2 \sqsubseteq \mathsf{CN}_3$ and $\mathsf{CN}_3 \sqsubseteq \mathsf{CN}_1$; CN_1 is a direct subsumee of CN_2 iff $\mathsf{CN}_1 \sqsubseteq \mathsf{CN}_2$ and there is no concept CN_3 such that $\mathsf{CN}_1 \sqsubseteq \mathsf{CN}_3$ and $\mathsf{CN}_3 \sqsubseteq \mathsf{CN}_2$. The concept hierarchy can be used to provide rapid answers to queries regarding classified concepts and to minimise the number of subsumption tests required to classify a new concept (see Section 6.3 on page 102).

2.3.1 Subsumption in General Terminologies

The semantics of DLs mean that subsumption can be formally defined in terms of the subset relationship between interpretations. Given a terminology $\mathcal{T}$ consisting of a finite set of terminological axioms, an interpretation $\mathcal{I}$ is a model of $\mathcal{T}$ if $\mathcal{I}$ satisfies all the terminological axioms in $\mathcal{T}$. A terminology $\mathcal{T}$ is satisfiable if and only if it has a non-empty model. C is subsumed by D in $\mathcal{T}$, written $C \sqsubseteq_{\mathcal{T}} D$ if and only if $C^{\mathcal{I}}$ is a subset of $D^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$:

$$C \sqsubseteq_{\mathcal{T}} D \iff C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \quad \text{for all models } \mathcal{I} \text{ of } \mathcal{T}$$

Note that if $\mathcal{T}$ is unsatisfiable, then every concept is subsumed by every other: in an empty model, $C^{\mathcal{I}} = \emptyset$ for all C and $\emptyset \subseteq \emptyset$.

This semantics for the subsumption relation, called *descriptive semantics*, can produce counter-intuitive results when the terminology contains cycles [Neb91]. For example, given a terminology $\mathcal{T}$ consisting of the axioms:

$$\begin{array}{rcl} \mathit{branch} & \sqsubseteq & \top \times \top \\ \mathsf{tree} & \sqsubseteq & \top \\ \mathsf{binary\text{-}tree} & \doteq & \mathsf{tree} \sqcap \leqslant 2\mathit{branch}\top \sqcap \forall \mathit{branch}.\mathsf{binary\text{-}tree} \\ \mathsf{ternary\text{-}tree} & \doteq & \mathsf{tree} \sqcap \leqslant 3\mathit{branch}\top \sqcap \forall \mathit{branch}.\mathsf{ternary\text{-}tree} \end{array}$$

descriptive semantics do *not* justify the intuitively obvious subsumption inference:

$$\mathsf{binary\text{-}tree} \sqsubseteq_{\mathcal{T}} \mathsf{ternary\text{-}tree}$$

because it is easy to construct a cyclical model of $\mathcal{T}$ in which $\mathsf{binary\text{-}tree}^{\mathcal{I}} \not\subseteq \mathsf{ternary\text{-}tree}^{\mathcal{I}}$:

$$\begin{array}{rcl} \Delta^{\mathcal{I}} & = & \{x\} \\ \mathsf{tree}^{\mathcal{I}} & = & \{x\} \\ \mathit{branch}^{\mathcal{I}} & = & \{\langle x, x \rangle\} \\ \mathsf{binary\text{-}tree}^{\mathcal{I}} & = & \{x\} \\ \mathsf{ternary\text{-}tree}^{\mathcal{I}} & = & \emptyset \end{array}$$

Alternative semantics based on least and greatest fixpoints have been proposed but it is not clear that any one semantics is universally preferable [Baa90c, Neb91, Sch93, BS96a]. Descriptive semantics are usually prefered as they are the most conceptually obvious (they correspond to the semantics of first order logic), and unlike fixpoint semantics, they are applicable to all DLs: fixpoints may not exist for all concept expressions when the description languages includes, for example, general concept negation or when the terminology includes GCIs [Baa90a, Sch93]. Subsumption based on descriptive semantics will therefore be assumed in the remainder of this thesis.

2.3.2 Subsumption in Unfoldable Terminologies

The evaluation of subsumption relationships is often simplified by restricting the kinds of axiom which can appear in a terminology. In particular, restricting a terminology so that it is *unfoldable* allows concept expressions to be *unfolded* so that they contain only atomic primitive concept names. Subsumption between unfolded concept expressions can then be evaluated independently of the terminology.

A terminology $\mathcal{T}$ is unfoldable if:

1. All concept axioms are introductions of the form:

$$\mathsf{CN} \sqsubseteq C \mid \mathsf{CN} \doteq C$$

   i.e., there are no GCIs or concept equations in $\mathcal{T}$.

2. All concept introduction axioms are *unique*: i.e., for each concept name CN there is at most one axiom in $\mathcal{T}$ of the form $\mathsf{CN} \sqsubseteq C$ or $\mathsf{CN} \doteq C$

3. All concept introduction axioms are *acyclic*:

   (a) an introduction axiom $\mathsf{CN} \sqsubseteq C$ or $\mathsf{CN} \doteq C$ *directly-uses* a concept name CN_1 if CN_1 appears in C (e.g., $\mathsf{CN} \doteq \exists R.\mathsf{CN}_1$ directly-uses CN_1);

   (b) an introduction axiom *uses* a concept name CN_1 if it directly-uses CN_1 or if it directly-uses a concept name CN_2 and CN_2 uses CN_1;

   (c) an introduction axiom $\mathsf{CN} \sqsubseteq C$ or $\mathsf{CN} \doteq C$ is acyclic unless it uses CN.

The process of unfolding a concept expression with respect to an unfoldable terminology $\mathcal{T}$ can most easily be described if $\mathcal{T}$ contains only non-primitive concept introduction axioms (i.e., axioms of the form $\mathsf{CN} \doteq C$). Primitive concept introduction axioms can be eliminated from $\mathcal{T}$ by replacing each axiom $\mathsf{CN} \sqsubseteq C \in \mathcal{T}$ with an equivalent non-primitive introduction axiom $\mathsf{CN} \doteq \mathsf{CN}' \sqcap C$, where CN' is a unique, new atomic primitive concept [Baa90a]. CN' represents the "primitiveness" of CN: the unspecified characteristic which differentiates CN from C. The interpretation of CN in the original terminology ($\mathsf{CN}^{\mathcal{I}} \subseteq C^{\mathcal{I}}$) remains the same in the modified terminology:

$$\mathsf{CN}^{\mathcal{I}} \;=\; (\mathsf{CN}' \sqcap C)^{\mathcal{I}} \;=\; \mathsf{CN}'^{\mathcal{I}} \cap C^{\mathcal{I}} \;\subseteq\; C^{\mathcal{I}}$$

A concept expression C can now be unfolded with respect to $\mathcal{T}$, to give $C_{u\mathcal{T}}$, by recursively substituting each non-primitive name in C with its definition, as stated in $\mathcal{T}$, until all concept names in the expression are atomic primitives [Baa90a, Neb90b]. The interpretation of C w.r.t. $\mathcal{T}$ is preserved by unfolding because $C^{\mathcal{I}}$ is defined by the interpretations of its components, and a component CN is only substituted with a concept expression D when $(\mathsf{CN} \doteq D) \in \mathcal{T}$, so $\mathsf{CN}^{\mathcal{I}} = D^{\mathcal{I}}$.

Example 2.6 Unfolding A Concept Expression

Given a terminology $\mathcal{T}$ containing the introductions from Examples 2.3 on page 33 and 2.4 on page 33:

$$\begin{aligned}\mathcal{T} &= \{\mathsf{human} \sqsubseteq \mathsf{animal} \sqcap \mathsf{biped},\\ &\quad \mathsf{woman} \doteq \mathsf{human} \sqcap \mathsf{female}\}\end{aligned}$$

the axiom $\mathsf{human} \sqsubseteq \mathsf{animal} \sqcap \mathsf{biped}$ can be replaced with $\mathsf{human} \doteq \mathsf{human}' \sqcap \mathsf{animal} \sqcap \mathsf{biped}$. The concept expression woman can then be unfolded with respect to $\mathcal{T}$ by substituting woman with its definition $\mathsf{human} \sqcap \mathsf{female}$ and then substituting human with its definition $\mathsf{human}' \sqcap \mathsf{animal} \sqcap \mathsf{biped}$ to give:

$$\mathsf{woman}_{u\mathcal{T}} = \mathsf{human}' \sqcap \mathsf{animal} \sqcap \mathsf{biped} \sqcap \mathsf{female}$$

In general, given an unfoldable terminology $\mathcal{T}$ in which all non-atomic primitive introduction axioms have been replaced with equivalent non-primitive introductions as described above, an unfolded concept expression $C_{u\mathcal{T}}$ is derived from a concept expression C using the following steps:

1. If C is an atomic primitive concept name CN, then $C_{u\mathcal{T}} \longrightarrow \mathsf{CN}$
2. If C is a concept name CN and $(\mathsf{CN} \doteq D) \in \mathcal{T}$, then $C_{u\mathcal{T}} \longrightarrow D_{u\mathcal{T}}$
3. If C is not a concept name, then $C_{u\mathcal{T}}$ is given by substituting each component concept expression D in C with $D_{u\mathcal{T}}$ (e.g., $\exists R.D \longrightarrow \exists R.D_{u\mathcal{T}}$)

The interpretation of an unfolded concept expression $C_{u\mathcal{T}}$ is defined by the interpretations of the roles and atomic primitive concepts of which it is composed and is independent of the concept introduction axioms in $\mathcal{T}$. If $\mathcal{T}'$ is a terminology derived from $\mathcal{T}$ by deleting all concept introduction axioms, then:

$$C \sqsubseteq_{\mathcal{T}} D \iff C_{u\mathcal{T}} \sqsubseteq_{\mathcal{T}'} D_{u\mathcal{T}}$$

For DLs such as $\mathcal{ALC}$, which do not support any form of role axiom, an unfoldable terminology $\mathcal{T}$ will consist entirely of concept introduction axioms and $\mathcal{T}'$ will therefore be an empty terminology. As any interpretation satisfies an empty terminology, subsumption of unfolded $\mathcal{ALC}$ concepts $C_{u\mathcal{T}}$ and $D_{u\mathcal{T}}$ is independent of $\mathcal{T}$:

$$C \sqsubseteq_{\mathcal{T}} D \iff C^{\mathcal{I}}_{u\mathcal{T}} \subseteq D^{\mathcal{I}}_{u\mathcal{T}} \quad \text{for all models } \mathcal{I}$$

Subsumption in such a terminology can be evaluated using a much simpler decision procedure: one which can only deal with a pair of concept expressions in which all names are atomic primitives.

Note that a primitive interpretation (an assignment of values to the interpretations of primitive concepts) leads, via the semantics, to a unique interpretation for all concept expressions in an unfoldable terminology: least fixpoint, greatest fixpoint and descriptive semantics therefore coincide in unfoldable terminologies [Neb90a].

2.4 Theoretical and Implemented DLs

This section introduces some of the DLs which have been described in the literature or which form the basis of implemented DL Knowledge Representation Systems (DLKRSs). As stated in Section 2.1.1 on page 25, a particular DL is characterised by the kinds of concept and role expression allowed in its description language and the kinds of axiom allowed in its terminologies.

2.4.1 The $\mathcal{ALC}$ Family of DLs

The $\mathcal{ALC}$ DL [SSS91] and its family of extensions are of particular interest:

- a wide variety of expressive possibilities can be achieved by augmenting the range of concept expressions, role expressions and terminological axioms which are supported by $\mathcal{ALC}$;
- $\mathcal{ALC}$ and its extensions have been the subject of detailed theoretical study:
  - decidability and complexity results are available for many possible extensions;
  - sound and complete subsumption testing algorithms have also been designed in many cases.

The syntax of $\mathcal{ALC}$ was described in detail in Section 2.1.1 on page 25. To recapitulate, if CN is a concept name, RN is a role name and C and D are concept expressions, then axioms are of the form:

$$C \sqsubseteq D \mid C \doteq D$$

and concept expressions are of the form:

$$\mathsf{CN} \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists \mathit{RN}.C \mid \forall \mathit{RN}.C$$

Extensions of $\mathcal{ALC}$ which have been studied include:

- $\mathcal{ALCN}$, $\mathcal{ALCR}$ and $\mathcal{ALCNR}$ [HN90]—$\mathcal{ALC}$ augmented with number restriction concept expressions ($\mathcal{N}$) or/and role conjunction ($\mathcal{R}$);
- $\mathcal{ALCF}$ [HN90]—$\mathcal{ALC}$ augmented with attributes (sometimes called features), attribute composition and attribute value map concept expressions;
- $\mathcal{ALCFN}$ [BH91c]—$\mathcal{ALCF}$ augmented with number restriction concept expressions;
- $\mathcal{ALCFNR}$ [HN90]—$\mathcal{ALCFN}$ augmented with role conjunction;

- $\mathcal{ALCN}(\circ)$ [BS96b]—$\mathcal{ALCN}$ augmented with role composition in number restriction concept expressions;

- $\mathcal{ALC}_{+}$ [Baa90a]—$\mathcal{ALC}$ augmented with union, composition and transitive closure role expressions;

- $\mathcal{ALC}_{R^+}$ [Sat96]—$\mathcal{ALC}$ augmented with transitively closed primitive roles (axioms of the form $\mathsf{RN} \in \mathbf{R}_{+}$);

- $\mathcal{ALC}_{\oplus}$ [Sat96]—$\mathcal{ALC}_{R^+}$ augmented with a restricted form of primitive role introduction axioms (see Section 4.1 on page 58);

- $\mathcal{TSL}$ [Sch91]—$\mathcal{ALC}$ augmented with union, composition, identity, transitive reflexive closure and inverse role expressions;

- $\mathcal{CIQ}$ [GL96]—$\mathcal{TSL}$ augmented with qualified number restriction concept expressions (inverse roles are the only form of role expression allowed in qualified number restrictions);

It should be noted that extending the syntax of a description language does not necessarily increase its expressiveness. For example, adding disjunction, composition and identity role expressions to $\mathcal{ALC}$ does not increase its expressiveness as they can be eliminated from concept expressions using the identities 2.4 on page 31. Characterising and comparing the expressiveness of different DLs is a non-trivial problem and has been the subject of a number of studies [Baa90b, Bor96, KdR97].

Figure 2.1 on the following page shows various members of the $\mathcal{ALC}$ family arranged in a hierarchy based on syntactic inclusion, although it is believed that the expressiveness of these DLs is also distinct. $\mathcal{CIQ}$ is probably the most expressive DL which is know to be decidable and for which a sound and complete subsumption testing algorithm is available [GL96].

Extensions of $\mathcal{ALC}$ which are introduced and will be the subject of detailed study in this thesis are $\mathcal{ALCH}_{R^+}$ and $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ (see Chapter 4). $\mathcal{ALCH}_{R^+}$ augments $\mathcal{ALC}_{\oplus}$ with unrestricted primitive role introduction axioms, allowing a complete role hierarchy[1] to be defined; $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ augments $\mathcal{ALCH}_{R^+}$ with attributes, or functional roles[2], and is the logic which is used in the FaCT system (see Chapter 6).

[1]The $\mathcal{H}$ in $\mathcal{ALCH}_{R^+}$ stands for role $\mathcal{H}$ierarchy.
[2]The $\boldsymbol{f}$ in $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ stands for $\boldsymbol{f}$unctional roles

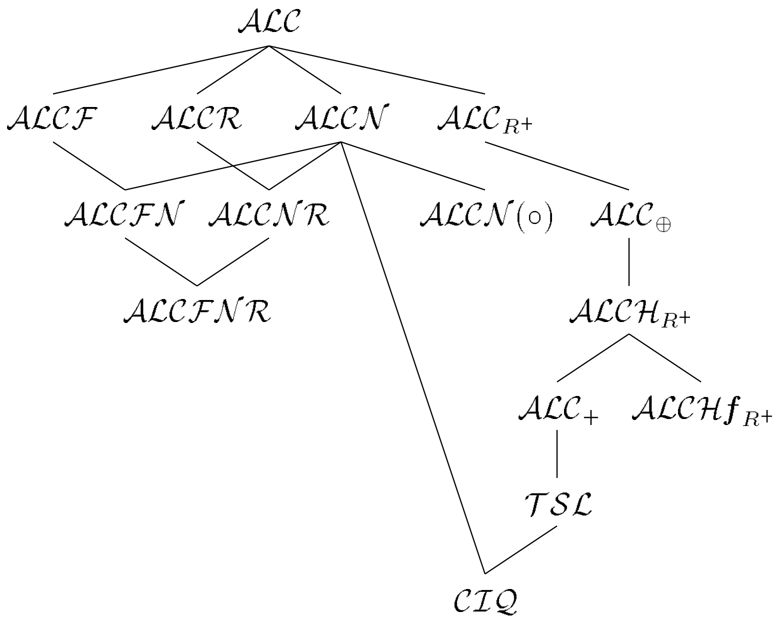

Figure 2.1: A hierarchy of DLs from the $\mathcal{ALC}$ family

2.4.2 Implemented DLKRSs

Other than FaCT, KRIS is the only available DLKRS which uses a sound and complete tableaux based subsumption testing algorithm[3], and for this reason it is used as one of the benchmarks when testing FaCT's performance (See Chapter 7). The DL implemented in the KRIS system is $\mathcal{ALCFN}$ augmented with the ability to refer to elements of Lisp enumerated types and to nonnegative integers in concept expressions (these are called *concrete domains* [BH91a]). The KRIS classifier is only able to evaluate subsumption with respect to unfoldable terminologies.

The other implemented DLKRSs mentioned in Section 2.1 on page 23 provide concept description languages of varying expressiveness: CLASSIC is more restricted than KRIS and does not support negation, disjunction or exists restrictions; unlike CLASSIC, BACK does not support attributes, but adds (in version 5) support for exists restrictions and qualified number restrictions, as well as conjunction, composition, inversion and transitive closure role expressions; LOOM is highly expressive, supporting all the concept expressions described in Section 2.2.1 on page 28, and even supporting additional expressions, such as role value maps ($R = S$), which are known to be generally undecidable [SS89].

[3]The only other tableaux based system of which the author is aware is CRACK [BFT95], which has been developed at IRST in Trento, Italy, and is not currently available for use outside IRST.

Classic, Back and Loom all restrict terminologies to those which are unfoldable and all use structural subsumption algorithms. Classic provides complete subsumption reasoning with respect to purely conceptual expressions but it also supports concepts which are extensionally defined (in terms of sets of individuals) and its reasoning with respect to these concepts is incomplete, or at least requires a weakening of the model theoretic semantics [BPS94]. Back and Loom both provide incomplete subsumption reasoning: they do not guarantee to find all the subsumption inferences which are justified by the semantics of their concept description languages. In the case of Loom, the degree of incompleteness can be controlled using the `power-level` function, which:

> ... allows the user to turn off (or down) certain expensive types of inferencing in order to gain a possibly substantial increase in classifier performance. [Bri93]

However it is known to be difficult to characterize the inferences missed by incomplete algorithms [Bor92] and in the case of Loom it is acknowledged that:

> It is difficult to precisely characterize the types of inferencing affected by the `power-level`. [Bri93]

Table 2.10 on the following page summarises and compares the expressiveness of the members of the $\mathcal{ALC}$ family of DLs described in Section 2.4.1, and of Kris, Classic, Back and Loom, in terms of their support for the concept expressions described in Section 2.2.1 on page 28; Table 2.11 on page 44 makes the same comparison in terms of their support for the role and attribute expressions described in Section 2.2.2 on page 30.

	Concept Expressions												
DL	$\sqcap$	$\sqcup$	$\neg$	$\exists_R$	$\forall_R$	$\geqslant_{nR}$	$\leqslant_{nR}$	$\geqslant_{nR.C}$	$\leqslant_{nR.C}$	$\exists_A$	$\forall_A$	$=$	$\neq$
$\mathcal{ALC}$	×	×	×	×	×								
$\mathcal{ALCN}$	×	×	×	×	×	×	×						
$\mathcal{ALCR}$	×	×	×	×	×								
$\mathcal{ALCNR}$	×	×	×	×	×	×	×						
$\mathcal{ALCF}$	×	×	×	×	×					×	×	×	×
$\mathcal{ALCFN}$	×	×	×	×	×	×	×			×	×	×	×
$\mathcal{ALCFNR}$	×	×	×	×	×	×	×			×	×	×	×
$\mathcal{ALCN}(\circ)$	×	×	×	×	×	×	×						
$\mathcal{ALC}_{+}$	×	×	×	×	×								
$\mathcal{ALC}_{R^+}$	×	×	×	×	×								
$\mathcal{ALC}_{\oplus}$	×	×	×	×	×								
$\mathcal{TSL}$	×	×	×	×	×								
$\mathcal{CIQ}$	×	×	×	×	×	×	×	×	×				
$\mathcal{ALCH}_{R^+}$	×	×	×	×	×								
$\mathcal{ALCH}\boldsymbol{f}_{R^+}$	×	×	×	×	×					×	×		
Kris	×	×	×	×	×	×	×			×	×	×	×
Classic	×				×	×	×			×	×	×	×
Back	×			×	×	×	×	×	×				
Loom	×	×	×	×	×	×	×	×	×	×	×	×	×

Table 2.10: Concept expressions supported by DLs

	Role and Attribute Expressions							
DL	$\sqcap$	$\sqcup$	$\circ$	id	-1	$+$	$*$	$A \circ B$
$\mathcal{ALC}$								
$\mathcal{ALCN}$								
$\mathcal{ALCR}$	$\times$							
$\mathcal{ALCNR}$	$\times$							
$\mathcal{ALCF}$								$\times$
$\mathcal{ALCFN}$								$\times$
$\mathcal{ALCFNR}$	$\times$							$\times$
$\mathcal{ALCN}(\circ)$			$\times$					
$\mathcal{ALC}_{+}$		$\times$	$\times$			$\times$		
$\mathcal{ALC}_{R^+}$								
$\mathcal{ALC}_{\oplus}$								
$\mathcal{TSL}$		$\times$	$\times$	$\times$	$\times$		$\times$	
$\mathcal{CIQ}$		$\times$	$\times$	$\times$	$\times$		$\times$	
$\mathcal{ALCH}_{R^+}$								
$\mathcal{ALCH}\boldsymbol{f}_{R^+}$								
Kris								$\times$
Classic								$\times$
Back	$\times$		$\times$		$\times$	$\times$		
Loom	$\times$		$\times$		$\times$			

Table 2.11: Role expressions supported by DLs

Chapter 3

Tableaux Algorithms

This chapter introduces tableaux subsumption testing algorithms and describes their operation. It is intended to be illustrative of the general principles involved rather than completely rigorous with respect to any particular algorithm: a rigorous treatment of a subsumption testing algorithm for $\mathcal{ALCH}_{R^+}$ will be presented in Chapter 4.

The chapter is organised as follows: Section 3.1 shows how a subsumption problem can be transposed into an equivalent satisfiability problem; Section 3.2 illustrates the tableaux method by describing a simple tableaux satisfiability testing algorithm; and finally, Section 3.3 shows how the algorithm can be extended to deal with general terminologies.

3.1 Tableaux Subsumption Testing

Most early DL systems used structural subsumption algorithms [Woo91] based on rules such as:

$$\begin{array}{rll} (C_1 \sqcap \ldots \sqcap C_n) \sqsubseteq_{\mathcal{T}} (D_1 \sqcap \ldots \sqcap D_m) & \textit{if} & \text{for every } D \text{ in } D_1, \ldots, D_n \\ & & \text{there is some } C \text{ in } C_1, \ldots, C_n \\ & & \text{such that } C \sqsubseteq_{\mathcal{T}} D \\ \exists R.C \sqsubseteq_{\mathcal{T}} \exists S.D & \textit{if} & R \sqsubseteq_{\mathcal{T}} S \text{ and } C \sqsubseteq_{\mathcal{T}} D \end{array}$$

The rules are used to recursively decompose the problem until it is reduced to axiomatic primitive subsumption relationships in the terminology $\mathcal{T}$.

An alternative approach, first used in the Kris system [BH91b], is to transpose the subsumption problem into an equivalent satisfiability problem:

$$C \sqsubseteq_{\mathcal{T}} D \iff (C \sqcap \neg D)^{\mathcal{I}} = \emptyset \quad \text{for all models } \mathcal{I} \text{ of } \mathcal{T}$$

The validity of this transformation is clear from the semantics of subsumption:

$$\begin{array}{rcl} C \sqsubseteq_{\mathcal{T}} D & \iff & C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \quad \text{for all models } \mathcal{I} \text{ of } \mathcal{T} \\ C^{\mathcal{I}} \subseteq D^{\mathcal{I}} & \iff & C^{\mathcal{I}} \cap \overline{D^{\mathcal{I}}} = \emptyset \\ C^{\mathcal{I}} \cap \overline{D^{\mathcal{I}}} = \emptyset & \iff & (C \sqcap \neg D)^{\mathcal{I}} = \emptyset \end{array}$$

Given an unfoldable terminology $\mathcal{T}$ (see Section 2.3.2 on page 37), subsumption between concept terms which have been unfolded with respect to $\mathcal{T}$ is independent of $\mathcal{T}$ and can therefore be evaluated by testing the satisfiability of a single unfolded concept expression:

$$C \sqsubseteq_{\mathcal{T}} D \iff (C \sqcap \neg D)^{\mathcal{I}}_{u\mathcal{T}} = \emptyset \quad \text{for all } \mathcal{I}$$

i.e., C is subsumed by D with respect to $\mathcal{T}$ iff $C \sqcap \neg D$, unfolded w.r.t. $\mathcal{T}$, is not satisfiable (has no non-empty model).

The resulting satisfiability problems can be solved using an algorithm based on the tableaux calculus [BH91b, HN90]. This approach has many advantages and has dominated recent DL research:

- it has a sound theoretical basis in first order logic [HN90];
- it can be relatively easily adapted to allow for a range of description languages by changing the set of tableaux expansion rules [HN90, BFT95];
- it can be adapted to deal with more expressive logics, including those which support transitively closed roles, by using more sophisticated control mechanisms to ensure termination [Baa90a, Sat96];
- it can be used to test knowledge base satisfiability and thus deal with DL languages which permit cyclical definitions and/or GCIs [BDS93];
- it has been shown to be optimal for a number of DL languages, in the sense that the worst case complexity of the algorithm is no worse than the known complexity of the satisfiability problem for the logic [HN90];
- the theoretical frontiers of decidability and tractability are well understood [Sch91, DHL+89].

3.2 General Method

Tableaux algorithms try to prove the satisfiability of a concept expression D by demonstrating a model—an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ in which $D^{\mathcal{I}} \neq \emptyset$; a *tableau* is a graph which represents such a model, with nodes corresponding to individuals and edges corresponding to relationships between individuals. A typical algorithm will start with a single individual satisfying D and try to construct a complete model "ground up" by inferring the existence of additional individuals or of additional constraints on individuals. The inference mechanism consists of applying a set of expansion rules which correspond to the logical constructs of the language and the algorithm terminates either when the model is complete (no further inferences are possible) or when an obvious contradiction has been revealed.

To simplify the algorithm, D is assumed to be an unfolded concept expression in *negation normal form*. A concept expression is in negation normal form when negations apply only to concept names and not to compound terms. Arbitrary concept expressions can be transformed into negation normal form using a combination of DeMorgan's laws and the identities 2.1 on page 29 (e.g., $\neg\exists R.C = \forall R.\neg C$ and $\neg\forall R.C = \exists R.\neg C$).

Example 3.1 Negation Normal Form

The negation normal form of $\neg(\exists R.C \sqcap \forall S.D)$ is obtained by first applying DeMorgan's laws:

$$\neg(\exists R.C \sqcap \forall S.D) \quad \longrightarrow \quad \neg\exists R.C \sqcup \neg\forall S.D$$

followed by identities 2.1:

$$\neg\exists R.C \sqcup \neg\forall S.D \quad \longrightarrow \quad \forall R.\neg C \sqcup \exists S.\neg D$$

3.2.1 A Tableaux Algorithm for $\mathcal{ALC}$

The tableaux method can best be illustrated by describing an algorithm for deciding the satisfiability of $\mathcal{ALC}$ concept expressions. The algorithm uses a tree to represent the model being constructed. Each node x in the tree represents an individual and is labelled with a set $\mathcal{L}(x)$ of $\mathcal{ALC}$ concept expressions which it must satisfy:

$$C \in \mathcal{L}(x) \Rightarrow x \in C^{\mathcal{I}}$$

Each edge $\langle x, y\rangle$ in the tree represents a pair of individuals in the interpretation of a role and is labelled with the role name:

$$R = \mathcal{L}(\langle x, y\rangle) \Rightarrow \langle x, y\rangle \in R^{\mathcal{I}}$$

To determine the satisfiability of a concept expression D, a tree $\mathbf{T}$ is initialised to contain a single node x_0, with $\mathcal{L}(x_0) = \{D\}$, and expanded by repeatedly applying the rules from Table 3.1. **T** is fully expanded when none of the rules can be applied. **T** contains an obvious contradiction or *clash* when, for some node x and some concept C, either $\bot \in \mathcal{L}(x)$ or $\{C, \neg C\} \subseteq \mathcal{L}(x)$.

$\sqcap$-rule	if 1.	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule	if 1.	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
	then	a. save **T**
		b. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1\}$
		If that leads to a clash then restore **T** and
		c. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_2\}$
$\exists$-rule	if 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \in \mathcal{L}(y)$
	then	create a new node y and edge $\langle x, y\rangle$
		with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}(\langle x, y\rangle) = R$
$\forall$-rule	if 1.	$\forall R.C \in \mathcal{L}(x)$
	2.	there is some y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \notin \mathcal{L}(y)$
	then	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Table 3.1: Tableaux expansion rules for $\mathcal{ALC}$

A fully expanded clash-free tree **T** can trivially be converted into a model which is a witness to the satisfiability of D:

$$
\begin{aligned}
\Delta^{\mathcal{I}} &= \{x \mid x \text{ is a node in } \mathbf{T}\} \\
\mathsf{CN}^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \mathsf{CN} \in \mathcal{L}(x)\} \text{ for all concept names } \mathsf{CN} \text{ in } D \\
R^{\mathcal{I}} &= \{\langle x, y\rangle \mid \langle x, y\rangle \text{ is an edge in } \mathbf{T} \text{ and } \mathcal{L}(\langle x, y\rangle) = R\}
\end{aligned}
$$

The interpretations of concept expressions follow directly from the semantics given in Table 2.3 on page 28.

Note that:

1. The second condition in each rule constitutes a control strategy which ensures that the algorithm does not fail to terminate due to an infinite repetition of the same expansion. Informally, the algorithm is guaranteed to terminate because:

(a) The $\sqcap$, $\sqcup$ and $\exists$ rules can only be applied once to any given concept expression C in $\mathcal{L}(x)$.

(b) The $\forall$-rule can be applied many times to a given $\forall R.C$ expression in $\mathcal{L}(x)$ but only once to any given edge $\langle x, y\rangle$.

(c) Applying a rule to a concept expression C extends the labelling with a concept expression which is always strictly smaller than C.

2. The $\sqcup$-rule is different from the other rules: it is non-deterministic and operates by performing a depth first backtracking search of the possible expansions resulting from disjunctions in D, halting when either a fully expanded tree is found or every possible expansion is shown to lead to a clash.

Example 3.2 Demonstrating Subsumption

Given an unfoldable terminology $\mathcal{T}$ containing the definition of vegan from Example 2.1:

$$\mathsf{vegan} \doteq \mathsf{person} \sqcap \forall \mathit{eats}.\mathsf{plant}$$

and the following definition of $\mathsf{vegetarian}$:

$$\mathsf{vegetarian} \doteq \mathsf{person} \sqcap \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy})$$

the algorithm can be used to show that $\mathsf{vegan} \sqsubseteq_{\mathcal{T}} \mathsf{vegetarian}$ by demonstrating that $(\mathsf{vegan} \sqcap \neg\mathsf{vegetarian})_{u\mathcal{T}}$ (i.e., $\mathsf{vegan} \sqcap \neg\mathsf{vegetarian}$ unfolded w.r.t. $\mathcal{T}$) is not satisfiable:

1. Unfold and normalise $\mathsf{vegan} \sqcap \neg\mathsf{vegetarian}$ to give:
$$\mathsf{person} \sqcap \forall \mathit{eats}.\mathsf{plant} \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}))$$

2. Initialise $\mathbf{T}$ to contain a single node x labelled:
$$\mathcal{L}(x) = \{\mathsf{person} \sqcap \forall \mathit{eats}.\mathsf{plant} \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}))\}$$

3. Apply the $\sqcap$-rule to $\mathsf{person} \sqcap \forall \mathit{eats}.\mathsf{plant} \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy})) \in \mathcal{L}(x)$:
$$\begin{aligned}\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \quad \cup \quad & \{\mathsf{person}, \forall \mathit{eats}.\mathsf{plant},\\ & \neg\mathsf{person} \sqcup \exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy})\}\end{aligned}$$

4. Apply the $\sqcup$-rule to $\neg\mathsf{person} \sqcup \exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}) \in \mathcal{L}(x)$:

(a) Save $\mathbf{T}$ and try:
$$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\neg\mathsf{person}\}$$
This is an obvious contradiction as $\{\mathsf{person}, \neg\mathsf{person}\} \subseteq \mathcal{L}(x)$.

(b) Restore $\mathbf{T}$ and try:
$$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}))\}$$

5. Apply the $\exists$-rule to $\exists \mathit{eats}.(\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}) \in \mathcal{L}(x)$:

   create a new node y and a new edge $\langle x, y\rangle$
   $\mathcal{L}(y) = \{\neg\mathsf{plant} \sqcap \neg\mathsf{dairy}\}$
   $\mathcal{L}(\langle x, y\rangle) = \mathit{eats}$

6. Apply the $\forall$-rule to $\forall \mathit{eats}.\mathsf{plant} \in \mathcal{L}(x)$ and $\mathcal{L}(\langle x, y\rangle) = \mathit{eats}$:

   $$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\mathsf{plant}\}$$

7. Apply the $\sqcap$-rule to $\neg\mathsf{plant} \sqcap \neg\mathsf{dairy} \in \mathcal{L}(y)$:

   $$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \quad \cup \quad \{\neg\mathsf{plant}, \neg\mathsf{dairy}\}$$

   This is an obvious contradiction as $\{\mathsf{plant}, \neg\mathsf{plant}\} \subseteq \mathcal{L}(y)$.

As all possible applications of the $\sqcup$-rule (step 4) have now been shown to lead to a contradiction, it can be concluded that, with respect to the definitions in $\mathcal{T}$, $\mathsf{vegan} \sqcap \neg\mathsf{vegetarian}$ is unsatisfiable and thus that $\mathsf{vegetarian}$ subsumes vegan.

Example 3.3 Demonstrating Non-subsumption

Conversely, it can be shown that $\mathsf{vegetarian} \not\sqsubseteq_{\mathcal{T}} \mathsf{vegan}$ by demonstrating that $(\mathsf{vegetarian} \sqcap \neg\mathsf{vegan})_{u\mathcal{T}}$ is satisfiable:

1. Unfold and normalise $\mathsf{vegetarian} \sqcap \neg\mathsf{vegan}$ to give:

   $$\mathsf{person} \sqcap \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}) \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant})$$

2. Initialise $\mathbf{T}$ to contain a single node x labeled:

   $$\mathcal{L}(x) = \{\mathsf{person} \sqcap \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}) \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant})\}$$

3. Apply the $\sqcap$-rule to $\mathsf{person} \sqcap \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}) \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant}) \in \mathcal{L}(x)$:

   $$\begin{aligned} \mathcal{L}(x) \longrightarrow \mathcal{L}(x) \quad \cup \quad & \{\mathsf{person}, \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}), \\ & \neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant}\} \end{aligned}$$

4. Apply the $\sqcup$-rule to $\neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant} \in \mathcal{L}(x)$:

   (a) Save $\mathbf{T}$ and try:

   $$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\neg\mathsf{person}\}$$

   This is an obvious contradiction as $\{\mathsf{person}, \neg\mathsf{person}\} \subseteq \mathcal{L}(x)$.

   (b) Restore $\mathbf{T}$ and try:

   $$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\exists \mathit{eats}.\neg\mathsf{plant}\}$$

5. Apply the $\exists$-rule to $\exists \mathit{eats}.\neg\mathsf{plant} \in \mathcal{L}(x)$:

   create a new node y and a new edge $\langle x, y\rangle$
   $\mathcal{L}(y) = \{\neg\mathsf{plant}\}$
   $\mathcal{L}(\langle x, y\rangle) = \mathit{eats}$

6. Apply the $\forall$-rule to $\forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}) \in \mathcal{L}(x)$ and $\mathcal{L}(\langle x, y\rangle) = \mathit{eats}$:
$$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\mathsf{plant} \sqcup \mathsf{dairy}\}$$
7. Apply the $\sqcup$-rule to $\mathsf{plant} \sqcup \mathsf{dairy} \in \mathcal{L}(y)$:
   (a) Save $\mathbf{T}$ and try:
   $$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\mathsf{plant}\}$$
   This is an obvious contradiction as $\{\mathsf{plant}, \neg\mathsf{plant}\} \subseteq \mathcal{L}(y)$.
   (b) Restore $\mathbf{T}$ and try:
   $$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\mathsf{dairy}\}$$

None of the expansion rules are now applicable to $\mathbf{T}$ so it is fully expanded. $\mathbf{T}$ consists of two nodes, x and y, and a single edge $\langle x, y\rangle$ with labels:

$$\begin{aligned}
\mathcal{L}(x) &= \{\mathsf{person} \sqcap \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}) \sqcap (\neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant}),\\
&\quad\ \mathsf{person}, \forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}), \neg\mathsf{person} \sqcup \exists \mathit{eats}.\neg\mathsf{plant},\\
&\quad\ \exists \mathit{eats}.\neg\mathsf{plant}\}\\
\mathcal{L}(y) &= \{\neg\mathsf{plant}, \mathsf{plant} \sqcup \mathsf{dairy}, \mathsf{dairy}\}\\
\mathcal{L}(\langle x, y\rangle) &= \mathit{eats}
\end{aligned}$$

$\mathbf{T}$ can trivially be converted into a model which is a witness to the satisfiability of $\mathsf{vegetarian} \sqcap \neg\mathsf{vegan}$ and thus proves that vegan does not subsume $\mathsf{vegetarian}$:

$$\begin{aligned}
\Delta^{\mathcal{I}} &= \{x, y\}\\
\mathsf{person}^{\mathcal{I}} &= \{x\}\\
\mathsf{dairy}^{\mathcal{I}} &= \{y\}\\
\mathsf{plant}^{\mathcal{I}} &= \emptyset\\
\mathit{eats}^{\mathcal{I}} &= \{\langle x, y\rangle\}
\end{aligned}$$

Interpretations of concept expressions, and thus of non-primitive concepts in $\mathcal{T}$, follow directly from the semantics given in Table 2.3 on page 28:

$$\begin{aligned}
(\mathsf{plant} \sqcup \mathsf{dairy})^{\mathcal{I}} &= \mathsf{plant}^{\mathcal{I}} \cup \mathsf{dairy}^{\mathcal{I}}\\
&= \{y\}\\
(\forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}))^{\mathcal{I}} &= \{d \in \Delta^{\mathcal{I}} \mid \quad \mathit{eats}^{\mathcal{I}}(d) \subseteq (\mathsf{plant} \sqcup \mathsf{dairy})^{\mathcal{I}}\}\\
&= \{x, y\}\\
\mathsf{vegetarian}^{\mathcal{I}} &= \mathsf{person}^{\mathcal{I}} \cap (\forall \mathit{eats}.(\mathsf{plant} \sqcup \mathsf{dairy}))^{\mathcal{I}}\\
&= \{x\}\\
(\forall \mathit{eats}.\mathsf{plant})^{\mathcal{I}} &= \{d \in \Delta^{\mathcal{I}} \mid \mathit{eats}^{\mathcal{I}}(d) \subseteq \mathsf{plant}^{\mathcal{I}}\}\\
&= \{y\}\\
\mathsf{vegan}^{\mathcal{I}} &= \mathsf{person}^{\mathcal{I}} \cap (\forall \mathit{eats}.\mathsf{plant})^{\mathcal{I}}\\
&= \emptyset\\
(\neg\mathsf{vegan})^{\mathcal{I}} &= \Delta^{\mathcal{I}} - \mathsf{vegan}^{\mathcal{I}}
\end{aligned}$$

$$
\begin{aligned}
&= \{x, y\} \\
(\mathsf{vegetarian} \sqcap \neg\mathsf{vegan})^{\mathcal{I}} &= \mathsf{vegetarian}^{\mathcal{I}} \cap (\neg\mathsf{vegan})^{\mathcal{I}} \\
&= \{x\}
\end{aligned}
$$

3.3 Dealing with General Terminologies

The algorithm described above tests the satisfiability of a single unfolded concept expression and can therefore only be used to evaluate subsumption with respect to an unfoldable terminology. To evaluate subsumption with respect to a general terminology, one which may contain cycles, concept equations and general concept inclusion axioms (GCIs), the algorithm can be extended by the addition of a *meta constraint* $\mathcal{M}$ and a more sophisticated control strategy known as *blocking*, both of which will be described in the following sections.

