ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

Optimisation Techniques for Expressive Description Logics

Ian Horrocks

Department of Computer Science

University of Manchester

Technical Report Series

UMCS-97-2-1

Optimisation Techniques for Expressive Description Logics*

Ian Horrocks

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
horrocks@cs.man.ac.uk

1st February 1997

*Refereed by: Graham Gough

Abstract

This report describes and evaluates optimisation techniques for a tableaux based satisfiability testing algorithm used to compute subsumption in GRAIL, an expressive description logic. Five techniques are studied in detail: normalisation and encoding, indexing, semantic branching, dependency directed backtracking and caching. The effectiveness of these techniques is evaluated by empirical testing using a large knowledge base from the GALEN project. The performance of the optimised classifier and subsumption test are also compared with that of the KRIS classifier and KSAT satisfiability testing procedure using both the GALEN knowledge base and randomly generated test data.

1 Introduction

As part of the European GALEN project the Medical Informatics Group at Manchester University have built a large concept model representing knowledge about medical terminology. The model is intended to promote sharing and re-use of medical data by acting as a flexible and extensible classification schema [40, 39].

In existing applications this function is often served by 'controlled vocabularies', of which more than 20 are currently in use. These vocabularies or 'coding schemes' are often very large and thus both difficult and costly to build and maintain. Moreover, in spite of the fact that many schemes are specific to particular purposes, specialities or even databases, their rigidity frequently compels application designers to use additional ad-hoc terms to cover fine detail [38].

The concept model has been built using GRAIL, a description logic (DL) based knowledge representation system. By using a DL based model it is hoped to avoid many of the pitfalls of existing static coding schemes and to provide additional benefits to applications:

- concept descriptions have clear semantics and more detailed descriptions can be constructed systematically to provide principled extensions to the terminology 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 [12] and query optimisation [11];
- the concept model can be used in intelligent data entry systems [35, 28] and in natural language processing [10, 37, 48];
- data can be shared between existing applications by using the concept model as an interlingua and providing mappings to a variety of coding schemes.

1.1 The GRAIL Description Logic

GRAIL was developed specifically for building the medical terminology model (although it is now being used in a variety of other applications [23, 22]). A detailed design study [34] led to the specification of a DL with a restricted set of concept and role forming operators but an unusually expressive set of terminological axioms. GRAIL is differentiated from other implemented DLs by its support for these axioms:

- Role inclusion axioms—used to construct hierarchies based on transitive relations other than is-a, such as the various part-whole and compositional relations which are essential for describing anatomy (and other complex physical structures). *e.g.* the axiom:

  $$\textit{has-part} \circ \textit{has-part} \sqsubseteq \textit{has-part}$$

  asserts that the *has-part* role is transitively closed.

- General Concept Inclusion axioms (GCIs)—used to add knowledge which does not form part of concept definitions. *e.g.* the GCI:

  $$\mathsf{Ulcer} \sqcap \exists \textit{hasLocation}.\mathsf{Stomach} \sqsubseteq \exists \textit{hasLocation}.\mathsf{StomachLining}$$

  asserts that all stomach ulcers are located in the lining of the stomach *without* making $\exists \textit{hasLocation}.\mathsf{StomachLining}$ part of the definition of a stomach ulcer.

It is worth noting that the requirement for similarly (or more) expressive DLs has been demonstrated for a wide range of other applications [30, 41].

1.2 Subsumption Algorithms

In common with the majority of implemented DL systems [31, 32, 36, 29] the original GRAIL classifier uses a structural subsumption algorithm. This algorithm is known to be incomplete and may even be unsound [27].

In contrast, the new GRAIL classifier converts the subsumption problem into an equivalent satisfiability problem which is solved using a sound and complete tableaux based algorithm [26]. Tableaux algorithms have the additional advantage that they can be relatively easily adapted to deal with different description languages [13]. Extensions to such algorithms which support both transitive closure [3] and GCIs [14] are well understood but the problem's inherent complexity (EXPTIME) appears to limit their practical applicability. However empirical studies using implemented DL systems (which do not support transitive closure or GCIs) have shown that:

- constructs which lead to pathological behaviour rarely occur in 'realistic' knowledge bases [24];

- optimising the implementation can significantly improve the performance of a tableaux algorithm [8].

In view of these considerations it seemed worthwhile to investigate tableaux optimisation techniques for a more expressive DL (Grail) via empirical testing with a real knowledge base (the Galen model). This report describes a number of optimisation techniques and presents some preliminary test results.

2 Description Logic Syntax and Semantics

Description Logics (DLs) are a family of formalisms closely related to semantic networks but with the distinguishing characteristic that the semantics of the concept (class) description language is sufficiently well defined that the resulting structured objects can be reasoned with [5]. In particular the *subsumption* (sub-class/super-class) relationship between two concept expressions can be computed by a suitable algorithm.