3.3.1 Meta Constraints

When testing satisfiability with respect to a general terminology $\mathcal{T}$, it is necessary to restrict the models $\mathcal{I}$ which can be generated by the algorithm to those which satisfy $\mathcal{T}$:

$$C \sqsubseteq_{\mathcal{T}} D \iff (C \sqcap \neg D)^{\mathcal{I}} = \emptyset \quad \text{for all models } \mathcal{I} \text{ of } \mathcal{T}$$

where a model $\mathcal{I}$ satisfies a terminology $\mathcal{T}$ if it satisfies all the axioms in $\mathcal{T}$ (see Section 2.3 on page 35). As all concept axioms can be transformed into equivalent GCIs using identity 2.5:

$$C \doteq D \;=\; \begin{cases} C & \sqsubseteq & D \\ D & \sqsubseteq & C \end{cases}$$

it is sufficient to consider the problem of restricting models to those which satisfy GCIs in $\mathcal{T}$.

A GCI $C \sqsubseteq D \in \mathcal{T}$ is satisfied by a model $\mathcal{I}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. This can be transposed into an equivalent satisfiability condition in the same way as for general subsumption testing: $C \sqsubseteq D$ is satisfied by $\mathcal{I}$ iff $(C \sqcap \neg D)^{\mathcal{I}} = \emptyset$. Negating both sides of this equality gives:

$$C \sqsubseteq D \text{ is satisfied by } \mathcal{I} \iff (\neg C \sqcup D)^{\mathcal{I}} = \Delta^{\mathcal{I}}$$

i.e., a GCI $C \sqsubseteq D$ is satisfied by a model $\mathcal{I}$ iff every individual in the model satisfies $\neg C \sqcup D$ [BDS93].

The trees which are constructed by the tableaux algorithm described in Section 3.2.1 can therefore be restricted to those which represent models satisfying the GCI $C \sqsubseteq D$ by imposing a meta constraint:

$$\text{for all nodes } x \text{ in } \mathbf{T},\ (\neg C \sqcup D) \in \mathcal{L}(x)$$

where $\neg C \sqcup D$ is transformed into negation normal form. The set of nodes x in $\mathbf{T}$ represent the domain $\Delta^{\mathcal{I}}$, and as $(\neg C \sqcup D)$ is in the label of every node, $(\neg C \sqcup D)^{\mathcal{I}} = \Delta^{\mathcal{I}}$. For convenience, we will abuse notation by writing such a meta constraint as:

$$\forall x \in \mathbf{T}.(\neg C \sqcap D) \in \mathcal{L}(x)$$

Example 3.4 Converting a GCI Into a Meta Constraint

The GCI $\mathsf{geometric\text{-}figure} \sqcap \exists \mathit{angles}.\mathsf{three} \sqsubseteq \exists \mathit{sides}.\mathsf{three}$ from Example 2.5 on page 34 would be converted into the meta constraint:

$$\forall x \in \mathbf{T}.(\exists \mathit{sides}.\mathsf{three} \sqcup \neg(\mathsf{geometric\text{-}figure} \sqcap \exists \mathit{angles}.\mathsf{three})) \in \mathcal{L}(x)$$

or, in negation normal form:

$$\forall x \in \mathbf{T}.(\exists \mathit{sides}.\mathsf{three} \sqcup (\neg \mathsf{geometric\text{-}figure} \sqcup \forall \mathit{angles}.\neg \mathsf{three})) \in \mathcal{L}(x)$$

which ensures that every individual in a valid model is either three-sided or not a three-angled geometric figure.

The satisfiability of a concept expression D with respect to a general $\mathcal{ALC}$ terminology $\mathcal{T}$ can be tested by transforming all the axioms in $\mathcal{T}$ into GCIs using identity 2.5 to give:

$$\mathcal{T} = \{C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n\}$$

and imposing the meta constraints:

$$\forall x \in \mathbf{T}.(\neg C_1 \sqcup D_1) \in \mathcal{L}(x), \ldots, \forall x \in \mathbf{T}.(\neg C_n \sqcup D_n) \in \mathcal{L}(x)$$

on the tree construction algorithm [BDS93]. These constraints can be combined to form a single meta constraint:

$$\forall x \in \mathbf{T}.\mathcal{M} \in \mathcal{L}(x)$$

where $\mathcal{M}$ is the negation normal form of:

$$(\neg C_1 \sqcup D_1) \sqcap \ldots \sqcap (\neg C_n \sqcup D_n)$$

The tree construction algorithm can be modified to deal with this constraint by:

1. Initialising $\mathbf{T}$ so that $\mathcal{L}(x_0) = \{D, \mathcal{M}\}$.

2. Adding $\mathcal{M}$ to the label of each new node created by the $\exists$-rule.

Note that it is not necessary to unfold D, because introduction axioms are represented by disjunctions in the label of every node. For example, an axiom $\mathsf{CN} \doteq C$ would be transformed into two GCIs, $\mathsf{CN} \sqsubseteq C$ and $C \sqsubseteq \mathsf{CN}$, leading to the disjunctions $\neg\mathsf{CN} \sqcup C$ and $\mathsf{CN} \sqcup \neg C$ being present in the label of every node.

3.3.2 Blocking

The modified tree construction algorithm described above no longer satisfies the termination conditions stated in Section 3.2.1 and it is easy to construct examples which would lead to non-termination.

Example 3.5 Non Terminating Tableaux Expansion

Given a terminology $\mathcal{T}$ containing the single cyclical axiom:

$$\mathsf{human} \sqsubseteq \exists \mathit{parent}.\mathsf{human}$$

then testing the satisfiability of human w.r.t. $\mathcal{T}$ (i.e., evaluating $\mathsf{human} \sqsubseteq_{\mathcal{T}} \bot$) leads to:

$$\begin{aligned} \mathcal{M} &= \neg\mathsf{human} \sqcup \exists \mathit{parent}.\mathsf{human} \\ \mathcal{L}(x_0) &= \{\mathsf{human}, \mathcal{M}\} \end{aligned}$$

Application of the $\sqcup$-rule to $\mathcal{M}$ in $\mathcal{L}(x_0)$ leads to $\exists \mathit{parent}.\mathsf{human}$ being added to $\mathcal{L}(x_0)$ (because adding $\neg\mathsf{human}$ causes a clash), and application of the $\exists$-rule leads to the creation of a new node x_1 with $\mathcal{L}(x_1) = \{\mathsf{human}, \mathcal{M}\}$. As $\mathcal{L}(x_1)$ is equal to the initial $\mathcal{L}(x_0)$, the same expansion rules will be applied and the process will continue indefinitely.

In order to ensure termination, the algorithm uses a more sophisticated control strategy called blocking [BDS93]. To describe blocking it is necessary to introduce some new terms:

- A *generating rule* is a tableaux expansion rule which extends the tree **T** by adding a new node or nodes; in the case of $\mathcal{ALC}$, the $\exists$-rule is the only generating rule.
- The *parent* of a node y is the node x such that $\langle x, y \rangle$ is an edge in **T**.
- *ancestor* is the transitive closure of parent.

Blocking imposes a new condition on generating rules: the rule can only be applied to a node y if it has no ancestor node x such that $\mathcal{L}(y) \subseteq \mathcal{L}(x)$ [BBH96]. If a node does not meet this condition it is said to be *blocked* and the ancestor node x is called the *blocking node*. When a node is blocked, it is being identified with the blocking node and the tree is describing a cyclical model. Intuitively, it can be seen that termination is now guaranteed because a finite terminology can only produce a finite number of different concept expressions and therefore a finite number of different labelling sets; all nodes must therefore eventually be blocked.

Note that there is no theoretical reason why the blocking node has to be an ancestor node. However, only checking ancestor nodes is sufficient to ensure termination, minimises the number of nodes which must be checked and therefore minimises the number of nodes which must be kept in memory: in practice, fully expanded branches of the tree will be discarded in order to reduce storage requirements (see Section 5.1 on page 78).

A modified $\exists$-rule, which adds $\mathcal{M}$ to new nodes and incorporates blocking, is shown in Table 3.2.

$\exists$-rule	if 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \in \mathcal{L}(y)$
	3.	there is no y s.t. y is an ancestor of x and $\mathcal{L}(x) \subseteq \mathcal{L}(y)$
	then	create a new node y and edge $\langle x, y\rangle$ with $\mathcal{L}(y) = \{C, \mathcal{M}\}$ and $\mathcal{L}(\langle x, y\rangle) = R$

Table 3.2: Modified $\exists$-rule with meta constraint

The mechanism for converting $\mathbf{T}$ into a model which is a witness to the satisfiability of D also needs to be extended to identify blocked nodes with the blocking ancestor nodes:

$$
\begin{aligned}
\Delta^{\mathcal{I}} &= \{x \mid x \text{ is a node in } \mathbf{T} \text{ and } x \text{ is not blocked}\} \\
\mathsf{CN}^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \mathsf{CN} \in \mathcal{L}(x)\} \text{ for all concept names } \mathsf{CN} \text{ in } \mathcal{T} \\
R^{\mathcal{I}} &= \{\langle x, y\rangle \mid \begin{array}[t]{ll} 1. & \langle x, y\rangle \text{ is an edge in } \mathbf{T} \text{ and } \mathcal{L}(\langle x, y\rangle) = R \\ & \text{and } y \text{ is not blocked } \textit{or} \\ 2. & \text{for some } z,\ \langle x, z\rangle \text{ is an edge in } \mathbf{T} \\ & \text{and } \mathcal{L}(\langle x, z\rangle) = R \text{ and } y \text{ blocks } z\} \end{array}
\end{aligned}
$$

The interpretations of concept expressions follow directly from the semantics given in Table 2.3 on page 28.

Example 3.6 Termination Resulting From Blocking

The tableaux expansion of the concept human described in Example 3.5 above, where:

$$
\begin{aligned}
\mathcal{T} &= \{\mathsf{human} \sqsubseteq \exists \mathit{parent}.\mathsf{human}\} \\
\mathcal{M} &= \neg\mathsf{human} \sqcup \exists \mathit{parent}.\mathsf{human}
\end{aligned}
$$

leads to a tree $\mathbf{T}$ containing two nodes x_0 and x_1, and an edge $\langle x_0, x_1\rangle$, with:

$$
\begin{aligned}
\mathcal{L}(x_0) &= \{\mathsf{human}, \neg\mathsf{human} \sqcup \exists \mathit{parent}.\mathsf{human}, \exists \mathit{parent}.\mathsf{human}\} \\
\mathcal{L}(x_1) &= \{\mathsf{human}, \neg\mathsf{human} \sqcup \exists \mathit{parent}.\mathsf{human}, \exists \mathit{parent}.\mathsf{human}\} \\
\mathcal{L}(\langle x_0, x_1\rangle) &= \mathit{parent}
\end{aligned}
$$

Node x_1 is now blocked because x_0 is an ancestor of x_1 and $\mathcal{L}(x_1) \subseteq \mathcal{L}(x_0)$. $\mathbf{T}$ therefore represents a cyclical model which demonstrates the satisfiability of human w.r.t. $\mathcal{T}$:

$$\Delta^{\mathcal{I}} = \{x_0\}$$

$$\begin{aligned}\mathsf{human}^{\mathcal{I}} &= \{x_0\} \\ \mathsf{\mathit{parent}}^{\mathcal{I}} &= \{\langle x_0, x_0\rangle\}\end{aligned}$$

3.3.3 Semi-unfoldable Terminologies

Testing satisfiability with respect to a general terminology by converting all the axioms into GCIs is extremely inefficient: each GCI causes a disjunction to be added to the label of every node, leading to an exponential increase in the number of possible expansions which might be explored by the $\sqcup$-rule. A more efficient technique is to divide $\mathcal{T}$ into an unfoldable part $\mathcal{T}_{\mathbf{U}}$ and a general part $\mathcal{T}_{\mathbf{G}}$ such that $\mathcal{T} = \mathcal{T}_{\mathbf{U}} \cup \mathcal{T}_{\mathbf{G}}$. The unfoldable part $\mathcal{T}_{\mathbf{U}}$ can be formed by taking from $\mathcal{T}$ at most one acyclic introduction axiom (see Section 2.3.2 on page 37) for each concept name CN which appears in $\mathcal{T}$; i.e., an axiom of the form $\mathsf{CN} \doteq C$ or $\mathsf{CN} \sqsubseteq C$ where CN is not used either directly or indirectly in C. $\mathcal{T}_{\mathbf{G}}$ is then defined as $\mathcal{T} - \mathcal{T}_{\mathbf{U}}$. Any terminology can be divided in this way although it is, of course, possible that $\mathcal{T}_{\mathbf{U}}$ will be empty and that $\mathcal{T}_{\mathbf{G}} = \mathcal{T}$. We will call the terminology $\mathcal{T}_{\mathbf{U}} \cup \mathcal{T}_{\mathbf{G}}$ a *semi-unfoldable* terminology.

The satisfiability of a concept expression D with respect to a semi-unfoldable terminology $\mathcal{T}_{\mathbf{U}} \cup \mathcal{T}_{\mathbf{G}}$ can be tested using the modified algorithm described above by forming the meta constraint from $\mathcal{T}_{\mathbf{G}}$ and unfolding both $\mathcal{M}$ and D w.r.t. $\mathcal{T}_{\mathbf{U}}$ so that they contain only primitive concept names. The trees constructed by the algorithm will be restricted to those which represent models satisfying $\mathcal{T}$: the meta constraint will ensure that the general part $\mathcal{T}_{\mathbf{G}}$ is satisfied and any interpretation of primitive concepts leads, via the semantics, to an interpretation of concept expressions and non-primitive concepts which satisfies $\mathcal{T}_{\mathbf{U}}$.

Chapter 4

The $\mathcal{ALCH}_{R^+}$ Description Logic

This chapter describes $\mathcal{ALCH}_{R^+}$, a DL which augments $\mathcal{ALC}$ with transitively closed primitive roles and primitive role introduction axioms. An algorithm for deciding the satisfiability of $\mathcal{ALCH}_{R^+}$ concept expressions is presented, along with a proof of its soundness and completeness. An extension of the algorithm to deal with $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, a DL which augments $\mathcal{ALCH}_{R^+}$ with attributes, will also be described.

The $\mathcal{ALCH}_{R^+}$ algorithm generalises an algorithm for $\mathcal{ALC}_{\oplus}$ which is presented in [Sat96]. $\mathcal{ALC}_{\oplus}$ also augments $\mathcal{ALC}$ with transitively closed primitive roles but only supports a very limited form of primitive role introduction axiom by associating each non-transitive role with a subsuming transitive role (its *transitive orbit*). The soundness and completeness proof for the $\mathcal{ALCH}_{R^+}$ algorithm is adapted and extended from a proof of the soundness and completeness of a satisfiability testing algorithm for $\mathcal{ALC}_{R^+}$ [Sat96], a DL which augments $\mathcal{ALC}$ with transitively closed primitive roles but does not support primitive role introduction axioms. The soundness and completeness of the $\mathcal{ALCH}_{R^+}$ algorithm also implicitly proves the soundness and completeness of the $\mathcal{ALC}_{\oplus}$ algorithm, which had not previously been demonstrated, as the $\mathcal{ALC}_{\oplus}$ algorithm can be seen as a special case of the $\mathcal{ALCH}_{R^+}$ algorithm.

The chapter is organised as follows: Section 4.1 discusses the relationship between $\mathcal{ALCH}_{R^+}$ and other transitive extensions to $\mathcal{ALC}$; Section 4.2 describes a tableaux algorithm for testing the satisfiability of $\mathcal{ALCH}_{R^+}$ concept expressions, and presents a proof of its soundness and completeness; Section 4.3 shows how the algorithm can be used with general terminologies; and finally, Section 4.4 describes how the algorithm can be extended to deal with attributes in the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ DL.

4.1 Transitive Extensions to $\mathcal{ALC}$

Extensions to $\mathcal{ALC}$ which support some form of transitive roles include $\mathcal{CIQ}$, $\mathcal{TSL}$, $\mathcal{ALC}_+$, $\mathcal{ALC}_{R^+}$ and $\mathcal{ALC}_\oplus$ (see Section 2.4.1 on page 39). Of these, $\mathcal{CIQ}$, $\mathcal{TSL}$ and $\mathcal{ALC}_+$ all support role expressions with transitive or transitive reflexive operators, and from correspondence to propositional dynamic logics their satisfiability problems are known to be EXPTIME-complete [Sch91, GL96]. The $\mathcal{ALC}_{R^+}$ and $\mathcal{ALC}_\oplus$ DLs were investigated in the hope that a more restricted form of transitive role might lead to a satisfiability problem in a lower complexity class [Sat96].

$\mathcal{ALC}_{R^+}$ augments $\mathcal{ALC}$ with transitively closed primitive roles: an $\mathcal{ALC}_{R^+}$ terminology may include axioms of the form $R \in \mathbf{R}_+$, where R is a role name and $\mathbf{R}_+$ is the set of transitive roles names in the terminology (see Section 2.2.5 on page 34). In [Sat96] an algorithm for deciding the satisfiability of $\mathcal{ALC}_{R^+}$ concept expressions is presented along with a proof of its soundness and completeness. It is also demonstrated that the complexity of the problem is PSPACE-complete, the same as for $\mathcal{ALC}$ [DLNN95].

$\mathcal{ALC}_\oplus$ extends $\mathcal{ALC}_{R^+}$ by associating each non-transitive role R with its transitive orbit. The transitive orbit of a role R, denoted $R^\oplus$, is a transitive role which subsumes R, and could be defined by the axioms $R^\oplus \in \mathbf{R}_+$ and $R \sqsubseteq R^\oplus$. The interpretation of $R^\oplus$ is therefore a superset of the interpretation of the transitive closure of R, i.e., $(R^\oplus)^\mathcal{I} \supseteq (R^+)^\mathcal{I}$. A means of extending the $\mathcal{ALC}_{R^+}$ satisfiability algorithm to deal with $\mathcal{ALC}_\oplus$ concept expressions is described in [Sat96] but the soundness and completeness of the extended algorithm is not proven. Unfortunately it is shown that the complexity of the problem is EXPTIME-complete, the same as for $\mathcal{ALC}_+$ [Sat96].

The relationship between roles and their transitive orbits in $\mathcal{ALC}_\oplus$ is equivalent to allowing a limited form of primitive role introduction axiom to appear in $\mathcal{ALC}_\oplus$ terminologies—the relationship between a role and its transitive orbit is described by a primitive role introduction axiom of the form $R \sqsubseteq R^\oplus$. $\mathcal{ALCH}_{R^+}$ generalises $\mathcal{ALC}_\oplus$ by allowing terminologies to include arbitrary primitive role introduction axioms of the form $R \sqsubseteq S$, where R and S are primitive role names. The subsumption ($\sqsubseteq$) relation defines a partial ordering in $\mathbf{R}$, the set of role names occuring in the terminology: the semantics of the subsumption relation mean that it is reflexive (for all roles R, $R \sqsubseteq R$), antisymmetric (for any two roles R and S, $R \sqsubseteq S$ and $S \sqsubseteq R \Rightarrow R = S$) and transitive (for any three roles R, R' and S, $R \sqsubseteq R'$ and $R' \sqsubseteq S \Rightarrow R \sqsubseteq S$). Like the concept subsumption relation, the role subsumption relation can be stored as a hierarchy, a directed acyclic graph in which each role is linked to its direct *super-roles* (subsuming roles) and *sub-roles* (subsumed roles). The GALEN medical terminology ontology includes transitively closed primitive roles and a role hierarchy, a fragment of which is illustrated in Figure 4.1 on the following page, with the notation $R^{(+)}$ being used to denote transitive roles (i.e., $R \in \mathbf{R}_+$).

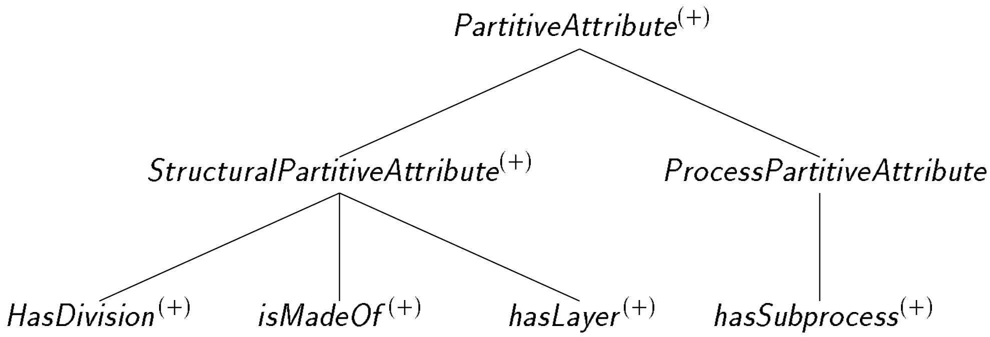

Figure 4.1: A fraction of the GALEN role hierarchy

As it generalises $\mathcal{ALC}_\oplus$, $\mathcal{ALCH}_{R^+}$ is clearly at least as expressive as $\mathcal{ALC}_\oplus$, and although no formal proof is available (see Section 2.4.1 on page 39), $\mathcal{ALCH}_{R^+}$ would appear to be more expressive than $\mathcal{ALC}_\oplus$. Given the roles *son*, *daughter* and *descendant*, $\mathcal{ALCH}_{R^+}$ is able to express the fact that *descendant* is a transitive role which includes both *son* and *daughter* with the axioms $\mathit{descendant} \in \mathbf{R}_+$, $\mathit{son} \sqsubseteq \mathit{descendant}$ and $\mathit{daughter} \sqsubseteq \mathit{descendant}$. This cannot be expressed in $\mathcal{ALC}_\oplus$, which is only able to assert an inclusion relation between a role and its transitive orbit, e.g., $\mathit{son} \sqsubseteq \mathit{son}^\oplus$, $\mathit{daughter} \sqsubseteq \mathit{daughter}^\oplus$ and $\mathit{descendant} \sqsubseteq \mathit{descendant}^\oplus$, where $\{\mathit{son}^\oplus, \mathit{daughter}^\oplus, \mathit{descendant}^\oplus\} \subseteq \mathbf{R}_+$. The role hierarchy allows $\mathcal{ALCH}_{R^+}$ to find subsumption relationships resulting from the interaction of a transitive role and multiple sub-roles, which would not be possible using $\mathcal{ALC}_\oplus$, e.g.:

$$\exists \mathit{daughter}.(\exists \mathit{son}.\mathsf{vegetarian}) \sqsubseteq \exists \mathit{descendant}.\mathsf{vegetarian}$$

It is also clear that $\mathcal{ALC}_+$ ($\mathcal{ALC}$ augmented with union, composition and transitive closure role expressions) is at least as expressive as $\mathcal{ALCH}_{R^+}$. $\mathcal{ALC}_+$ can express a primitive hierarchy identical to that defined in $\mathcal{ALCH}_{R^+}$ by using role union and transitive closure; e.g., the primitive role *descendant* could be represented by the expression $(\mathit{son} \sqcup \mathit{daughter} \sqcup \mathit{descendant})^+$. $\mathcal{ALC}_+$ is also able to express the fact that *descendant* is exactly equal to the transitive closure of $\mathit{son} \sqcup \mathit{daughter}$ by using the expression $(\mathit{son} \sqcup \mathit{daughter})^+$ to represent *descendant*. This allows a subsumption relationship such as:

$$\exists(\mathit{son} \sqcup \mathit{daughter})^+.\top \sqsubseteq \exists \mathit{son}.\top \sqcup \exists \mathit{daughter}.\top$$

to be found, whereas $\mathcal{ALCH}_{R^+}$ will not find the subsumption relationship:

$$\exists \mathit{descendant}.\top \sqsubseteq \exists \mathit{son}.\top \sqcup \exists \mathit{daughter}.\top$$

i.e., in $\mathcal{ALCH}_{R^+}$, having a *descendant* does not imply having either a *son* or a *daughter*, because $\mathit{descendant}^\mathcal{I} \supseteq \mathit{son}^\mathcal{I} \cup \mathit{daughter}^\mathcal{I}$.

As $\mathcal{ALCH}_{R^+}$ is at least as expressive as $\mathcal{ALC}_\oplus$, and no more expressive than $\mathcal{ALC}_+$, the complexity of its satisfiability problem is clearly EXPTIME-complete, the same as for both $\mathcal{ALC}_\oplus$ and $\mathcal{ALC}_+$. However the satisfiability testing algorithm for $\mathcal{ALCH}_{R^+}$ is much simpler than for $\mathcal{ALC}_+$ [Baa90a]: dealing with role expressions in $\mathcal{ALC}_+$ is quite complex (a technique which uses finite state automata is suggested in [Baa90a]) and the blocking strategy for $\mathcal{ALC}_+$ necessitates differentiating between cycles which lead to a model and those which do not. $\mathcal{ALCH}_{R^+}$ is sufficiently expressive to represent the primitive role hierarchy from the GALEN ontology, and the simplicity of its satisfiability testing algorithm means that it is easy to implement and amenable to a wide range of optimisation techniques (see Chapter 5).

4.2 A Tableaux Algorithm for $\mathcal{ALCH}_{R^+}$

In this section a tableaux algorithm for testing the satisfiability of $\mathcal{ALCH}_{R^+}$ concept expressions is described and a proof of its soundness and completeness is presented. $\mathcal{ALCH}_{R^+}$ is the DL obtained by augmenting $\mathcal{ALC}$ with transitively closed primitive roles and primitive role introduction axioms. An $\mathcal{ALCH}_{R^+}$ terminology is defined by the following formation rules:

- Axioms are of the form:

  $$C \sqsubseteq D \mid C \doteq D \mid R \sqsubseteq S \mid R \in \mathbf{R}_+$$

  where C and D are concept expressions, R and S are role names and $\mathbf{R}_+$ is the set of transitive role names.

- Concept expressions are of the form:

  $$\mathsf{CN} \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

  where CN is a concept name, C and D are concept expressions and R is a role name.

To simplify the description of the algorithm, it will be assumed that $\mathbf{R}_+$ and the $\sqsubseteq$ relation have been defined by an $\mathcal{ALCH}_{R^+}$ terminology $\mathcal{T}$ so that:

$$\mathbf{R}_+ = \{R \mid R \in \mathbf{R}_+ \text{ is an axiom in } \mathcal{T}\}$$

and for two roles R and S, $R \sqsubseteq S$ iff $R \sqsubseteq S$ is an axiom in $\mathcal{T}$ or there is a role R' such that $R \sqsubseteq R'$ is an axiom in $\mathcal{T}$ and $R' \sqsubseteq S$. It will also be assumed that concept expressions are fully unfolded and in negation normal form.

The semantics of $\mathcal{ALCH}_{R^+}$ concept expressions are the same as for $\mathcal{ALC}$ (they are described in full in Chapter 2 and summarised in Table 4.1), but $\mathcal{ALCH}_{R^+}$'s role axioms place additional constraints on the interpretation of roles:

Syntax	Semantics
CN	$\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
$\top$	$\Delta^{\mathcal{I}}$
$\bot$	$\emptyset$
$\neg C$	$\Delta^{\mathcal{I}} - C^{\mathcal{I}}$
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\exists R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$
$\forall R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$

Table 4.1: Semantics of $\mathcal{ALCH}_{R^+}$ concept expressions

Definition 4.1 As well as being correct for $\mathcal{ALC}$ concept expressions, an $\mathcal{ALCH}_{R^+}$ interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ must satisfy the additional conditions:

1. if $\langle d, e\rangle \in R^{\mathcal{I}}$ and $\langle e, f\rangle \in R^{\mathcal{I}}$ and $R \in \mathbf{R}_+$, then $\langle d, f\rangle \in R^{\mathcal{I}}$
2. if $R \sqsubseteq S$, then $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$

Like other tableaux algorithms, the $\mathcal{ALCH}_{R^+}$ algorithm tries to prove the satisfiability of a concept expression D by demonstrating a model of D—an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ such that $D^{\mathcal{I}} \neq \emptyset$. The model is represented by a tree whose nodes correspond to individuals in the model, each node being labelled with a set of $\mathcal{ALCH}_{R^+}$-concepts. When testing the satisfiability of an $\mathcal{ALCH}_{R^+}$-concept D, these sets are restricted to subsets of $sub(D)$, where $sub(D)$ is the closure of the subexpressions of D and is defined as follows:

1. if D is of the form $\neg C$, $\exists R.C$ or $\forall R.C$, then C is a subexpression of D, and $sub(D) = \{D\} \cup sub(C)$;
2. if D is of the form $C_1 \sqcap C_2$ or $C_1 \sqcup C_2$, then C_1 and C_2 are subexpressions of D, and $sub(D) = \{D\} \cup sub(C_1) \cup sub(C_2)$;
3. otherwise $sub(D) = \{D\}$.

The soundness and completeness of the algorithm will be proved by showing that it creates a *tableau* for D:

Definition 4.2 If D is an $\mathcal{ALCH}_{R^+}$-concept and $\mathbf{R}_D$ is the set of role names occuring in D, a tableau T for D is defined to be a triple $(\mathbf{S}, \mathcal{L}, \mathcal{E})$ such that: $\mathbf{S}$ is a set of individuals, $\mathcal{L} : \mathbf{S} \to 2^{sub(D)}$ maps each individual to a set of concept expressions which is a subset of $sub(D)$, $\mathcal{E} : \mathbf{R}_D \to 2^{\mathbf{S}\times\mathbf{S}}$ maps each role name occuring in D to a set of pairs of individuals, and there is some individual $s \in \mathbf{S}$ such that $D \in \mathcal{L}(s)$. For all $s \in \mathbf{S}$ it holds that:

1. $\perp \notin \mathcal{L}(s)$, and if $C \in \mathcal{L}(s)$, then $\neg C \notin \mathcal{L}(s)$
2. if $C_1 \sqcap C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ and $C_2 \in \mathcal{L}(s)$
3. if $C_1 \sqcup C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ or $C_2 \in \mathcal{L}(s)$
4. if $\forall R.C \in \mathcal{L}(s)$ and $\langle s, t\rangle \in \mathcal{E}(R)$, then $C \in \mathcal{L}(t)$
5. if $\exists R.C \in \mathcal{L}(s)$, then there is some $t \in \mathbf{S}$ s.t. $\langle s, t\rangle \in \mathcal{E}(R)$ and $C \in \mathcal{L}(t)$
6. if $\forall R.C \in \mathcal{L}(s)$, $\langle s, t\rangle \in \mathcal{E}(R)$ and $R \in \mathbf{R}_+$, then $\forall R.C \in \mathcal{L}(t)$
7. if $R \sqsubseteq S$ then $\mathcal{E}(R) \subseteq \mathcal{E}(S)$

Lemma 4.1 *An $\mathcal{ALCH}_{R^+}$-concept D is satisfiable iff there exists a tableau for D.*

Proof: For the *if* direction, if $T = (\mathbf{S}, \mathcal{L}, \mathcal{E})$ is a tableau for D, a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of D can be defined as:

$$\begin{aligned} \Delta^{\mathcal{I}} &= \mathbf{S} \\ \mathsf{CN}^{\mathcal{I}} &= \{s \mid \mathsf{CN} \in \mathcal{L}(s)\} \text{ for all concept names } \mathsf{CN} \text{ in } sub(D) \\ R^{\mathcal{I}} &= \begin{cases} \mathcal{E}(R)^+ & \text{if } R \in \mathbf{R}_+ \\ \mathcal{E}(R) & \text{otherwise} \end{cases} \end{aligned}$$

where $\mathcal{E}(R)^+$ denotes the transitive closure of $\mathcal{E}(R)$.

By induction on the structure of concepts it can be shown that $\mathcal{I}$ is well defined and that $D^{\mathcal{I}} \neq \emptyset$. For concepts of the form $\neg C$, $C_1 \sqcap C_2$, $C_1 \sqcup C_2$ and $\exists R.C$, the correctness of their interpretations follows directly from Definition 4.2 on the preceding page and the semantics of $\mathcal{ALCH}_{R^+}$ concept expressions given in Table 4.1 above:

1. For concepts of the form $\neg C$, if $\neg C \in \mathcal{L}(s)$, then $C \notin \mathcal{L}(s)$, so $s \in \Delta^{\mathcal{I}} - C^{\mathcal{I}}$ and $\neg C$ is correctly interpreted.
2. For concepts of the form $C_1 \sqcap C_2$, if $C_1 \sqcap C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ and $C_2 \in \mathcal{L}(s)$, so $s \in C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ and $C_1 \sqcap C_2$ is correctly interpreted.
3. For concepts of the form $C_1 \sqcup C_2$, if $C_1 \sqcup C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ or $C_2 \in \mathcal{L}(s)$, so $s \in C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ and $C_1 \sqcup C_2$ is correctly interpreted.
4. For concepts of the form $\exists R.C$, if $\exists R.C \in \mathcal{L}(s)$, then there is some $t \in S$ such that $\langle s, t\rangle \in \mathcal{E}(R)$ and $C \in \mathcal{L}(t)$, so $s \in \{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$ and $\exists R.C$ is correctly interpreted.

For concepts of the form $\forall R.C$, the correctness of their interpretations follows from Definition 4.2, the semantics of $\mathcal{ALCH}_{R^+}$ concept expressions and the additional condition imposed by property 1 in Definition 4.1 on the preceding page:

1. if $\forall R.C \in \mathcal{L}(s)$ and $\langle s, t\rangle \in \mathcal{E}(R)$, then $C \in \mathcal{L}(t)$
2. if $\forall R.C \in \mathcal{L}(s)$, $\langle s, t\rangle \in \mathcal{E}(R)$, $\langle t, u\rangle \in \mathcal{E}(R)$ and $R \in \mathbf{R}_+$, then $\forall R.C \in \mathcal{L}(t)$ and $C \in \mathcal{L}(u)$

so $s \in \{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$ and $\forall R.C$ is correctly interpreted.

Finally, from Definition 4.2, there is some individual $s \in \mathbf{S}$ such that $D \in \mathcal{L}(s)$, so $s \in D^{\mathcal{I}}$ and $D^{\mathcal{I}} \neq \emptyset$.

For the converse, if $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model of D, then a tableau $T = (\mathbf{S}, \mathcal{L}, \mathcal{E})$ for D can be defined as:

$$
\begin{array}{rcl}
S & = & \Delta^{\mathcal{I}} \\
\mathcal{E}(R) & = & R^{\mathcal{I}} \\
\mathcal{L}(s) & = & \{C \in sub(D) \mid s \in C^{\mathcal{I}}\}
\end{array}
$$

It only remains to demonstrate that T is a tableau for D:

1. T satisfies properties 1–5 in Definition 4.2 on page 61 as a direct consequence of the semantics of the $\neg C$, $C_1 \sqcap C_2$, $C_1 \sqcup C_2$, $\forall R.C$ and $\exists R.C$ concept expressions.
2. If $d \in (\forall R.C)^{\mathcal{I}}$, $\langle d, e\rangle \in R^{\mathcal{I}}$ and $R \in \mathbf{R}_+$, then $e \in (\forall R.C)^{\mathcal{I}}$ unless there is some f such that $\langle e, f\rangle \in R^{\mathcal{I}}$ and $f \notin C^{\mathcal{I}}$. However, if $\langle d, e\rangle \in R^{\mathcal{I}}$, $\langle e, f\rangle \in R^{\mathcal{I}}$ and $R \in \mathbf{R}_+$, then from property 1 of Definition 4.1 on page 61 $\langle d, f\rangle \in R^{\mathcal{I}}$ and $d \notin (\forall R.C)^{\mathcal{I}}$. T therefore satisfies property 6 in Definition 4.2.
3. T satisfies property 7 in Definition 4.2 as a direct consequence of property 2 in Definition 4.1. ■

4.2.1 Constructing an $\mathcal{ALCH}_{R^+}$ Tableau

From Lemma 4.1 on the preceding page, an algorithm which constructs a tableau for an $\mathcal{ALCH}_{R^+}$-concept D can be used as a decision procedure for the satisfiability of D. Such an algorithm will now be described in detail.

The algorithm builds a tree where each node x of the tree is labelled with a set $\mathcal{L}(x) \subseteq sub(D)$ and may, in addition, be marked *satisfiable*. The tree is initialised with a single node x_0, where $\mathcal{L}(x_0) = \{D\}$, and expanded either by extending $\mathcal{L}(x)$ for some leaf node x or by adding new leaf nodes. From this tree a tableau for D will be constructed.

For a node x, $\mathcal{L}(x)$ is said to contain a *clash* if it does not satisfy property 1 of Definition 4.2 on page 61, i.e., $\bot \subseteq \mathcal{L}(x)$ or, for some concept C, $\{C, \neg C\} \subseteq \mathcal{L}(x)$. $\mathcal{L}(x)$ is called a *pre-tableau* if it satisfies properties 1–3 of Definition 4.2, i.e., it is clash-free and contains no unexpanded conjunction or disjunction concepts. Note that $\emptyset$ is a pre-tableau.

Edges of the tree are either unlabelled or labelled R for some role name R occuring in $sub(D)$. Unlabelled edges are added when expanding $C_1 \sqcup C_2$ concepts in $\mathcal{L}(x)$ and are the mechanism whereby the algorithm searches possible alternative expansions. Labelled edges are added when expanding $\exists R.C$ terms in $\mathcal{L}(x)$ and correspond to relationships between pairs of individuals.

A node y is called an *R-successor* of a node x if there is an edge $\langle x, y\rangle$ labelled R; y is called a *$\sqcup$-successor* of x if there is a path, consisting of unlabelled edges, from x to y. A node x is an *ancestor* of a node y if there is a path from x to y regardless of the labelling of the edges. Note that both the $\sqcup$-successor and ancestor relations are reflexive as nodes are connected to themselves by the empty path.

The algorithm initialises a tree $\mathbf{T}$ to contain a single node x_0, called the *root* node, with $\mathcal{L}(x_0) = \{D\}$. $\mathbf{T}$ is then expanded by repeatedly applying the rules from Table 4.2 on the following page until either the root node is marked *satisfiable* or none of the rules is applicable. If the root node is marked *satisfiable* then the algorithm returns *satisfiable*; otherwise it returns *unsatisfiable*.

A few remarks about the expansion rules may be useful:

1. The $\exists$-rule incorporates the actions of both the $\exists$ and $\forall$-rules in the $\mathcal{ALC}$ algorithm described in Section 3.2.1 on page 47.

2. The $\exists$-rule deals with transitive roles by propagating $\forall R.C$ labels to R-successors when $R \in \mathbf{R}_+$. It deals with the role hierarchy by processing an $\forall S.C$ label whenever an edge $\langle x, y\rangle$ is added such that $\mathcal{L}(\langle x, y\rangle) = R$ and $R \sqsubseteq S$.

3. Part b of the $\exists$-rule constitutes a blocking strategy, preventing non-termination in cases such as $D = \exists R.C \sqcap \forall R.(\exists R.C)$ and $R \in \mathbf{R}_+$.

4. The combination of the $\sqcup$ and SAT-rules constitute a search of the possible alternative models. As the expansion of a node x cannot affect any of its predecessors, only alternative sub-trees below x are searched and these are represented by $\sqcup$-successors. The SAT-rule marks fully expanded clash-free leaf nodes as *satisfiable* and works back up the tree marking pre-tableau nodes (those with R-successors) *satisfiable* if *all* of their sub-trees were satisfiable and marking search nodes (those with $\sqcup$-successors) *satisfiable* if *any* of their sub-trees were satisfiable.

Several examples of tableaux expansion will be provided in Section 4.2.3 on page 67.

$\sqcap$-rule:	if 1. x is a leaf of $\mathbf{T}$, $\mathcal{L}(x)$ is clash-free, $C_1 \sqcap C_2 \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$ then $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule:	if 1. x is a leaf of $\mathbf{T}$, $\mathcal{L}(x)$ is clash-free, $C_1 \sqcup C_2 \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ then create two $\sqcup$-successors y, z of x with: $\mathcal{L}(y) = \mathcal{L}(x) \cup \{C_1\}$ $\mathcal{L}(z) = \mathcal{L}(x) \cup \{C_2\}$
$\exists$-rule:	if x is a leaf of $\mathbf{T}$ and $\mathcal{L}(x)$ is a pre-tableau then for each $\exists R.C \in \mathcal{L}(x)$ do: a. $\ell_{Rx} := \{C\} \cup \{D \mid \forall S.D \in \mathcal{L}(x) \text{ and } R \sqsubseteq S\}$ $\cup \{\forall S.D \mid \forall S.D \in \mathcal{L}(x),\ S \in \mathbf{R}_+ \text{ and } R \sqsubseteq S\}$ b. if for some ancestor w of x, $\ell_{Rx} \subseteq \mathcal{L}(w)$ then create an R-successor y of x with $\mathcal{L}(y) = \emptyset$ c. otherwise create an R-successor y of x with $\mathcal{L}(y) = \ell_{Rx}$
SAT-rule:	if a node x is not marked *satisfiable* and either: a. $\mathcal{L}(x)$ is a pre-tableau containing no concepts of the form $\exists R.C$ b. $\mathcal{L}(x)$ is a pre-tableau which has successors, and all successors of x are marked *satisfiable* c. $\mathcal{L}(x)$ is not a pre-tableau and some $\sqcup$-successor of x is marked *satisfiable* then mark x *satisfiable*

Table 4.2: Tableaux expansion rules for $\mathcal{ALCH}_{R^+}$

4.2.2 Soundness and Completeness

The soundness and completeness of the algorithm will be demonstrated by proving that, for an $\mathcal{ALCH}_{R^+}$-concept D, it always terminates and that it returns *satisfiable* if and only if D is satisfiable.

Lemma 4.2 *For each $\mathcal{ALCH}_{R^+}$-concept D, the tableau construction terminates.*

Proof: Let $m = |sub(D)|$. As nodes are labelled with subsets of $sub(D)$, $|\mathcal{L}(x)| \leq m$ for all nodes x. For any node x the $\sqcap$-rule can therefore be applied at most m times. The size of any sub-trees is also limited by m: the $\sqcup$-rule can also be applied at most m times along an unlabelled path and the $\exists$-rule can be applied at most 2^m times along any path before there must be some ancestor y s.t. $\ell_{Rx} \subseteq \mathcal{L}(y)$ for any R.

Lemma 4.3 *An $\mathcal{ALCH}_{R^+}$-concept D is satisfiable iff the tableau construction for D returns* satisfiable.

Proof: For the *if* direction (the algorithm returns *satisfiable*), let $\mathbf{T}$ be the tree constructed by the tableaux algorithm for D. A tableau $T = (\mathbf{S}, \mathcal{L}, \mathcal{E})$ can be defined with:

$$
\begin{array}{rcll}
\mathbf{S} & = & \{x \mid & x \text{ is a node in } \mathbf{T} \text{ and } x \text{ is marked } \textit{satisfiable} \\
 & & & \text{and } \mathcal{L}(x) \text{ is a non-empty, pre-tableau.}\} \\
\mathcal{E}(R) & = & \{\langle x, y\rangle \in \mathbf{S} \times \mathbf{S} \mid & 1. \;\; y \text{ is a } \sqcup\text{-successor of an } R\text{-successor of } x \textit{ or} \\
 & & & 2. \;\; x \text{ has an } R\text{-successor } z \text{ with } \mathcal{L}(z) = \emptyset \\
 & & & \text{and } y \text{ is an ancestor of } x \\
 & & & \text{and } \ell_{Rx} \subseteq \mathcal{L}(y) \textit{ or} \\
 & & & 3. \;\; \langle x, y\rangle \in \mathcal{E}(S) \text{ and } S \sqsubseteq R\}
\end{array}
$$

and it can be shown that T is a tableau for D:

1. $D \in \mathcal{L}(x)$ for the root x_0 of $\mathbf{T}$ and for all $\sqcup$-successors of x_0. As x_0 is marked *satisfiable* one of these must be a non-empty pre-tableau marked *satisfiable*, so $D \in \mathcal{L}(s)$ for some $s \in \mathbf{S}$.

2. Properties 1–3 of Definition 4.2 on page 61 are satisfied because each $x \in \mathbf{S}$ is a pre-tableau.

3. Property 4 in Definition 4.2 is satisfied because $\{C \mid \forall R.C \in \mathcal{L}(x)\} \subseteq \ell_{Rx}$ and $\ell_{Rx} \subseteq \mathcal{L}(y)$ for all y with $\langle x, y\rangle \in \mathcal{E}(R)$.

4. Property 5 in Definition 4.2 is satisfied by the $\exists$-rule which, for all $x \in \mathbf{S}$, creates for each $\exists R.C \in \mathcal{L}(x)$ a new R-successor y with either:

   (a) $C \in \mathcal{L}(y)$ *or*

   (b) $\mathcal{L}(y) = \emptyset$, $C \in \ell_{Rx}$ and $\ell_{Rx} \subseteq \mathcal{L}(z)$ for some ancestor z of x.

5. Property 6 in Definition 4.2 is satisfied because $\{\forall R.C \mid \forall R.C \in \mathcal{L}(x)$ and $R \in \mathbf{R}_+\} \subseteq \ell_{Rx}$ and $\ell_{Rx} \subseteq \mathcal{L}(y)$ for all y with $\langle x, y\rangle \in \mathcal{E}(R)$.

6. Property 7 in Definition 4.2 is satisfied because $\langle x, y\rangle \in \mathcal{E}(S)$ for all $\langle x, y\rangle \in \mathcal{E}(R)$ and $R \sqsubseteq S$.

For the converse (the algorithm returns *unsatisfiable*), it can be shown by induction on $h(x)$, the height of the sub-tree below x, that if x is not marked *satisfiable* then the concept $X = \sqcap_{C \in \mathcal{L}(x)} C$ is not satisfiable:

1. If $h(x) = 0$ (x is a leaf) and x is not marked *satisfiable*, then $\mathcal{L}(x)$ contains a clash and X is clearly unsatisfiable.

2. If $h(x) > 0$, $\mathcal{L}(x)$ is not a pre-tableau and x is not marked *satisfiable*, then none of its $\sqcup$-successors is marked *satisfiable*; hence $C_1 \sqcup C_2 \in \mathcal{L}(x)$ and neither y with $\mathcal{L}(y) = \mathcal{L}(x) \cup \{C_1\}$ nor z with $\mathcal{L}(z) = \mathcal{L}(x) \cup \{C_2\}$ is marked *satisfiable*. It follows by induction that X is not satisfiable.

3. If $h(x) > 0$, $\mathcal{L}(x)$ is a pre-tableau and x is not marked *satisfiable*, then there is some R-successor of x which is not marked *satisfiable* and it follows by induction that X is not satisfiable. ∎

4.2.3 Worked Examples

In this section worked examples will be used to illustrate some of the features of the tableaux algorithm.

Example 4.1 Complex Role Interactions

This example illustrates a subsumption inference which results from the interaction of transitive roles and the role hierarchy. Given a terminology $\mathcal{T}$:

$$\{R \sqsubseteq Q, S \sqsubseteq Q, Q \in \mathbf{R}_+\} \subseteq \mathcal{T}$$

the role hierarchy represented by $\mathcal{T}$ is shown in Figure 4.2, with the notation $Q^{(+)}$ being used to denote that $Q \in \mathbf{R}_+$.

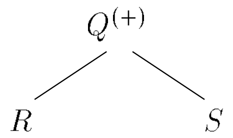

Figure 4.2: The role hierarchy represented by $\mathcal{T}$

It can be shown that $\exists R.(\exists S.C) \sqsubseteq_{\mathcal{T}} \exists Q.C$ by demonstrating that $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$ is unsatisfiable w.r.t. $\mathcal{T}$:

1. Convert $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$ to negation normal form:
$$\exists R.(\exists S.C) \sqcap \forall Q.\neg C$$

2. Initialise $\mathbf{T}$ to contain a single node x_0 labeled:
$$\mathcal{L}(x_0) = \{\exists R.(\exists S.C) \sqcap \forall Q.\neg C\}$$

3. Apply the $\sqcap$-rule to $\exists R.(\exists S.C) \sqcap \forall Q.\neg C \in x_0$:
$$\mathcal{L}(x_0) \longrightarrow \mathcal{L}(x_0) \cup \{\exists R.(\exists S.C), \forall Q.\neg C\}$$

4. Apply the $\exists$-rule to $\exists R.(\exists S.C) \in \mathcal{L}(x_0)$:

   (a) $\ell_{Rx_0} := \{\exists S.C, \neg C, \forall Q.\neg C\}$

(b) There is no ancestor w of x_0 with $\ell_{Rx_0} \subseteq \mathcal{L}(w)$, so create an R-successor x_1 of x_0 with $\mathcal{L}(x_1) = \ell_{Rx_0}$

5. Apply the $\exists$-rule to $\exists S.C \in \mathcal{L}(x_1)$:

   (a) $\ell_{Sx_1} := \{C, \neg C, \forall Q.\neg C\}$

   (b) There is no ancestor w of x_1 with $\ell_{Sx_1} \subseteq \mathcal{L}(w)$, so create an S-successor x_2 of x_1 with $\mathcal{L}(x_2) = \ell_{Sx_1}$

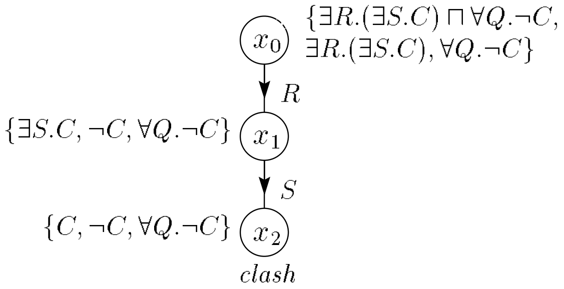

Figure 4.3: Expanded tree $\mathbf{T}$ for $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$

None of the expansion rules are now applicable to $\mathbf{T}$, so it is fully expanded: the fully expanded tree is shown in Figure 4.3. Because $\mathcal{L}(x_2)$ contains a clash ($\{C, \neg C\} \subseteq \mathcal{L}(x_2)$) it is not a pre-tableau and the SAT-rule does not apply. As a result, neither x_1 nor x_0 is marked *satisfiable* and the algorithm returns *unsatisfiable*.

Example 4.2 Blocking

This example illustrates the operation of the blocking mechanism. Given a terminology $\mathcal{T}$:

$$\{R \in \mathbf{R}_+\} \subseteq \mathcal{T}$$

it can be shown that $\exists R.C \not\sqsubseteq_{\mathcal{T}} \exists R.(\forall R.\neg C)$ by demonstrating that $\exists R.C \sqcap \neg\exists R.(\forall R.\neg C)$ is satisfiable w.r.t. $\mathcal{T}$:

1. Convert $\exists R.C \sqcap \neg\exists R.(\forall R.\neg C)$ to negation normal form:
$$\exists R.C \sqcap \forall R.(\exists R.C)$$

2. Initialise $\mathbf{T}$ to contain a single node x_0 labeled:
$$\mathcal{L}(x_0) = \{\exists R.C \sqcap \forall R.(\exists R.C)\}$$

3. Apply the $\sqcap$-rule to $\exists R.C \sqcap \forall R.(\exists R.C) \in x_0$:
$$\mathcal{L}(x_0) \longrightarrow \mathcal{L}(x_0) \cup \{\exists R.C, \forall R.(\exists R.C)\}$$

4. Apply the $\exists$-rule to $\exists R.C \in \mathcal{L}(x_0)$:
   (a) $\ell_{Rx_0} := \{C, \exists R.C, \forall R.(\exists R.C)\}$
   (b) There is no ancestor w of x_0 with $\ell_{Rx_0} \subseteq \mathcal{L}(w)$, so create an R-successor x_1 of x_0 with $\mathcal{L}(x_1) = \ell_{Rx_0}$

5. Apply the $\exists$-rule to $\exists R.C \in \mathcal{L}(x_1)$:
   (a) $\ell_{Rx_1} := \{C, \exists R.C, \forall R.(\exists R.C)\}$
   (b) There is an ancestor x_1 of x_1 (recall that ancestor is reflexive) with $\ell_{Rx_1} \subseteq \mathcal{L}(x_1)$, so create an R-successor x_2 of x_1 with $\mathcal{L}(x_2) = \emptyset$

6. Apply the SAT-rule to x_2: x_2 is not marked *satisfiable* and $\mathcal{L}(x_2)$ is a pre-tableau (recall that $\emptyset$ is a pre-tableau) containing no concepts of the form $\exists R.C$, so mark x_2 *satisfiable*.

7. Apply the SAT-rule to x_1: x_1 is not marked *satisfiable* and $\mathcal{L}(x_1)$ is a pre-tableau with successors, all of which are marked *satisfiable*.

8. Apply the SAT-rule to x_0: x_0 is not marked *satisfiable* and $\mathcal{L}(x_0)$ is a pre-tableau with successors, all of which are marked *satisfiable*.

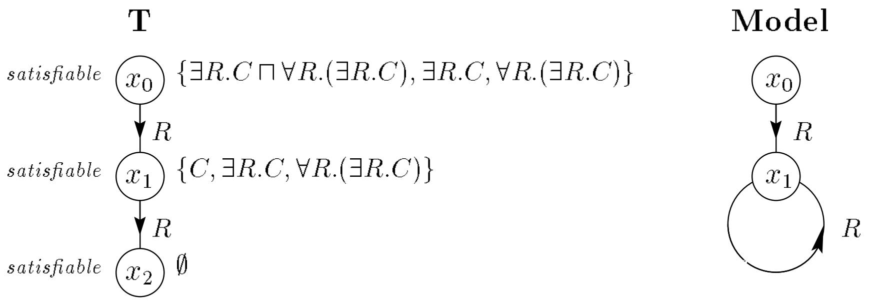

Figure 4.4: Expanded tree **T** and model for $\exists R.C \sqcap \forall R.(\exists R.C)$

None of the expansion rules are now applicable to **T**, so it is fully expanded. Because $\mathcal{L}(x_0)$ is marked *satisfiable*, the algorithm returns *satisfiable*: both the fully expanded **T** and the model which it represents are shown in Figure 4.4. Note that $\mathcal{L}(x_2) = \emptyset$ represents a cycle in the model. The model represented by **T** is:

$$
\begin{aligned}
\Delta^{\mathcal{I}} &= \{x_0, x_1\} \\
C^{\mathcal{I}} &= \{x_1\} \\
R^{\mathcal{I}} &= \{\langle x_0, x_1\rangle, \langle x_1, x_1\rangle\}
\end{aligned}
$$

The interpretations of concept expressions follow directly from the semantics given in 4.1 on page 61 and is consistent with the labeling of the nodes in **T**.

Example 4.3 Disjunctions

This example illustrates how the algorithm uses $\sqcup$-successors to search possible alternative models. Given a terminology $\mathcal{T}$ containing no role axioms, it can be shown that $\exists R.(C \sqcup D) \sqcap \forall R.\neg C$ is satisfiable w.r.t. $\mathcal{T}$:

1. Initialise **T** to contain a single node x_0 labeled:
$$\mathcal{L}(x_0) = \{\exists R.(C \sqcup D) \sqcap \forall R.\neg C\}$$
2. Apply the $\sqcap$-rule to $\exists R.(C \sqcup D) \sqcap \forall R.\neg C \in x_0$:
$$\mathcal{L}(x_0) \longrightarrow \mathcal{L}(x_0) \cup \{\exists R.(C \sqcup D), \forall R.\neg C\}$$
3. Apply the $\exists$-rule to $\exists R.(C \sqcup D) \in \mathcal{L}(x_0)$:
   (a) $\ell_{Rx_0} := \{C \sqcup D, \neg C\}$
   (b) There is no ancestor w of x_0 with $\ell_{Rx_0} \subseteq \mathcal{L}(w)$, so create an R-successor x_1 of x_0 with $\mathcal{L}(x_1) = \ell_{Rx_0}$
4. Apply the $\sqcup$-rule to $C \sqcup D \in \mathcal{L}(x_1)$: create 2 $\sqcup$-successors x_2 and x_3 with $\mathcal{L}(x_2) = \mathcal{L}(x_1) \cup \{C\}$ and $\mathcal{L}(x_3) = \mathcal{L}(x_1) \cup \{D\}$.
5. Apply the SAT-rule to x_3: x_3 is not marked *satisfiable* and $\mathcal{L}(x_3)$ is a pre-tableau containing no concepts of the form $\exists R.C$, so mark x_3 *satisfiable.*
6. Apply the SAT-rule to x_1: x_1 is not marked *satisfiable*, $\mathcal{L}(x_1)$ is a not pre-tableau and x_1 has a successor (x_3) which is marked *satisfiable.*
7. Apply the SAT-rule to x_0: x_0 is not marked *satisfiable* and $\mathcal{L}(x_0)$ is a pre-tableau with successors, all of which are marked *satisfiable.*

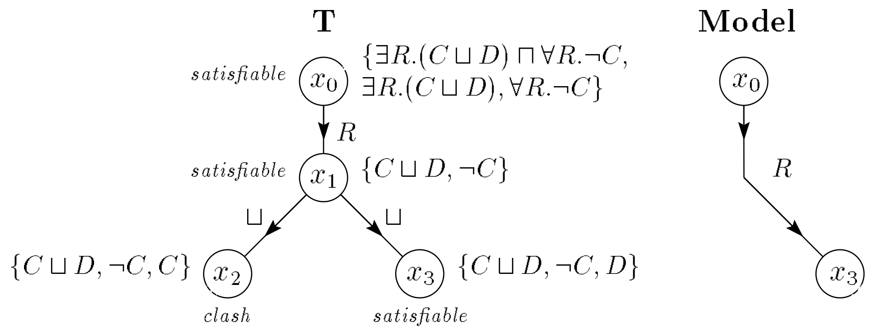

Figure 4.5: Expanded tree **T** and model for $\exists R.(C \sqcup D) \sqcap \forall R.\neg C$

None of the expansion rules are now applicable to **T**, so it is fully expanded. Because $\mathcal{L}(x_0)$ is marked *satisfiable*, the algorithm returns *satisfiable*: both the fully expanded **T** and the model which it represents are shown in Figure 4.5. The model represented by **T** is:

$$\begin{array}{rcl} \Delta^{\mathcal{I}} & = & \{x_0, x_3\} \\ C^{\mathcal{I}} & = & \emptyset \\ D^{\mathcal{I}} & = & \{x_3\} \\ R^{\mathcal{I}} & = & \{\langle x_0, x_3 \rangle\} \end{array}$$

The interpretations of concept expressions follow directly from the semantics given in 4.1 on page 61 and is consistent with the labeling of the nodes in **T**.

4.3 Internalising GCIs

The algorithm described above tests the satisfiability of a single unfolded concept expression but it can, without modification, be used to test satisfiability with respect to a general terminology $\mathcal{T}$, one which may contain cycles, concept equations and general concept inclusion axioms (GCIs).

It has already been shown (in Section 3.3.1 on page 52) how an arbitrary set of concept axioms in $\mathcal{T}$ can be converted into a single concept expression $\mathcal{M}$ which must be satisfied by every individual in a model which satisfies $\mathcal{T}$. This condition can be imposed on individuals in a model of an $\mathcal{ALCH}_{R^+}$ concept expression D using a procedure called *internalisation* [Baa90a]. Internalisation works by testing the satisfiability of a new concept expression $D \sqcap \mathcal{M} \sqcap \forall U.\mathcal{M}$ which incorporates (internalises) both $\mathcal{M}$ and D. The role U is a specially defined transitive role which subsumes all the other roles in $\mathbf{R}$ (the set of role names occuring in $\mathcal{T}$):

$$U \in \mathbf{R}_+ \text{ is an axiom in } \mathcal{T} \text{ and for all } R \in \mathbf{R},\ R \sqsubseteq U \text{ is an axiom in } \mathcal{T}$$

Applying the $\sqcap$-rule to $D \sqcap \mathcal{M} \sqcap \forall U.\mathcal{M} \in \mathcal{L}(x_0)$ will lead to $\{\mathcal{M}, \forall U.\mathcal{M}\} \subseteq \mathcal{L}(x_0)$, and due to the properties of the $\sqcup$-rule and the $\exists$-rule, all of x_0's successors will also have $\mathcal{M}$ in their labels.

As discussed in Section 3.3.3 on page 56, dealing with a general terminology $\mathcal{T}$ in this way is extremely inefficient and it is preferable to convert $\mathcal{T}$ into a semi-unfoldable terminology by dividing it into an unfoldable part $\mathcal{T}_{\mathbf{U}}$ and a general part $\mathcal{T}_{\mathbf{G}}$ such that $\mathcal{T} = \mathcal{T}_{\mathbf{U}} \cup \mathcal{T}_{\mathbf{G}}$. $\mathcal{M}$ can then be formed from $\mathcal{T}_{\mathbf{G}}$ and both $\mathcal{M}$ and D unfolded with respect to $\mathcal{T}_{\mathbf{U}}$ before testing the satisfiability of $D \sqcap \mathcal{M} \sqcap \forall U.\mathcal{M}$.

4.4 $\mathcal{ALCH}_{R^+}$ Extended with Attributes

In order to provide a DL whose expressive possibilities are closer to those of the GRAIL DL, $\mathcal{ALCH}_{R^+}$ can be extended with limited support for attributes (functional roles) to give $\mathcal{ALCH}\boldsymbol{f}_{R^+}$. Unlike $\mathcal{ALCF}$ [HN90], $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ does not include support for attribute value map concept forming operators: this is in line with GRAIL and only requires a small extension to the algorithm.

$\mathcal{ALCH}\boldsymbol{f}_{R^+}$ extends the syntax of $\mathcal{ALCH}_{R^+}$ by allowing axioms of the form:

$$A \in \mathbf{F}$$

to appear in terminologies, and by supporting concept expressions of the form:

$$\exists A.C \mid \forall A.C$$

where A is an attribute name, C is a concept expression and $\mathbf{F}$ is the set of attribute names.

As before, it will be assumed that $\mathbf{F}$, $\mathbf{R}_+$ and the $\sqsubseteq$ relation have been defined by an $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ terminology $\mathcal{T}$. The set of attribute names $\mathbf{F}$ is defined by:

$$\mathbf{F} = \{A \mid \begin{array}{ll} 1. & A \in \mathbf{F} \text{ is an axiom in } \mathcal{T} \textit{ or} \\ 2. & A \sqsubseteq B \text{ is an axiom in } \mathcal{T} \\ & \text{and } B \in \mathbf{F}\} \end{array}$$

Note that any role which has an attribute as a super-role must itself be an attribute: if $B^{\mathcal{I}}$ is functional and $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ then $A^{\mathcal{I}}$ must also be functional.

In $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, the set of transitive roles is made disjoint from the set of attributes ($\mathbf{R}_+ \cap \mathbf{F} = \emptyset$) by making a small alteration to the definition of $\mathbf{R}_+$ given in section 4.2 above, so that:

$$\mathbf{R}_+ = \{R \mid R \in \mathbf{R}_+ \text{ is an axiom in } \mathcal{T} \text{ and } R \notin \mathbf{F}\}$$

This restriction does not seem unreasonable: in general, the transitive closure of an attribute is a role [BHH+91], and it is difficult to imagine the meaning of a transitively closed attribute. The restriction also greatly simplifies the algorithm because, if transitive roles and attributes are not disjoint, then chains of A-successors for a transitive attribute A (i.e., $A \in \mathbf{F}$ and $A \in \mathbf{R}_+$) would have to be collapsed:

$$A^{\mathcal{I}}(x) = y \text{ and } A^{\mathcal{I}}(y) = z \;\;\Rightarrow\;\; A^{\mathcal{I}}(x) = z \;\;\Rightarrow\;\; y = z$$

The semantics of $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ concept expressions are the same as for $\mathcal{ALCH}_{R^+}$, extended to deal with $\exists A.C$ and $\forall A.C$ expressions as shown in Table 4.3 on the following page, but $\mathcal{ALCH}\boldsymbol{f}_{R^+}$'s attribute axioms place an additional constraint on $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ interpretations:

Definition 4.3 As well as being correct for $\mathcal{ALCH}_{R^+}$ concept expressions, an $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ must satisfy the additional condition that, for all $A \in \mathbf{F}$, $A^{\mathcal{I}}$ is a single valued partial function:

$$A^{\mathcal{I}} : dom\ A^{\mathcal{I}} \longrightarrow \Delta^{\mathcal{I}}$$

Syntax	Semantics
$\exists A.C$	$\{d \in dom\ A^{\mathcal{I}} \mid A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$
$\forall A.C$	$\{d \in \Delta^{\mathcal{I}} \mid d \in dom\ A^{\mathcal{I}} \Rightarrow A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$

Table 4.3: Semantics of $\mathcal{ALCH}\boldsymbol{f}_{R+}$ concept expressions

The $\exists$-rule in the $\mathcal{ALCH}_{R+}$ tree construction algorithm can be extended to deal with attributes in $\mathcal{ALCH}\boldsymbol{f}_{R+}$. Expressions of the form $\exists R.C$, where R is a role, are dealt with exactly as before, but expressions of the form $\exists A.C$, where A is an attribute, require special treatment. The extended rule treats attributes in a similar way to roles: $\exists A.C$ expressions in the label of a pre-tableau node x will lead to the creation of new $\mathbf{A}$-successor nodes y_i and labelled edges $\langle x, y_i \rangle$. However, it may group together multiple $\exists A.C$ expressions in x's label to create a single $\mathbf{A}$-successor y, labeling the edge $\langle x, y \rangle$ with a set of attribute names $\mathbf{A}$.

Multiple $\exists A.C$ expressions must be grouped together when, in the model represented by the tree, the $A^{\mathcal{I}}(x)$ are constrained to be the same individual, for example when there are multiple $\exists A.C$ expressions containing the same attribute A. The interaction between attributes and the role hierarchy (defined by the $\sqsubseteq$ relation) means that for two expressions $\exists A.C_1 \in \mathfrak{L}(x)$ and $\exists B.C_2 \in \mathfrak{L}(x)$, $A^{\mathcal{I}}(x)$ and $B^{\mathcal{I}}(x)$ are also constrained to be the same individual when $A \sqsubseteq B$ (because $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$) or $B \sqsubseteq A$ (because $B^{\mathcal{I}} \subseteq A^{\mathcal{I}}$). We will say that an attribute B is *directly-constrained* by an attribute A in $\mathfrak{L}(x)$ if:

$$(\exists A.C \in \mathfrak{L}(x) \text{ and } A \sqsubseteq B) \text{ or } (\exists B.C \in \mathfrak{L}(x) \text{ and } B \sqsubseteq A)$$

and we will say that an attribute B is *constrained* by an attribute A in $\mathfrak{L}(x)$ if B is directly-constrained by A in $\mathfrak{L}(x)$ or if, for some attribute A', A' is directly-constrained by A in $\mathfrak{L}(x)$ and B is constrained by A' in $\mathfrak{L}(x)$. For an attribute B and a node x, the set of attributes which are constrained by B in $\mathfrak{L}(x)$ will be denoted $\mathbf{A}_{Bx}$:

$$\mathbf{A}_{Bx} = \{A \mid A \text{ is constrained by } B \text{ in } \mathfrak{L}(x)\}$$

The extended $\exists$-rule for $\mathcal{ALCH}\boldsymbol{f}_{R+}$ is shown in Table 4.4 on the following page.

Note that:

1. The $\mathcal{ALCH}\boldsymbol{f}_{R+}$ extension is orthogonal to $\mathcal{ALCH}_{R+}$: for all concept expressions not containing attributes, the trees constructed by the two algorithms will be identical.

2. The treatment of roles and attributes is sufficiently similar that, for most purposes, it is convenient not to distinguish between the two. In the remainder of this thesis $\exists R.C$ and $\forall R.C$ expressions will therefore be taken to refer to either the role or attribute case unless it is explicitly stated that they refer only to roles.

if	x is a leaf of $\mathbf{T}$ and $\mathcal{L}(x)$ is a pre-tableau
then	for each $\exists R.C \in \mathcal{L}(x)$ where $R \notin \mathbf{F}$ do:
	a. $\ell_{Rx} := \{C\} \cup \{D \mid \forall S.D \in \mathcal{L}(x) \text{ and } R \sqsubseteq S\}$ $\cup \{\forall S.D \mid \forall S.D \in \mathcal{L}(x),\ S \in \mathbf{R}_+ \text{ and } R \sqsubseteq S\}$
	b. if for some ancestor w of x, $\ell_{Rx} \subseteq \mathcal{L}(w)$ then create an R-successor y of x with $\mathcal{L}(y) = \emptyset$
	c. otherwise create an R-successor y of x with $\mathcal{L}(y) = \ell_{Rx}$
and	for each $\exists A.D \in \mathcal{L}(x)$ where $A \in \mathbf{F}$ do:
	a. if for some $\mathbf{A}$-successor y of x, $A \in \mathbf{A}$ then do nothing.
	b. otherwise
	i. $\mathbf{A} := \mathbf{A}_{Ax}$
	ii. $\ell_{Ax} := \bigcup_{B \in \mathbf{A}} (\{C \mid \exists B.C \in \mathcal{L}(x)\} \cup \{C \mid \forall S.C \in \mathcal{L}(x) \text{ and } B \sqsubseteq S\} \cup \{\forall S.C \mid \forall S.C \in \mathcal{L}(x),\ S \in \mathbf{R}_+ \text{ and } B \sqsubseteq S\})$
	iii. if for some ancestor w of x, $\ell_{Ax} \subseteq \mathcal{L}(w)$ then create an $\mathbf{A}$-successor y of x with $\mathcal{L}(y) = \emptyset$
	iv. otherwise create an $\mathbf{A}$-successor y of x with $\mathcal{L}(y) = \ell_{Ax}$

Table 4.4: Extended $\exists$-rule for $\mathcal{ALCH}\boldsymbol{f}_{R^+}$

3. In the context of $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, the first part of the SAT-rule, which states that a node x can be marked satisfiable if $\mathcal{L}(x)$ is a pre-tableau containing no concepts of the form $\exists R.C$, is taken to refer to concepts where R is either a role or an attribute.

When demonstrating the satisfiability of an $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ concept expression D, the algorithm will construct a tree $\mathbf{T}$. If D contains disjunctive terms, then $\mathbf{T}$ may represent multiple possible models, as illustrated in Figure 4.6 on the following page for $D = \exists A.(C \sqcup D)$. In $\mathcal{ALCH}_{R^+}$ the model obtained by merging all these possibilities together is still valid, but in $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ the merged model can lead to a violation of the additional constraint in Definition 4.3 on page 72 which states that if A is an attribute, then its interpretation must be a single valued partial function.

Before using $\mathbf{T}$ to derive an $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ tableau, multiple possible models must therefore be eliminated by deleting all but one of the successors of any node which has $\sqcup$-successors, and by deleting all of the successors of deleted $\sqcup$-successors. The tableau construction described in the proof of Lemma 4.3 on page 66 can then be extended to deal with $\mathbf{A}$-successors in $\mathbf{T}$:

$$\mathbf{S} = \{x \mid x \text{ is a node in } \mathbf{T} \text{ and } x \text{ is marked } \textit{satisfiable} \text{ and } \mathcal{L}(x) \text{ is a non-empty, pre-tableau.}\}$$

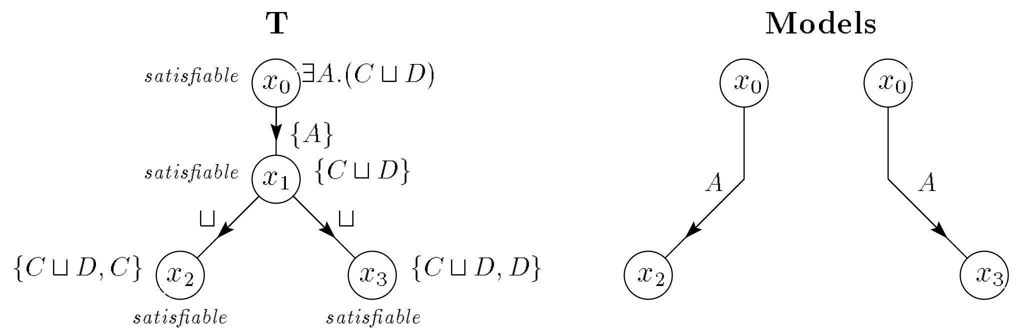

Figure 4.6: Expanded tree **T** and models for $\exists A.(C \sqcup D)$

$$
\begin{array}{rcll}
\mathcal{E}(R) & = & \{\langle x,y\rangle \in \mathbf{S}\times\mathbf{S} \mid & 1.\ \ y \text{ is a } \sqcup\text{-successor of an } R\text{-successor of } x \textit{ or} \\
 & & & 2.\ \ x \text{ has an } R\text{-successor } z \text{ with } \mathcal{L}(z) = \emptyset \\
 & & & \quad\ \ \text{and } y \text{ is an ancestor of } x \\
 & & & \quad\ \ \text{and } \ell_{Rx} \subseteq \mathcal{L}(y) \textit{ or} \\
 & & & 3.\ \ \langle x,y\rangle \in \mathcal{E}(S) \text{ and } S \sqsubseteq R \textit{ or} \\
 & & & 4.\ \ \langle x,y\rangle \in \mathcal{E}(A) \text{ and } A \sqsubseteq R\} \\
\mathcal{E}(A) & = & \{\langle x,y\rangle \in \mathbf{S}\times\mathbf{S} \mid & 1.\ \ y \text{ is a } \sqcup\text{-successor of an } \mathbf{A}\text{-successor of } x \\
 & & & \quad\ \ \text{and } A \in \mathbf{A} \textit{ or} \\
 & & & 2.\ \ x \text{ has an } \mathbf{A}\text{-successor } z \text{ with } \mathcal{L}(z) = \emptyset \\
 & & & \quad\ \ \text{and } y \text{ is an ancestor of } x \text{ and } \ell_{Ax} \subseteq \mathcal{L}(y) \\
 & & & \quad\ \ \text{and } A \in \mathbf{A}\}
\end{array}
$$

Example 4.4 Attribute Interactions

This example illustrates a subsumption inference which results from a subtle interaction of attributes and the role hierarchy. Given a terminology $\mathcal{T}$:

$$\{B_1 \sqsubseteq A, B_2 \sqsubseteq A, A \in \mathbf{F}, B_1 \in \mathbf{F}, B_2 \in \mathbf{F}\} \subseteq \mathcal{T}$$

the role hierarchy represented by $\mathcal{T}$ is shown in Figure 4.7.

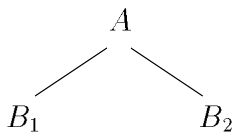

Figure 4.7: The role hierarchy represented by $\mathcal{T}$

It can be shown that $\exists B_1.C \sqsubseteq_{\mathcal{T}} \forall B_2.C$ by demonstrating that $\exists B_1.C \sqcap \neg\forall B_2.C$ is unsatisfiable w.r.t. $\mathcal{T}$:

1. Convert $\exists B_1.C \sqcap \neg\forall B_2.C$ to negation normal form:
$$\exists B_1.C \sqcap \exists B_2.\neg C$$

2. Initialise $\mathbf{T}$ to contain a single node x_0 labeled:
$$\mathcal{L}(x_0) = \{\exists B_1.C \sqcap \exists B_2.\neg C\}$$

3. Apply the $\sqcap$-rule to $\exists B_1.C \sqcap \exists B_2.\neg C \in \mathcal{L}(x_0)$:
$$\mathcal{L}(x_0) \longrightarrow \mathcal{L}(x_0) \cup \{\exists B_1.C, \exists B_2.\neg C\}$$

4. Apply the $\exists$-rule to $\exists B_1.C \in \mathcal{L}(x_0)$:

   (a) There is no $\mathbf{A}$-successor y of x_0 with $B_1 \in \mathbf{A}$ so:

      i. $\mathbf{A} := \mathbf{A}_{B_1 x_0}$, the set of attributes which are constrained by B_1 in $\mathcal{L}(x_0)$. Both B_1 and A are directly constrained by B_1 because $\exists B_1.C \in \mathcal{L}(x_0)$, $B_1 \sqsubseteq B_1$ and $B_1 \sqsubseteq A$; B_2 is directly constrained by A because $\exists B_2.\neg C \in \mathcal{L}(x_0)$ and $B_2 \sqsubseteq A$; B_1 and A are constrained by B_1 because they are directly constrained by B_1, and B_2 is constrained by B_1 because A is directly constrained by B_1 and B_2 is constrained by A. Therefore:
$$\mathbf{A}_{B_1 x_0} = \{B_1, A, B_2\}$$

      ii. $\ell_{B_1 x_0} := \{C, \neg C\}$

      iii. There is no ancestor w of x_0 with $\ell_{B_1 x_0} \subseteq \mathcal{L}(w)$, so create an $\mathbf{A}$-successor x_1 of x_0 with $\mathcal{L}(x_1) = \ell_{B_1 x_0}$

5. Apply the $\exists$-rule to $\exists B_2.\neg C \in \mathcal{L}(x_0)$:

   (a) There is an $\mathbf{A}$-successor x_1 of x_0 with $B_2 \in \mathbf{A}$ so do nothing.

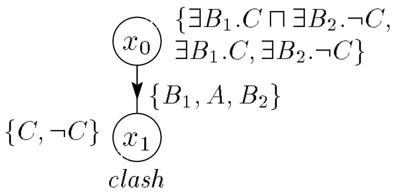

Figure 4.8: Expanded tree $\mathbf{T}$ for $\exists B_1.C \sqcap \exists B_2.\neg C$

None of the expansion rules are now applicable to $\mathbf{T}$, so it is fully expanded: the fully expanded tree is shown in Figure 4.8. Because $\mathcal{L}(x_1)$ contains a clash ($\{C, \neg C\} \subseteq \mathcal{L}(x_1)$) it is not a pre-tableau and the SAT-rule does not apply. As a result, x_0 is not marked *satisfiable* and the algorithm returns *unsatisfiable*.

Chapter 5

Optimising Tableaux Algorithms

In addition to dealing with specific queries, a DL classifier uses its subsumption testing algorithm to compute a concept hierarchy, a partial ordering of named concepts based on the subsumption relation. Techniques for improving the performance of a DL classifier therefore divide naturally into two categories: those which try to optimise the subsumption testing algorithm and those which try to reduce the number of subsumption tests required to compute the partial ordering [BHNP92]. This chapter describes techniques which can be used to optimise tableaux subsumption testing algorithms in general and the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm in particular; techniques for reducing the number of subsumption tests required to compute the partial ordering are described in Chapter 6.

To improve the performance of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ subsumption testing algorithm, a range of known, adapted and novel optimisations have been employed. It has been suggested [BHNP92] that such techniques can be further sub-divided into those which try to avoid general subsumption tests by using a cheaper test (typically a structural comparison) and those which try to improve the performance of the general subsumption testing algorithm, but this distinction is blurred by some of the novel techniques described here (e.g., caching) which operate at both levels. Of the techniques described, those which are believed to be novel, or whose adaption to tableaux subsumption testing is novel, include:

- Lexically normalising and encoding concept expressions—a novel technique which detects structurally obvious satisfiability and unsatisfiability as well as enhancing both the efficiency and effectiveness of other optimisations.

- Absorbing GCI axioms into primitive concept introduction axioms—a novel technique which can eliminate most GCIs (in the case of the Grail DL, all GCIs) from a terminology.

- Semantic branching—a search technique adapted from the Davis-Putnam-Logemann-Loveland procedure (DPL) commonly used to solve propositional satisfiability (SAT) problems [DLL62]. A similar technique is also used in the KSAT algorithm [GS96a].

- Dependency directed backtracking—a technique adapted from constraint satisfiability problem solving [Bak95].

- Caching and re-using partial models—a novel technique which takes advantage of the repetitive structure of the satisfiability problems generated by terminological classification.

The chapter is organised as follows: Sections 5.1 and 5.2 describe known techniques for reducing the algorithm's storage requirements and for optimising tableaux subsumption testing; Sections 5.3–5.7 describe the optimisation techniques listed above, which have been developed in this thesis; and finally, Section 5.8 discusses interactions between the various techniques.

5.1 Reducing Storage Requirements

When considering the performance of an algorithm, its execution time and its storage requirement may both be significant factors. In practice, execution time is the limiting factor as regards the usefulness of tableaux satisfiability testing algorithms (see Section 7.2 on page 111), and minimising execution time is the primary objective of the optimisation techniques developed in this thesis. However, it is still sensible to use an improved search technique which reduces the storage requirements of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm: storage requirements may become more significant if the optimised algorithm can be used to solve sufficiently large problems, and the improved search technique can also reduce execution time.