Although implemented DLs use various forms of syntax in their description languages a standard infix notation is commonly used in theoretical papers. The interpretation of concept expressions and of the subsumption relationship is also widely based on a Tarski style model theoretic semantics [47].

2.1 Syntax

DLs support the logical description of concepts, roles (relationships) and attributes (single-valued roles). Concepts, roles and attributes can be combined, using using a variety of operators, to form more complex expressions [50] which can be used in terminological axioms to add information to the knowledge base. The operators supported by DLs usually include some or all of the standard logical connectives along with one or both of universal and existential quantification (often called value restrictions and exists restrictions). Using standard infix notation the concept `parent`, a `person` who is related via the `child` role to another `person`, might be introduced by the axiom:

$$\mathtt{parent} \doteq \mathtt{person} \sqcap \exists \mathtt{child}.\mathtt{person}$$

Concepts can be viewed as corresponding to unary predicates in first order logic and roles as corresponding to binary predicates. Using this correspondence the above axiom could be translated into first order logic as:

$$\forall x.(\mathtt{parent}(x) \iff \mathtt{person}(x) \wedge \exists y(\mathtt{person}(y) \wedge \mathtt{child}(x, y)))$$

2.2 Model Theoretic Semantics

A Tarski style model theoretic semantics [47] 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 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 a concept expression, R a role expression and A an attribute expression.

The meaning of concepts, roles and attributes is given by an interpretation $\mathcal{I}$ which consists of a set $\Delta^{\mathcal{I}}$, the domain, and an interpretation function $\cdot^{\mathcal{I}}$ [5]. 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:

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

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

The interpretations of concept, role and attribute expressions can be derived from that of their components using the semantics of concept, role and attribute forming operators as described in the following section.

2.2.1 Concept and Role Forming Operators

DLs can support a wide variety of operators for building and combining concept expressions. The semantics of some of the most common operators is described in Table 1 on the following page.

Less commonly, DLs can also support a range of operators for building and combining role and attribute expressions. A selection of these operators and their semantics is described in Table 2 on the next page. All of the operators can be used with either roles or attributes but it should be noted that $A_1 \sqcup A_2$, A^{-1}, A^{+} and A^{*} are in general roles and not attributes.

2.2.2 Terminological Axioms

Terminological axioms are used to add information to the knowledge base. Most DLs only support a restricted set of axioms which introduce new names and associate them with

Operator	Notation	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
conjunction	$C_1 \sqcap \ldots \sqcap C_n$	$C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$
disjunction	$C_1 \sqcup \ldots \sqcup C_n$	$C_1^{\mathcal{I}} \cup \ldots \cup C_n^{\mathcal{I}}$
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	$\geq nR.C$	$\{d \in \Delta^{\mathcal{I}} \mid \|R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\| \geq n\}$
number restriction	$\leq nR.C$	$\{d \in \Delta^{\mathcal{I}} \mid \|R^{\mathcal{I}}(d) \cap C^{\mathcal{I}}\| \leq n\}$

Table 1: Concept forming operators

Operator	Notation	Semantics
conjunction	$R_1 \sqcap \ldots \sqcap R_n$	$R_1^{\mathcal{I}} \cap \ldots \cap R_n^{\mathcal{I}}$
disjunction	$R_1 \sqcup \ldots \sqcup R_n$	$R_1^{\mathcal{I}} \cup \ldots \cup R_n^{\mathcal{I}}$
composition	$R_1 \circ \ldots \circ R_n$	$R_1^{\mathcal{I}} \circ \ldots \circ R_n^{\mathcal{I}}$
inverse	R^{-1}	$\{(d, d') \mid (d', d) \in R^{\mathcal{I}}\}$
transitive closure	R^{+}	$\bigcup_{n \geq 1} (R^{\mathcal{I}})^n$
transitive reflexive closure	R^{*}	$\bigcup_{n \geq 0} (R^{\mathcal{I}})^n$

Table 2: Role and attribute forming operators

Axiom	Notation	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	$\mathsf{AN} \doteq A$	$\mathsf{AN}^{\mathcal{I}} = A^{\mathcal{I}}$
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	$\mathsf{AN} \sqsubseteq A$	$\mathsf{AN}^{\mathcal{I}} = A^{\mathcal{I}}$

Table 3: Terminological axioms

an expression through either an equality or a subsumption relationship. The semantics of these axioms is described in Table 3 on this page.