Implementing the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm exactly as described in the theoretical presentation in Chapter 4 would involve a very large storage requirement as all the possible models explored are represented in the tree which is constructed by the algorithm. The algorithm's storage requirement can be reduced by modifying the way in which successor nodes are explored. With less expressive DLs, a trace technique can be used which discards the labels of all fully expanded nodes [HN90], but this is not possible when access to the labels of predecessor nodes is required by the blocking mechanism. It is possible, however, to reduce storage requirements by performing a depth first search of a node's successors and discarding fully expanded sub-trees. For example, if applying the $\exists$-rule to a pre-tableau node x generates several R-successors and $\mathbf{A}$-successors, these can be expanded one at a time, with the sub-tree generated by each expansion being discarded as soon as it is complete. The search of is halted, and x shown to be unsatisfiable, if any of its successors proves unsatisfiable; x is marked satisfiable if all its successors were satisfiable.

Similarly, the $\sqcup$-successors of a node y can be explored one at a time and discarded when fully expanded. In this case the search is halted, and y marked satisfiable, if any of its successors proves satisfiable; y is unsatisfiable if all its successors were unsatisfiable.

Depth first search can also reduce execution time, because there is no wasted expansion of nodes whose satisfiability cannot affect the satisfiability of their ancestor nodes. Without depth first search there is nothing, for example, to prevent the expansion of an R-successor of a node when another R-successor has already been shown to be unsatisfiable, or the expansion of a $\sqcup$-successor of a node when another $\sqcup$-successor has already been shown to be satisfiable.

5.2 Lazy Unfolding

Although theoretical descriptions of tableaux algorithms generally assume that the concept expression to be tested is fully unfolded, in practice it is usual to unfold the expression only as required by the progress of the algorithm. For example, when testing the satisfiability of an expression $\exists R.\mathsf{CN}$, where CN is a concept name, the unfolding of CN can be delayed until the $\exists$-rule has created an R-successor y with $\mathcal{L}(y) = \{\mathsf{CN}\}$. At this point the negation normal form of CN's definition can be substituted for CN in $\mathcal{L}(y)$. This method, which will be called *lazy unfolding*, is more efficient as it can avoid wasted unfolding in sub-trees which are not expanded by the depth first search described in the previous section.

It has been shown (in the KRIS system) that a significant improvement in performance can be obtained simply by leaving concept names in node labels and adding, instead of substituting, their definitions when they are lazily unfolded [BHNP92]. This is because obvious contradictions can often be detected much earlier by comparing concept names rather than their unfolded definitions. This can save a great deal of wasted expansion if the unfolded definitions are large and complex.

Example 5.1 Contradiction Detection Due To Lazy Unfolding

When testing the satisfiability of the concept expression:

$$\exists R.\mathsf{CN} \sqcap \forall R.\neg\mathsf{CN}$$

the optimised algorithm will detect a contradiction as soon it creates an R-successor y because $\{\mathsf{CN}, \neg\mathsf{CN}\} \subseteq \mathcal{L}(y)$. This could save a lot of wasted work if unfolding CN produces a large and complex expression.

5.3 Normalisation and Encoding

In realistic terminologies, large and complex concepts are seldom described monolithically, but are built up from a hierarchy of named concepts whose descriptions are less complex. The lazy unfolding optimisation described above can use this structure to provide more rapid detection of inconsistencies. The hierarchical structuring of terminologies, and thus the effectiveness of the lazy unfolding optimisation, can be taken to its logical conclusion by lexically normalising and encoding all concept expressions and, recursively, their sub-expressions so that:

1. All sub-expressions are named concepts; e.g., $\exists R.(C_1 \sqcap C_2)$ would be encoded as $\exists R.D$ where $D \doteq C_1 \sqcap C_2$.
2. All concept expressions are in a standard form; e.g., all exists restrictions ($\exists R.C$ expressions) are converted to value restrictions ($\forall R.C$ expressions), so $\exists R.D$ would be normalised to $\neg\forall R.\neg D$.

Adding normalisation (step 2) allows lexically equivalent expressions to be recognised and identically encoded and can also lead to the detection of concept descriptions which are obviously satisfiable or unsatisfiable. A functional definition of the normalisation and encoding process is given in Table 5.1 on the following page.

Note that:

1. Expressions of the form $\exists R.C$ and $\forall R.C$ refer to the case where R is either a role or an attribute.
2. The *Encode* function minimises the number of new concept definitions added to the knowledge base $\mathcal{T}$ by identifying lexically equivalent expressions which have already been encoded. In the case of conjunctive expressions, this is facilitated by sorting the conjuncts and eliminating any redundancies.
3. Non-primitive concepts introduced by the encoding process are are *not* classified in the concept hierarchy. Their function is to act as place-holders or macros, representing the encoded concept expression.

Example 5.2 Detecting Unsatisfiability Via Normalisation and Encoding

Normalising and encoding the expression $\exists R.(C \sqcap E \sqcap \neg D) \sqcap \forall R.(D \sqcup \neg C \sqcup \neg E)$, where C, D and E are all concept names, would proceed as follows (assuming an empty terminology $\mathcal{T}$):

Normalise(C) :

$$
\begin{array}{lll}
C = \text{concept name} & & \\
 & \longrightarrow & C \\
C = \neg D & \longrightarrow & \mathsf{CN} \ \ \textit{if} \ \ \text{Normalise}(D) = \neg\mathsf{CN} \\
 & & \bot \ \ \textit{if} \ \ \text{Normalise}(D) = \top \\
 & & \top \ \ \textit{if} \ \ \text{Normalise}(D) = \bot \\
 & & \textit{otherwise} \ \ \neg\text{Normalise}(D) \\
C = \forall R.D & \longrightarrow & \top \ \ \textit{if} \ \ \text{Normalise}(D) = \top \\
 & & \textit{otherwise} \ \ \text{Encode}(\forall R.\text{Normalise}(D)) \\
C = D_1 \sqcap \ldots \sqcap D_n & & \\
 & \longrightarrow & \bot \ \ \textit{if} \ \ \bot \in \{\text{Normalise}(D_1), \ldots, \text{Normalise}(D_n)\} \\
 & & \bot \ \ \textit{if} \ \ \exists D.(\{D, \neg D\} \subseteq \{\text{Normalise}(D_1), \ldots, \text{Normalise}(D_n)\}) \\
 & & \textit{otherwise} \ \ \text{Encode}(\text{Normalise}(D_1) \sqcap \ldots \sqcap \text{Normalise}(D_n)) \\
C = \exists R.D & \longrightarrow & \text{Normalise}(\neg\forall R.\neg D) \\
C = D_1 \sqcup \ldots \sqcup D_n & & \\
 & \longrightarrow & \text{Normalise}(\neg(\neg D_1 \sqcap \ldots \sqcap \neg D_n))
\end{array}
$$

Encode(C) :

$$
\begin{array}{lll}
C = \forall R.D & \longrightarrow & \mathsf{CN} \ \ \textit{if} \ \ \mathsf{CN} \doteq \forall R.D \in \mathcal{T} \\
 & & \textit{otherwise} \ \ \mathsf{CN}' \text{ where } \mathsf{CN}' \text{ is a new concept name } \textit{and} \\
 & & \qquad\qquad\qquad \mathcal{T} \longrightarrow \mathcal{T} \cup \{\mathsf{CN}' \doteq \forall R.D\} \\
C = D_1 \sqcap \ldots \sqcap D_n & & \\
 & \longrightarrow & \mathsf{CN} \ \ \textit{if} \ \ \ \mathsf{CN} \doteq (D'_1 \sqcap \ldots \sqcap D'_n) \in \mathcal{T} \ \textit{and} \\
 & & \qquad\quad \forall D.(D \in \{D_1, \ldots, D_n\} \Leftrightarrow D \in \{D'_1, \ldots, D'_n\}) \\
 & & \textit{otherwise} \ \ \mathsf{CN}' \text{ where } \mathsf{CN}' \text{ is a new concept name } \textit{and} \\
 & & \qquad\qquad\qquad \mathcal{T} \longrightarrow \mathcal{T} \cup \{\mathsf{CN}' \doteq D_1 \sqcap \ldots \sqcap D_n\}
\end{array}
$$

Table 5.1: Normalisation and encoding

1. Normalise the first sub-expression:

   (a) Normalise$(\exists R.(C \sqcap E \sqcap \neg D)) \longrightarrow \neg$Normalise$(\forall R.\neg(C \sqcap E \sqcap \neg D))$

   (b) Normalise$(\forall R.\neg(C \sqcap E \sqcap \neg D)) \longrightarrow$ Encode$(\forall R.$Normalise$(\neg(C \sqcap E \sqcap \neg D)))$

   (c) Normalise$(\neg(C \sqcap E \sqcap \neg D)) \longrightarrow \neg$Encode$(C \sqcap E \sqcap \neg D)$

   (d) Encode$(C \sqcap E \sqcap \neg D) \longrightarrow \mathsf{CN}_1$ where CN_1 is a new concept name and $\mathcal{T} \longrightarrow \mathcal{T} \cup \{\mathsf{CN}_1 \doteq C \sqcap \neg D \sqcap E\}$

   (e) Encode$(\forall R.$Normalise$(\neg(C \sqcap E \sqcap \neg D))) \longrightarrow \mathsf{CN}_2$ where CN_2 is a new concept name and $\mathcal{T} \longrightarrow \mathcal{T} \cup \{\mathsf{CN}_2 \doteq \forall R.\neg\mathsf{CN}_1\}$

   (f) Normalise$(\exists R.(C \sqcap E \sqcap \neg D)) \longrightarrow \neg\mathsf{CN}_2$

2. Normalise the second sub-expression:

   (a) Normalise($\forall R.(D \sqcup \neg C \sqcup \neg E)$) $\longrightarrow$ Encode($\forall R.$Normalise($D \sqcup \neg C \sqcup \neg E$))

   (b) Normalise($D \sqcup \neg C \sqcup \neg E$) $\longrightarrow$ Normalise($\neg(\neg D \sqcap C \sqcap E)$)

   (c) Normalise($\neg(\neg D \sqcap C \sqcap E)$) $\longrightarrow$ $\neg$Encode($\neg D \sqcap C \sqcap E$)

   (d) Encode($\neg D \sqcap C \sqcap E$) $\longrightarrow$ Encode($C \sqcap \neg D \sqcap E$), which is recognised as CN_1

   (e) Normalise($\forall R.(D \sqcup \neg C \sqcup \neg E)$) $\longrightarrow$ Encode($\forall R.\neg \mathsf{CN}_1$), which is recognised as CN_2

3. Recombine the two sub-expressions:

   (a) Normalise($\exists R.(C \sqcap E \sqcap \neg D) \sqcap \forall R.(D \sqcup \neg C \sqcup \neg E)$) $\longrightarrow$ Normalise($\neg \mathsf{CN}_2 \sqcap \mathsf{CN}_2$)

   (b) Normalise($\neg \mathsf{CN}_2 \sqcap \mathsf{CN}_2$) $\longrightarrow \bot$

4. The concept expression is identified as unsatisfiable without recourse to the tableaux expansion algorithm.

Normalisation and encoding has a number of advantages:

- Structurally obvious satisfiability and unsatisfiability can be detected without using the tableaux algorithm—in effect a general subsumption test has been avoided by using a cheaper structural test.
- Knowledge bases with large amounts of repetitive structure can be stored more compactly.
- Other optimisation techniques are facilitated (see Section 5.8 on page 95).
- More efficient data structures can be used (see Section 6.2.4 on page 101).

The disadvantages of normalisation and encoding are that there is obviously some cost associated with the process, although this is likely to be small relative to the cost of classifying the knowledge base (see Section 7.3.1 on page 115), and that the size of the knowledge base could be increased if it contains very little repetitive structure. However, any increase in knowledge base size should be relatively small because new concept definitions added by encoding will result in a compaction of existing definitions. For example, if the concept expression $C_1 \sqcap \ldots \sqcap C_n$ occurs in a knowledge base $\mathcal{T}$, a non-primitive concept definition $\mathsf{CN} \doteq C_1 \sqcap \ldots \sqcap C_n$ may be added to $\mathcal{T}$ (if $C_1 \sqcap \ldots \sqcap C_n$ is not recognised as the definition of an existing concept), but the original occurrence of $C_1 \sqcap \ldots \sqcap C_n$ will be replaced by CN.

5.4 GCI Absorption

GCIs are a major cause of intractability: each GCI causes a disjunctive expression to be added to the label of every node in the tree generated by the tableaux algorithm and this leads to an exponential increase in the size of the search space to be explored by the $\sqcup$-rule. The aim of this technique is to reduce the number of GCIs by absorbing them into primitive concept introduction axioms whenever possible. It was suggested by the structure of GCIs in the GALEN terminology and by restrictions on the structure of GCIs imposed by the syntax of the GRAIL concept description language:

- Many GCIs in the GALEN terminology have only a primitive concept name as their antecedent.
- When the antecedent of a GRAIL GCI is not a primitive concept it is either a conjunctive concept expression or a non-primitive concept name whose definition is a conjunctive concept expression.
- The first conjunct of a GRAIL conjunctive concept expression is always either a primitive concept name or a non-primitive concept name whose definition is a conjunctive concept expression.

This structure means that the antecedent of a GRAIL GCI can always be unfolded so that it is either a primitive concept name or a conjunctive concept expression one of whose terms is a primitive concept name.

When the antecedent of a GCI is a primitive concept name, the GCI simply states additional necessary conditions for the primitive concept and can be absorbed into the primitive concept's introduction axiom using the identity:

$$\mathsf{CN} \sqsubseteq C \text{ and } \mathsf{CN} \sqsubseteq D \quad = \quad \mathsf{CN} \sqsubseteq C \sqcap D \tag{5.1}$$

the validity of which is obvious from the semantics:

$$\mathsf{CN}^{\mathcal{I}} \subseteq C^{\mathcal{I}} \wedge \mathsf{CN}^{\mathcal{I}} \subseteq D^{\mathcal{I}} \quad \Longleftrightarrow \quad \mathsf{CN}^{\mathcal{I}} \subseteq C^{\mathcal{I}} \cap D^{\mathcal{I}}$$

Furthermore, when the antecedent of a GCI is a conjunctive concept expression one of whose terms is a primitive concept, it can be transformed into a GCI with a primitive concept name as its antecedent using the identity:

$$\mathsf{CN} \sqcap C \sqsubseteq D \quad = \quad \mathsf{CN} \sqsubseteq D \sqcup \neg C \tag{5.2}$$

the validity of which is again obvious from the semantics:

$$\mathsf{CN}^{\mathcal{I}} \cap C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \quad \Longleftrightarrow \quad \mathsf{CN}^{\mathcal{I}} \subseteq D^{\mathcal{I}} \cup (\neg C)^{\mathcal{I}}$$

Example 5.3 Absorbing a GCI

The GCI:

$$\mathsf{geometric\text{-}figure} \sqcap \exists \textsf{\textit{angles}}.\mathsf{three} \sqsubseteq \exists \textsf{\textit{sides}}.\mathsf{three}$$

from Example 2.5 can be absorbed into the primitive concept introduction axiom:

$$\mathsf{geometric\text{-}figure} \sqsubseteq \mathsf{shape}$$

by first transforming the GCI using identity 5.2 on the preceding page to give:

$$\mathsf{geometric\text{-}figure} \sqsubseteq \exists \textsf{\textit{sides}}.\mathsf{three} \sqcup \neg \exists \textsf{\textit{angles}}.\mathsf{three}$$

and then absorbing this into the introduction axiom using identity 5.1 on the preceding page to give:

$$\mathsf{geometric\text{-}figure} \sqsubseteq \mathsf{shape} \sqcap (\exists \textsf{\textit{sides}}.\mathsf{three} \sqcup \neg \exists \textsf{\textit{angles}}.\mathsf{three})$$

Absorption can also be performed on GCIs whose consequent is a negated primitive concept name, or a disjunctive expression one of whose elements is a negated primitive concept name. This is done using using the identity:

$$C \sqsubseteq \neg \mathsf{CN} \sqcup D \quad = \quad \mathsf{CN} \sqsubseteq D \sqcup \neg C$$

In general, a GCI $C \sqsubseteq D$ can be absorbed into a primitive concept introduction axiom, whenever possible, by using the following steps:

1. Initialise a set $\mathbf{G}$ to contain the GCI's consequent and negated antecedent:

   $$\mathbf{G} := \{D, \neg C\}$$

2. If for some $w \in \mathbf{G}$, w is a negated primitive concept name with an introduction axiom of the form $w \sqsubseteq C_w$, absorb the GCI into w's introduction so that it becomes:

   $$w \sqsubseteq C_w \sqcap \left(\bigsqcup_{y \in \mathbf{G} - \{w\}} y \right)$$

   and exit.

3. If for some $w \in \mathbf{G}$, w is a negated conjunctive concept expression $\neg(C_1 \sqcap \ldots \sqcap C_n)$ or a negated non-primitive concept name whose definition is a conjunctive concept expression $C_1 \sqcap \ldots \sqcap C_n$, then expand w:

   $$\mathbf{G} \longrightarrow (\mathbf{G} - \{w\}) \cup \{\neg C_1, \ldots, \neg C_n\}$$

   and return to step 2.

4. If for some $w \in \mathbf{G}$, w is a disjunctive concept expression $C_1 \sqcup \ldots \sqcup C_n$ or a non-primitive concept name whose definition is a disjunctive concept expression $C_1 \sqcup \ldots \sqcup C_n$, then expand w:

   $$\mathbf{G} \longrightarrow (\mathbf{G} - \{w\}) \cup \{C_1, \ldots, C_n\}$$

   and return to step 2.

5. The GCI could not be absorbed. It must be added to the meta constraint $\mathcal{M}$:
$$\mathcal{M} \longrightarrow \mathcal{M} \sqcap (D \sqcup \neg C)$$
and dealt with using internalisation (see Section 4.3 on page 71).

The advantage of absorption is that it avoids adding large numbers of irrelevant disjunctions to the label of every node in a tree $\mathbf{T}$ being constructed by the tableaux expansion algorithm. Using internalisation to deal with the GCI $\mathsf{geometric\text{-}figure} \sqcap \exists \mathit{angles}.\mathsf{three} \sqsubseteq \exists \mathit{sides}.\mathsf{three}$ from example 5.3 on the preceding page would result in the disjunction:

$$\exists \mathit{sides}.\mathsf{three} \sqcup \neg \mathsf{geometric\text{-}figure} \sqcup \neg \exists \mathit{angles}.\mathsf{three}$$

being added to the label of every node in $\mathbf{T}$. Absorbing the GCI into $\mathsf{geometric\text{-}figure}$'s introduction axiom on the other hand results in the disjunction:
$$\exists \mathit{sides}.\mathsf{three} \sqcup \neg \exists \mathit{angles}.\mathsf{three}$$
being added to a node x only when $\mathsf{geometric\text{-}figure} \in \mathcal{L}(x)$—this will happen automatically when $\mathsf{geometric\text{-}figure}$ is unfolded.

If there is more than one primitive concept into whose definition the GCI could be absorbed, the primitive which is chosen is irrelevant as regards the correctness if not of the efficiency of absorption[1]. For example the GCI $\mathsf{CN}_1 \sqcap \mathsf{CN}_2 \sqsubseteq C$, where CN_1 and CN_2 are both primitive concept names, could be absorbed into the the introduction axioms of either CN_1 or CN_2. If absorbed into CN_1's introduction then all nodes x s.t. $\mathsf{CN}_1 \in \mathcal{L}(x)$ will have $C \sqcup \neg \mathsf{CN}_2$ in their labels; if absorbed into CN_2's introduction then all nodes x s.t. $\mathsf{CN}_2 \in \mathcal{L}(x)$ will have $C \sqcup \neg \mathsf{CN}_1$ in their labels. In either case the GCI is clearly satisfied:

$$\begin{array}{lll} \mathsf{CN}_1 \sqcap \mathsf{CN}_2 \in \mathcal{L}(x) \text{ and } \mathsf{CN}_1 \sqsubseteq C \sqcup \neg \mathsf{CN}_2 & \text{leads to} & \mathcal{L}(x) \cup \{C\} \\ \mathsf{CN}_1 \sqcap \mathsf{CN}_2 \in \mathcal{L}(x) \text{ and } \mathsf{CN}_2 \sqsubseteq C \sqcup \neg \mathsf{CN}_1 & \text{leads to} & \mathcal{L}(x) \cup \{C\} \end{array}$$

5.5 Semantic Branching Search

Standard tableaux algorithms are inherently inefficient as they use a search technique called syntactic branching. Syntactic branching works by choosing an unexpanded disjunction and searching the different models obtained by adding each of the disjuncts [GS96a]. As the alternative branches of the search tree are not disjoint, there is nothing to prevent the recurrence of an unsatisfiable disjunct in different branches. The resulting wasted expansion could be costly if discovering the unsatisfiability requires the solution of a complex sub-problem.

[1]Intuitively, it would seem sensible to choose the primitive which is least general or which occurs least frequently in the terminology, but this has not been investigated.

Example 5.4 Wasted Search Due To Syntactic Branching

Tableaux expansion of a tree **T** with a leaf node x, where $\{(C \sqcup D_1), (C \sqcup D_2)\} \subseteq \mathcal{L}(x)$ and C is an unsatisfiable concept, expression could lead to the search pattern shown in Figure 5.1.

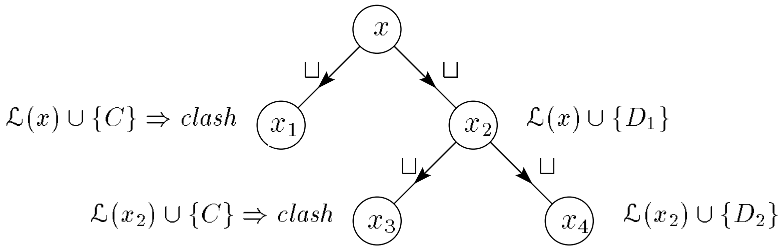

Figure 5.1: Syntactic branching

This problem is dealt with by using a semantic branching technique adapted from the Davis-Putnam-Logemann-Loveland procedure (DPL) commonly use to solve propositional satisfiability (SAT) problems [DLL62, Fre96]. Instead of choosing an unexpanded disjunction in $\mathcal{L}(x)$, a single disjunct D is chosen from one of the unexpanded disjunctions in $\mathcal{L}(x)$. The two possible sub-trees obtained by adding either D or $\neg D$ to $\mathcal{L}(x)$ are then searched. The resulting search pattern will be shown (in Figure 5.2 on page 88) after discussing some enhancements to the basic technique.

Semantic branching has a number of advantages:

- At each branching point in the search tree the two branches are strictly disjoint, so there is no possibility of wasted search as in syntactic branching.
- A great deal is known about the implementation and optimisation of this algorithm. In particular, both *boolean constraint propagation* and *heuristic guided search* can be used to minimise the size of the search tree.

5.5.1 Boolean Constraint Propagation

Boolean constraint propagation (BCP) [Fre95] is a technique used to maximise deterministic expansion, and thus pruning of the search tree via clash detection, before performing non-deterministic expansion (branching).

Before the $\sqcup$-rule is applied to the label of a node x, BCP deterministically expands disjunctions in $\mathcal{L}(x)$ which present only one expansion possibility and detects a clash when a disjunction in $\mathcal{L}(x)$ has no expansion possibilities. The number of expansion possibilities presented by a disjunction $(C_1 \sqcup \ldots \sqcup C_n) \in \mathcal{L}(x)$

is equal to the number of disjuncts C_i such that $\neg C_i \notin \mathcal{L}(x)$. In effect, BCP is applying the inference rule:

$$\frac{\neg C, C \sqcup D}{D}$$

to $\mathcal{L}(x)$ or, in other words, performing some propositional resolution on the disjunctive clauses in $\mathcal{L}(x)$.

Example 5.5 Boolean Constraint Propagation

Given a node x such that:

$$\{(C \sqcup (D_1 \sqcap D_2)), (\neg D_1 \sqcup \neg D_2), \neg C\} \subseteq \mathcal{L}(x)$$

BCP deterministically expands the disjunction $(C \sqcup (D_1 \sqcap D_2))$, because $\neg C \in \mathcal{L}(x)$:

$$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{(D_1 \sqcap D_2)\}$$

After $(D_1 \sqcap D_2)$ is expanded to give $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{D_1, D_2\}$, BCP identifies $(\neg D_1 \sqcup \neg D_2)$ as a clash because $D_1 \in \mathcal{L}(x)$ and $D_2 \in \mathcal{L}(x)$.

5.5.2 Heuristic Guided Search

Heuristic techniques can be used to guide the search in a way which tries to minimise the size of the search tree. One possible method is to branch on the disjunct which has the Maximum number of Occurrences in disjunctions of Minimum Size—the well known MOMS heuristic [Fre95]. By choosing a disjunct which occurs frequently in small disjunctions, MOMS heuristic tries to maximise the effect of BCP. For example, if the label of a node x contains the unexpanded disjunctions:

$$\{C \sqcup D_1, \ldots, C \sqcup D_n\} \subseteq \mathcal{L}(x)$$

then branching on C leads to their deterministic expansion in a single step. When C is added to $\mathcal{L}(x)$, all of the disjunctions are fully expanded and when $\neg C$ is added to $\mathcal{L}(x)$, BCP will expand all of the disjunctions, causing $D_1, \ldots, D_n$ to be added to $\mathcal{L}(x)$. Branching first on any of $D_1, \ldots, D_n$, on the other hand, would only cause a single disjunction to be expanded.

When branching on a given disjunct, the order in which the two branches are explored is also heuristically determined. The branch which maximises the potential for BCP expansion is explored first, as the concepts added by BCP may result in further expansion. In the above example, adding $\neg C$ to $\mathcal{L}(x)$ would be tried first as this will cause BCP to add $D_1, \ldots, D_n$ to $\mathcal{L}(x)$. Adding $D_1, \ldots, D_n$ to $\mathcal{L}(x)$ will, in turn, satisfy any other disjunctions in $\mathcal{L}(x)$ containing these concepts and could lead to further BCP expansion if any of $\neg D_1, \ldots \neg D_n$ occur in other disjunctions in $\mathcal{L}(x)$.

Example 5.6 Semantic Branching Search

When applied to node x in the tree $\mathbf{T}$ from Example 5.4 on page 86, where $\{(C \sqcup D_1), (C \sqcup D_2)\} \subseteq \mathcal{L}(x)$, MOMS heuristic would select C as the disjunct on which to branch because it occurs in 2 disjunctions of size 2. The $\sqcup$-successor in which $\neg C$ is added to $\mathcal{L}(x)$ would be explored first as BCP also causes D_1 and D_2 to be added to $\mathcal{L}(x)$. The modified search pattern is illustrated in Figure 5.2.

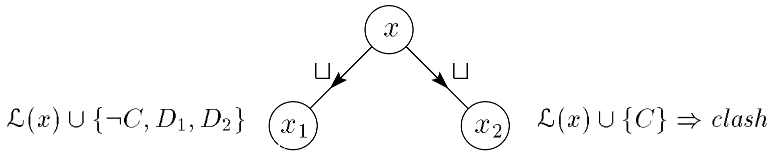

Figure 5.2: Semantic branching with BCP

5.5.3 The Optimised Search Algorithm

In general, for a node x in a tree $\mathbf{T}$, the optimised search algorithm proceeds as follows:

1. Perform all deterministic expansions; e.g., apply the $\sqcap$-rule from Table 4.2 on page 65 as many times as possible.

2. Perform Boolean Constraint Propagation (BCP):

   (a) Search $\mathcal{L}(x)$ for unexpanded disjunctions $(C_1 \sqcup \ldots \sqcup C_n)$ where:

   $$\begin{array}{rcl} \{\neg C_1, \ldots, \neg C_{i-1}\} & \subseteq & \mathcal{L}(x) \\ \{\neg C_{i+1}, \ldots, \neg C_n\} & \subseteq & \mathcal{L}(x) \\ C_i & \notin & \mathcal{L}(x) \end{array}$$

   (b) If such a disjunction is found and $\neg C_i \in \mathcal{L}(x)$, return *clash*.

   (c) If such a disjunction is found and $\neg C_i \notin \mathcal{L}(x)$, expand the disjunction:

   $$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_i\}$$

   and return to step 1.

3. Perform heuristic guided search if there are unexpanded disjunctions $(C_{11} \sqcup \ldots \sqcup C_{1n}), \ldots, (C_{m1} \sqcup \ldots \sqcup C_{mn})$ in $\mathcal{L}(x)$.

(a) Use MOMS heuristic to select C_{ij} such that $\neg C_{ij} \notin \mathcal{L}(x)$. If there is more than one candidate with the same MOMS priority, try to break the tie by using Jeroslow and Wang's weighted occurrences heuristic [JW90].

(b) Create two $\sqcup$-successors y and z with:

$$\begin{array}{rcl} \mathcal{L}(y) & = & \mathcal{L}(x) \cup \{C_{ij}\} \\ \mathcal{L}(z) & = & \mathcal{L}(x) \cup \{\neg C_{ij}\} \end{array}$$

(c) Explore y and z in a heuristically determined order: explore y first if the number of occurrences of $\neg C_{ij}$ is greater than the number of occurrences of C_{ij}, otherwise explore z first.

5.6 Dependency Directed Backtracking

Inherent unsatisfiability concealed in sub-problems can lead to large amounts of unproductive backtracking search known as thrashing. The problem is exacerbated when blocking is used to guarantee termination, because blocking requires that sub-problems only be explored after all other forms of expansion have been performed. In the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm, for example, the $\exists$-rule can only be applied to a pre-tableau, a node label which does not contain any unexpanded conjunctions or disjunctions.

Example 5.7 Thrashing

Expanding a node x, where:

$$\mathcal{L}(x) = \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists R.(C \sqcap D), \forall R.\neg C\}$$

would lead to the fruitless exploration of 2^n pre-tableau $\sqcup$-successors of x before the inherent unsatisfiability is discovered[2]. The search tree under x created by the tableaux expansion algorithm is illustrated in Figure 5.3 on the following page.

This problem is addressed by adapting a form of dependency directed backtracking called *backjumping*, which has been used in solving constraint satisfiability problems [Bak95]. Backjumping works by labeling concept expressions with a dependency set indicating the $\sqcup$-nodes (nodes with $\sqcup$-successors) on which they depend. A concept expression depends on a $\sqcup$-node x when it was added to the label of x's $\sqcup$-successor by the search algorithm, or when it depends an another concept expression which depends on x. A concept expression C depends on

[2]Note that if $\mathcal{L}(x)$ simply included $\exists R.C$ instead of $\exists R.(C \sqcap D)$, then the inherent unsatisfiability would have been detected immediately due to the encoding of $\exists R.C$ and $\forall R.\neg C$ as CN and $\neg\mathsf{CN}$ for some concept name CN.

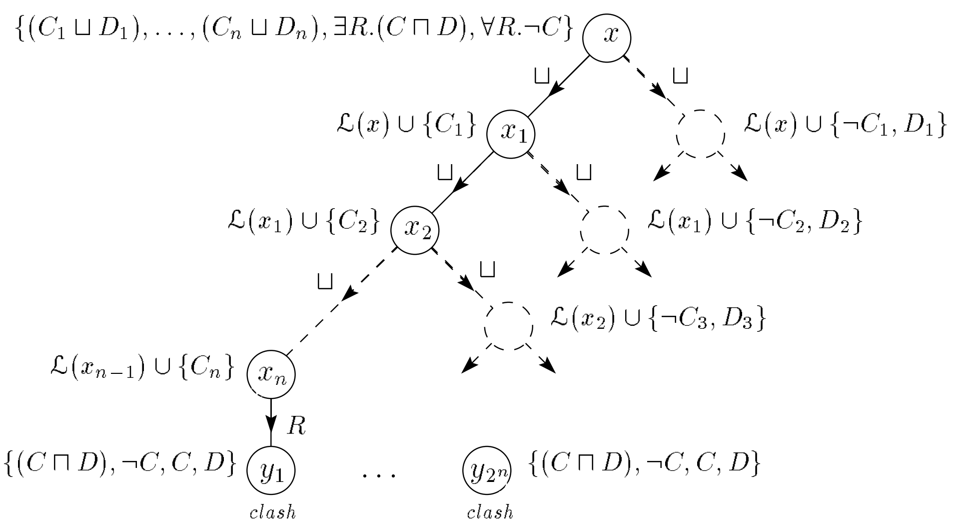

Figure 5.3: Thrashing in backtracking search

a concept expression D when C was added to a node label by a deterministic expansion (an application of the $\sqcap$-rule, $\exists$-rule or BCP) which used D.

When a clash is discovered, the dependency sets of the clashing concepts can be used to identify the most recent $\sqcup$-node where exploring an alternative $\sqcup$-successor might alleviate the cause of the clash. The algorithm can then jump back over intervening $\sqcup$-nodes *without* exploring alternative $\sqcup$-successors.

For example, when expanding the $\sqcup$-node x from Example 5.7 on the preceding page, where:

$$\mathcal{L}(x) = \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists R.(C \sqcap D), \forall R.\neg C\}$$

the search algorithm will create a sequence of $\sqcup$-successors and $\sqcup$-nodes, eventually leading to a pre-tableau node x_n with $\{\exists R.(C \sqcap D), \forall R.\neg C\} \subseteq \mathcal{L}(x_n)$. When x_n is expanded the algorithm will generate an R-successor y_1 with $\{C, \neg C\} \subseteq \mathcal{L}(y_1)$ and a clash will be detected. As neither of the clashing concepts in $\mathcal{L}(y_1)$ will have the $\sqcup$-nodes leading from x to x_n in their dependency sets, the algorithm can either return *unsatisfiable* immediately (if both the dependency sets were empty) or jump directly back to the most recent $\sqcup$-node on which one of the clashing concepts did depend. Figure 5.4 on the following page illustrates how the search tree below x is pruned by backjumping, with only $n+2$ nodes being explored instead of $2^n + 2^{n+1} - 1$ nodes.

In more general terms, backjumping works as follows:

1. The initial concept expressions in $\mathcal{L}(x_0)$ have their dependency sets initialised to $\emptyset$.

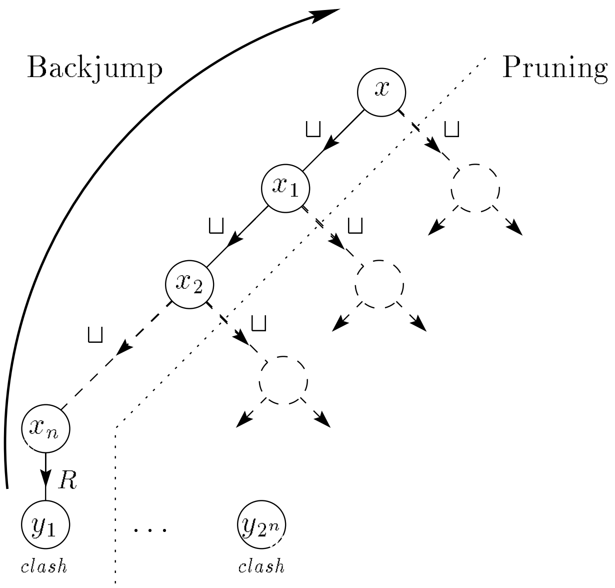

Figure 5.4: Pruning the search using backjumping

2. Concept expressions added to node labels by deterministic expansion (the $\sqcap$-rule, $\exists$-rule and BCP) are labelled with the (union of the) label(s) from the concept expression(s) which are used in the expansion.

3. If x_n is a $\sqcup$-node, the concept expressions C and $\neg C$ added to the labels of the $\sqcup$-successors of x_n by the search algorithm are labeled $\{n\}$.

4. After a clash, return a dependency set $\mathcal{D}$ consisting of the (union of the) label(s) from the clashing concept(s).

5. If the first $\sqcup$-successor of a $\sqcup$-node x_n is unsatisfiable, and returns a dependency set $\mathcal{D}_1$ such that $n \notin \mathcal{D}_1$, backtrack immediately returning the dependency set $\mathcal{D}_1$.

6. If the second $\sqcup$-successor of x_n is unsatisfiable, and returns a dependency set $\mathcal{D}_2$, return the dependency set $(\mathcal{D}_1 \cup \mathcal{D}_2) - \{n\}$.

5.7 Caching

The combination of normalisation, encoding and lazy unfolding facilitates the rapid detection of "obvious" unsatisfiability (subsumption), but detecting "obvious" satisfiability (non-subsumption) is more difficult for tableaux algorithms. This is unfortunate as, when classifying realistic terminologies:

- most tests are satisfiable (e.g., a ratio of 3:1 when classifying the GALEN terminology);
- satisfiable tests are generally much more expensive (e.g., a ratio of 7:1 when classifying the GALEN terminology).

This problem is tackled by trying to use cached results from previous tableaux tests to demonstrate the satisfiability of a concept expression. This is possible because satisfiability problems in sub-trees are completely independent and have no effect on ancestor nodes, other than to determine their satisfiability. A considerable amount of work can be saved when large or complex models are re-used in this way.

Example 5.8 Merging Models

Given two concepts:

$$\begin{array}{rcl} C & \doteq & D_1 \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2 \\ D & \doteq & \neg(D_2 \sqcap \exists R_3.C_3) \end{array}$$

the satisfiability of $C \sqcap \neg D$ (and thus the non-subsumption $C \not\sqsubseteq D$) can be demonstrated by a model consisting of models of C and $\neg D$ joined at their roots, as shown in Figure 5.5. Note that $\neg D \doteq D_2 \sqcap \exists R_3.C_3$.

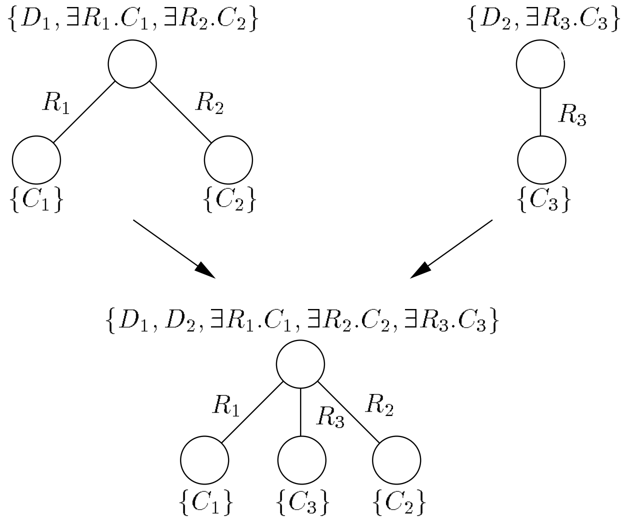

Figure 5.5: Merging models of C and $\neg D$

When the root node x_0 of a tree $\mathbf{T}$ is marked satisfiable, it must have a pre-tableau $\sqcup$-successor which is marked satisfiable—this node represents the individual which is the root of the model described by $\mathbf{T}$ and will be denoted x_r. To demonstrate that two models joined at their roots result in a valid (non-clashing) model, it is only necessary to examine the labels on their x_r nodes, which we will call their x_r-labels, and check that the union of these labels is still a satisfiable pre-tableau. Given two models represented by trees $\mathbf{T}^C$ and $\mathbf{T}^D$, with x_r-labels $\mathcal{L}(x_r^C)$ and $\mathcal{L}(x_r^D)$, $\mathcal{L}(x_r^C) \cup \mathcal{L}(x_r^D)$ is a satisfiable pre-tableau, and the two models can be joined, unless either:

1. the union of their x_r-labels contains an immediate contradiction, e.g.:

$$C_1 \in \mathcal{L}(x_r^C) \wedge \neg C_1 \in \mathcal{L}(x_r^D)$$

2. exists and value restrictions in the x_r-labels might interact, e.g.:

$$\exists R.C_1 \in \mathcal{L}(x_r^C) \text{ and } \forall S.C_2 \in \mathcal{L}(x_r^D) \text{ and } R \sqsubseteq S$$

   where R and S are either roles or attributes;

3. attribute exists restrictions in the x_r-labels might interact, e.g.:

$$\exists A.C_1 \in \mathcal{L}(x_r^C) \text{ and } \exists B.C_2 \in \mathcal{L}(x_r^D) \text{ and } (\mathbf{A}_{Ax_r^C} \cap \mathbf{A}_{Bx_r^D} \neq \emptyset)$$

Note that conditions 2 and 3 state that restrictions *might* interact—it would be possible to merge models in a wider range of cases by examining the potential interaction in more detail (e.g., when condition 2 is violated, by checking if $\forall S.C_2$ is already in $\mathcal{L}(x_r^C)$), or by considering additional nodes. However, the test described is fast and requires a relatively small amount of space for storing cached models.