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 equivalence relation are known as *non-primitives*. Non-primitives are fully defined by their characteristics—the characteristics are said to be both necessary and sufficient. *e.g.* the introduction `woman` $\doteq$ `human` $\sqcap$ `female` states that a `woman` is necessarily both `human` and `female` and that the conjunction of `human`(x) and `female`(x) is sufficient to infer `woman`(x). For this reason, the concept expression associated with a non-primitive concept name will be called its definition: in this case $\texttt{human} \sqcap \texttt{female}$ is the definition of `woman`.

Primitives on the other hand are not fully defined by their characteristics—they have only necessary characteristics. *e.g.* the introduction $\texttt{human} \sqsubseteq \texttt{featherless} \sqcap \texttt{biped}$ states that a human is necessarily both `featherless` and a `biped` but the conjunction of $\texttt{featherless}(x)$ and $\texttt{biped}(x)$ is not sufficient to infer $\texttt{human}(x)$. Natural kinds, such as `human`, are often primitives as it is difficult to describe them definitively.

2.2.3 General Concept Inclusion Axioms

Expressive DLs may support more general terminological axioms of the form $C_1 \sqsubseteq C_2$ where C_1 and C_2 are arbitrary concept expressions. These axioms are variously known as general concept inclusions (GCIs), concept equations or universal terminological axioms: their semantics is defined in Table 4 on the current page.

Example 2.1 The knowledge that all stomach ulcers occur in the lining of the stomach could be represented by the general concept inclusion axiom:

$$\mathsf{Ulcer} \sqcap \exists \mathit{hasLocation}.\mathsf{Stomach} \sqsubseteq \exists \mathit{hasLocation}.\mathsf{StomachLining}$$

Notation	Semantics
$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$

Table 4: General concept inclusion axioms

The concept introduction axioms described in Section 2.2.2 are special forms of GCI—primitive introductions are simply GCIs where the left hand expression is a concept name while non-primitive introductions are equivalent to a pair of GCIs:

$$\mathsf{CN} \doteq C \quad \equiv \quad \left\{ \begin{array}{rcl} \mathsf{CN} & \sqsubseteq & C \\ C & \sqsubseteq & \mathsf{CN} \end{array} \right.$$

2.3 Subsumption and Classification

Subsumption is the subclass-superclass relationship between concept, role and attribute expressions; classification is the computation of a partial ordering based on the subsumption relation. In practice, subsumption between and classification of concept expressions is the usual focus of interest. A DL classifier will cache the computed partial order in the form of a concept hierarchy which can then 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 [8].

The semantics of DLs mean that subsumption can be formally defined in terms of the subset-superset 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 $\mathcal{T}$. C_1 is subsumed by C_2 in $\mathcal{T}$, written $C_1 \preceq_{\mathcal{T}} C_2$ if and only if $C_1^{\mathcal{I}}$ is a subset of $C_2^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$:

$$C_1 \preceq_{\mathcal{T}} C_2 \iff C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}} \quad \text{for all models } \mathcal{I} \text{ of } \mathcal{T}$$

Most implemented DLs restrict terminological axioms to unique and acyclic concept, role and attribute introductions—a name can only appear once on the left hand side of an axiom (unique) and an expression on the right hand side of an axiom cannot refer either directly or indirectly to the name on the left hand side (acyclic). Given these restrictions, any $\mathcal{I}$ is a model of any $\mathcal{T}$ and subsumption between concept expressions is therefore independent of $\mathcal{T}$:

$$C_1 \preceq C_2 \iff C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}} \quad \text{for all } \mathcal{I}$$

This semantics for the subsumption relation, called descriptive semantics, can produce counter-intuitive results when the terminology contains cycles [33]. Alternative semantics based on least and greatest fixpoints have been proposed [2, 9] but it is not clear that any one semantics is universally preferable [44]. 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 (but see [44]), are applicable to all DLs, including those supporting GCIs. Subsumption based on descriptive semantics will be assumed in the remainder of this report.

3 Tableaux Subsumption Algorithms

Most early DL systems used structural subsumption algorithms [49] based on rules such as:

$$C \succeq (A \sqcap B) \iff C \succeq A \lor C \succeq B$$

—C subsumes $A \sqcap B$ if and only if C subsumes either A or B. An alternative approach, first used in the Kris system [7], transposes the subsumption problem into an equivalent satisfiability problem:

$$C \succeq D \iff \neg(D \sqcap \neg C)$$

—C subsumes D if and only if $D \sqcap \neg C$ is not satisfiable. The satisfiability problem can be solved using an algorithm based on the tableaux calculus [7, 26]. This approach has many advantages and has dominated recent DL research:

- it has a sound theoretical basis in first order logic [26];
- it can be relatively easily adapted to allow for different expressive possibilities by changing the set of tableaux expansion rules [13];

- 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 [26];
- the theoretical frontiers of decidability and tractability are well understood [43, 15].