Given a concept D, a model of which is represented by the tree $\mathbf{T}^D$ containing the x_r node x_r^D, the space required for caching can be minimised by only storing those components of $\mathcal{L}(x_r^D)$ which need to be checked before merging can be performed, i.e., concept names and their negations, roles and attributes which occur in exists or value restrictions, and attributes which are constrained by attribute exists restrictions. These are cached as the sets $\mathbf{S}_{\mathsf{C}}^D$, $\mathbf{S}_{\neg\mathsf{C}}^D$, $\mathbf{S}_{\exists}^D$, $\mathbf{S}_{\forall}^D$ and $\mathbf{S}_{A}^D$, where:

$$\begin{aligned}
\mathbf{S}_{\mathsf{C}}^D &= \{\mathsf{CN} \mid \mathsf{CN} \in \mathcal{L}(x_r^D)\} \\
\mathbf{S}_{\neg\mathsf{C}}^D &= \{\mathsf{CN} \mid \neg\mathsf{CN} \in \mathcal{L}(x_r^D)\} \\
\mathbf{S}_{\exists}^D &= \{R \mid \exists R.C \in \mathcal{L}(x_r^D)\} \cup \{A \mid \exists A.C \in \mathcal{L}(x_r^D)\} \\
\mathbf{S}_{\forall}^D &= \{R \mid \forall R.C \in \mathcal{L}(x_r^D)\} \cup \{A \mid \forall A.C \in \mathcal{L}(x_r^D)\} \\
\mathbf{S}_{A}^D &= \{A \mid \exists B.C \in \mathcal{L}(x_r^D) \text{ and } A \in \mathbf{A}_{Bx_r^D}\}
\end{aligned}$$

In more general terms, when testing if C is subsumed by D, caching works as follows:

1. If models for C, $\neg C$, D or $\neg D$ have not been cached, perform the satisfiability test(s) and, in each case, cache $\mathbf{S}_{\mathsf{C}}$, $\mathbf{S}_{\neg\mathsf{C}}$, $\mathbf{S}_{\exists}$, $\mathbf{S}_{\forall}$ and $\mathbf{S}_{A}$. If a satisfiability test fails, the concept being tested is equal to $\bot$ and its negation to $\top$.

2. Return *true* ($C \sqsubseteq D$) or *false* ($C \not\sqsubseteq D$) if obvious subsumption or non-subsumption is detected:

$$\begin{aligned} \neg D = \bot &\;\Rightarrow\; \textit{true}\ (C \sqsubseteq D) \\ C = \bot &\;\Rightarrow\; \textit{true}\ (C \sqsubseteq D) \\ D = \bot \text{ and } C \neq \bot &\;\Rightarrow\; \textit{false}\ (C \not\sqsubseteq D) \\ \neg C = \bot \text{ and } \neg D \neq \bot &\;\Rightarrow\; \textit{false}\ (C \not\sqsubseteq D) \end{aligned}$$

3. Return *false* ($C \not\sqsubseteq D$) if the cached models of C and $\neg D$ can be merged. The models can be merged unless either:

   (a) $\mathbf{S}^{C}_{\mathsf{C}} \cap \mathbf{S}^{\neg D}_{\neg\mathsf{C}} \neq \emptyset$

   (b) $\mathbf{S}^{C}_{\neg\mathsf{C}} \cap \mathbf{S}^{\neg D}_{\mathsf{C}} \neq \emptyset$

   (c) there is some R, S s.t. $R \in \mathbf{S}^{C}_{\exists}$ and $S \in \mathbf{S}^{\neg D}_{\forall}$ and $R \sqsubseteq S$

   (d) there is some R, S s.t. $R \in \mathbf{S}^{C}_{\forall}$ and $S \in \mathbf{S}^{\neg D}_{\exists}$ and $S \sqsubseteq R$

   (e) $\mathbf{S}^{C}_{A} \cap \mathbf{S}^{\neg D}_{A} \neq \emptyset$

4. Perform a tableaux satisfiability test on $C \sqcap \neg D$ returning *false* ($C \not\sqsubseteq D$) if it is satisfiable and *true* ($C \sqsubseteq D$) if it is not.

5.7.1 Using Caching in Sub-problems

The caching technique can be extended in order to avoid the construction of obviously satisfiable sub-trees during tableaux expansion. For example, if some leaf node x is about to be expanded, and $\mathcal{L}(x) = \{\mathsf{CN}\}$, unfolding and expanding CN is clearly unnecessary if CN is already known to be satisfiable, i.e., a model of CN has already been cached.

The technique can be further extended by trying to merge cached models of all the concepts in a leaf node's label before starting the expansion process. The method generalises that described in section 5.7 in the obvious way—models of concepts $D_1, \ldots, D_n$ can be merged unless either:

1. there is some i, j s.t. $\mathbf{S}^{Di}_{\mathsf{C}} \cap \mathbf{S}^{Dj}_{\neg\mathsf{C}} \neq \emptyset$

2. there is some i, j, R, S s.t. $R \in \mathbf{S}^{Di}_{\exists}$ and $S \in \mathbf{S}^{Dj}_{\forall}$ and $R \sqsubseteq S$

3. there is some i, j s.t. $\mathbf{S}^{Di}_{A} \cap \mathbf{S}^{Dj}_{A} \neq \emptyset$

5.8 Interactions Between Optimisations

The behaviour of the optimisation techniques is not orthogonal—some interfere while others are complementary:

- The normalisation and encoding of concept expressions complements most of the other optimisation techniques:
  - The effect of lazy unfolding is enhanced because it is not limited to named concepts—clashes between encoded concept expressions can also be rapidly detected.
  - Semantic branching, BCP and heuristic guided search can be performed more efficiently because the large numbers of comparisons of disjuncts required by these procedures can be performed on a concept name level instead of a structural level—normalisation and encoding effectively performs all structural comparisons prior to classification and caches the results in the form of new non-primitive concept introductions.
  - The granularity of caching is maximised—when classifying a terminology, models of every concept expression and subexpression in the terminology may eventually be cached.
  - The effect of the told subsumer classification optimisation is enhanced (see Section 6.3 on page 102).
- Using MOMS heuristic to select branching disjuncts interacts with and degrades the performance of backjumping (see Section 7.3.1 on page 115). For this reason, the heuristic guided search optimisation is disabled by default in the FaCT classifier (see Section 6.4.1 on page 104).
- Several of the optimisations increase storage requirements. Backjumping, in particular, requires a dependency set to be stored with every concept expression in a node label. However, as the same dependency sets will be used in whole or in part by large numbers of concept expressions, the storage requirement can be reduced by using a pointer based implementation such as Lisp lists.

Chapter 6

The FaCT Classifier

This chapter describes FaCT[1], a DL classifier which has been implemented as a test-bed for the optimised $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ subsumption testing algorithm, and explains how FaCT uses the algorithm to classify a terminological knowledge base (TKB).

The chapter is organised as follows: Section 6.1 provides a brief overview of the FaCT system and describes the syntax of FaCT TKBs; Section 6.2 explains the pre-processing which FaCT performs on axioms in the TKB prior to classification; Section 6.3 describes the optimised algorithm which FaCT uses to classify the TKB; and finally, Section 6.4 explains how FaCT can be configured to facilitate testing and to adapt its behaviour to different reasoning tasks.

6.1 FaCT Overview

FaCT has been designed to be closely compatible with the KRIS system [BH91c]: it uses the same, relatively standard, concept description syntax (with the exception of number restrictions and attribute value maps which are not supported by FaCT) and a similar function and macro interface, extended as necessary to deal with the additional features provided by FaCT. This facilitates comparing the two systems by allowing TKBs which use a common subset of the two languages ($\mathcal{ALC}$ augmented with attributes but restricted to unfoldable terminologies) to be classified by either system.

As the purpose of FaCT is to evaluate the feasibility of using the optimised $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm for subsumption reasoning, only a terminological classifier (TBox) has been implemented: FaCT reasons about concept, role and attribute descriptions, and maintains a concept hierarchy based on the subsumption relation (see Section 1.1 on page 16). FaCT provides a wide range of functions and macros for managing TKBs, adding axioms to a TKB, performing inferences and

[1] **Fa**st **C**lassification of **T**erminologies.

answering queries. FaCT's main features are described in the following sections and a full reference manual is reproduced in Appendix A.

6.1.1 Concept Description Syntax

Like Kris, FaCT is implemented in Lisp, and its concept descriptions use the same list based syntax employed by Kris. The correspondence between this form and the standard infix notation is shown in Table 6.1.

FaCT syntax	Standard notation
`*TOP*`	$\top$
`*BOTTOM*`	$\bot$
(`and` $C_1 \ldots C_n$)	$C_1 \sqcap \ldots \sqcap C_n$
(`or` $C_1 \ldots C_n$)	$C_1 \sqcup \ldots \sqcup C_n$
(`not` C)	$\neg C$
(`some` R C)	$\exists R.C$
(`all` R C)	$\forall R.C$
(`some` A C)	$\exists A.C$
(`all` A C)	$\forall A.C$

Table 6.1: FaCT concept expressions

6.1.2 Knowledge Base Syntax

A FaCT TKB is defined by a set of macro calls (or their functional equivalents—see Appendix A) which add axioms to the terminology. Table 6.2 on the following page describes the syntax of FaCT's primary axiomatic macros and how they correspond to terminological axioms in the standard notation. The macros extend those provided by Kris where necessary to deal with the additional features of FaCT. In particular the key-word arguments `:supers` and `:transitive` have been added to role introduction macros, allowing the role hierarchy and transitive roles to be specified, and an `implies` macro has been provided for adding GCIs to the TKB.

FaCT forbids the multiple introduction of concepts, roles and attributes: only one `defconcept` or `defprimconcept` macro is allowed for a given concept name and only one `defprimrole` or `defprimattribute` macro for a given role/attribute name. The restriction on concept introductions is not strictly necessary, as FaCT can deal with general terminologies. However, it has the benefit of alerting users to inadvertent multiple introductions, and it does not reduce expressiveness—if additional primitive introductions are required they can be stated directly as

FaCT syntax	Standard notation
`(defprimconcept` CN C`)`	$\mathsf{CN} \sqsubseteq C$
`(defconcept` CN C`)`	$\mathsf{CN} \doteq C$
`(implies` C D`)`	$C \sqsubseteq D$
`(defprimrole` *RN*`)`	$\mathit{RN} \sqsubseteq \top \times \top$
`(defprimrole` *RN* `:supers` $(R_1 \ldots R_n)$ `:transitive` T`)`	$\begin{cases} \mathit{RN} \in \mathbf{R}_+ \\ \mathit{RN} \sqsubseteq R_1 \\ \vdots \\ \mathit{RN} \sqsubseteq R_n \end{cases}$
`(defprimattribute` *AN*`)`	$\begin{cases} \mathit{AN} \in \mathbf{F} \\ \mathit{AN} \sqsubseteq \top \times \top \end{cases}$
`(defprimattribute` *AN* `:supers` $(R_1 \ldots R_n)$`)`	$\begin{cases} \mathit{AN} \in \mathbf{F} \\ \mathit{AN} \sqsubseteq R_1 \\ \vdots \\ \mathit{AN} \sqsubseteq R_n \end{cases}$

Table 6.2: FaCT axiomatic macros

GCIs, and if addition non-primitive introductions are required they can be transposed into a pairs of GCIs using identity 2.5:

$$\mathsf{CN} \doteq C \quad = \quad \begin{cases} \mathsf{CN} & \sqsubseteq & C \\ C & \sqsubseteq & \mathsf{CN} \end{cases}$$

Note that `defconcept` and `defprimconcept` macros do not cause the concepts they introduce to be classified: to classify the TKB a call must be made to the `classify-tkb` function. When `classify-tkb` is called, FaCT first performs some pre-processing on the axioms in the TKB (see Section 6.2) and then computes the concept hierarchy (see Section 6.3 on page 102).

6.2 Pre-processing Terminological Axioms

This section describes the pre-processing FaCT performs on terminological axioms prior to classification. Although the restriction which FaCT places on introduction axioms ensures that they are unique, the TKB may still contain cycles and GCIs. Pre-processing transforms the TKB into a form which can be dealt with by the $\mathcal{ALCH}f_{R^+}$ algorithm, traps (and if possible corrects) simple errors and performs all the optimisation procedures which must precede classification. The basic steps are:

1. Deal with terminological cycles;

2. Pre-process roles and attributes;

3. Absorb GCIs and form the meta constraint $\mathcal{M}$;

4. Normalise and encode concept expressions.

Each of these steps is described in detail in the following sections.

6.2.1 Terminological Cycles

A method of dealing with terminological cycles by converting concept introduction axioms into GCIs was described in Section 3.3 on page 52. However, this method would run counter to one of the main optimisation strategies described in Chapter 5, which is to reduce the number of GCIs in the terminology. Because lazy unfolding is used in the satisfiability testing algorithm, FaCT is able to deal with terminological cycles in a much more efficient manner.

Treating cyclical primitive concept introduction axioms as GCIs is not necessary because a combination of lazy unfolding and blocking ensures that the algorithm terminates and that the model it constructs satisfies all the primitive introduction axioms in the terminology. When building a tree $\mathbf{T}$, lazy unfolding ensures that if the terminology $\mathcal{T}$ contains an axiom $\mathsf{CN} \sqsubseteq C$, then for any node x in $\mathbf{T}$, $\mathsf{CN} \in \mathcal{L}(x) \Rightarrow C \in \mathcal{L}(x)$. Therefore, in the model represented by $\mathbf{T}$, $\mathsf{CN}^{\mathcal{I}} \subseteq C^{\mathcal{I}}$ and the axiom is satisfied. If C refers either directly or indirectly to CN, termination of the tree construction algorithm is still guaranteed because of the blocking strategy.

Lazy unfolding also takes care of non-primitive introduction axioms $\mathsf{CN} \doteq C \in \mathcal{T}$, provided that C can be unfolded so that it contains only primitive concepts. When building a tree $\mathbf{T}$, lazy unfolding again ensures that, for any node x in $\mathbf{T}$, $\mathsf{CN} \in \mathcal{L}(x) \Rightarrow C \in \mathcal{L}(x)$. Any primitive interpretation (an assignment of values to the interpretations of primitive concepts) will then lead, via the semantics, to an interpretation for CN such that $\mathsf{CN}^{\mathcal{I}} = C^{\mathcal{I}}$. However, if C cannot be unfolded so that it contains only primitive concepts, then it cannot be guaranteed that the model represented by $\mathbf{T}$ satisfies $\mathcal{T}$.

Example 6.1 Unsatisfiability Due To Cyclical Non-primitive Axioms

If a terminology $\mathcal{T}$ is defined as:

$$\mathcal{T} = \{\mathsf{CN}_1 \sqsubseteq \top, \mathsf{CN}_2 \doteq \neg\mathsf{CN}_2\}$$

then $\mathcal{T}$ is obviously unsatisfiable (it has only an empty model):

$$\mathsf{CN}_2^{\mathcal{I}} = \Delta^{\mathcal{I}} - \mathsf{CN}_2^{\mathcal{I}} \quad \Rightarrow \quad \Delta^{\mathcal{I}} = \emptyset$$

Testing the satisfiability of CN_1, however, would cause the algorithm to build a tree representing the model:

$$\begin{array}{rcl} \Delta^{\mathcal{I}} & = & \{x\} \\ \mathsf{CN}_1^{\mathcal{I}} & = & \{x\} \end{array}$$

which is not valid with respect to $\mathcal{T}$.

FaCT deals with this problem by checking each non-primitive concept introduction axiom to see if the definition can be unfolded until it contains only primitive concepts. If this is not the case, then the axiom is transformed into a primitive introduction and a GCI using the identity 2.5 on page 34.

Example 6.2 Eliminating Cyclical Non-primitive Axioms

The terminology $\mathcal{T}$ from Example 6.1 would be transformed to give:

$$\mathcal{T} = \{\mathsf{CN}_1 \sqsubseteq \top, \mathsf{CN}_2 \sqsubseteq \neg\mathsf{CN}_2, \neg\mathsf{CN}_2 \sqsubseteq \mathsf{CN}_2\}$$

The GCI $\neg\mathsf{CN}_2 \sqsubseteq \mathsf{CN}_2$ cannot be absorbed and would would be added to the meta constraint: $\mathcal{M} \longrightarrow \mathcal{M} \sqcap (\mathsf{CN}_2 \sqcup \mathsf{CN}_2)$, which can be simplified to $\mathcal{M} \longrightarrow \mathcal{M} \sqcap \mathsf{CN}_2$. When combined with the primitive concept introduction $\mathsf{CN}_2 \sqsubseteq \neg\mathsf{CN}_2$ this would mean that, for any node x in any tree constructed by the tableaux algorithm w.r.t. $\mathcal{T}$, $\{\mathsf{CN}_2, \neg\mathsf{CN}_2\} \subseteq \mathcal{L}(x)$. The algorithm would therefore return *unsatisfiable* when testing the satisfiability of any concept expression.

6.2.2 Pre-processing Roles and Attributes

Prior to classification, FaCT computes the set of attributes $\mathbf{F}$, the set of transitive roles $\mathbf{R}_+$, and the role hierarchy. FaCT also performs consistency checking and, in some cases, takes remedial action.

When computing the role hierarchy, FaCT caches *all* the super-roles (subsuming roles and attributes) of each role and attribute—normally only the direct (most specific) super-roles would be cached. This does not require a great deal of additional storage, because role hierarchies are generally much smaller than concept hierarchies, and it facilitates the operation of the $\exists$-rule in the tableaux expansion algorithm (see Table 4.4 on page 74).

For each role or attribute R introduced by a `defprimrole` or `defprimattribute` macro, FaCT performs the following steps:

1. The set of all super-roles of R is computed by taking the union of the set of roles and attributes S such that $R \sqsubseteq S$ is an axiom in the terminology, and all of their recursively computed super-roles. A role or attribute appearing in its own super-role set is trapped as an error although, in principal, a set of cyclically defined roles all have the same interpretation and could be replaced by a single role [Neb90a].

2. If $R \in \mathbf{F}$ is an axiom in the terminology (R was introduced by a `defprimattribute` macro), R is added to $\mathbf{F}$. If $R \in \mathbf{F}$ is not an axiom in the terminology (R was introduced by a `defprimrole` macro), but it has an attribute in its super-role set, a warning is issued and R is added to $\mathbf{F}$ (it becomes an attribute); if $R \in \mathbf{R}_+$ is an axiom in the terminology (R

had `:transitive T` in its introduction macro), a further warning is issued as attributes cannot be transitive.

3. If $R \in \mathbf{R}_+$ is an axiom in the terminology (R had `:transitive T` in its introduction macro), R is added to $\mathbf{R}_+$ *unless* it has already been added to $\mathbf{F}$ (transformed into attribute) by step 2.

6.2.3 Absorbing GCIs

FaCT absorbs GCIs into primitive concept introduction axioms, wherever possible, using the procedure described in Section 5.4 on page 83. GCIs which cannot be absorbed are combined to form the meta constraint $\mathcal{M}$:

$$C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n \quad \Rightarrow \quad \mathcal{M} = (D_1 \sqcup \neg C_1) \sqcap \ldots \sqcap (D_n \sqcup \neg C_n)$$

6.2.4 Normalisation and Encoding

FaCT normalises and encodes all concept expressions, including the meta constraint $\mathcal{M}$, extending the procedure described in section 5.3 on page 80 so that concept names are also encoded: non-primitive names are simply replaced with their encoded definitions while primitive names are replaced with a unique new name. For example, if a terminology $\mathcal{T}$ contains the non-primitive introduction axiom $\mathsf{CN} \doteq C$, then C will be encoded as CN_x, where CN_x is a system generated concept name, and CN_x will be substituted for CN wherever it appears in $\mathcal{T}$. If $\mathcal{T}$ contains the primitive introduction axiom $\mathsf{CN} \sqsubseteq C$, then C will be encoded as CN_x, where CN_x is a system generated concept name, and a new system generated concept name CN_y will be substituted for CN wherever it appears in $\mathcal{T}$.

The advantage of this is that FaCT can use internally whatever form of name is most convenient, maintaining a mapping to "external" names only for the purposes of input and output. FaCT actually encodes concepts as integers, with positive concepts being encoded as even integers and their negations as odd integers, so that:

$$\text{Encode}(\neg C) = \begin{cases} \text{Encode}(C) + 1 \text{ if Encode}(C) \text{ is even} \\ \text{Encode}(C) - 1 \text{ if Encode}(C) \text{ is odd} \end{cases}$$

For example, $\top$ is encoded as 0 and $\bot$ is encoded as 1. This has a number of benefits:

- Concepts can be negated simply by inverting their least significant bit.
- Lazy unfolding requires large numbers of accesses to concept definitions, and this can be made more efficient by storing them in an array instead of a hash table.

- The negation normal form of each negated concept definition can be pre-computed and stored in the array.

Note that only the (encoded version of) concept names which are introduced in `defconcept` and `defprimconcept` macros are classified in the concept hierarchy.

6.3 Classifying the Knowledge Base

Like other DL classifiers, FaCT uses its subsumption testing algorithm to compute a partial ordering of the set of named concepts in the TKB. The computed partial ordering is cached in the form of a concept hierarchy, a directed acyclic graph in which each concept is linked to its direct subsumers and subsumees (see Section 2.3 on page 35). This process is called classification.

As subsumption testing is always potentially costly, it is important to ensure that the classification process uses the smallest possible number of tests. Minimising the number of subsumption tests required to classify a concept in the concept hierarchy can be treated as an abstract order-theoretic problem which is independent of the ordering relation (subsumption in this case). However, some additional optimisation can be achieved by using the structure of concept expressions to reveal obvious subsumption relationships and to control the order in which concepts are added to the hierarchy.

The classification algorithm used by FaCT is *enhanced traversal*, an optimised algorithm which is used in the Kris system [BHNP92]. Enhanced traversal computes a concept's subsumers by searching down the hierarchy from the $\top$ concept (the *top search* phase) and its subsumees by searching up the hierarchy from the $\bot$ concept (the *bottom search* phase).

When classifying a concept D, the top search takes advantage of the transitivity of the subsumption relation by propagating failed results from preceding tests. It concludes, without performing a subsumption test, that D is not subsumed by C if it has already been shown that D is not subsumed by some subsumer of C (a more general concept than C):

$$D \not\sqsubseteq C_1 \text{ and } C \sqsubseteq C_1 \Rightarrow D \not\sqsubseteq C$$

To maximise the effect of this strategy, a modified breadth first search is used [Ell92] which ensures that a subsumption test $D \sqsubseteq C$? is not performed until it has been established that D is subsumed by all of C's subsumers.

The bottom search uses a corresponding technique, testing if D subsumes C only when D is already known to subsume all of C's subsumees. Information from the top search is also used by confining the bottom search to those concepts which are subsumed by all of D's subsumers:

$$C \not\sqsubseteq C_1 \text{ and } D \sqsubseteq C_1 \Rightarrow C \not\sqsubseteq D$$

FaCT also uses an enhancement of this technique, described in [BHNP92], which takes advantage of the structural subsumption relationships:

$$\begin{array}{rcl}(C_1 \sqcap \ldots \sqcap C_n) & \sqsubseteq & C_1 \\ & \vdots & \\ (C_1 \sqcap \ldots \sqcap C_n) & \sqsubseteq & C_n\end{array}$$

When a concept CN is introduced with an axiom of the form $\mathsf{CN} \sqsubseteq C$ or $\mathsf{CN} \doteq C$, C is said to be a *told subsumer* of CN, and if C is a conjunctive concept expression $(C_1 \sqcap \ldots \sqcap C_n)$, then $C_1, \ldots, C_n$ are also told subsumers of CN. Moreover, due to the transitivity of the subsumption relation, any told subsumers of $C_1, \ldots, C_n$ are also told subsumers of CN. Before classifying CN, all of its told subsumers which have already been classified, and all their subsumers, are marked as subsumers of CN; subsumption tests with respect to these concepts are therefore rendered unnecessary. The effect of this procedure is enhanced by normalisation and encoding because it is no longer limited to named concepts—a concept whose definition is structurally equivalent to one of the conjuncts in another concept's conjunctive definition will also be recognised as a told subsumer.

FaCT extends this technique in the obvious way to take advantage of structural subsumption relations with respect to disjunctive concept expressions: when a concept CN is introduced with an axiom of the form $\mathsf{CN} \doteq C$ and C is a disjunctive concept expression $(C_1 \sqcup \ldots \sqcup C_n)$, CN is a told subsumer of $C_1, \ldots, C_n$.

To maximise the effect of this technique, concepts are classified in *definition order*—that is, a concept CN is not classified until all of its told subsumers have been classified. KRIS takes advantage of this ordering by omitting the bottom search phase when classifying a primitive concept: its subsumees are simply set to $\{\bot\}$. This is possible with unfoldable terminologies because a primitive concept can only subsume concepts for which it is a told subsumer, and because concepts are classified in definition order, a primitive concept will be classified before any of its subsumees. FaCT cannot use this additional optimisation because, in the presence of GCIs, it cannot be guaranteed that a primitive concept will only subsume concepts for which it is a told subsumer.

Example 6.3 Primitive Subsumption Caused By GCIs

If a terminology $\mathcal{T}$ is defined such that:

$$\{\mathsf{CN}_1 \sqsubseteq \exists R.C, \mathsf{CN}_2 \sqsubseteq \top, \exists R.C \sqsubseteq \mathsf{CN}_2\} \subseteq \mathcal{T}$$

CN_2 is not a told subsumer of CN_1 so CN_1 may be classified first. However, when CN_2 is classified the bottom search phase will discover that it subsumes CN_1 due to the GCI $\exists R.C \sqsubseteq \mathsf{CN}_1$.

6.4 Configuring FaCT

Many of FaCT's features are configurable. This facilitates testing and allows FaCT to adapt its behavior to different reasoning tasks.

6.4.1 Configuring Optimisations

In order to facilitate testing, the GCI absorption, backjumping, heuristic guided search and caching optimisations can be completely disabled, and the normalisation and encoding optimisation can be partially disabled. Normalisation and encoding cannot be completely disabled as many of FaCT's data structures take advantage of the integer encoding of concept expressions, but it is possible to disable the lexical matching which the encode function normally performs and this will eliminate most of the optimisation's benefits (see Section 5.3 on page 80 and Section 5.8 on page 95).

Most of the optimisations are enabled by default, the exceptions being heuristic guided search and caching. Heuristic guided search is disabled by default due to its adverse interaction with the backjumping optimisation (see Section 7.3.1 on page 115). Caching is normally enabled by default but is automatically disabled when FaCT is used to solve individual satisfiability problems (see Section 7.5 on page 124): caching is of little benefit unless multiple tests are being performed on concepts with structural similarities, as is the case when classifying a TKB.

6.4.2 Configuring Reasoning Power

The blocking mechanism in the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm's $\exists$-rule works by comparing the label which would be applied to an R-successor or an $\mathbf{A}$-successor of node x with the labels of all of x's ancestor nodes. This can be costly and is not required for a simpler description language such as $\mathcal{ALC}$. FaCT takes advantage of this by automatically disabling blocking when classifying TKBs, or solving individual satisfiability problems, which do not include transitive roles, GCIs or cyclical definitions.

Chapter 7

Test Methodology and Data

This chapter describes the empirical testing procedures which have been used to evaluate the performance of the FaCT classifier and the effectiveness of the various optimisation techniques.

The test procedures divide into four categories:

- Classifying a realistic TKB—the performance of FaCT was measured using the GALEN ontology [RGG+94], a large TKB from a real application.
- Comparing FaCT with KRIS—the performance of FaCT was compared with that of the KRIS system [BH91c], another tableaux based DL classifier. As KRIS does not support either transitive roles or GCIs, a simplified version of the GALEN ontology was used for this purpose.
- Solving randomly generated satisfiability problems—the performance of FaCT's satisfiability test was measured using randomly generated satisfiability problems, an established test method [GS96b, HS97], and compared with that of other algorithms, in particular KSAT [GS96a] and the FLOTTER/SPASS theorem prover [HS97].
- Testing for correctness—the correctness of FaCT's algorithms was checked empirically using a test suite of "hard problems" adapted from [HKNP94] and by comparing, whenever possible, the results obtained using FaCT with those obtained using other systems.

The chapter is organised as follows: Section 7.1 describes the GALEN ontology and explains how it was translated from GRAIL into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$; Section 7.2 provides details of the testing methodology; Section 7.3 describes the tests performed on FaCT using the GALEN terminology; Section 7.4 describes the tests which compare the performance of FaCT and KRIS; Section 7.5 describes the tests which compare the performance of FaCT and KSAT; Section 7.6 describes the correctness testing procedures; and finally, Section 7.7 summarises the results of the various tests and discusses the effectiveness of the optimisation techniques.

7.1 The GALEN Ontology

Many of the tests in this chapter use a real TKB developed as part of the GALEN project. The GALEN TKB is is a high level ontology which has been designed to form the foundation of a large concept model representing medical terminology; it contains 2,740 concepts, 413 roles and 1,214 GCIs.

The GALEN TKB has been created using the specially developed GRAIL DL which supports a primitive role hierarchy, transitive roles and GCIs. GRAIL has a limited terminological language—only conjunction and exists restrictions are allowed in concept expressions—and an unusual syntax which restricts the way concept expressions can be formed. Before using the GALEN TKB with FaCT it was necessary to translate it into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$.

7.1.1 Translating GRAIL into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$

Concept Expressions and Introduction Statements

Although GRAIL's semantics are operationally rather than formally defined, translating GRAIL concept expressions and introduction statements into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ is straightforward and uncontroversial: Table 7.1 gives $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ equivalents for GRAIL concept expressions and for its most important terminological statements. The `which` statement is GRAIL's only mechanism for forming concept expressions, and they are restricted by the syntax of the statement to a combination of conjunction and exists restrictions. The `newSub` and `name` statements serve as primitive and non-primitive concept introduction axioms while the `topicNecessarily` statement is used to add GCI axioms to the TKB.

GRAIL	$\mathcal{ALCH}\boldsymbol{f}_{R^+}$
C `which` $\langle R_1\, C_1 \ldots R_n\, C_n \rangle$	$C \sqcap \exists R_1.C_1 \sqcap \ldots \sqcap \exists R_n.C_n$
C `newSub` CN	$\mathsf{CN} \sqsubseteq C$
C `name` CN	$\mathsf{CN} \doteq C$
C `topicNecessarily` $\langle R_1\, C_1 \ldots R_n\, C_n \rangle$	$C \sqsubseteq \exists R_1.C_1 \sqcap \ldots \sqcap \exists R_n.C_n$

Table 7.1: GRAIL concept statements and equivalent $\mathcal{ALCH}\boldsymbol{f}_{R^+}$

Role Introduction Statements

Unfortunately, the semantics of GRAIL's role statements are less clear. In GRAIL, pairs of primitive roles are introduced with the `newAttribute` statement, e.g.:

$$\textsf{\textit{StructuralPartitiveAttribute}}\ \texttt{newAttribute}\ \textsf{\textit{hasLayer}}\ \textsf{\textit{isLayerOf}}\ \texttt{manyMany} \quad (7.1)$$

where $\mathsf{StructuralPartitiveAttribute}$ is an existing role, $\mathsf{hasLayer}$ and $\mathsf{isLayerOf}$ are a pair of new roles, and `manyMany` is a keyword which specifies that they are to be roles rather than attributes. The role hierarchy is established by making $\mathsf{hasLayer}$ a sub-role of $\mathsf{StructuralPartitiveAttribute}$ (see Figure 4.1 on page 59) and $\mathsf{isLayerOf}$ a sub-role of $\mathsf{InverseStructuralPartitiveAttribute}$, where $\mathsf{InverseStructuralPartitiveAttribute}$ is the role which was paired with $\mathsf{StructuralPartitiveAttribute}$ when it was introduced. The built-in most general role, $\mathsf{Attribute}$, is provided as a foundation for the role hierarchy and is assumed to be paired with itself. The general form of `newAttribute` statements is:

$$\mathsf{RN}\ \texttt{newAttribute}\ \mathsf{RN}_1\ \mathsf{RN}_2\ k$$

where RN_1 and RN_2 are two new primitive sub-roles of, respectively, RN and the role with which RN was paired in its introduction axiom. The keyword k determines whether RN_1 and RN_2 are roles or attributes and can be one of `oneOne` (both RN_1 and RN_2 are attributes), `manyMany` (neither are attributes), `oneMany` (RN_1 is an attribute) or `manyOne` (RN_2 is an attribute).

The syntax of the new attribute statement, and its use in the Galen model, seems to imply that RN_2 is the inverse of RN_1 and suggests that statement (7.1) could be taken to mean:

$$\begin{array}{rcl} \mathsf{hasLayer} & \sqsubseteq & \mathsf{StructuralPartitiveAttribute} \\ \mathsf{isLayerOf} & \doteq & \mathsf{hasLayer}^{-1} \end{array}$$

However, this is not the case: the Grail classifier does not reason about inverse roles[1] and the two new roles are treated semantically as though they were unconnected. What Grail does do, is to use the pairing of roles to expand certain statements so that their syntactic inverse is also added to the TKB. For example, as well as the `topicNecessarily` statement described in Table 7.1 on the preceding page, Grail also provides the `necessarily` statement which makes use of the pairing of roles to add a pair of `topicNecessarily` statements. Given the role introduction statement (7.1) above, the statement:

$$\mathsf{GramPositiveBacterialCell}\ \texttt{necessarily}\ \langle \mathsf{hasLayer}\ \mathsf{GramPositiveCellWall}\rangle$$

would be expanded by Grail into the two `topicNecessarily` statements:

$$\begin{array}{c} \mathsf{GramPositiveBacterialCell}\ \texttt{topicNecessarily}\ \langle \mathsf{hasLayer}\ \mathsf{GramPositiveCellWall}\rangle \\ \mathsf{GramPositiveCellWall}\ \texttt{topicNecessarily}\ \langle \mathsf{isLayerOf}\ \mathsf{GramPositiveBacterialCell}\rangle \end{array}$$

The approach taken when translating the Galen ontology into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ has been to translate a `newAttribute` axiom as two role introduction axioms:

$$\mathsf{RN}\ \texttt{newAttribute}\ \mathsf{RN}_1\ \mathsf{RN}_2\ k \quad\longrightarrow\quad \left\{ \begin{array}{l} \mathsf{RN}_1 \sqsubseteq \mathsf{RN} \\ \mathsf{RN}_2 \sqsubseteq \mathsf{RN}' \end{array} \right.$$

[1]Reasoning with inverse roles, attributes and transitivity is known to be problematical [Sch91, GL96].

where RN' is the role which was paired with RN in its introduction axiom. Whether RN_1 and RN_2 are roles or attributes is determined by the value of k. GRAIL `necessarily` statements are then expanded into two `topicNecessarily` statements and translated as per Table 7.1 on page 106:

$$C \texttt{ necessarily } \langle \mathsf{RN}_1\, D\rangle \quad\longrightarrow\quad \left\{ \begin{array}{lcl} C & \sqsubseteq & \exists \mathsf{RN}_1.D \\ D & \sqsubseteq & \exists \mathsf{RN}_2.C \end{array} \right.$$

Specialisation

A further problem arises when attempting to translate the GRAIL `specialisedBy` statement, which is used to declare the *specialisation* of one role by another. For example, the GRAIL statement:

$$\mathsf{hasLocation} \texttt{ specialisedBy } \mathsf{isPart}$$

is intended to capture the meaning that, if object_1 *hasLocation* in object_2 and object_2 *isPart* of object_3, then object_1 also *hasLocation* in object_3: e.g., if a fracture is located in a part of a bone then it is also located in the bone itself. Similar expressiveness is provided in the CycL DL [LG89] by the `transfersThro` statement, which allows relationships established with one role to be "transfered through" those established with another. However, CycL's `transfersThro` statement does not affect terminological reasoning: it only affects reasoning about individuals in CycL's ABox (see Section 1.1 on page 16).

Unfortunately the formal semantics of the `specialisedBy` statement is not clear. It is perhaps most obvious to think of it as a role inclusion axiom:

$$R \texttt{ specialisedBy } S \quad\longrightarrow\quad R \circ S \sqsubseteq R$$

While this appears to capture the intended meaning very neatly, it has the obvious disadvantage that the resulting axioms are not supported by $\mathcal{ALCH}\boldsymbol{f}_{R^+}$. In fact there is no known subsumption algorithm for a DL supporting such axioms, nor is it known if such a logic would be decidable.

An alternative approach is to use `specialisedBy` statements to expand concept expressions so that, given R `specialisedBy` S, the expression $\exists R.C$ would be expanded to give $\exists R.C \sqcup \exists R.(\exists S.C)$, or perhaps $\exists R.(\exists S^*.C)$. The problems with this approach are that it is uncertain which, if any, of the two expanded expressions best capture the intended meaning; that $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ does not support the transitive reflexive role forming operator required by the second expansion; and that there are cycles in the `specialisedBy` statements in the GALEN ontology (e.g., R `specialisedBy` S, S `specialisedBy` R), so the expansion process would be non-terminating[2].

[2]The designers of the GALEN ontology have stated, in a personal communication, that the existence of cycles in `specialisedBy` statements is an error which will be corrected in future versions of the ontology.

Given the problems with `specialisedBy` statements, it was decided to discard them when translating the Galen ontology. The resulting ontology can still be considered "realistic": it will be identical to the original apart from the loss of a small number of subtle subsumption inferences.

Transitivity and the Role Hierarchy

Grail supports two more role statements which can be straightforwardly translated into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$. The `transitiveDown` statement asserts that a given role is transitive and is translated as:

$$\mathsf{RN}\ \texttt{transitiveDown} \quad \longrightarrow \quad \mathsf{RN} \in \mathbf{R}_+$$

The `addSub` statement asserts a role inclusion, forming part of the role hierarchy, and is translated as:

$$\mathsf{RN}_1\ \texttt{addSub}\ \mathsf{RN}_2 \quad \longrightarrow \quad \mathsf{RN}_2 \sqsubseteq \mathsf{RN}_1$$

The translation of Grail role statements into $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ is summarised in Table 7.2

Grail	$\mathcal{ALCH}\boldsymbol{f}_{R^+}$
RN `newAttribute` RN_1 RN_2 k	$\begin{cases} \mathsf{RN}_1 \sqsubseteq \mathsf{RN} \\ \mathsf{RN}_2 \sqsubseteq \mathsf{RN}^{-1} \end{cases}$
RN `transitiveDown`	$\mathsf{RN} \in \mathbf{R}_+$
RN_1 `addSub` RN_2	$\mathsf{RN}_2 \sqsubseteq \mathsf{RN}_1$
RN_1 `specialisedBy` RN_2	*ignored*

Table 7.2: Grail role axioms and equivalent $\mathcal{ALCH}\boldsymbol{f}_{R^+}$

7.1.2 Test Knowledge Bases

Three TKBs were used in the various experiments. The first of these, **TKB0**, includes both transitive roles and GCIs, which means that it can be used with FaCT but not with Kris (see Section 2.4.2 on page 41); **TKB1** and **TKB2**, are simplified versions of **TKB0** which can be used with either FaCT or Kris. Table 7.3 provides details of the composition of each of the TKBs.

TKB0 was derived from the Galen ontology by translating the Grail TKB as described in Section 7.1.1. **TKB1** is a simplification of **TKB0** which is suitable for use with Kris. The simplification steps were as follows:

Characteristic	**TKB0**	**TKB1**	**TKB2**
Concepts	2,740	2,719	3,917
primitive	2,041	2,041	2,041
non-primitive	699	678	1,876
Roles	413	413	413
attributes	150	150	150
transitive	26	0	0
GCIs	1,214	1,214	0
absorbed	0	1,214	0
non-absorbed	1,214	0	0

Table 7.3: Test TKB characteristics

1. Role inclusion axioms were discarded as KRIS does not support a role hierarchy.
2. Transitivity axioms were discarded as KRIS does not support transitive roles.
3. GCI axioms were pre-absorbed into primitive concept introduction axioms, as described in Section 6.2.3 on page 101.
4. Terminological cycles were eliminated from the TKB as KRIS is unable to deal with them.
5. Synonyms such as $\mathsf{Haemoglobin} \doteq \mathsf{Hemoglobin}$ were eliminated from the TKB as KRIS does not deal with these correctly. This accounts for the slightly reduced number of non-primitive concepts.
6. The TKB was ordered so that a concept's introduction axiom always appears before the concept is used in a concept expression. The resulting TKB is more convenient for detailed performance analyses as it is possible to classify concepts one at a time in the order in which their introductions occur in the TKB.