3.1 Basic Method

Tableaux algorithms test the satisfiability of a concept expression by trying to construct a model—a domain in which some individual satisfies the concept expression. They use a constraint system S to describe possible models and a set of expansion rules which are applied to S until it is fully expanded, and thus describes a complete model, or demonstrates obvious contradictions which prove that a model cannot be constructed.

For a relatively simple DL such as $\mathcal{ALC}$ [45] only two forms of constraint are required, $x : C$ and xRy where x and y are variables corresponding to unique individuals in the domain. The constraint $x : C$ states that the individual x is of type C ($x \in C^{\mathcal{I}}$) while the constraint xRy states that x is related to y by role R ($y \in R^{\mathcal{I}}(x)$). A constraint system consists of a finite set of constraints.

To determine the satisfiability of a concept expression C, S is initialised to contain a single constraint $x : C$. This states that the model must include some individual x such that $x \in C^{\mathcal{I}}$. A set of expansion rules are repeatedly applied to S until it is fully expanded or an obvious contradiction is detected. S is fully expanded when none of the expansion rules are applicable and contains a contradiction when, for some x and some C, either $x : \bot$ or both $x : C$ and $x : \neg C$ are in S. A fully expanded constraint system can trivially be converted into a model which is a witness to the satisfiability of C.

3.2 Expansion Rules

The expansion rules for $\mathcal{ALC}$ are shown in Table 5 on the following page. There are 4 rules corresponding to the operators supported by the logic: $\sqcap$, $\sqcup$, $\exists$ and $\forall$—the need for a $\neg$ rule is eliminated by internalising negations using a combination of de Morgan's laws and the axioms:

$$\begin{aligned} \neg\exists R.C &\iff \forall R.\neg C \\ \neg\forall R.C &\iff \exists R.\neg C \end{aligned}$$

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. It can be intuitively seen that the algorithm is guaranteed to terminate because:

$\sqcap$	if	1.	$x : (C \sqcap D) \in S$
		2.	$x : C \notin S \quad \mathit{or} \quad x : D \notin S$
	then		$S \longrightarrow S \cup \{x : C,\ x : D\}$
$\sqcup$	if	1.	$x : (C \sqcup D) \in S$
		2.	$x : C \notin S \quad \mathit{and} \quad x : D \notin S$
	then		$S \longrightarrow S \cup \{x : C\} \quad \mathit{or} \quad S \longrightarrow S \cup \{x : D\}$
$\exists$	if	1.	$x : \exists R.C \in S$
		2.	$\neg\exists y(xRy \in S\ \wedge\ y : C \in S)$
	then		$S \longrightarrow S \cup \{xRz,\ z : C\}$
			where z is a new variable
$\forall$	if	1.	$x : \forall R.C \in S \quad \mathit{and} \quad xRy \in S$
		2.	$y : C \notin S$
	then		$S \longrightarrow S \cup \{y : C\}$

Table 5: Tableaux expansion rules for $\mathcal{ALC}$

(a) The $\sqcap$, $\sqcup$ and $\exists$ rules can only be applied once to a given $x : C$ constraint.

(b) The $\forall$ rule can be applied many times to a given $x : \forall R.C$ constraint but only once to a given xRy constraint.

(c) The constraints added by each rule are always smaller than the constraints to which the rule was applied.

2. The $\sqcup$ rule for disjunctive constraints is different from the other rules: it is non-deterministic and operates by searching different possible expansions.

3.3 Extended Tableaux Algorithms

In order to deal with GCIs the basic $\mathcal{ALC}$ algorithm can be extended by the addition of a new kind of constraint called a universal constraint and a more sophisticated control strategy known as blocking.

3.3.1 Universal Constraints

When a terminology contains GCIs, they must all be satisfied in any valid model. A GCI $C_1 \sqsubseteq C_2$ is satisfied in a model $\mathcal{I}$ if and only if every individual in $\mathcal{I}$ is either in $C_2^{\mathcal{I}}$ or not in $C_1^{\mathcal{I}}$. This condition is imposed in a constraint system S by adding a universal constraint $\forall x.x : C$ where $C \doteq C_2 \sqcap \neg C_1$ [14].

Example 3.1 The GCI from Example 2.1 on page 6 would be converted into the universal constraint:

$$\forall x.x : \neg\mathsf{Ulcer} \sqcup \forall \mathit{hasLocation}.\neg\mathsf{Stomach} \sqcup \exists \mathit{hasLocation}.\mathsf{StomachLining}$$

This constraint states that every individual in a valid model is either not an Ulcer or not located in the Stomach or is located in the StomachLining.

The expansion rule for universal constraints, given in Table 6 on this page, ensures that when S contains the universal constraint $\forall x.x : C$, every variable y in S will be subject to the constraint $y : C$.