In practice, trying to classify **TKB1** using KRIS proved to be highly intractable and had to be abandoned. In order to create a TKB which KRIS was able to classify, a more radical simplification was applied to **TKB0** to produce **TKB2**. In **TKB2**, instead of absorbing GCIs into primitive introductions, they were converted into non-primitive concept introductions:

$$C \sqsubseteq D \quad \longrightarrow \quad \mathsf{CN} \doteq C \sqcap D$$

where CN is a unique, system generated name. This method of eliminating GCIs was chosen because it increased the size of the TKB by adding "realistic" concepts: if the TKB includes the assertion that C implies D, it is reasonable to assume that concepts of type $C \sqcap D$ are realistic. Non-primitive concepts were prefered, as their classification on the basis of subsumption reasoning is a key characteristic of DLs. After GCIs had been dealt with, role inclusions, transitivity, cycles and synonyms were eliminated and the TKB was ordered, in the same way as when converting **TKB0** to **TKB1**. The elimination of synonyms accounts for the fact that the number of non-primitive concepts in **TKB2** is less than the sum of the non-primitive concepts and GCIs in **TKB0**.

7.2 Testing Methodology

The majority of the tests described in this chapter use CPU time as their measure of performance. CPU time has been the most widely used performance indicator in earlier analyses [BHNP92, HKNP94, Spe95, SvRvdVM95, GS96b, HS97]; it has the advantage of being easy to measure and of being common to all algorithms, although its value will clearly be affected by implementation style, language and hardware. Empirically, it can also been seen to be the limiting factor as regards the usefulness of tableaux algorithms: the tests conducted in earlier analyses, as well as those described here, indicate that computational intractability is a serious problem in terminological reasoning, whereas excessive storage requirements are not. For example, when classifying **TKB0** with the encoding optimisation disabled, some individual satisfiability tests took over 18,000s, while the largest tree ever stored in memory contained only 417 pre-tableau nodes. It is worth noting, however, that storage requirements could become a more important factor if further improvements in the optimisation techniques allow significantly larger problems to be solved.

Other performance measures used include:

- Search space—the amount of backtracking search performed during satisfiability tests. This indicates the extent to which the various optimisations are successful in pruning the potentially very large search space.
- Satisfiability tests—the number of satisfiability tests performed when classifying a concept. This indicates the extent to which classification times depend on the “hardness” of individual satisfiability tests and the extent to which they depend on the number of tests performed. It is also useful to compare the number of satisfiability tests with the number of subsumption tests as this indicates the success rate of the caching optimisation.

7.2.1 Percentile Plots

Many of the graphs in the following sections plot percentile CPU times. The 50th percentile is the time taken by the problem in the test set such that 50% of the problems took a *shorter* time (this is the median time); the 100th percentile is the time taken by the hardest problem in the test set. This is more informative than simply plotting the average time or the median time: average time is often distorted by a small number of very hard problems, and so does not indicate the typical solution time; median time masks hard problems, and so does not indicate either the total time or the worst-case time.

In TKB classification tests, the percentile classification times have been calculated for groups of concepts, usually of size 100, and plotted against the number of concepts which have already been classified (classified TKB size). These plots necessarily exclude pre-processing time (see Section 6.2 on page 98), but this is small in comparison to total classification time: e.g., the time required to pre-process **TKB0** was ≈12s compared to ≈361s to classify it. Presenting the data in this way shows how both typical and worst case performance varies as the size of the classified concept hierarchy increases. This is of interest because one of the major problems with existing DL systems is the degradation in their performance as TKB size increases [HKNP94, SvRvdVM95].

In tests using randomly generated satisfiability problems, groups of 100 problems are generated at each data point, using the same random generation parameters. The median solution times, or in some cases the 50th, 60th 70th, 80th, 90th and 100th percentile solution times, are plotted against L/N, the ratio of the number of disjunctive clauses to the number of atomic primitive concepts (see Section 7.5 on page 124).

7.2.2 Data Gathering

FaCT has an integrated profiling facility, and for tests which only involved FaCT (i.e., using **TKB0**) this facility was used to gather performance data. Kris has no such facility, so when comparing FaCT with Kris it was necessary to use an external timing procedure which measures the time taken to classify each concept using an `add-concept` statement (see Appendix A), which forces the concept to be classified immediately. Imposing an order on the classification of concepts degrades performance as it prevents the classifier from maximising the benefit of the told subsumer optimisation by classifying concepts in definition order (see Section 6.3 on page 102). For this reason, the external timing mechanism was used with both systems when comparing the performance of FaCT and Kris (i.e., using **TKB1** and **TKB2**).

7.2.3 System Specification

The current version of FaCT is written in Lisp and compiled using Allegro Lisp with the compiler optimisation settings speed=3, safety=0, compilation-speed=0, space=0, and debug=0. No attempt has been made at low-level optimisation or tuning of the Lisp code, for example by using type declarations or by trying to minimise the amount of consing [Nor92]. All tests have been performed on a Sun SPARCstation 20/61 equipped with a 60MHz superSPARC processor, a 1Mbyte off-chip cache and 128Mbytes of RAM.

7.3 Classifying the GALEN Terminology

The purpose of these tests was to analyse and evaluate the performance of FaCT, and the effectiveness of the various optimisation techniques, when classifying a large "realistic" TKB, i.e., **TKB0**.

Function	Time(s)
Load	6.93
Pre-process	11.87
Classify	360.53
TOTAL	379.33

Table 7.4: Total classification times (**TKB0**/FaCT)

Table 7.4 shows the time taken by FaCT to classify **TKB0**; Figure 7.1 shows the 50th percentile (median) and 100th percentile classification time per concept plotted against the number of concepts which have already been classified (classified TKB size).

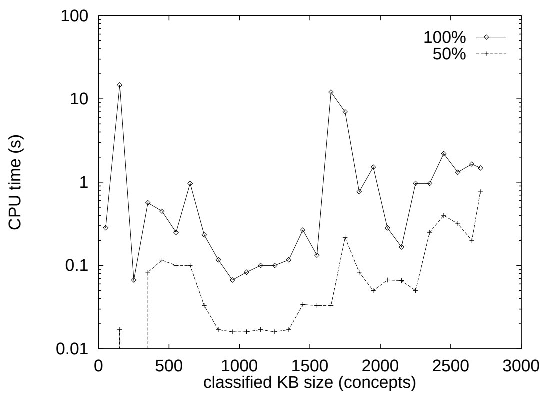

Figure 7.1: Classification time per concept -v- TKB size (**TKB0**/FaCT)

The time required to classify single concepts ranges from ≈0s to ≈15s and does not appear to be directly correlated with the number of classified concepts. The variation in classification time is due in part to variations in the "hardness" of individual satisfiability tests and in part to variations in the number of satisfiability tests performed when classifying concepts in different parts of the TKB.

The number and hardness of satisfiability tests required to classify different concepts reflects the kinds of concepts being classified, the complexity of the hierarchy in the area of the TKB to which they belong and the order in which they are classified. The consistently high median classification times in the region of classified TKB size 350–650, for example, are caused by the classification of part of the hierarchy concerned with human anatomy. Concepts concerned with anatomy form a populous and complex sub-hierarchy, and many of the primitive concepts near the the top of this sub-hierarchy have large numbers of disjunctive clauses in their introduction axioms due to the absorption of GCIs. The most general anatomical concept, $\mathsf{BodyStructure}$, is at the top of a sub-hierarchy containing 780 concepts, and has 10 disjunctive clauses in its introduction axiom. Classification of a concept in the anatomy sub-hierarchy below $\mathsf{BodyStructure}$ therefore tends to require a greater than average number of satisfiability tests, due the size and complexity of the hierarchy, and more expensive satisfiability tests, due to the search space which may have to be explored when expanding disjunctive expressions.

The low median classification times in the region of classified TKB size 850–1,350, on the other hand, are caused by the classification of a primitive sub-hierarchy which contains state value concepts such as acute, often and $\mathsf{persistent}$. The concept State, for example, is at the top of a sub-hierarchy containing 340 concepts, none of which have disjunctive clauses in their introduction axioms. This sub-hierarchy consists entirely of asserted primitive subsumption relations, e.g.:

$$\begin{aligned}
\mathsf{ProcessState} &\sqsubseteq \mathsf{State}\\
\mathsf{TemporalState} &\sqsubseteq \mathsf{ProcessState}\\
\mathsf{ChronicityState} &\sqsubseteq \mathsf{TemporalState}\\
\mathsf{acute} &\sqsubseteq \mathsf{ChronicityState}
\end{aligned}$$

Classification therefore requires relatively few, simple satisfiability tests, most of which are avoided by a combination of the told subsumer and caching optimisations.

Peaks in the 100th percentile classification time are largely the result of the order in which concepts are classified: it takes longer to classify the first concepts in a sub-hierarchy because many of the concepts referred to in their introduction axioms will not have been previously encountered by the classifier and so will not have cached models. In complex and highly connected parts of the TKB, this can cause a cascade of satisfiability testing and caching before classification can continue.

The peak of 14.7s which appears at classified KB size = 150, for example, is caused by the classification of the concept BodyCavity. FaCT first tests the satisfiability of BodyCavity in order to cache the resulting model for use in avoiding subsequent subsumption tests. During the tableaux expansion many as yet uncached concepts are encountered, in particular in an area of the anatomical hierarchy which describes the knee joint in considerable detail. This causes FaCT to performs satisfiability tests on, and cache models for, more than 170 additional concepts before continuing with the classification of BodyCavity.

Although caching will cause some overhead, FaCT is not performing a great deal of additional work during these satisfiability tests, as the concepts encountered in the tableaux expansion would have to be unfolded and expanded in any case. Moreover, caching means that the work FaCT is performing can be used to speed up subsequent tests, both during the classification of BodyCavity and of subsequent anatomical concepts. Without caching, classifying BodyCavity takes ≈150s, and classifying many of the other anatomical concepts in the TKB size 350–850 region takes at least as long (compare Figure 7.1 with Figure 7.4 on page 119).

The peak in classification time for BodyCavity reflects FaCT's "on demand" method of organising the work which it must perform in order to classify a TKB. If FaCT was being used interactively to add concepts to a hierarchy, such peaks would be less likely to arise as introduction axioms could only refer to concepts which had already been classified. This effect can be observed in Figure 7.8 on page 123, which shows FaCT's performance with **TKB1**, a TKB which has been sorted so that all concepts are classified before being used in introduction axioms: the 100th percentile classification time plot for this TKB is much flatter than that for **TKB0**.

7.3.1 Testing Optimisation Techniques

In order to test the effect of some of the optimisation techniques on FaCT's performance, the above test was repeated several times with one or more of the optimisations disabled. It only proved possible to perform a detailed analysis with respect to the encoding and caching optimisations: it is not possible to disable semantic branching, and with either GCI absorption or backjumping disabled the resulting intractability made it impossible to classify the TKB.

It will be recalled that in FaCT, the heuristic guided search optimisation, using MOMS heuristic, is disabled by default due to its interaction with backjumping (see Section 6.4.1 on page 104). The adverse effect of MOMS heuristic was demonstrated by repeating the TKB classification test with the optimisation enabled.

GCI Absorption

With GCI absorption disabled, FaCT failed to classify a single concept. This is not particularly surprising as each of the 1,214 GCIs will cause a disjunction to be added to the label of the root node and of every R-successor and $\mathbf{A}$-successor node in the tree constructed by the tableaux algorithm. The size of the resulting search space, which is exponential in the number of disjunctions, might best be described as "discouraging".

Backjumping

With backjumping disabled ≈150 concepts were successfully classified, but serious intractability was soon encountered: the attempted classification was abandoned after the algorithm had used 100 hours of CPU time trying to classify the concept BodyCavity.

Normalisation and Encoding

It is not possible to completely disable normalisation and encoding (see Section 6.4.1 on page 104), but the lexical matching which the encode function performs can be disabled, and this will eliminate most of the optimisation's benefits. Table 7.5 shows the time taken by FaCT to classify **TKB0** with lexical encoding disabled: the total classification time increased by a factor of ≈158.

Function	Time(s)
Load	7
Pre-process	12
Classify	59,807
TOTAL	59,826

Table 7.5: Total classification times without encoding (**TKB0**/FaCT)

Figure 7.2 on the following page contrasts FaCT's normal performance with its performance when lexical encoding is disabled; in both cases the 50th percentile (median) and 100th percentile classification time per concept is plotted against classified TKB size. It can be seen from these graphs that, while lexical encoding improves performance generally, its most significant effect is to smooth peaks in the 100th percentile time. Without lexical encoding the concept KneeJointRecessus, for example, took over 29,250s to classify compared to 2.2s with lexical encoding.

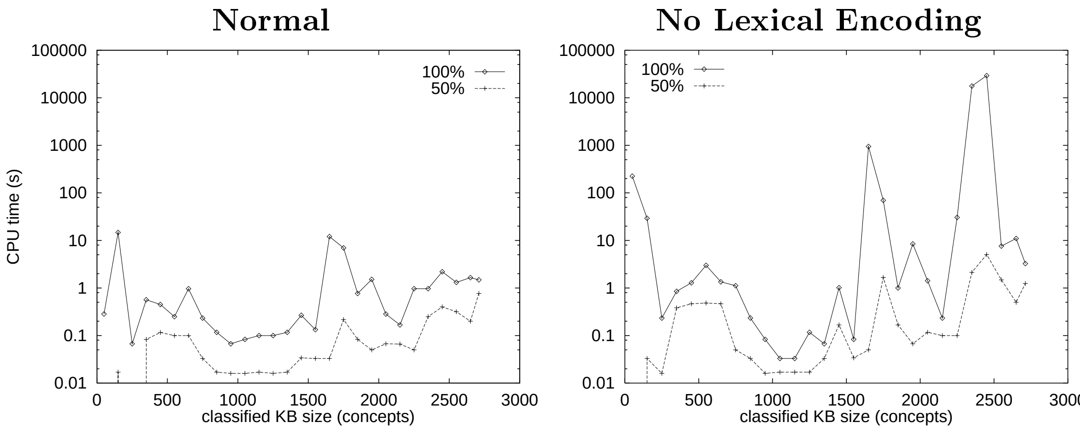

Figure 7.2: Classification with and without lexical encoding (**TKB0**/FaCT)

The increased time taken to classify concepts without encoding is primarily the result of a small number of very costly failed satisfiability tests: e.g., discovering that $\mathsf{KneeJointRecessus} \sqcap \neg\mathsf{MirrorImagedBodyStructure}$ is not satisfiable took $\approx$19,000s. Unsatisfiable tests are normally less expensive than satisfiable ones because lexically obvious contradictions in concept expressions are quickly discovered by the lazy unfolding optimisation. Without lexical encoding, however, this optimisation is much less effective, and more unfolding and expansion may be required in order to discover unsatisfiability. Where the expansion encounters disjunctive concepts, the unsatisfiability of every $\sqcup$-successor must be demonstrated and this can lead to an explosion in the search space.

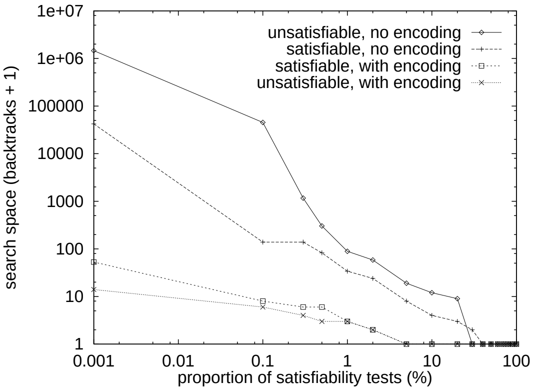

Figure 7.3: Search space with and without lexical encoding (**TKB0**/FaCT)

The increase in search space caused by disabling lexical encoding is illustrated by Figure 7.3 which plots search space (number of back-tracks + 1) against the percentage of satisfiability tests which led to at least that size of search space being explored. The plots are separated into satisfiable and unsatisfiable tests, with and without lexical encoding. Note that both x and y scales are logarithmic. It can be seen that with lexical encoding enabled over 95% of satisfiability problems are solved without any backtracking, and that in the worst case this increases to only 13 backtracks for an unsatisfiable problem and 52 backtracks for a satisfiable problem. When lexical encoding is disabled only 60% of problems are solved without backtracking, and in the worst case this increases to 42,270 backtracks for a satisfiable problem and 1,458,394 backtracks for an unsatisfiable problem.

Caching

Table 7.6 shows the time taken by FaCT to classify **TKB0** with caching disabled: the total classification time increased by a factor of ≈177.

Function	Time(s)
Load	7
Pre-process	12
Classify	67,147
TOTAL	67,166

Table 7.6: Total classification times without caching (**TKB0**/FaCT)

Figure 7.4 on the following page contrasts FaCT's normal performance with its performance when caching is disabled; in both cases the 50th percentile (median) and 100th percentile classification time per concept is plotted against classified TKB size. Like lexical encoding, caching has the effect of smoothing peaks in the classification times; although disabling caching does not cause such high peaks (a maximum of ≈2,255s), its effect is more uniform than that of lexical encoding and its overall effect is slightly greater.

The increased time taken to classify concepts without caching is primarily the result of an increase in the number of satisfiability tests performed. This is illustrated by Figure 7.5 on the following page, which plots the median number of satisfiability tests required to classify each concept against classified TKB size, both with and without caching. When classifying **TKB0** with caching enabled, FaCT performs a total of 122,695 subsumption tests but only 23,492 satisfiability tests; with caching disabled, the number of satisfiability tests is equal to the number of subsumption tests.

Heuristic Guided Search

Heuristic guided search, using MOMS heuristic, is disabled by default in the FaCT system. Table 7.7 on page 120 shows the time taken by FaCT to classify

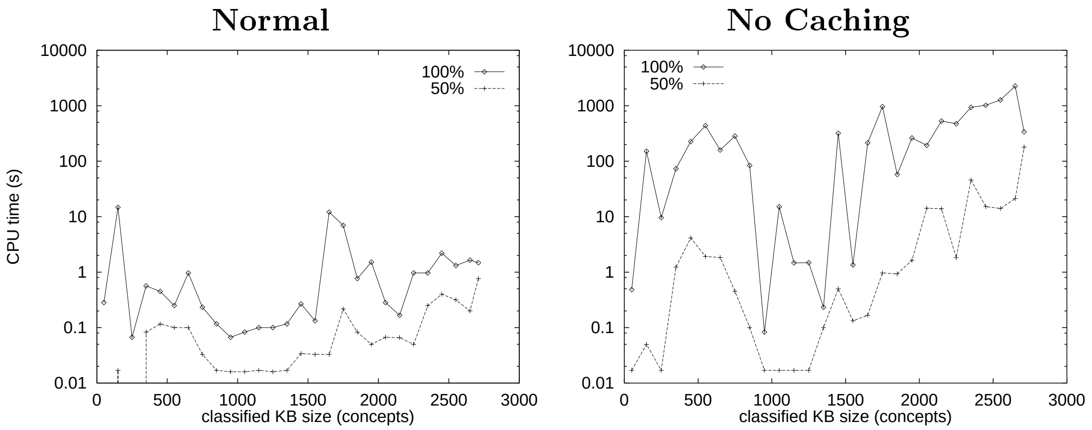

Figure 7.4: Classification with and without caching (**TKB0**/FaCT)

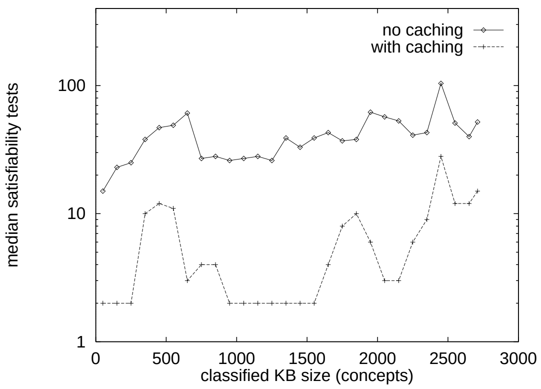

Figure 7.5: Satisfiability tests with and without caching (**TKB0**/FaCT)

TKB0 with MOMS heuristic enabled: the total classification time increased by a factor of $\approx$4.

Figure 7.6 on the following page contrasts FaCT's normal performance with its performance when MOMS heuristic is enabled; in both cases the 50th percentile (median) and 100th percentile classification time per concept is plotted against classified TKB size. It can be seen from these graphs that the use of MOMS heuristic degrades performance generally, with the effect being most significant in the peak 100th percentile times. With MOMS heuristic enabled, the worst case classification time (for the concept BodyCavity) increased from 14.7s to 159.2s.

Although some time will clearly be required to evaluate the heuristic function, the increased time taken to classify concepts with MOMS heuristic enabled is primarily the result of an increase in the size of the search space. This is illustrated by Figure 7.7 on page 121 which plots search space (number of back-tracks + 1)

Function	Time(s)
Load	7
Pre-process	12
Classify	1,574
TOTAL	1,593

Table 7.7: Total classification times with MOMS heuristic (**TKB0**/FaCT)

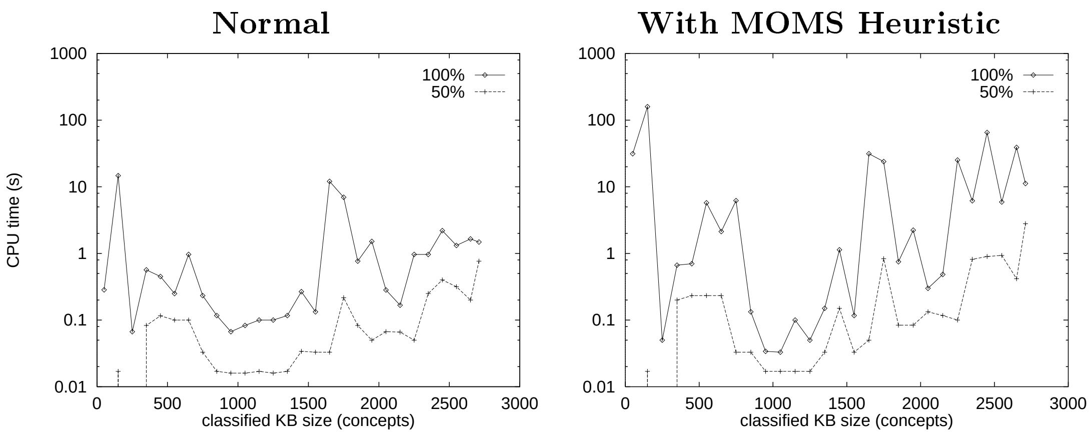

Figure 7.6: Classification with and without MOMS heuristic (**TKB0**/FaCT)

against the percentage of satisfiability tests which led to at least that size of search space being explored. The plots are separated into satisfiable and unsatisfiable tests, with and without MOMS heuristic. Note that both x and y scales are logarithmic. It can be seen that with MOMS heuristic enabled, the number of backtracks required to solve the hardest problems increased dramatically, particularly for satisfiable problems: in the worst case (testing the satisfiability of BodyCavity) the number of backtracks increased from 52 to 4,833.

The increase in the size of the search space when MOMS heuristic is enabled is due to an interaction between the heuristic and the backjumping optimisation. When MOMS heuristic is *not* used to select the disjunct on which to branch, the algorithm chooses a disjunct from the first unexpanded disjunction in $\mathcal{L}(x)$. As a result of the implementation, this will usually be the "oldest" unexpanded disjunction, that is the disjunction whose dependency set contains the smallest maximum dependency value. This is quite a useful heuristic for selecting branching disjuncts because it maximises the pruning effect of backjumping when a clash is discovered. Using MOMS heuristic can cause disjuncts to be selected from "newer" unexpanded disjunctions, and this reduces the effectiveness of backjumping.

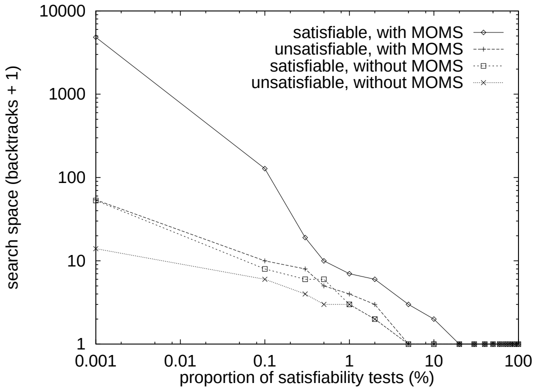

Figure 7.7: Search space with and without MOMS heuristic (**TKB0**/FaCT)

Because of the limited effectiveness of MOMS heuristic in typical concept satisfiability problems, the adverse effect of the interaction with backjumping outweighs the benefit of the heuristic. The heuristic relies for its effectiveness on finding the same disjuncts recurring in multiple unexpanded disjunctions: this is likely in SAT problems, where the disjuncts are propositional variables, and where the number of different variables is usually small compared to the number of disjunctive clauses (otherwise problems would, in general, be trivially satisfiable); it is unlikely in concept satisfiability problems, where the disjuncts are concept expressions, and where the number of different concept expressions is usually large compared to the number of disjunctive clauses. As a result, the heuristic will often discover that all disjuncts have similar or equal MOMS priorities, and the guidance it provides is not particularly useful.

7.4 Comparing FaCT and KRIS

Although the Galen TKB was created using Grail, a direct comparison between FaCT and Grail would be of limited value:

- They use different host environments: FaCT is Lisp based whereas Grail is SmallTalk based. This makes it difficult to separate differences in the performance of the algorithms from differences in the performance of the environments.
- They perform different subsumption reasoning tasks: FaCT performs complete tableaux based subsumption tests whereas Grail performs incomplete structural tests.
- They perform different classification tasks: Grail has an integrated sanctioning mechanism so that, as well as classifying concepts and checking that

they are logically coherent, it also uses sanctioning information to check that descriptions are "sensible" [RBG+97].

- They use different description languages: they cannot be tested with an identical TKB.

It was therefore decided to test FaCT by comparing it with KRIS[3]. In contrast to GRAIL, KRIS is Lisp based, uses a sound and complete tableaux subsumption test (it is the only other available system to do so) and can classify a simplified FaCT TKB.

In practice, trying to classify **TKB1** using KRIS proved to be highly intractable and had to be abandoned: after approximately 100 hours of CPU time KRIS had only classified a small proportion of the TKB. Table 7.8 shows the relative performances of FaCT and KRIS without the overhead imposed by gathering performance data; Figure 7.8 on the following page shows the 50th percentile (median) and 100th percentile classification time and the 50th and 100th percentile number of satisfiability tests per concept classification plotted against TKB size. It can be seen that when using KRIS the classification times, particularly the 100th percentile time, escalated rapidly and soon reached the point where the algorithm was effectively non-terminating: after classifying only ≈10% of **TKB1**, KRIS had already taken over 1,900 times longer than FaCT.

	Time (s)	
Function	FaCT	KRIS
Load	6.03	135.90
Pre-process	0.85	–
Classify	204.03	≫400,000
TOTAL	210.91	≫400,000

Table 7.8: Total classification times (**TKB1**)

As little detailed information could be derived from this experiment, a further comparison was carried out using **TKB2**. Although the complexity of **TKB2** is significantly reduced compared to the GALEN TKB, comparing the performance of KRIS and FaCT when classifying this TKB is still of interest for several reasons:

- It provides a useful correctness test for the FaCT classifier.
- It indicates the impact the optimisation techniques would be likely to have on a less expressive DL and a TKB which does not contain numerous large disjunctions resulting from the absorption of GCIs.

[3]For completeness, it should be noted that the GRAIL classifier takes approximately 24 hours to classify the GALEN ontology TKB using the standard test hardware (see Section 7.2.3 on page 112).

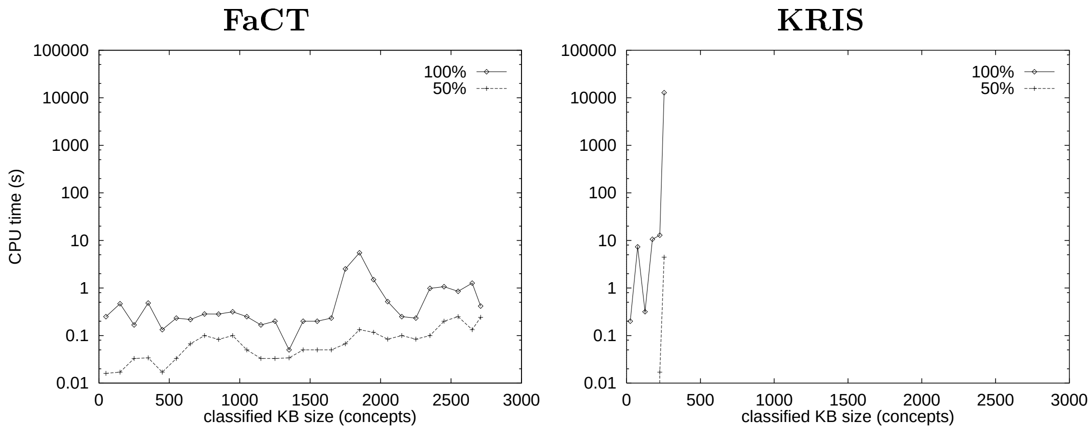

Figure 7.8: Classification time per concept -v- classified TKB size (**TKB1**)

- It demonstrates the extent to which FaCT's performance improves when only a subset of the available expressiveness is used: the ability to provide "pay as you go" performance has been identified as a desirable characteristic for knowledge representation systems [DP91].

Table 7.9 on the following page shows the times taken by both FaCT and Kris to classify **TKB2** without the overhead imposed by gathering performance data; Figure 7.9 on the following page shows the 50th percentile (median) and 100th percentile classification times plotted against TKB size. Overall, FaCT outperformed Kris by a factor of ≈6.6.

It can be seen that much of FaCT's improved performance is achieved by smoothing out the large peaks which occur in Kris's classification times. This is particularly noticeable looking at the 100th percentile which is relatively consistent (0.02–0.50s) with FaCT and widely varying (0.02–46.18s) with Kris: in a small number of cases Kris can take several hundred times longer than the median time to classify a concept and in these cases Kris can be >100 times slower than FaCT. Median classification times on the other hand are relatively similar, with Kris actually out-performing FaCT for the first 1,300 concepts classified. This is probably due to the fact that, as most of these concepts are primitives, Kris can avoid the bottom search phase of classification (see Section 6.3 on page 102). FaCT does not perform this optimisation because it would not be valid if the TKB contained GCIs.

Figure 7.10 on page 125 uses **TKB2** to compare the performance of Kris and FaCT with its encoding, caching and backjumping optimisations disabled. While the lack of these optimisations does degrade FaCT's performance, it is still considerably faster than Kris (classification time was 444.68s) and does not display such serious worst case intractability in the 100th percentile times. Given the

	Time (s)	
Function	FaCT	KRIS
Load	7.60	135.90
Pre-process	1.08	–
Classify	146.37	887.17
TOTAL	155.05	1,023.07

Table 7.9: Total classification times (**TKB2**)

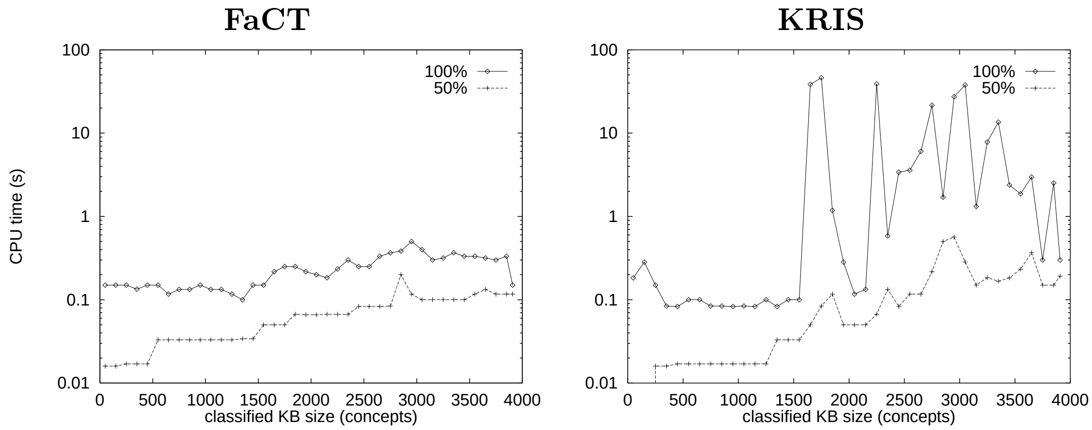

Figure 7.9: Classification time per concept -v- classified TKB size (**TKB2**)

similarity of the FaCT and KRIS systems, it seems reasonable to assume that this is the result of FaCT's semantic branching search optimisation, which could not be disabled. This optimisation is likely to be most effective when the search space is largest and could account for FaCT's improved performance with harder classification problems.

7.5 Solving Satisfiability Problems

The correspondence between $\mathcal{ALC}$ and the propositional modal logic $\mathbf{K_{(m)}}$ [Sch91] means that FaCT can also be used as a $\mathbf{K_{(m)}}$ theorem prover: $\mathbf{K_{(m)}}$ formulae correspond to $\mathcal{ALC}$ concept expressions as shown in Table 7.10 on the following page. Note that the modal operators $\Box$ and $\Diamond$ correspond to exists and value restrictions ($\exists R.C$ and $\forall R.C$ expressions), with different roles corresponding to distinct modalities or accessibility relations. Standard modal $\mathbf{K}$ ($\mathbf{K_{(1)}}$) has only one modality, so modal $\mathbf{K}$ formulae correspond to $\mathcal{ALC}$ concept expressions containing a single role.

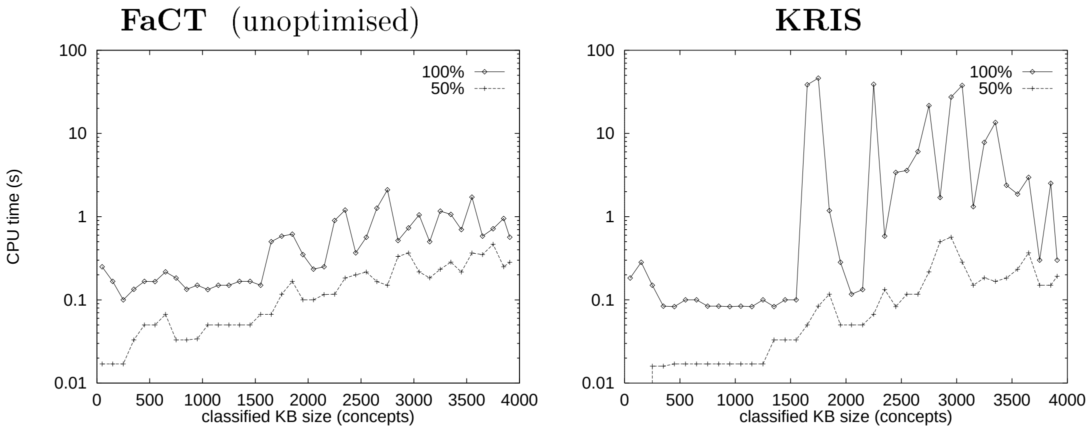

Figure 7.10: Classification without optimisations (**TKB2**)

$\mathbf{K_{(m)}}$	$\mathcal{ALC}$	$\mathbf{K_{(m)}}$	$\mathcal{ALC}$
True	$\top$	False	$\bot$
ϕ	C	$\neg\phi$	$\neg C$
$\phi \wedge \varphi$	$C \sqcap D$	$\phi \vee \varphi$	$C \sqcup D$
$\Box_i\phi$	$\forall R_i.C$	$\Diamond_i\phi$	$\exists R_i.C$

Table 7.10: The correspondence between modal $\mathbf{K_{(m)}}$ and $\mathcal{ALC}$

The performance of FaCT's satisfiability testing algorithm when used as a $\mathbf{K_{(m)}}$ theorem prover was evaluated by comparing it with the KSAT $\mathbf{K_{(m)}}$ theorem prover [GS96a]. The evaluation is of interest for several reasons:

- It provides a useful correctness test for the FaCT classifier.
- A range of carefully designed and controlled experiments can be conducted in a way which is not possible when using a realistic TKB.
- It demonstrates how FaCT's performance compares with that of a dedicated $\mathbf{K_{(m)}}$ theorem prover.

The comparison uses randomly generated $\mathbf{K_{(m)}}$ formulae transposed into $\mathcal{ALC}$ concept expressions, a test method devised by KSAT's developers and derived from a widely used procedure for testing propositional satisfiability (SAT) decision procedures [GS96b, Fre96]. The transposition of $\mathbf{K_{(m)}}$ formulae into $\mathcal{ALC}$ concept expressions is performed in accordance with the correspondences given in Table 7.10.

Two random generation procedures were used in the tests, the first devised by Giunchiglia and Sebastiani [GS96b] and the second by Hustadt and

Schmidt [HS97]. Both procedures use a number of parameters to control the size and complexity of the generated expressions; the Hustadt and Schmidt procedure is designed to eliminate trivially unsatisfiable formulae and so generate more consistently difficult problems.

The random generation procedures both produce conjunctive concept expressions of the form $(D_1 \sqcap \ldots \sqcap D_L)$ where each D_i is a K-disjunctive expression of the form $(C_1 \sqcup \ldots \sqcup C_K)$. Each disjunct C_j can be either a literal (an atomic primitive concept or its negation) or a modal atom (a value restriction concept or its negation). Modal atoms are of the form $\forall R_i.D$ where R_i is a primitive role corresponding to one of the modalities and D is another K-disjunctive expression.

Generation is controlled by six parameters:

N — the number of different primitive concepts (propositional variables);

M — the number of different roles (modalities);

K — the size of the K-disjunctive expressions;

D — the maximum depth of nested value restrictions (modal depth);

P — the probability of a disjunct being a primitive concept or a negated primitive concept (literal) rather than a value restriction or negated value restriction (modal atom);

L — the number of K-disjunctive expressions in the top-level conjunction.

If P is 1, the formulae generated are purely propositional and, it is claimed by Giunchiglia and Sebastiani, are of the kind widely used for testing SAT decision procedures [GS96b]. It has been shown that when K is fixed, the probability of such a randomly generated propositional formula being unsatisfiable is proportional to the ratio L/N, and that, independent of the algorithm being used, the hardest satisfiability problems are found in the *critically-constrained* region where the probability of satisfiability is $\approx$0.5 [Fre96]. Well designed algorithms have been shown to exhibit an easy-hard-easy behavior, called a *phase transition*: for given values of K and N, problems with a value of L which makes them either under-constrained ($\gg$50% satisfiable) or over-constrained ($\ll$50% satisfiable) are generally much easier to solve than critically constrained problems [Fre96, SML96]. The phase transition phenomenon has also been observed in a range of other NP-COMPLETE problems [HHW96].

7.5.1 Using the Giunchiglia and Sebastiani Generator

The experiments devised by Giunchiglia and Sebastiani are designed to test the performance of $\mathbf{K_{(m)}}$ satisfiability testing algorithms and to discover if a phase transition can be observed. Three sets of experiments are performed by varying one of the parameters N, M and D while keeping the others fixed. The values of the fixed parameters is chosen so that varying L to give values L/N in the range 1–40 produces problems ranging from ≈100% satisfiable to ≈0% satisfiable. In all the experiments, K is fixed at 3 (disjunctions are of size 3) and P is fixed at 0.5 (half the disjuncts are literals, and half are modal atoms). The values used for the three sets of experiments are: N = 3, 4, 5 and 8, with M fixed at 1 and D at 2; M = 1, 2, 5, 10 and 20, with N fixed at 4 and D at 2; and D = 2, 3, 4 and 5, with N fixed at 3 and M at 1. As there is some overlap in the parameter settings (e.g., $N = 4$, $M = 1$ and $D = 2$ occurs in both the first and second sets of experiments) this only gives rise to 11 different experiments. The parameter settings used in the 11 experiments are summarised in Table 7.11.