$\forall x$	if 1.	$\forall x.x : C \in S$ *and*	y is a variable in S
	2.	$y : C \notin S$	
	then	$S \longrightarrow S \cup \{y : C\}$	

Table 6: Tableaux expansion rule for universal constraints

To determine the satisfiability of a concept expression C with respect to a terminology $\mathcal{T}$ containing the GCIs $C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n$, S is initialised to contain the constraint $x : C$ plus the universal constraints $\forall x.x : D_1 \sqcup \neg C_1, \ldots, \forall x.x : D_n \sqcup \neg C_n$. Note that each universal constraints will cause a disjunctive constraint to be applied to every variable in S.

3.3.2 Blocking

Universal constraints do not satisfy the termination conditions stated in Section 3.2 and it is easy to see that a more sophisticated control strategy is required in order to guarantee termination; *e.g.* if $\forall x.x : \exists R.C \in S$, an infinite sequence of applications of the $\forall x$ and $\exists$ rules would result in non-termination.

In order to avoid this the algorithm uses a more sophisticated control strategy called blocking. To describe blocking it is necessary to introduce some new terms and notation:

- A generating rule is a tableaux expansion rule which introduces a new variable to the constraint system; in the case of $\mathcal{ALC}$ the $\exists$ rule is the only generating rule.
- The parent of a variable x is the variable which triggered the application of the generating rule which introduced x; ancestor is the transitive closure of parent.
- The constraints on variable v in S, written S_v, is the set of constraints $x : C$ or xRy such that $x = v$.

Blocking imposes a new condition on generating rules: the rule can only be applied to a variable x if it has no ancestor variable y such that $S_x \subseteq S_y$ [14, 4]. If a variable does not meet this condition it is said to be blocked. When a variable is blocked it is in effect being identified with the blocking variable and the constraint system 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 constraints and therefore a finite number of different sets of constraints; all variables must therefore eventually be blocked.

4 The New GRAIL Classifier

Key GRAIL statements and their equivalent abstract forms are summarised in Table 7 on the current page. In the (confusingly named) `newAttribute` statement K is a cardinality keyword which determines whether R_1 and R_2 are roles (many valued) or attributes (single valued): K can be one of *oneOne* (both R_1 and R_2 are attributes), *manyMany* (neither are attributes), *oneMany* (R_1 is an attribute) or *manyOne* (R_2 is an attribute).

GRAIL Statement	Abstract form
C `which` $\langle R_1\, C_1 \ldots R_n\, C_n \rangle$	$C \sqcap \exists R_1.C_1 \sqcap \ldots$ $\ldots \sqcap \exists R_n.C_n$
C `newSub` CN	$\mathsf{CN} \sqsubseteq C$
C `name` CN	$\mathsf{CN} \doteq C$
C `topicNecessarily` $\langle R_1\, C_1 \ldots R_n\, C_n \rangle$	$C \sqsubseteq \exists R_1.C_1 \sqcap \ldots$ $\ldots \sqcap \exists R_n.C_n$
RN `newAttribute` RN_1 RN_2 K	$\begin{cases} \mathsf{RN}_1 \sqsubseteq \mathsf{RN} \\ \mathsf{RN}_2 \doteq \mathsf{RN}_1^{-1} \end{cases}$
RN_1 `addSub` RN_2	$\mathsf{RN}_2 \sqsubseteq \mathsf{RN}_1$
RN_1 `specialisedBy` RN_2	$\mathsf{RN}_1 \circ \mathsf{RN}_2 \sqsubseteq \mathsf{RN}_1$

Table 7: GRAIL statements and equivalent abstract forms

From Table 7 it can be seen that the only concept forming operators in the GRAIL language are conjunction ($\sqcap$) and exists restriction ($\exists$); GRAIL concept expressions are therefore a subset of $\mathcal{ALC}$ expressions. GRAIL's concept axioms can also be seen to be a subset of the concept introduction and GCI axioms described in Table 3 on page 5 and Table 4 on page 6. The new GRAIL classifier uses an extended tableaux subsumption algorithm based on [14, 3, 42, 4] which supports the full set of $\mathcal{ALC}$ concept forming operators as well as GCIs.

GRAIL's role forming operators and axioms are rather more complex. Although GRAIL's syntax implies support for the inverse role forming operator the subsumption algorithm does not support reasoning about inverse roles—the two new roles introduced by a `newAttribute` statement are treated independently. As reasoning with inverse roles is problematical [43, 18] the same approach has been adopted in the new GRAIL classifier.

Grail's role inclusion axioms are dealt with by extensions to the tableaux expansion rules similar to those used to deal with transitively closed roles [42]; these extensions do not affect the operation of the optimisation techniques and their description is beyond the scope of this report.

5 Intractability

When classifying the Galen knowledge base the large number of GCIs is a major cause of intractability. Constraint systems used in subsumption/satisfiability testing will include a set of universal constraints corresponding to the set of GCIs in the knowledge base; each universal constraint will cause a disjunctive constraint to be added to every variable in the constraint system, resulting in an exponential increase in the number of constraint systems explored by the $\sqcup$ expansion rule.

If there are n GCIs in the knowledge base n disjunctions will be applied to each variable. Each disjunction could be expanded in at least 2 possible ways so this gives at least $\#\text{variables}\times 2^n$ possible constraint systems which may have to be searched. As there are several hundred GCIs in the Galen knowledge base it is not surprising that a naive implementation of the tableaux algorithm results in effective non termination.

6 Optimisation Techniques

The new Grail classifier includes a number of adapted and novel optimisations which try to take advantage of the structure of a 'realistic' knowledge base. These include:

- Lexically normalising and encoding all concept expressions;
- Indexing GCI constraints so only 'relevant' GCIs are fully expanded;
- Semantic branching, a search technique adapted from the Davis-Putnam-Logeman-Loveland procedure (DPL) commonly use to solve propositional satisfiability (SAT) problems;
- Dependency directed backtracking (backjumping);
- Caching and re-using partial constraint systems.

The new classifier also includes optimisation techniques which have already been demonstrated to be effective [8]. These include:

- Enhanced traversal—a modified breadth first search technique which minimises the number of subsumption tests required to classify a concept.

- Lazy expansion—early clash detection is facilitated by substituting definitions for concept names in constraints only as required by the expansion algorithm and by retaining a copy of the original constraint in the constraint system.

6.1 Normalisation and Encoding

GRAIL lexically normalises and encodes all concept expressions and, recursively, their sub-expressions. This ensures that:

1. All concept expressions are in a standard form; *e.g.* all exists restrictions are converted to value restrictions so $\exists R.A$ would be normalised to $\neg\forall R.\neg A$;
2. All sub-expressions are atomic concept names; *e.g.* $\forall R.A \sqcap \forall R.B$ would be encoded as $C_1 \sqcap C_2$ where $C_1 \doteq \forall R.A$ and $C_2 \doteq \forall R.B$.

A functional definition of the normalisation and encoding process is given in Table 8 on the following page. Note that:

- the *Encode* function minimises the number of new concept definitions added to the knowledge base $\mathcal{T}$ by identifying lexically equivalent conjunctive expressions;
- non-primitive concepts introduced by the encoding process are are *not* classified.

Example 6.1 Normalising the expression $\exists R.(A \sqcap C \sqcap \neg B) \sqcap \forall R.(B \sqcup \neg A \sqcup \neg C)$ leads to the normalisation and encoding of the two sub-expressions $\exists R.(A \sqcap C \sqcap \neg B)$ and $\forall R.(B \sqcup \neg A \sqcup \neg C)$. After normalisation the first sub-expression becomes $\neg C_y$, where $C_y \doteq \forall R.\neg C_x$ and $C_x \doteq A \sqcap \neg B \sqcap C$. When the second sub-expression is normalised $(B \sqcup \neg A \sqcup \neg C)$ becomes $\neg(\neg B \sqcap A \sqcap C)$ and is encoded as $\neg C_x$; $\forall R.\neg C_x$ is then encoded as C_y. The combined expression $\neg C_y \sqcap C_y$ is then normalised as $\bot$ which means that the expression is identified as being unsatisfiable without recourse to the tableaux expansion algorithm.

Normalisation and encoding has a number of advantages:

- syntactically obvious unsatisfiability can be detected without using the tableaux procedure;
- the effect of the lazy expansion optimisation (described in Section 6 on the page before) is maximised—clashing concepts can be detected without fully expanding their definitions;
- knowledge bases with large amounts of repetitive structure may be more compactly stored;