	N	M	K	D	P
PS0	5	1	3	2	0.5
PS1	3	1	3	5	0.5
PS2	3	1	3	4	0.5
PS3	3	1	3	3	0.5
PS4	3	1	3	2	0.5
PS5	4	1	3	2	0.5
PS6	4	4	3	2	0.5
PS7	4	5	3	2	0.5
PS8	4	10	3	2	0.5
PS9	4	20	3	2	0.5
PS10	8	1	3	2	0.5

Table 7.11: Parameter settings **PS0**–**PS11**

For each parameter setting, 100 problems were generated at each data point (integer values of L/N), and solved using both KSAT and FaCT. A total of 44,000 problems were therefore generated, and in order to keep the CPU-time required by the experiment within reasonable bounds, a 1,000s time limit was imposed on individual problems. The results of these experiments are shown in Figures 7.11–7.13 and the percentages of the generated problems which were satisfiable are shown in Figure 7.14 on page 129

Several points emerge from these results:

- FaCT outperformed KSAT in all cases and by the greatest factor when the problems were hardest: KSAT's median solution times were >1,000 times larger than those of FaCT for problems generated using **PS10** with L/N in the range 8–23.

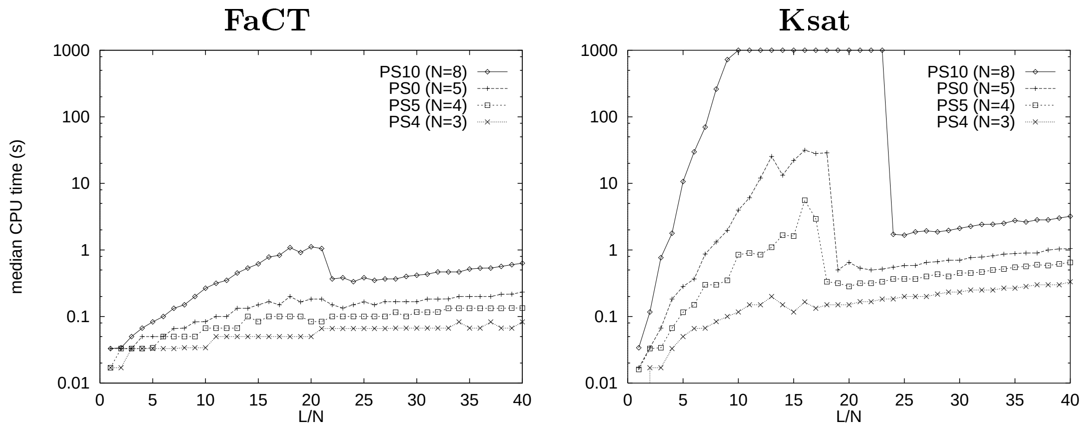

Figure 7.11: Median solution times — varying N

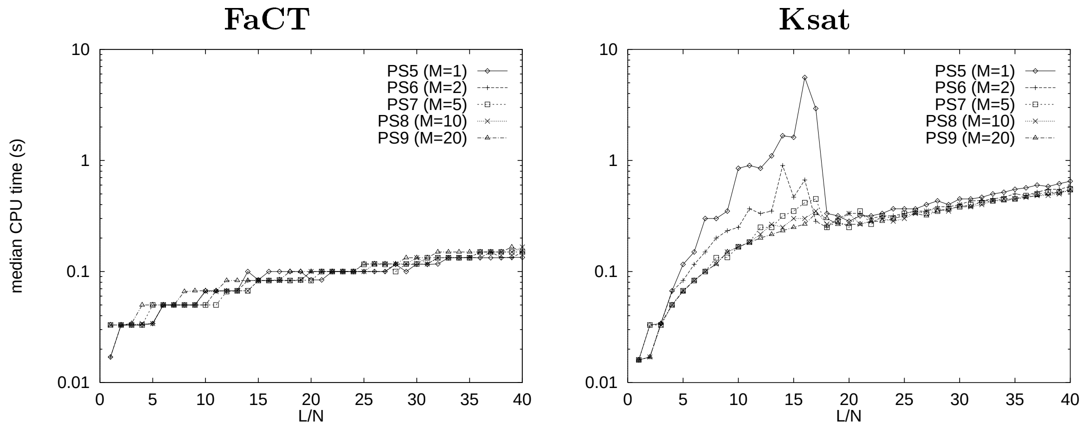

Figure 7.12: Median solution times — varying M

- Varying either M or D makes little difference to the difficulty of the problems generated, at least for FaCT.
- With the possible exception of **PS10**, the results obtained using FaCT do not show a pronounced phase transition.
- Increasing N causes much harder problems to be generated. **PS10** (N=8) generated by far the most challenging problems, and when L/N was in the range 10–23, KSAT failed to solve more than half the problems within the 1,000s time limit. For the same problems, FaCT's median solution time was ≈1s.

Two serious weaknesses in Giunchiglia and Sebastiani's experimental method have been pointed out [HS97]. Firstly, because the random generation procedure

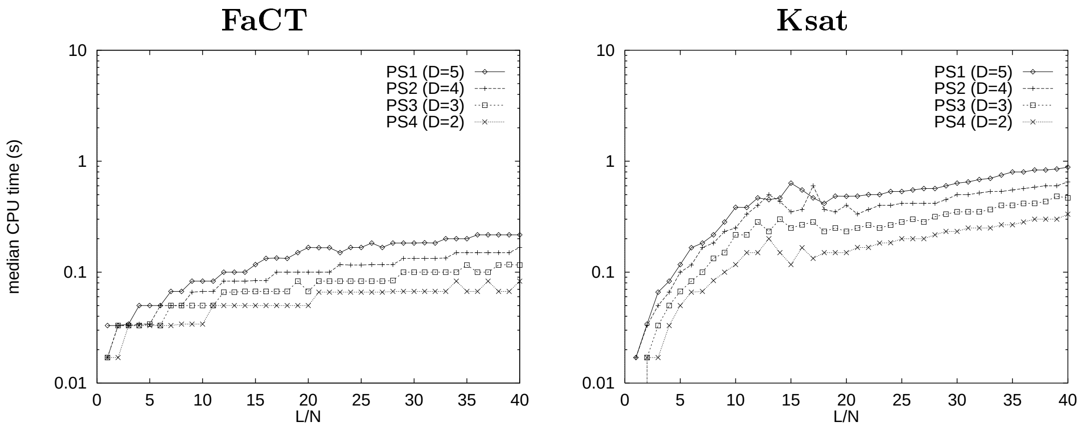

Figure 7.13: Median solution times — varying D

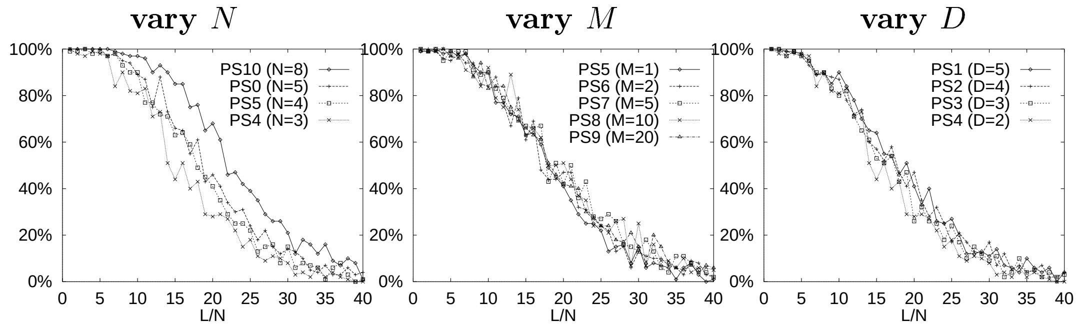

Figure 7.14: Percentages of satisfiable problems generated

chooses individual literals at random from the set of N propositional variables and their negations, the K-disjunctions which it generates can be tautological, containing both a proposition (primitive concept) and its negation, e.g., $(\phi_1 \vee \phi_2 \vee \neg\phi_1)$. Replacing these kinds of expression with True ($\top$), and performing further simplification using the identities 2.2 on page 30, can considerably reduce the effective size of many formulae and is sufficient to demonstrate that some formulae are satisfiable (they simplify to True/$\top$) or unsatisfiable (they simplify to False/$\bot$).

This method of randomising literals is not, as Giunchiglia and Sebastiani mistakenly claim, the same as the method widely employed in testing SAT decision procedures [GS96b, HS97, Fre96]. The standard method for generating random SAT problems, which is apparently due to Franco and Paull [FP83], chooses the *combination* of K propositional variables in a given K-disjunction from the ${}^{N}\mathbf{C}_K$ different possibilities, then negates each variable with a probability of 0.5 to give

K literals. Choosing combinations of propositional variables eliminates the duplication of variables within K-disjunctions and the possibility of generating trivial tautologies.

The second weakness in Giunchiglia and Sebastiani's experimental method is that, when using a value of P greater than 0, there is a probability that some of the top level K-disjunctions will contain only propositional variables. If the conjunction of these clauses is unsatisfiable, then the whole formula can be shown to be unsatisfiable without expanding any of the modal atoms. It can be shown that, when N is small, the solution of this kind of problem is trivial, even using the truth table method [HS97]. When $P = 0.5$, the value used by Giunchiglia and Sebastiani, and $L \gg N$, the probability of these kinds of formulae being generated is quite large, and as a result, most of the problems generated as L/N tends to 40 are trivially unsatisfiable.

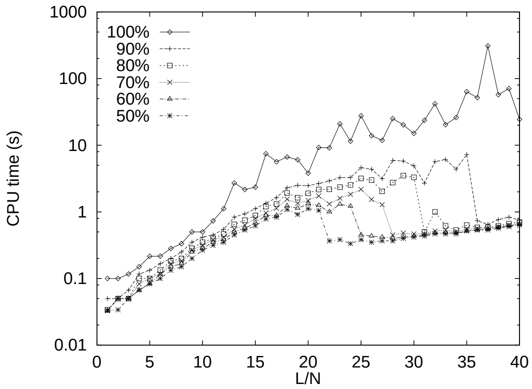

Figure 7.15: Percentile solution times — **PS10**/FaCT

The effect of these weaknesses on the performance analyses can be seen in Figure 7.15, which plots the percentile solution times obtained using FaCT with **PS10**. For problems in the range $L/N > 21$ the majority of problems are trivially unsatisfiable and this accounts for the rapid fall off in the 50th percentile (median) solution time. However a small number of non-trivial problems are generated and these become increasingly challenging as the problem size increases. With $L/N = 37$ for example, the median solution time across all 100 problems was <0.6s, but for the 7 satisfiable and thus non-trivial problems, the median solution time increased to 28.2s and the hardest problem took FaCT >308s to solve. KSAT failed to solve any of these 7 problems within the 1,000s time limit.

7.5.2 Using the Hustadt and Schmidt Generator

An alternative generator designed by Hustadt and Schmidt produces many more hard problems, particularly in the over-constrained region, by eliminating trivially unsatisfiable cases [HS97]. Hustadt and Schmidt eliminate the possibility of generating tautological K-disjunctions by using the combination method of choosing literals, as described in the previous section. The possibility of unsatisfiability being determined by purely propositional K-disjunctions at the top level is also eliminated by setting $P = 0$ so that K-disjunctions consist entirely of modal atoms, except those nested at depth D which consist entirely of propositional variables.

Hustadt and Schmidt used their improved random problem generator to evaluate four different modal logic decision procedures, one of which was KSAT. Their experiments were similar to those performed by Giunchiglia and Sebastiani: each experiment fixed the parameters N, M, K, D and P, so that varying L to give values of L/N in the range 1–30 generated problems ranging from ≈100% satisfiable to ≈0% satisfiable. As it was shown that the value of N is by far the most significant factor in determining the "hardness" of problems [HS97], only two experiments were performed, one with $N = 4$ (**PS12**) and the other with $N = 6$ (**PS13**). The two parameter settings used are given in full in Table 7.12.

	N	M	K	D	P
PS12	4	1	3	1	0
PS13	6	1	3	1	0

Table 7.12: Parameter settings **PS12** and **PS13**

The experiments using **PS12** and **PS13** were repeated using both KSAT and FaCT. The results of the experiment with **PS12** are shown in Figure 7.16 on the following page. The difference in performance between FaCT and KSAT is even more marked than for **PS10**. Not only did FaCT outperform KSAT quantitatively (for example with $L/N = 16$ FaCT's median solution time was ≈2s compared to KSAT's >1,000s) but also qualitatively: FaCT's performance shows evidence of a phase transition, with all the percentile solution times diminishing for probabilities of satisfiability <0.5. For $L/N = 30$, FaCT's median solution time had fallen to <0.3s whereas KSAT's was still >1,000s.

The best performing decision procedure in Hustadt and Schmidt's evaluation was a "translation approach" which used an optimised functional translation of the modal formula into FOPC [OS95], and then solved the FOPC formula using the FLOTTER/SPASS theorem prover [WGR96]. In contrast to FaCT, the median solution time using the translation approach with **PS12** reached a peak of ≈30s for $L/N = 20$ and showed little sign of decreasing. This was in spite of the fact that Hustadt and Schmidt used a much faster machine for their evaluation (a Sun Ultra 1/170E with 196MB of main memory) and that FLOTTER/SPASS used compiled C code, which would normally be expected to outperform Lisp code.

Increasing N, the number of propositional variables, from 4 (**PS12**) to 6 (**PS13**)

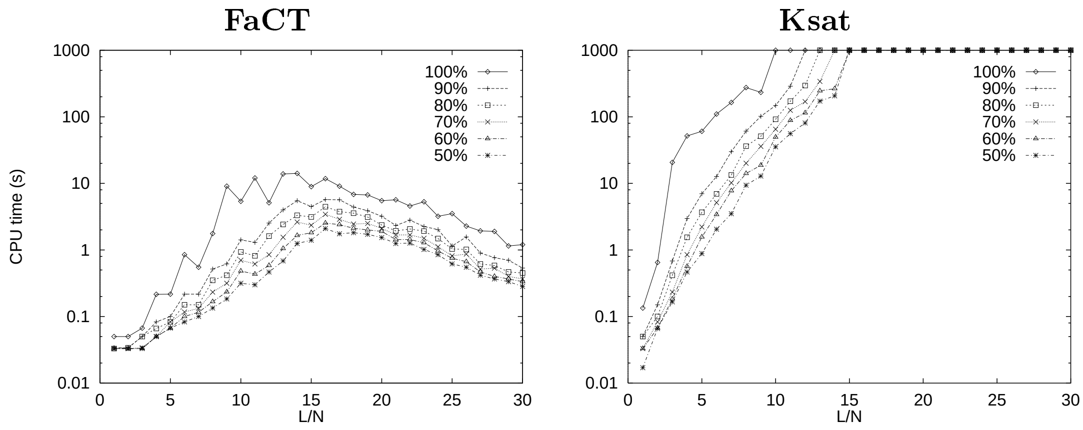

Figure 7.16: Percentile solution times — **PS12**

dramatically increased the difficulty of the problems generated, as can be seen from Figure 7.17. For values of $L/N > 7$ when using KSAT, and $L/N > 11$ when using FaCT, more than half the problems were not solved within the 1,000s time limit. From Hustadt and Schmidt's results it would appear that the translation approach fared a little better with its median solution exceeding 1,000s for values of $L/N > 13$.

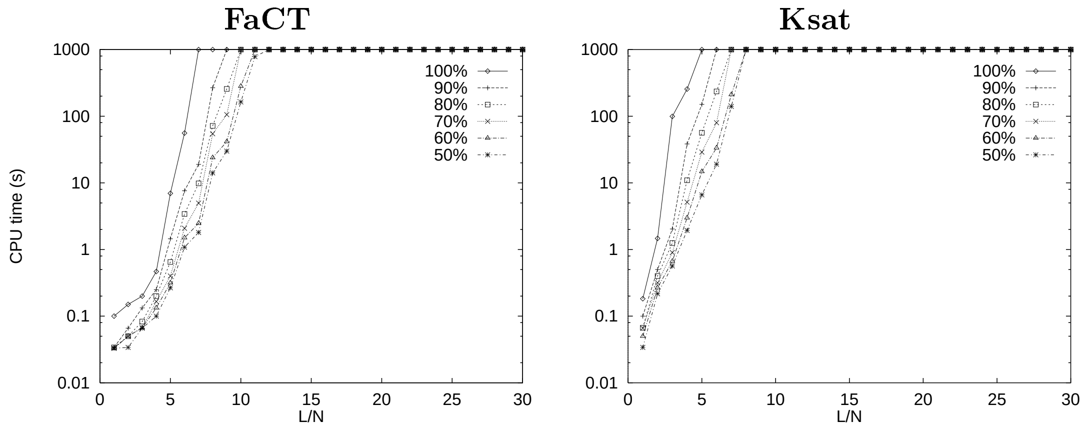

Figure 7.17: Percentile solution times — **PS13**

It is not difficult to understand why the problems generated with **PS13** are so much more challenging than those generated with **PS12**. Both parameter settings generate modal formulae of the form:

$$((\phi_{11} \vee \phi_{12} \vee \phi_{13}) \wedge \ldots \wedge (\phi_{L1} \vee \phi_{L2} \vee \phi_{L3}))$$

where each ϕ_{ij} is a (possibly negated) modal atom of the form:

$$\Box(\varphi_a \vee \varphi_b \vee \varphi_c)$$

When $N = 4$, the purely propositional disjunctive clauses $(\varphi_a \vee \varphi_b \vee \varphi_c)$ at depth 1 contain a combination of 3 propositional variables chosen from 4 candidates, giving only $\frac{4!}{(4-3)!3!} = 4$ different possibilities. Each of the 3 variables is then negated with a probability of 0.5, which gives a total of $4 \times 2^3 = 32$ possibilities. As there is only 1 modality, this means that there are 64 possible ϕ_{ij} corresponding to the 32 possible $\Box(\varphi_a \vee \varphi_b \vee \varphi_c)$ and their negations. The size of the worst case search space at depth 0 is therefore 2^{32} regardless of the size of the problem, and given the constrainedness of the propositional sub-problems at modal depth 1, which contain at most 4 variables, this search space can be effectively pruned by the backjumping optimisation. Figure 7.18 plots the percentile search space sizes (the number of backtracks before a solution is found) measured for FaCT with **PS12**.

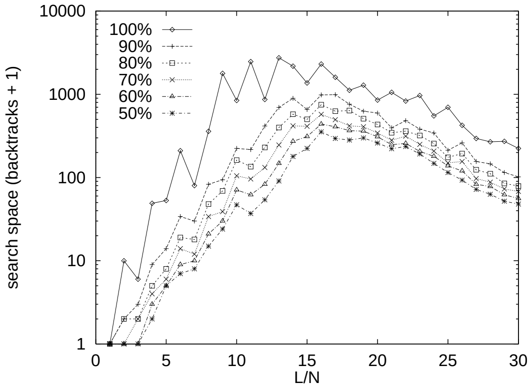

Figure 7.18: Search space — **PS12**/FaCT

When $N = 6$, the total number of possible propositional clauses at depth 1 increases to $\frac{6!}{(6-3)!3!} \times 2^3 = 160$ and the size of the worst case search space at depth 0 is therefore 2^{160}. As the propositional sub-problems at modal depth 1 are less constrained the pruning of this search space will also be less effective and larger problems soon become highly intractable.

7.6 Testing for Correctness

Although the underlying $\mathcal{ALCH}_{R^+}$ algorithm has been shown to be correct, errors might have been introduced by the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ extension, the optimisation techniques or the implementation process. Empirical testing cannot completely eliminate this possibility (some errors might be very obscure), but extensive empirical testing does suggest a high probability that the implementation is correct.

7.6.1 Hard Problems

During development, FaCT's correctness was regularly checked using a suite of "hard problems", some adapted from [HKNP94] and some designed specifically to test FaCT's features. These tests include:

- Detecting unsatisfiability caused by disjoint concepts. e.g., if $\{C \sqsubseteq \neg D\} \subseteq \mathcal{T}$, is $C \sqcap D$ satisfiable w.r.t. $\mathcal{T}$?
- Detecting unsatisfiability caused by disjoint concepts and attributes. e.g., if $\{C \sqsubseteq \neg D, A \in \mathbf{F}\} \subseteq \mathcal{T}$, is $\exists A.C \sqcap \exists A.D$ satisfiable w.r.t. $\mathcal{T}$?
- Detecting subsumption relations caused by GCIs (see Example 6.3 on page 103).
- Detecting subsumption relations caused by the interaction of transitive roles and the role hierarchy (see Example 4.1 on page 67).
- Detecting subsumption relations caused by complex interactions between attributes and the role hierarchy (see Example 4.4 on page 75).

7.6.2 Classification

The correctness of terminological classification in FaCT was checked by comparing the **TKB0** concept hierarchies computed by FaCT with and without various optimisations and by comparing the hierarchies computed by FaCT and Kris for **TKB2** (see Section 7.4 on page 121).

Unfortunately, the usefulness of the latter comparison is limited by the fact that **TKB2** did not contain a role hierarchy, transitive roles or GCIs. It did not, therefore, test the correctness of the FaCT classifier with respect to these constructs.

7.6.3 Satisfiability Testing

The correctness of satisfiability testing in FaCT was checked by comparing it with the Ksat algorithm. The results computed by FaCT for 50,000 randomly generated satisfiability problems were compared with those computed by Ksat. The results were identical in all those cases where both Grail and Ksat were able to compute answers before exceeding the CPU-time limit of 1,000s.

The usefulness of the test is again limited by the fact that KSAT only supports a subset of the language implemented in FaCT, namely $\mathcal{ALC}$, so that those parts of the algorithm which deal with attributes, transitivity, the role hierarchy and non-absorbed GCIs were not tested. However the extensions to the basic $\mathcal{ALC}$ algorithm required to support these features are relatively minor: the main complexity of the algorithm is in the optimisation techniques and this area of the code is thoroughly exercised by these tests.

7.7 Summary

The results of the experiments presented in this chapter demonstrate that, with the exception of MOMS heuristic, the optimisation techniques dramatically improve the performance of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm, and that they are most effective with the hardest problems. The performance improvement for these problems is at least three orders of magnitude: it is impossible to be more precise because, without optimisation, many problems are so intractable as to be effectively non-terminating.

7.7.1 GCIs and Absorption

When classifying a TKB containing GCIs, absorption is by far the most important optimisation technique: without absorption, FaCT failed to classify a single concept from **TKB0**. Unlike the other optimisations, absorption actually simplifies the problem by eliminating GCIs from the terminology. Without this simplification, the potential size of the search space which may have to be explored when classifying **TKB0** makes it unlikely that any other optimisation techniques could succeed in making the problem tractable.

The success of this optimisation depends on GCIs being of an appropriate form, specifically on their antecedents being either primitive concepts or conjunctive concept expressions which contain, or can be unfolded so that they contain, a primitive conjunct. It is difficult to assess whether or not this is a serious restriction without analysing other realistic TKBs which contain GCIs, and as GRAIL and FaCT are the only known DL implementations which support GCIs, it is unlikely that such TKBs exist. It seems probable, however, that most GCIs in a realistic TKB would be amenable to absorption:

1. The majority of concepts in realistic knowledge bases are primitive, and the definitions of most non-primitive concepts are conjunctive concept expressions [HKNP94]; in fact many implemented DLs find it acceptable not to support disjunctive concept expressions (e.g., CLASSIC).

2. The restricted form of concept expression supported by GRAIL, which results in all GRAIL GCIs being amenable to absorption, is the result of a detailed design study which concluded that this form of expression was adequate for describing medical terminology [Now93]. GRAIL has also proved useful in a number of other applications domains, and where problems have been encountered it has been due to the lack of number restrictions rather than the lack of disjunction [BG96, GHB96].

3. It is difficult to imagine situations where non-absorbable GCIs would be necessary: if a GCI is not amenable to absorption, it means that it represents a general assertion which might be applicable to every concept in the TKB. Even when a GCI is not of the appropriate form, it would normally be possible to apply a reasonable restriction to the assertion which would allow it to be absorbed. e.g., given the GCI $\exists \mathit{degree}.\mathsf{MSc} \sqsubseteq \exists \mathit{degree}.\mathsf{BSc}$ [BDS93] it seems reasonable to assume than having degrees is only applicable to persons, and to restrict the GCI to $\mathsf{person} \sqcap \exists \mathit{degree}.\mathsf{MSc} \sqsubseteq \exists \mathit{degree}.\mathsf{BSc}$.

7.7.2 Backjumping and MOMS Heuristic

When solving individual subsumption/satisfiability problems, backjumping is the most important optimisation. Even when absorption succeeds in eliminating all the GCIs from a terminology, the resulting classification problem can still be highly intractable, as large numbers of disjunctive expressions are added to concept introduction axioms. The backjumping optimisation is highly effective in pruning the search space which can result from expanding these disjunctions: without backjumping, single satisfiability problems are encountered which are so intractable as to be effectively non-terminating.

The use of MOMS heuristic to guide the exploration of the search space interacts with backjumping, reduces its effectiveness, and degrades FaCT's performance. As a result, the use of MOMS heuristic is disabled by default in the FaCT system.

7.7.3 Other Optimisations

The normalisation and encoding optimisation and the caching optimisation also make significant contributions to FaCT's performance: disabling either optimisation degrades performance by a factor of over 100 when classifying **TKB0**.

The caching optimisation is particularly important because it can cheaply eliminate obvious non-subsumers and concentrate more expensive subsumption testing in areas of the TKB where subsumers are likely to be found. As a result, the time taken to classify a new concept is relatively independent of the total size of the classified TKB and is instead dependent on the size and complexity of the sub-hierarchy to which the new concept belongs.

Chapter 8

Discussion

This thesis concludes with a review of the work presented and an assessment of the extent to which the objectives set out in Chapter 1 have been met. The significance of the major results is summarised, outstanding issues are discussed and directions for future work are suggested.

8.1 Thesis Overview

The objective of this thesis was to investigate the practicability of providing sound, complete and empirically tractable subsumption reasoning for a DL with an expressive concept description language and to demonstrate that, in spite of the worst-case intractability of reasoning in such languages, a suitably optimised tableaux algorithm could provide acceptable performance with a realistic terminological knowledge base (TKB).

This claim has been strongly supported by the successful development of the FaCT system and by results from the empirical testing of FaCT using a large TKB from the GALEN project. It has also been demonstrated that, when used with a simplified TKB, FaCT's performance compares favourably with that of a less expressive DL (KRIS) and that, when used to solve randomly generated $\mathbf{K_{(m)}}$ satisfiability problems, FaCT's performance compares favourably with that of dedicated $\mathbf{K_{(m)}}$ theorem provers (KSAT and FLOTTER/SPASS).

8.2 Significance of Major Results

The major results of this thesis are:

- the design of the $\mathcal{ALCH}_{R^+}$ and $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ satisfiability testing algorithms and the soundness and completeness proof for $\mathcal{ALCH}_{R^+}$;

- the development of a range of optimisation techniques which dramatically improve the performance of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm;
- the evaluation of the optimised algorithm through extensive empirical testing, including a detailed comparison with the Kris DL classifier and the KSAT $\mathbf{K_{(m)}}$ theorem prover.

The FaCT system, which was implemented as a test-bed for the optimised $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm, is also of significant interest in its own right, both as a DL knowledge representation tool and as a propositional modal logic theorem prover.

8.2.1 Satisfiability Testing Algorithms

A tableaux satisfiability testing algorithm has been designed for $\mathcal{ALCH}_{R^+}$, a DL which supports transitive roles, a role hierarchy and GCIs, and a proof of the soundness and completeness of this algorithm has been presented. An extension to the $\mathcal{ALCH}_{R^+}$ algorithm has also been designed, which allows it to be used for satisfiability testing in $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, a DL which augments $\mathcal{ALCH}_{R^+}$ with support for functional roles.

These theoretical results generalise and extend those already available for the $\mathcal{ALC}_{R^+}$ and $\mathcal{ALC}_{\oplus}$ DLs [Sat96]. Instead of the restricted form of role inclusions provided by $\mathcal{ALC}_{\oplus}$, $\mathcal{ALCH}_{R^+}$ supports general role inclusion axioms, allowing a complete role hierarchy to be defined. The soundness and completeness proof for the $\mathcal{ALCH}_{R^+}$ satisfiability testing algorithm also provides implicit proof of the soundness and completeness of the $\mathcal{ALC}_{\oplus}$ algorithm, which can be seen as a special case of the $\mathcal{ALCH}_{R^+}$ algorithm. The soundness and completeness of the $\mathcal{ALC}_{\oplus}$ algorithm had not previously been demonstrated.

8.2.2 Optimisation Techniques

A range of novel and adapted optimisation techniques have been developed which dramatically improve the performance of the $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm. Absorption and backjumping are the most effective of these techniques, but normalisation and encoding, semantic branching and caching are also valuable: quantitative results from the empirical testing of FaCT are summarised in Section 8.2.3.

Many of the optimisation techniques developed for $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ could be used with other tableaux satisfiability testing algorithms and should become standard in future tableaux based DL implementations. Some of these techniques have been shown to be applicable to propositional modal logic theorem proving and may also be relevant to automated deduction in other logics, for example temporal logics, where tableaux methods are widely used [Gou84].

Absorption Absorption could be used with any DL which supports full negation in concept descriptions and is independent of the subsumption testing algorithm employed. It could even be performed as a pre-processing step: this was demonstrated in the derivation of **TKB1**, one of the test TKBs used to compare FaCT with KRIS (see Section 7.1.2 on page 109). The only caveat is that absorption of GCIs can introduce cycles into primitive concept introduction axioms, resulting in a TKB which can not be classified using an algorithm which is restricted to unfoldable terminologies (such cycles had to be eliminated from **TKB1** before attempting to classify it using KRIS).

Normalisation and Encoding Normalisation and encoding performed as a pre-processing step could be applied to any DL, including existing implementations. Used in this way the technique is also independent of the subsumption testing algorithm, although its effect on performance may not be. Integrating normalisation and encoding with the classifier is preferable, however, in order to avoid the classification of system generated concepts and to facilitate the use of more efficient data structures.

Semantic Branching Semantic branching search and boolean constraint propagation could be used with any tableaux algorithm, although it calls for a significant change in the way algorithms are implemented.

Backjumping Backjumping could easily be used with other tableaux algorithms, although some minor adaptions may be necessary to deal with the dependencies caused by different kinds of concept expression (or formula in logics other than DLs).

Caching Caching could also be used with other tableaux algorithms, but the adaptions required to deal with different kinds of concept expression (or formula) may be more complex: it may be necessary to change both the information which is cached and the algorithm which decides when cached models can be merged.

8.2.3 Empirical Evaluation

The FaCT system was implemented as a test-bed for the optimised $\mathcal{ALCH}f_{R^+}$ algorithm and has been used to perform extensive empirical testing, including a detailed comparison with the KRIS DL classifier and the KSAT $\mathbf{K_{(m)}}$ theorem prover. These experiments demonstrate that:

1. Although the satisfiability problem for $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ is EXPTIME-complete, FaCT can provide tractable performance when used to classify a large realistic TKB: a 2,740 concept TKB developed as part of the GALEN project was classified by FaCT in approximately 380s (see Table 7.4 on page 113).

2. Absorption and backjumping are the most effective of the investigated optimisation techniques: with either of these optimisations disabled it was impossible to classify the GALEN TKB in a reasonable time (see Section 7.3.1 on page 115). Encoding and caching are also of significant value: the time taken to classify the GALEN TKB increased by a factor of approximately 158 when encoding was disabled (see Table 7.5 on page 116) and by a factor of approximately 177 when caching was disabled (see Table 7.6 on page 118). Guided search using MOMS heuristic proved to be ineffective: its use actually degraded performance by a factor of approximately 4 (See Table 7.7 on page 120).

3. When used to classify a TKB containing large numbers of disjunctive concept expressions, FaCT was at least 1,900 times faster than KRIS (see Table 7.8 on page 122); the actual performance difference is probably much greater than this, as KRIS had only classified approximately 10% of the test TKB when the experiment was terminated. Even when used with a simplified TKB containing no disjunctive concept expressions, FaCT was still more than 6 times faster than KRIS overall (see Table 7.9 on page 124) and more than 100 times faster for the concepts which took KRIS longest to classify (see Figure 7.9 on page 124). As KRIS's performance has been shown to compare favourably with that of other DL classifiers, including BACK, CLASSIC and LOOM [BHNP92], it seems reasonable to assume that FaCT would also outperform these systems by a similar or greater margin.

4. When used to solve the hardest of the randomly generated $\mathbf{K_{(m)}}$ satisfiability problems, FaCT was at least 500 times faster than KSAT (see Figure 7.16 on page 132) and in some cases more than 1,000 times faster than KSAT (see Figure 7.11 on page 128). Although it has not been possible to repeat all the experiments performed in a recent evaluation of $\mathbf{K_{(m)}}$ theorem provers [HS97], it would appear that FaCT compares favourably with the other systems evaluated including FLOTTER/SPASS, the best performing system.

8.2.4 The FaCT System

Although primarily intended as a test-bed for the optimised $\mathcal{ALCH}\boldsymbol{f}_{R^+}$ algorithm, the FaCT system is of significant interest as a practical DL knowledge representation tool: FaCT is the only available system which supports transitive roles and GCIs, it provides sound and complete reasoning, it is able to handle large TKBs and it has a simple but comprehensive functional interface which is compatible with that of the Kris system. A detailed user manual is available for the FaCT system (see Appendix A).

FaCT is also of interest as a propositional modal logic theorem prover. FaCT's performance has been shown to compare favourably with that of other $\mathbf{K_{(m)}}$ theorem provers, and as it can reason with transitive roles, FaCT can also be used for theorem proving in propositional modal $\mathbf{K4_{(m)}}$ and $\mathbf{S4_{(m)}}$. $\mathbf{K4_{(m)}}$ is equivalent to $\mathbf{K_{(m)}}$ with a transitive accessibility relation and is dealt with by transposing formulae into $\mathcal{ALC}$ concept expressions as for $\mathbf{K_{(m)}}$ (see Table 7.10 on page 125), but using transitive roles. $\mathbf{S4_{(m)}}$ is equivalent to $\mathbf{K_{(m)}}$ with a transitive reflexive accessibility relation and is also dealt with by transposing formulae into $\mathcal{ALC}$ concept expressions as for $\mathbf{K_{(m)}}$, but using transitive reflexive roles; identities 2.3 (see Section 2.2.2 on page 30) can then be used to transpose expressions using transitive reflexive roles into equivalent expressions using transitive roles.

FaCT has been made available via the world wide web[1] and copies of FaCT have already been obtained by researchers in Colombia, France, Germany, Holland, Israel, Italy, Switzerland and the U.S.A.

8.3 Outstanding Issues

This thesis does not, of course, represent a complete answer to the problems associated with the use of expressive DLs. There remain a number of outstanding issues, several of which are discussed below.

8.3.1 The Spectre of Intractability

The complexity of the subsumption reasoning problem in $\mathcal{ALCH}\boldsymbol{f}_{R^+}$, and other expressive DLs, is immutable: regardless of the optimisation techniques employed, the spectre of intractability will always be present. In a complete DLKRS, there would need to be some mechanism for specifying the maximum acceptable time to be spent on any single subsumption or classification problem and for dealing with exceptions should they arise.

What constitutes an acceptable time for the system to spend on a single problem would depend very much on the application: clearly it would be relatively short if the TKB were being queried and updated interactively, as is anticipated in some applications of the Galen ontology [RSNR94]. However, regardless of the time limit which is imposed, it will always be possible to generate subsumption problems which take too long to solve and which can only be dealt with by accepting a degree of incompleteness.

[1] The FaCT home page is: `http://www.cs.man.ac.uk/~horrocks/FaCT`

Even in applications where this circumstance arises, a theoretically complete but intractable algorithm is at least able to inform the user when its reasoning is incomplete: in response to the question "does C subsume D?" it can answer "yes", "no" or "don't know". In response to the same question, the incomplete structural algorithms used in most implemented systems can only answer "yes" or "don't know", as their failure to find a subsumption relationship is no guarantee that one does not exist.

8.3.2 GALEN and GRAIL

The use of DLs in ontological engineering, in particular the construction of a large medical terminology ontology as part of the GALEN project, was one of the primary motivations for investigating the tractability of sound and complete reasoning for an expressive DL. However, before FaCT, or some future development of FaCT, could be used in place of GRAIL, there are a number of outstanding issues which would need to be resolved:

- The semantics of the GRAIL `specialisedBy` statement need to be clarified (see Section 7.1.1 on page 106) and a method devised for representing them in the FaCT system. This may require a logic which is more expressive than $\mathcal{ALCH}f_{R^+}$.
- GRAIL has an integrated sanctioning mechanism: as well as classifying concepts and checking that they are logically coherent, it also uses sanctioning information to check that descriptions are "sensible" [RBG+97]. This is considered to be an important part of the GRAIL system and before FaCT could be used instead of GRAIL it would need to be equipped with an equivalent mechanism.
- Although FaCT's more expressive concept description language addresses some of GRAIL's shortcomings, for example its lack of negation, requirements for additional constructs have already been identified in the GALEN project: number restrictions, attribute value maps and inverse roles would be particularly useful [RBG+97].

8.3.3 Tools and Environments

The way in which knowledge is represented can have a major impact on the tractability of reasoning. The effectiveness of many of the optimisations depends on the structure of the TKB: absorption is most effective when the antecedents of GCIs are made as specific as possible and the ability of caching to isolate expensive subsumption testing in sub-hierarchies depends on there being a well designed primitive hierarchy at the "top" of the TKB.

Ideally, users of knowledge representation systems in general, and of expressive description logics in particular, should be aware of these issues and understand the impact their design choices may have on the likely tractability of the resulting TKB. However, as users are typically domain experts and not knowledge engineers, it is essential for a complete DLKRS to include tools and environments, such as those provided with the GRAIL system [GBS+94, ST97, Sol97], which can insulate unsophisticated users from the complexity of the underlying representation (e.g., $\mathcal{ALCH}\boldsymbol{f}_{R^+}$) and help them to represent knowledge more efficiently. The system should detect constructs which are likely to cause intractability, and where possible, automatically modify them to give an equivalent but more tractable representation: the absorption optimisation can be seen as an example of such a modification.

8.4 Future Work

Development of the FaCT system has suggested several promising avenues for further research:

- Extending the optimisation techniques to deal with a variety of different concept description languages. Adding number restrictions and attribute value maps to FaCT would be particularly useful.

- Investigating new optimisation techniques and improving the the existing techniques. Dependency directed backtracking is one of the most effective of the existing techniques and it may be worthwhile investigating more sophisticated backtracking procedures, such as dynamic backtracking [Gin93], which try to preserve useful work while returning to the source of the discovered contradiction. Heuristic guided search, on the other hand, proved ineffective or even counter productive and more work is required in order to ascertain if this problem can be remedied by the use of improved heuristics.

- Investigating the potential of parallelism. The time taken to solve hard satisfiability problems can depend heavily on early branching choices made by the algorithm. One possible approach to this problem is to explore multiple branching choices simultaneously by using coarse grained parallelism to run the algorithm on multiple processors or machines. Using parallelism to improve the performance GRAIL's structural subsumption algorithm has already been investigated with some success in the PAEPR project [RN95].

- FaCT has only been tested with the GALEN TKB. More extensive testing with realistic TKBs from other application domains is required in order to demonstrate the general effectiveness of the optimisation techniques and

FaCT's ability to provide tractable reasoning in a range of application domains. Due to its unique expressiveness (among implemented systems) this may have to wait for the development of applications which use FaCT.

- Building a complete DLKRS based on FaCT. The existing FaCT system has a number of shortcomings: it has been implemented as an experimental prototype; it only supports a single concept description language, which is designed to accommodate the GALEN medical terminology application (but see Section 8.3.2 on page 142); and it does not provide any support for reasoning about individuals (an ABox). A project has already been funded[2] which proposes to address these issues by developing a more robustly engineered modular system, which will be able to support a number of different concept description languages. The design of this system includes a database interface module, which will allow the classifier to be coupled to an Object Oriented Database instead of a traditional ABox, and an environment interface module, which will allow the classifier to be used with a range of knowledge engineering tools and user environments [HGG+97].

Demonstrating that an expressive DL with a sound and complete subsumption testing algorithm can provide acceptable performance in a realistic application is an encouraging result for the DL community: it suggests that the substantial body of theoretical work which has emerged in recent years may have more direct practical applicability than had previously been realised. It is hoped that the design of the FaCT system will provide a firm foundation for ongoing research, leading to the development of DL knowledge representation systems which are useful in a wide range of application domains.

[2]EPSRC Grant reference GR/L54516.

Appendix A

FaCT Reference Manual

A.1 Introduction

FaCT is a prototype description logic knowledge representation system which uses an optimised tableaux subsumption algorithm to provide complete inference for a relatively expressive concept description language. In particular, FaCT supports transitively closed roles, a role/attribute hierarchy and general concept inclusion axioms (implications of the form $C \Rightarrow D$ where C and D are arbitrary concept descriptions).