$$
\begin{array}{l}
\text{Normalise}(A): \\
A = \text{atomic concept name} \Rightarrow A \\
A = \neg B \Rightarrow \begin{array}{ll}
C & \textit{if } \text{Normalise}(B) = \neg C \\
\bot & \textit{if } \text{Normalise}(B) = \top \\
\top & \textit{if } \text{Normalise}(B) = \bot \\
\textit{otherwise} & \neg\text{Normalise}(B)
\end{array} \\
A = \forall R.B \Rightarrow \begin{array}{ll}
\top & \textit{if } \text{Normalise}(B) = \top \\
\textit{otherwise} & \text{Encode}(\forall R.\text{Normalise}(B))
\end{array} \\
A = B_1 \sqcap \ldots \sqcap B_n \Rightarrow \begin{array}{ll}
\bot & \textit{if } \bot \in \{\text{Normalise}(B_1), \ldots, \text{Normalise}(B_n)\} \\
\bot & \textit{if } \exists C.(\{C, \neg C\} \subseteq \{\text{Normalise}(B_1), \ldots, \text{Normalise}(B_n)\}) \\
\textit{otherwise} & \text{Encode}(\text{Normalise}(B_1) \sqcap \ldots \sqcap \text{Normalise}(B_n))
\end{array} \\
A = \exists R.B \Rightarrow \text{Normalise}(\neg\forall R.\neg B) \\
A = B_1 \sqcup \ldots \sqcup B_n \Rightarrow \text{Normalise}(\neg(\neg B_1 \sqcap \ldots \sqcap \neg B_n)) \\
\\
\text{Encode}(A): \\
A = \forall R.B \Rightarrow \begin{array}{ll}
C & \textit{if } C \doteq \forall R.B \in \mathcal{T} \\
\textit{otherwise} & D \text{ where } D \text{ is a new concept name } \textit{and} \\
& \qquad \mathcal{T} \longrightarrow \mathcal{T} \cup D \doteq \forall R.B
\end{array} \\
A = B_1 \sqcap \ldots \sqcap B_n \Rightarrow \begin{array}{ll}
C & \textit{if } C \doteq (B'_1 \sqcap \ldots \sqcap B'_n) \in \mathcal{T} \textit{ and} \\
& \qquad \forall B.(B \in \{B_1, \ldots, B_n\} \Leftrightarrow B \in \{B'_1, \ldots, B'_n\}) \\
\textit{otherwise} & D \text{ where } D \text{ is a new concept name } \textit{and} \\
& \qquad \mathcal{T} \longrightarrow \mathcal{T} \cup D \doteq B_1 \sqcap \ldots \sqcap B_n
\end{array}
\end{array}
$$

Table 8: Normalisation and encoding

- more efficient data structures can be used—*e.g.* if all concepts are encoded as integers, hash tables can be replaced with arrays;
- other optimisation techniques are facilitated—*e.g.* semantic branching and its associated heuristic functions (see Section 6.3 on page 16).

6.2 Indexing GCI Constraints

This technique uses knowledge about the structure and function of GCIs in the GALEN terminology:

1. The left hand side of a GCI is always a conjunctive concept expression, one element of which is either a primitive or is a defined concept which can be expanded into a conjunctive expression containing a primitive.
2. Each GCI represents additional knowledge about some specific concept—*e.g.* the GCI in Example 2.1 on page 6 represents the knowledge that stomach ulcers always

occur in the lining of the stomach—and consequently is irrelevant to the majority of subsumption tests.

As described in section 5 on page 12, testing satisfiability with respect to a terminology which contains a GCI $C \sqsubseteq D$ is achieved by applying the universal constraint $\forall x.x : D \sqcup \neg C$ to any model being constructed by the tableaux expansion algorithm. When, as in the GALEN terminology, C is a conjunctive expression of the form $P \sqcap A_1 \sqcap \ldots \sqcap A_n$, where P is primitive, this leads to a universal constraint of the form:

$$\forall x.x : D \sqcup \neg P \sqcup \neg A_1 \sqcup \ldots \sqcup \neg A_n$$

The effect of irrelevant GCIs can be minimised by ordering the search of possible expansions of the resulting disjunctive constraints on each variable x so that $S \longrightarrow S \cup \{x : \neg P\}$ is chosen first. Because P is primitive, $x : \neg P$ requires no further expansion and can remain implicit—an explicit expansion of the remainder of the disjunction is only necessary when $x : P$ is added to the constraint system.

For example, testing satisfiability w.r.t. a terminology containing the GCI from Example 2.1 on page 6 means that the universal constraint from Example 3.1 on page 10 will apply and that every variable x in a constraint system S generated by the tableaux expansion algorithm will be subject to the disjunctive constraint:

$$x : \exists \mathit{hasLocation}.\mathsf{StomachLining} \sqcup \neg\mathsf{Ulcer} \sqcup \forall \mathit{hasLocation}.\neg\mathsf{Stomach}$$

The implicit expansion $S \longrightarrow S \cup \{x : \neg\mathsf{Ulcer}\}$ can be assumed unless and until $x : \mathsf{Ulcer}$ is added to S, when explicit expansion of the remainder of the disjunction becomes necessary:

$$S \longrightarrow S \cup \{x : \exists \mathit{hasLocation}.\mathsf{StomachLining} \sqcup \forall \mathit{hasLocation}.\neg\mathsf{Stomach}\}$$

This technique can be extended to deal with a set of n GCIs:

$$\begin{gathered} P_1 \sqcap A_{11} \sqcap \ldots \sqcap A_{1m} \sqsubseteq B_1 \\ \vdots \\ P_n \sqcap A_{n1} \sqcap \ldots \sqcap A_{nm} \sqsubseteq B_n \end{gathered}$$