FaCT is primarily intended as a tool for conceptual schema design and ontological engineering and thus provides only concept based reasoning services: there is a TBox but no ABox. The interface to the TBox is designed to be broadly compatible with that of the KRIS system [BH91c].

The correspondence between $\mathcal{ALC}$ and the propositional modal logic $\mathbf{K_{(m)}}$ means that FaCT can also be used as a highly efficient decision procedure for $\mathbf{K_{(m)}}$ [GS96b, HS97]. FaCT's support for transitively closed roles means that it can also be used as a decision procedure for modal $\mathbf{K4_{(m)}}$ and $\mathbf{S4_{(m)}}$.

A.1.1 Obtaining FaCT

FaCT can be obtained from the following world wide web site:

http://www.cs.man.ac.uk/∼horrocks/FaCT

After downloading the file `FaCT.tar.gz` use the command:

```
gunzip FaCT.tar.gz ; tar xvf FaCT.tar
```

to create a directory `FaCT` containing the distribution files and directories.

To install FaCT follow the instructions in the file `FaCT/README`.

FaCT has been written in Common LISP and has been tested with Allegro Common Lisp, Harlequin LispWorks and Gnu Common Lisp.

A.2 Concept Descriptions

FaCT uses the same list based concept description syntax as the KRIS system. If CN is the name of a defined concept, *R* is the name of a defined role, *A* is the name of a defined attribute and $C, C_1, \ldots, C_n$ are concept descriptions, then the following are also valid concept descriptions:

> *TOP*
> *BOTTOM*
> CN
> (and $C_1 \ldots C_n$)
> (or $C_1 \ldots C_n$)
> (not C)
> (some *R* C)
> (all *R* C)
> (some *A* C)
> (all *A* C)

The correspondence between this form and the standard infix notation is shown in Table A.1.

FaCT syntax	Standard notation
`*TOP*`	$\top$
`*BOTTOM*`	$\bot$
(`and` $C_1 \ldots C_n$)	$C_1 \sqcap \ldots \sqcap C_n$
(`or` $C_1 \ldots C_n$)	$C_1 \sqcup \ldots \sqcup C_n$
(`not` C)	$\neg C$
(`some` R C)	$\exists R.C$
(`all` R C)	$\forall R.C$
(`some` A C)	$\exists A.C$
(`all` A C)	$\forall A.C$

Table A.1: FaCT concept expressions

A.3 Function and Macro Interface

This section describes the built-in concepts, macros and functions which provide the user interface to the TBox.

A.3.1 Built-in Concepts

`*TOP*` *concept*

Description: The name of the top concept ($\top$).

Remarks: Every concept in the TBox is subsumed by `*TOP*`.

`*BOTTOM*` *concept*

Description: The name of the bottom concept ($\bot$).

Remarks: Every concept in the TBox subsumes `*BOTTOM*`. Note that unsatisfiable concepts become synonyms for `*BOTTOM*`.

A.3.2 Knowledge Base Management

---

init-tkb *function*

Description: Initialises the TBox.

Syntax: (**init-tkb**)

Remarks: All user defined concepts, roles, attributes and implications are deleted from the TBox leaving only `*TOP*` and `*BOTTOM*`.

Examples: `(init-tkb)`

---

load-tkb *function*

Description: Loads a TBox from a file.

Syntax: (**load-tkb** *name* `&key` (*verbose* `T`) (*overwrite* `nil`))

Arguments: *name* - Name of the TBox file (a character string).

verbose	- Keyword which, if non-`nil`, causes the classifier to print symbols indicating the progress of the load operation: `P` for each primitive concept, `C` for each non-primitive concept, `R` for each role, `R+` for each transitive role, `A` for each attribute and `I` for each implication. If omitted, *verbose* defaults to `T`.
overwrite	- Keyword which, if non-`nil`, causes the classifier to clear the existing TBox from memory (by performing an **init-tkb**) before loading the new TBox. If omitted, *overwrite* defaults to `nil`.

Return: `T` if the TBox is successfully loaded; `nil` otherwise.

Examples: `(load-tkb "demo.kb" :verbose T)`

A.3.3 TBox Definitions

---

defprimconcept *macro*

Description: Defines a primitive concept.

Syntax: (**defprimconcept** *name* `&optional` (*description* `*TOP*`))

Arguments:

name	- Name of the new concept.
description	- Optional description of the new concept. If omitted it defaults to `*TOP*`.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: The new concept is not classified until a call is made to **classify-tkb**. It is an error if a concept *name* has already been defined.

Examples:

```
(defprimconcept ANIMAL)
(defprimconcept MALE)
(defprimconcept FEMALE)
(defprimconcept BIPED)
(defprimconcept HUMAN (and ANIMAL BIPED))
```

---

defprimconcept-f *function*

Description: Functional equivalent of **defprimconcept**.

Remarks: Note that all arguments have to be quoted.

Examples:

```
(defprimconcept-f 'ANIMAL)
(defprimconcept-f 'MALE)
(defprimconcept-f 'FEMALE)
(defprimconcept-f 'BIPED)
(defprimconcept-f 'HUMAN '(and ANIMAL BIPED))
```

defconcept *macro*

Description: Defines a non-primitive concept.

Syntax: (**defconcept** *name description*)

Arguments: *name* - Name of the new concept.

description - Description of the new concept.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: The new concept is not classified until a call is made to **classify-tkb**. It is an error if a concept *name* has already been defined. Note that in contrast to **defprimconcept** it is also an error if *description* is omitted.

Examples:

```
(defconcept MAN (and MALE HUMAN))
(defconcept WOMAN (and FEMALE HUMAN))
```

defconcept-f *function*

Description: Functional equivalent of **defconcept**.

Remarks: Note that all arguments have to be quoted.

Examples:

```
(defconcept-f 'MAN '(and MALE HUMAN))
(defconcept-f 'WOMAN '(and FEMALE HUMAN))
```

defprimrole *macro*

Description: Defines a primitive role.

Syntax: (**defprimrole** *name* `&key` (*supers* `nil`) (*transitive* `nil`))

Arguments: *name* - Name of the new primitive role.

supers - Keyword list of super-roles. If omitted it defaults to `nil`.

transitive - Keyword which, if non-`nil`, makes the role transitive. If omitted it defaults to `nil`.

Return: A role structure `r[`*name*`]` is returned.

Remarks: It is an error if a role or attribute *name* has already been defined.

Examples:

```
(defprimrole Relation :transitive T)
(defprimrole Close-relation :supers (Relation))
(defprimrole Ancestor :supers (Relation)
    :transitive T)
(defprimrole Parent :supers (Close-relation Ancestor))
```

defprimrole-f *function*

Description: Functional equivalent of **defprimrole**.

Remarks: Note that all arguments have to be quoted.

Examples:

```
(defprimrole-f 'Relation :transitive T)
(defprimrole-f 'Close-relation :supers '(Relation))
(defprimrole-f 'Ancestor :supers '(Relation)
    :transitive T)
(defprimrole-f 'Parent :supers '(Close-relation
    Ancestor))
```

defprimattribute *macro*

Description: Defines a primitive attribute (functional role).

Syntax: (**defprimattribute** *name* `&key` (*supers* `nil`))

Arguments: *name* - Name of the new primitive attribute.

supers - Keyword list of super-attributes or super-roles. If omitted it defaults to `nil`.

Return: A role structure `r[`*name*`]` is returned.

Remarks: It is an error if a role or attribute *name* has already been defined. Note that unlike roles, attributes cannot be transitive.

Examples:
```
(defprimattribute Best-friend)
(defprimattribute Father :supers (Parent))
(defprimattribute Mother :supers (Parent))
```

defprimattribute-f *function*

Description: Functional equivalent of **defprimattribute**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(defprimattribute-f 'Best-friend)
(defprimattribute-f 'Father :supers '(Parent))
(defprimattribute-f 'Mother :supers '(Parent))
```

implies *macro*

Description: An implication/subsumption axiom between two concept descriptions.

Syntax: (**implies** *antecedent consequent*)

Arguments: *antecedent* - Description of the antecedent concept.

consequent - Description of the consequent concept.

Return: *antecedent* is returned.

Remarks: Asserts that *antecedent* implies *consequent* (*antecedent* $\Rightarrow$ *consequent*) or, equivalently, that the *antecedent* is subsumed by *consequent* (*antecedent* $\sqsubseteq$ *consequent*). Note that adding implications after the TBox has been classified may cause strange and unpredictable results.

Examples:
```
(implies (and POLYGON ((some Angles 3)))
    (some Sides 3))
```

implies-f *function*

Description: Functional equivalent of **implies**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(implies-f '(and POLYGON ((some Angles 3)))
     '(some Sides 3))
```

---

disjoint *macro*

Description: A disjointness axiom between concept descriptions.

Syntax: (**disjoint** *description-1* ... *description-n*)

Arguments: *description-i*- A concept description.

Remarks: Asserts that the extensions of *description-1*,...,*description-n* are disjoint.

Examples:
```
(disjoint MALE FEMALE)
(disjoint CAT DOG RABBIT HAMSTER)
```

---

disjoint-f *function*

Description: Functional equivalent of **disjoint**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(disjoint-f 'MALE 'FEMALE)
(disjoint 'CAT 'DOG 'RABBIT 'HAMSTER)
```

A.3.4 TBox Inferences

---

classify-tkb *function*

Description: Classifies the TBox.

Syntax: (**classify-tkb** `&key` (*mode* `:nothing`))

Arguments: *mode* - Keyword which controls the output from the classifier:

`:nothing` - None;

`:stars`	- A symbol for each concept classified: `P` for a primitive concept, `C` for a non-primitive concept and `S` for a synonym;
`:names`	- The name of each concept classified followed by `-P` for a primitive concept, `-C` for an non-primitive concept and `-S` for a synonym;
`:count`	- The number of subsumption and satisfiability tests.

Warnings are always output regardless of the setting of *mode*. If *mode* is either `:stars` or `:names` then symbols are also output during the pre-processing of roles (`r` for each role `a` for each attribute), implications (`p` for each implication absorbed into a primitive concept, `i` for each non-absorbed implication) and concept terms (`c` for each concept term). If omitted, *mode* defaults to `:nothing`.

Remarks: If any new implications have been added to the TBox (**implies** or **f-implies**) since the last TBox classification (**classify-tkb**), all concepts will be reclassified; otherwise classification will be incremental.

Examples: `(classify-tkb :mode :stars)`

---

direct-supers *function*

Description: Finds the direct super-concepts of a classified concept.

Syntax: (**direct-supers** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of the direct subsumers (super-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(direct-supers 'MAN)` ⇒ `(c[HUMAN] c[MALE])`

---

all-supers *function*

Description: Finds all the super-concepts of a classified concept.

Syntax: (**all-supers** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the subsumers (super-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(all-supers 'MAN)` ⇒ `(c[ANIMAL] c[HUMAN] c[BIPED] c[MALE] c[*TOP*])`

---

direct-subs *function*

Description: Finds the direct sub-concepts of a classified concept.

Syntax: (**direct-subs** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of the direct subsumees (sub-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(direct-subs 'MALE)` ⇒ `(c[MAN])`

---

all-subs *function*

Description: Finds all the sub-concepts of a classified concept.

Syntax: (**all-subs** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the subsumees (sub-concepts) of the concept *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

Examples: `(all-subs 'MALE)` ⇒ `(c[MAN] c[*BOTTOM*])`

equivalences *function*

Description: Finds those concepts which are equivalent to a classified concept.

Syntax: (**equivalences** *name*)

Arguments: *name* - A concept name.

Return: Returns a list of all the concepts which are equivalent to (synonyms for) *name*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *name* must be quoted.

satisfiable *function*

Description: Tests if a concept description is satisfiable.

Syntax: (**satisfiable** *description*)

Arguments: *description* - A concept description.

Return: Returns `T` if *description* is satisfiable w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description* must be quoted.

Examples:

```
(satisfiable '(and MALE FEMALE)) ⇒ nil
(satisfiable '(and MALE ANIMAL)) ⇒ T
```

subsumes *function*

Description: Tests if one concept description subsumes another.

Syntax: (**subsumes** *description-1 description-2*)

Arguments: *description-1* - A concept description.

description-2 - A concept description.

Return: Returns `T` if *description-1* subsumes *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(subsumes '(and MALE ANIMAL) 'MAN)` ⇒ `T`

---

equivalent-concepts *function*

Description: Tests if two concept descriptions are equivalent.

Syntax: (**equivalent-concepts** *description-1 description-2*)

Arguments: *description-1* - A concept description.

description-2 - A concept description.

Return: Returns `T` if *description-1* is equivalent to *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(equivalent-concepts '(and MALE HUMAN) 'MAN)` ⇒ `T`
`(equivalent-concepts 'HUMAN '(and ANIMAL BIPED))` ⇒ `nil`

---

disjoint-concepts *function*

Description: Tests if two concept descriptions are disjoint.

Syntax: (**disjoint-concepts** *description-1 description-2*)

Arguments: *description-1* - A concept description.

description-2 - A concept description.

Return: Returns `T` if *description-1* is disjoint from *description-2* w.r.t. the current TBox, `nil` otherwise.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description-1* and *description-2* must be quoted.

Examples: `(disjoint-concepts 'MALE 'FEMALE)` ⇒ `T`

classify-concept *function*

Description: Finds where a concept description would classify without adding it to the TBox.

Syntax: (**classify-concept** *description*)

Arguments: *description* - A concept description.

Return: Returns three values: a list of all the direct-supers of *description*; a list of all the direct-subs of *description*; a list of all the concepts which are equivalent to *description*.

Remarks: The TBox must have been classified using **classify-tkb**. Note that *description* must be quoted.

Examples:
```
(classify-concept '(and MALE ANIMAL)) ⇒ (c[MALE]
    c[ANIMAL]) (c[MAN]) nil
```

add-concept *macro*

Description: Defines a new concept and classifies the TBox using **classify-tkb**.

Syntax: (**add-concept** *name description* `&key` (*primitive* `nil`))

Arguments:

name	- Name of the new concept.
description	- Description of the new concept.
primitive	- Keyword which, if non-`nil`, makes the concept primitive. If omitted it defaults to `nil`.

Return: A concept structure `c[`*name*`]` is returned.

Remarks: It is an error if a concept *name* has already been defined.

Examples:
```
(add-concept VEGETABLE *TOP*:primitive T)
(add-concept WOMAN (and FEMALE HUMAN))
```

add-concept-f *function*

Description: Functional equivalent of **add-concept**.

Remarks: Note that all arguments have to be quoted.

Examples:
```
(add-concept-f 'VEGETABLE '*TOP*  :primitive T)
(add-concept-f 'WOMAN '(and FEMALE HUMAN))
```

A.3.5 TBox Queries

get-concept *function*

Description: Retrieves a concept from the TBox.

Syntax: (**get-concept** *name*)

Arguments: *name* - Concept name (a LISP atom).

Return: A concept structure `c[`*name*`]` if a concept *name* is defined in the TBox; `nil` otherwise.

get-role *function*

Description: Retrieves a role or attribute from the TBox.

Syntax: (**get-role** *name*)

Arguments: *name* - Role or attribute name (a LISP atom).

Return: A role structure `r[`*name*`]` if a role or attribute *name* is defined in the TBox; `nil` otherwise.

get-all-concepts *function*

Description: Retrieves all concepts from the TBox.

Syntax: (**get-all-concepts**)

Return: A list of all the concepts defined in the TBox.

get-all-roles *function*

Description: Retrieves all roles and attributes from the TBox.

Syntax: (**get-all-roles**)

Return: A list of all the roles and attributes defined in the TBox.

classified-tkb? *function*

Description: Tests if the TBox is classified.

Syntax: (**classified-tkb?**)

Return: `T` if the TBox is classified; `nil` otherwise.

what-is? *function*

Description: Determines the type of a concept, or role structure.

Syntax: (**what-is?** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The type of structure, one of `CONCEPT`, `PRIMITIVE`, `ROLE` or `FEATURE`.

Remarks: Returns `FEATURE` if *structure* is an attribute.

is-primitive? *function*

Description: Determines if a structure is a primitive concept (of type `PRIMITIVE`).

Syntax: (**is-primitive?** *structure*)

Arguments: *structure* - Concept structure.

Return: `T` if *structure* is a primitive concept (of type `PRIMITIVE`); `nil` otherwise.

is-concept? *function*

Description: Determines if a structure is a non-primitive concept (of type `CONCEPT`).

Syntax: (**is-concept?** *structure*)

Arguments: *structure* - Concept structure.

Return: `T` if *structure* is a non-primitive concept (of type `CONCEPT`); `nil` otherwise.

---

is-role? *function*

Description: Determines if a structure is a role (of type `ROLE`).

Syntax: (**is-role?** *structure*)

Arguments: *structure* - Role structure.

Return: `T` if *structure* is a role (of type `ROLE`); `nil` otherwise.

---

is-feature? *function*

Description: Determines if a structure is an attribute (of type `FEATURE`).

Syntax: (**is-feature?** *structure*)

Arguments: *structure* - Role structure.

Return: `T` if *structure* is an attribute (of type `FEATURE`); `nil` otherwise.

---

name *function*

Description: Retrieves the name of a concept or role structure.

Syntax: (**name** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The name of *structure* if it is a concept or role structure; `nil` otherwise.

---

description *function*

Description: Retrieves the description of a concept or role structure.

Syntax: (**description** *structure*)

Arguments: *structure* - Concept or role structure.

Return: The description of *structure* if it is a concept or role structure; `nil` otherwise. The format of the description depends on the type of structure:

`PRIMITIVE` - The *description* given in the **defprimconcept** or **defprimconcept-f** definition, possibly extended by the absorption of **implies** axioms.

`CONCEPT` - The *description* given in the **defconcept** or **defconcept-f** definition.

`ROLE` - A list consisting of the role's name, the keyword `:supers` followed by a list of the role's *supers* given in the **defprimrole** or **defprimrole-f** definition and the keyword `:transitive` followed by `T` if the role is transitive (the last 2 items will be omitted if their values are `nil`).

`FEATURE` - A list consisting of the attribute's name and the keyword `:supers` followed by a list of the attribute's *supers* given in the **defprimattribute** or **defprimattribute-f** definition (the last item will be omitted if its value is `nil`).

A.4 Controlling FaCT's Behavior

This section describes functions and macros which allow FaCT's features to be customised and which control performance profiling.

---

set-features *function*

Description: Enables classifier features.

Syntax: (**set-features** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is enabled. The available keywords are as follows:

```
:transitivity :concept-eqn
:subset-s-equivalent :backjumping
:obvious-subs :top-level-caching
:full-caching :blocking :taxonomic-encoding
:gci-absorption :cyclical-definitions
:auto-configure :moms-heuristic
```

`:prefer-pos-lits :minimise-clashes`
`:auto-install-primitives`
`:auto-install-transitive`

By default all features are enabled except:

`:moms-heuristic`, `:auto-install-primitives`
`:auto-install-transitive`

Return: A list of all enabled features.

Remarks: See source code for details of the effect of each feature.

Examples:
```
(set-features)
(set-features :transitivity :backjumping)
```

---

reset-features *function*

Description: Disables classifier features.

Syntax: (**reset-features** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is disabled.

Return: A list of all enabled features.

Remarks: See **set-features** for a list of features. See source code for details of the effect of each feature.

Examples:
```
(reset-features :transitivity :backjumping)
```

---

features *function*

Description: Prints information about feature settings.

Syntax: (**features** `&optional` (*stream* `T`))

Arguments: *stream* - Optional output stream; default is `T`, which writes output to the `*terminal-io*` stream.

Examples:
```
(features)
```

---

set-profiling *function*

Description: Controls performance profiling.

Syntax: (**set-profiling** `&key` (*level* 1) (*file* "profile.out"))

Arguments: *level* - Keyword which controls the amount of profiling data which is output:

0	- Disables profiling.
1	- Outputs profiling data for each concept classification.
2	- Outputs profiling data for each subsumption test.
3	- Outputs profiling data for each satisfiability test.

If omitted, *level* defaults to 1.

file - Keyword specifying the name of a file to which the profiling data is to be written. If explicitly `nil`, it is written to the `*terminal-io*` stream. If omitted, *file* defaults to "profile.out".

Remarks: Note that profiling can generate a large amount of data, particularly if `level` is >1. For each satisfiability test the profiler outputs the number of backtracks, the maximum model size, the maximum model depth, the number of cache accesses, the number of cache hits, the CPU time used, the result (`T` or `nil`) and whether blocking was triggered (`T` or `nil`). See source code for more details.

Examples:
```
(set-profiling)
(set-profiling :level 2 :file nil)
```

---

reset-profiling *function*

Description: Disables performance profiling.

Syntax: (**reset-profiling**)

Remarks: Equivalent to `(set-profiling :level 0)`.

Examples:
```
(reset-profiling)
```

set-verbosity *function*

Description: Increases the verbosity of the classifier.

Syntax: (**set-verbosity** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The verbosity feature associated with each supplied keyword is enabled. The available keywords are as follows:

```
:warnings :notes :synonyms :reclassifying
:features :rc-counts :test-counts
:cache-counts :classify-1 :classify-2
```

By default, the of enabled verbosity features are:

```
:warnings :synonyms :reclassifying
:rc-counts :test-counts :cache-counts
```

Return: A list of all enabled verbosity features.

Remarks: The verbosity setting is temporarily overridden by the *verbose* argument to **load-tkb** and the *mode* argument to **classify-tkb**. See source code for details of the effect of each verbosity feature.

Examples:

```
(set-verbosity)
(set-verbosity :notes :classify-2)
```

reset-verbosity *function*

Description: Reduces the verbosity of the classifier.

Syntax: (**reset-verbosity** `&rest` *features*)

Arguments: *features* - Zero or more keywords. The feature associated with each supplied keyword is disabled. If no keywords are supplied, all verbosity features are disabled.

Return: A list of all enabled verbosity features.

Remarks: See **set-features** for a list of verbosity features. See source code for details of the effect of each feature.

Examples:

```
(reset-verbosity)
(reset-verbosity :notes :classify-2)
```

A.5 Modal Logic Theorem Proving

FaCT can also be used as a decision procedure for the propositional modal logics $\mathbf{K_{(m)}}$, $\mathbf{K4_{(m)}}$ and $\mathbf{S4_{(m)}}$. Although the functions and macros described in Section A.3 could be used for this purpose, FaCT provides a more convenient functional interface for performing single satisfiability tests on $\mathbf{K_{(m)}}$, $\mathbf{K4_{(m)}}$ and $\mathbf{S4_{(m)}}$ formulae encoded as $\mathcal{ALC}$ concept descriptions.

---

alc-concept-coherent *function*

Description: Tests the satisfiability of a $\mathbf{K_{(m)}}$, $\mathbf{K4_{(m)}}$ or $\mathbf{S4_{(m)}}$ formula encoded as an $\mathcal{ALC}$ concept description.

Syntax: (**alc-concept-coherent** *description* `&key` (*logic* `'K`))

Arguments: *description* - $\mathbf{K_{(m)}}$, $\mathbf{K4_{(m)}}$ or $\mathbf{S4_{(m)}}$ formula encoded as an $\mathcal{ALC}$ concept description.

logic - Keyword which specifies the logic to be used:

- `'K` - modal $\mathbf{K_{(m)}}$;
- `'K4` - modal $\mathbf{K4_{(m)}}$ (all roles are transitive);
- `'S4` - modal $\mathbf{S4_{(m)}}$ (all roles are transitive reflexive).

If omitted, *logic* defaults to `'K`.

Return: Returns four values: `T` if *description* is satisfiable, `nil` otherwise; run time (seconds) excluding concept encoding; number of backtracks; maximum model size (nodes).

Remarks: All user defined concepts, roles, attributes and implications are deleted from the TBox (using **init-tkb**); some or all of the concepts and roles occuring in *description* are then automatically installed in the TBox. The verbosity setting is temporarily overridden and set to `nil`. Profiling is temporarily disabled.

Examples:

```
(alc-concept-coherent '(and (some R (some R C1))
    (all R (not C1)))) ⇒ T
(alc-concept-coherent '(and (some R (some R C1))
    (all R (not C1))) :logic 'k4) ⇒ nil
```

Bibliography

[ADS96] L. C. Aiello, J. Doyle, and S. Shapiro, editors. *Principals of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*. Morgan Kaufmann, November 1996.

[Baa90a] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. Research Report RR-90-13, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1990.

[Baa90b] F. Baader. A formal definition of the expressive power of knowledge representation languages. Research Report RR-90-05, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), April 1990.

[Baa90c] F. Baader. Terminological cycles in KL-ONE-based knowledge representation languages. Research Report RR-90-01, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), January 1990.

[Bak95] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995.

[BBH96] F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1–2):195–213, 1996.

[BBJN96] F. Baader, M. Buchheit, M.A. Jeusfeld, and W. Nutt, editors. *Reasoning about structured objects: knowledge representation meets databases. Proceedings of the 3rd Workshop KRDB'96*, 1996.

[BBLS94] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Terminological logics for schema design and query processing in OODBs. In F. Baader, M. Buchheit, M.A. Jeusfeld, and W. Nutt, editors, *Reasoning about structured objects: knowledge representation meets databases. Proceedings of the KI'94 Workshop KRDB'94*, pages 58–62, September 1994.

[BBN+91] F. Baader, H.-J. Burckert, B. Nebel, W. Nutt, and G. Smolka. On the expressivity of feature logics with negation, functional uncertainty and sort equations. Research Report RR-91-01, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1991.

[BDS93] M. Buchheit, F. M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[BFL83] R. J. Brachman, R. E. Fikes, and H. J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, October 1983.

[BFT95] P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive description logics: a preliminary report. In Ellis et al. [ELFD95], pages 28–39.

[BG96] S. Bechhofer and C. Goble. Description logics and multimedia—applying lessons learnt from the GALEN project. In *Proceedings of the workshop on Knowledge Representation for Interactive Multimedia Systems (KRIMS'96), at ECAI'96*, pages 1–9, Budapest, Hungary, August 1996.

[BGL85] R. J. Brachman, V. P. Gilbert, and H. J. Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 532–539, August 1985.

[BH91a] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In Proceedings of IJCAI-91 [Pro91], pages 452–457.

[BH91b] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In *Processing declarative knowledge: International workshop PDK'91*, number 567 in Lecture Notes in Artificial Intelligence, pages 67–86, Berlin, 1991. Springer-Verlag.

[BH91c] F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3):8–14, 1991.

[BHH+91] F. Baader, H.-J. Heinsohn, B. Hollunder, J. Muller, B. Nebel, W. Nutt, and H.-J. Profitlich. Terminological knowledge representation: A proposal for a terminological logic. Technical Memo

TM-90-04, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1991.

[BHNP92] F. Baader, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems. In B. Nebel, C. Rich, and W. Swartout, editors, *Principals of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 270–281. Morgan-Kaufmann, 1992. Also available as DFKI RR-93-03.

[BIG94] J. M. Blanco, A. Illarramendi, and A. Goni. Building a federated database system: an approach using a knowledge based system. *Journal of Intelligent and Cooperative Information Systems*, 3(4):415–455, 1994.

[BIW94] J. I. Berman, H. H. Moore IV, and J. R. Wright. Classic and PROSE stories: Enabling technologies for knowledge based systems. *AT&T Technical Journal*, pages 69–78, January/February 1994.

[BJNS94] M. Buchheit, M. A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented databases. *Information Systems*, 19(1):33–54, 1994.

[Bor92] A. Borgida. Description logics are not just for the flightless-birds: A new look at the utility and foundations of description logics. Technical Report DCS-TR-295, New Brunswick Department of Computer Science, Rutgers University, 1992.

[Bor96] A. Borgida. On the relative expressiveness of description logics and first order logics. *Artificial Intelligence*, 82:353–367, 1996.

[BPS94] A. Borgida and P. F. Patel-Schneider. A semantics and complete algorithm for subsumption in the Classic description logic. *Journal of Artificial Intelligence Research*, 1:277–308, 1994.

[Bre95] P. Bresciani. Querying databases from description logics. In F. Baader, M. Buchheit, M.A. Jeusfeld, and W. Nutt, editors, *Reasoning about structured objects: knowledge representation meets databases. Proceedings of the 2nd Workshop KRDB'95*, pages 1–4, 1995.

[Bri93] D. Brill. Loom *reference manual version 2.0.* University of Southern California, Information Sciences Institute, December 1993.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April 1985.

[BS96a] F. Baader and U. Sattler. Description logics with symbolic number restrictions. In Wahlster [Wah96], pages 283–287.

[BS96b] F. Baader and U. Sattler. Number restrictions on complex roles in description logics. In Aiello et al. [ADS96], pages 328–339.

[Cal96] D. Calvanese. Reasoning with inclusion axioms in description logics: Algorithms and complexity. In Wahlster [Wah96], pages 303–307.

[CL94] T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 2(4):375–398, 1994.

[DBBS91] P. Devanbu, R. J. Brachman, B. Ballard, and P. G. Selfridge. LaSSIE - a knowledge-based software information system. *Communications of the ACM*, 34(5):35–49, 1991.

[DHL+89] F. Donini, B. Hollunder, M. Lenzerini, A. M. Spaccamela, D. Nardi, and W. Nutt. The frontier of tractability for concept description languages. Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1989.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DLNN95] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. Research Report RR-95-07, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1995.

[DP91] J. Doyle and R. Patil. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence*, 48:261–297, 1991.

[ELFD95] Gerard Ellis, Robert A. Levinson, Andrew Fall, and Veronica Dahl, editors. *Knowledge Retrieval, Use and Storage for Efficiency: Proceedings of the First International KRUSE Symposium*, 1995.

[Ell92] G. Ellis. Compiled hierarchical retrieval. In T. Nagle, J. Nagle, L. Gerholz, and P. Eklund, editors, *Conceptual Structures: Current Research and Practice*, pages 285–310. Ellis Horwood, 1992.

[FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[Fre96] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.

[GBBI96] A. Goni, J. Bermúdez, J. M. Blanco, and A. Illarramendi. Using reasoning of description logics for query processing in multidatabase systems. In Baader et al. [BBJN96], pages 18–23.

[GBS+94] C. A. Goble, S. K. Bechhofer, W. D. Solomon, A. L. Rector, W. A. Nowlan, and A. J. Glowinski. Conceptual, semantic and information models for medicine. In *Proceedings of the 4th European-Japanese Seminar on Information Modelling and Knowledge Bases*, pages 257–286, Stockholm, Sweden, 31st May – 3rd June 1994.

[GHB96] C. A. Goble, C. Haul, and S. Bechhofer. Describing and classifying multimedia using the description logic GRAIL. In *Proceedings of IS&T/SPIE, vol 2670, Storage and Retrieval for Still Image and Video Databases* {*IV*}, pages 132–143, San Jose, California, USA, February 1996.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[GINR96] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Moving a robot: The KR&R approach at work. In Aiello et al. [ADS96], pages 198–209.

[GL90] R. V. Guha and D. B. Lenat. Cyc: A midterm report. *AI Magazine*, pages 32–59, October 1990.

[GL94] R. V. Guha and D. B. Lenat. Enabling agents to work together. *Communications of the ACM*, 37(7):127–142, July 1994.

[GL96] G. De Giacomo and M. Lenzerini. TBox and ABox reasoning in expressive description logics. In Aiello et al. [ADS96], pages 316–327.

[Gou84] G. Gough. Decision procedures for temporal logic. Master's thesis, University of Manchester, 1984.

[GS96a] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures—the case study of modal K. In McRobbie and Slaney [MS96], pages 583–597.

[GS96b] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for $\mathcal{ALC}$. In Aiello et al. [ADS96], pages 304–314.

[HGG+97] I. Horrocks, C. Goble, G. Gough, N. Paton, and A. Rector. CAMELOT research proposal. http://www.cs.man.ac.uk/~horrocks/camelot.html, 1997.

[HHW96] T. Hogg, B. A. Huberman, and C. P. Williams. Phase transitions and the search problem. *Artificial Intelligence*, 81:1–15, 1996. Editorial.

[HKNP94] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich. An empirical analysis of terminological representation systems. *Artificial Intelligence*, 68:367–397, 1994.

[HN90] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Research Report RR-90-04, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), April 1990.

[Hor94] P. Hors. Description logics to specify the part–whole relation. In *Proceedings of the ECAI'94 Workshop on Parts and Wholes: Conceptual Part–Whole Relations and Formal Mereology*, pages 103–109, 1994.

[Hor95] I. Horrocks. A comparison of two terminological knowledge representation systems. Master's thesis, University of Manchester, 1995.

[HRG96] I. Horrocks, A. Rector, and C. Goble. A description logic based schema for the classification of medical data. In Baader et al. [BBJN96], pages 24–28.

[HS97] U. Hustadt and R. A. Schmidt. On evaluating decision procedures for modal logic. Technical Report MPI-I-97-2-003, Max-Planck-Institut Für Informatik, Im Stadtwald, D 66123 Saarbrücken, Germany, February 1997.

[JW90] R. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KdR97] N. Kurtonina and M. de Rijke. Expressiveness of first-order description languages. Research Report RR-325, Department of Computer Science, University of Warwick, 1997.

[Kob91] A. Kobsa. First experiences with the Sb-One knowledge representation workbench in natural-language applications. *SIGART Bulletin*, 2(3):70–76, 1991.

[LG89] D. B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, 1989.

[LG91] D. B. Lenat and R. V. Guha. The evolution of CycL, the Cyc representation language. *SIGART Bulletin*, 2(3):84–87, 1991.

[LR96] A. Y. Levy and M.-C. Rousset. Using description logics to model and reason about views. In Baader et al. [BBJN96], pages 48–49.

[Mac91a] R. M. MacGregor. The evolving technology of classification-based knowledge representation systems. In Sowa [Sow91], chapter 13, pages 385–400.

[Mac91b] R. M. MacGregor. Inside the Loom description classifier. *SIGART Bulletin*, 2(3):88–92, 1991.

[MB92] R. M. MacGregor and D. Brill. Recognition algorithms for the Loom classifier. Technical report, University of Southern California, Information Sciences Institute, 1992.

[MDW91] E. Mays, R. Dionne, and R. Weida. K-rep system overview. *SIGART Bulletin*, 2(3):93–97, 1991.

[Mos83] M. G. Moser. An overview of Nikl: the new implementation of Kl-One. Technical report 5421, Bolt, Beranek and Newman, Cambridge, MA, 1983.

[MS96] Michael McRobbie and John Slaney, editors. *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-13)*, number 1104 in Lecture Notes in Artificial Intelligence. Springer, 1996.

[MWD+96] E. Mays, R. Weida, R. Dionne, M. Laker, B. White, C. Liang, and F. J. Oles. Scalable and expressive medical terminologies. In *Proceedings of the 1996 AMAI Fall Symposium*, 1996.

[Neb90a] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Number 422 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 1990.

[Neb90b] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, 1990.

[Neb91] B. Nebel. Terminological cycles: Semantics and computational properties. In Sowa [Sow91], chapter 11, pages 331–361.

[Nor92] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo, CA, 1992.

[Now93] W. A. Nowlan. *Structured methods of information management for medical records.* PhD thesis, University of Manchester, 1993.

[N.R96] N.Rychtyckyj. DLMS: An evaluation of KL-ONE in the automobile industry. In Aiello et al. [ADS96], pages 328–339.

[NSA+94] S. Navathe, A. Savasere, T. Anwar, H. Beck, and S. Gala. Object modeling using clasification in CANDIDE and its applications. In A. Dogac, T. Ozsu, A. Briliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, pages 435–476. NATO ASI Series Vol 130, 1994.

[OS95] H. J. Ohlbach and R. A. Schmidt. Functional translation and second-order frame properties of modal logics. Research Report MPI-I-95-2-002, Max-Planck-Institut Für Informatik, Im Stadtwald, D 66123 Saarbrücken, Germany, January 1995.

[Pel91] C. Peltason. The BACK system—an overview. *SIGART Bulletin*, 2(3):114–119, 1991.

[PL94] L. Padgham and P. Lambrix. A framework for part–of hierarchies in terminological logics. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Principals of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR'94)*, pages 485–496. Morgan-Kaufmann, 1994.

[Pro91] *Proceedings of the 12th International Joint Conference on Artificial Intelligence, (IJCAI-91)*, 1991.

[PS89] P. F. Patel-Schneider. A four-valued semantics for terminological logics. *Artificial Intelligence*, 38(3):319–351, 1989.

[PS91] P. F. Patel-Schneider. The CLASSIC knowledge representation system: Guiding principals and implementation rationale. *SIGART Bulletin*, 2(3):108–113, 1991.

[RBG+97] A. Rector, S. Bechhofer, C. A. Goble, I. Horrocks, W. A. Nowlan, and W. D. Solomon. The Grail concept modelling language for medical terminology. *Artificial Intelligence in Medicine*, 9:139–171, 1997.

[RGG+94] A. L. Rector, A. Gangemi, E. Galeazzi, A. J. Glowinski, and A Rossi-Mori. The Galen core model schemata for anatomy: towards a re-useable application-independent model of medical concepts. In P. Barahona, M. Veloso, and J. Bryant, editors, *Proceedings of Medical Informatics in Europe (MIE 94)*, pages 229–233, 1994.

[RH97] A. Rector and I. Horrocks. Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium (AAAI'97)*. AAAI Press, Menlo Park, California, 1997.

[RN95] G. Riley and A. Nisbet. Paepr concept module. Paepr Deliverable 6, Centre for Novel Computing, University of Manchester, October 1995.

[RNG93] A. L. Rector, W A Nowlan, and A Glowinski. Goals for concept representation in the Galen project. In *Proceedings of the 17th Annual Symposium on Computer Applications in Medical Care (SCAMC'93)*, pages 414–418, Washington DC, USA, 1993.

[RSNR94] A. L. Rector, W. D. Solomon, W. A. Nowlan, and T. W. Rush. A terminology server for medical language and medical information systems. In *Proceedings of IMIA WG6*, Geneva, Switzerland, May 1994.

[Sat95] U. Sattler. A concept language for engineering applications with part–whole relations. In *Proceedings of the International Conference on Description Logics—DL'95*, pages 119–123, Roma, Italy, 1995.

[Sat96] U. Sattler. A concept language extended with different kinds of transitive roles. In G. Görz and S. Hölldobler, editors, *20. Deutsche Jahrestagung für Künstliche Intelligenz*, number 1137 in Lecture Notes in Artificial Intelligence, pages 333–345. Springer Verlag, 1996.

[Sch91] K. Schild. A correspondence theory for terminological logics: Preliminary report. In Proceedings of IJCAI-91 [Pro91], pages 466–471.

[Sch93] K. Schild. Terminological cycles and the propositional μ-calculus. Technical Report RR-93-18, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), April 1993.

[SML96] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996.

[Smu68] R. M. Smullyan. *First-Order Logic.* Springer-Verlag, 1968.

[Sol97] W. D. Solomon. Tigger: Motivation, functionality and interface. Available from: Project Coordinator, Medical Informatics Group, Department of Computer Science, University of Manchester, Manchester M13 9PL, UK, 1997.

[Sow91] J. F. Sowa, editor. *Principals of Semantic Networks: Explorations in the representation of knowledge.* Morgan-Kaufmann, 1991.

[Spe95] P-H. Speel. *Selecting Knowledge Representation Systems.* PhD thesis, Universiteit Twente, 1995.

[SS89] M. Schmidt-Schauß. Subsumption in Kl-One is undecidable. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Principals of Knowledge Representation and Reasoning: Proceedings of the First International Conference (KR'89)*, pages 421–431. Morgan-Kaufmann, 1989.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[ST97] W. D. Solomon and C. J. Toft. *The* Grail *KnoME: Overview and User Guide*, 1997. Available from: Project Coordinator, Medical Informatics Group, Department of Computer Science, University of Manchester, Manchester M13 9PL, UK.

[SvRvdVM95] P.-H. Speel, F. van Raalte, P. E. van der Vet, and N. J. I. Mars. Runtime and memory usage performance of description logics. In Ellis et al. [ELFD95], pages 13–27.

[Tar56] A. Tarski. *Logic, Semantics, Mathemetics: Papers from 1923 to 1938.* Oxford University Press, 1956.

[Wah96] Wolfgang Wahlster, editor. *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96).* John Wiley & Sons Ltd., 1996.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER version 0.42. In McRobbie and Slaney [MS96], pages 141–145.

[Woo91] W. A. Woods. Understanding subsumption and taxonomy: a framework for progress. In Sowa [Sow91], chapter 1, pages 45–94.

[WS92] W. A. Woods and J. G. Schmolze. The Kl-One family. *Computers and Mathematics with Applications – Special Issue on Artificial Intelligence*, 23(2–5):133–177, 1992.