The GCIs are transformed into n universal constraints, each containing a negated primitive $\neg P_j$:

$$\begin{gathered} \forall x.x : \neg P_1 \sqcup \neg A_{11} \sqcup \ldots \sqcup \neg A_{1m} \sqcup B_1 \\ \vdots \\ \forall x.x : \neg P_n \sqcup \neg A_{n1} \sqcup \ldots \sqcup \neg A_{nm} \sqcup B_n \end{gathered}$$

These constraints are used to construct a table (which can be an array if all concept names are encoded as integers—see section 6.1 on page 13—or a hash table otherwise) indexed by $P_1 \dots P_n$. The entry corresponding to each P_j is a (conjunction of) disjunction(s) from the universal constraint(s) containing P_j:

index	*entry*
P_1	$(\neg A_{11} \sqcup \dots \sqcup \neg A_{1m} \sqcup B_1) \sqcap \dots$
$\vdots$	$\vdots$
P_n	$(\neg A_{n1} \sqcup \dots \sqcup \neg A_{nm} \sqcup B_n) \sqcap \dots$

The expansion algorithm makes the default assumption that every variable $x_i \in S$ is subject to the constraints $x_i : \neg P_1, \dots, x_i : \neg P_n$. Whenever a constraint $x_i : P_j$ is added to S, P_j is looked up in the table. If there is an entry for P_j it is added as a constraint of the form:

$$x_i : (\neg A_{j1} \sqcup \dots \sqcup \neg A_{jm} \sqcup B_j) \sqcap \dots$$

Although this optimisation depends on the structure of GCIs in the Galen terminology there is no loss of generality as GCIs which do not meet condition 1 on page 14 can still be dealt with in the usual way by adding the full disjunctive constraint to every individual.

It is perhaps worth noting that this optimisation is in fact a form of dynamic backtracking [19]: a branching choice is made (*i.e.* the selection of the negated primitive from a GCI disjunction); search continues, perhaps involving further branching choices; a contradiction is discovered; the earlier choice is changed *without* discarding subsequent searching.

6.3 Semantic Branching

Standard tableaux algorithms are inherently inefficient as the search technique uses syntactic branching—choosing an unexpanded disjunction and searching all the possible constraint systems obtained by adding constraints corresponding to each of the disjuncts [20]. As the alternative constraint systems are not disjoint there is nothing to prevent the recurrence of clashing constraints in different branches of the search tree. The resulting wasted expansion could be costly if discovering the clash requires the solution of a complex sub-problem.

Example 6.2 Tableaux expansion of a constraint system S, where:

$$\{x : (A \sqcup B),\ x : (A \sqcup C)\} \subseteq S$$

could lead to the search pattern shown in Figure 1 on the next page.

Grail deals with this problem by using a semantic branching technique adapted from DPL SAT algorithms [17]. Instead of choosing an unexpanded disjunction, Grail chooses a single disjunct from one of the unexpanded disjunctions and searches the two possible constraint systems obtained by adding constraints corresponding to the chosen disjunct and its negation. *e.g.* if $x : (A \sqcup B) \in S$ the algorithm might choose A and search the two constraint systems $S \cup \{x : A\}$ and $S \cup \{x : \neg A\}$.

Using semantic branching has a number of advantages:

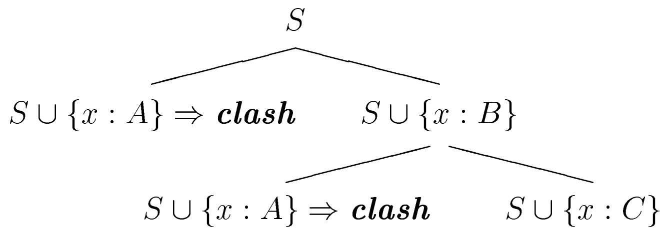

Figure 1: syntactic branching

- 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 [16].

6.3.1 Boolean Constraint Propagation

Boolean constraint propagation (BCP) [16] is a technique used to maximise deterministic expansion, and thus pruning of the search tree via clash detection, before performing non-deterministic expansion (branching). BCP deterministically expands disjunctive constraints presenting only one expansion possibility and detects a clash when a disjunctive constraint has no expansion possibilities. The number of expansion possibilities presented by a disjunctive constraint $x : C_1 \sqcup \ldots \sqcup C_n$ in a constraint system S is equal to the number of disjuncts C_i such that $x : \neg C_i \notin S$.

Example 6.3 Given a constraint system S such that:

$$\{x : (A \sqcup (B \sqcap C)),\ x : (\neg B \sqcup \neg C),\ x : \neg A\} \subseteq S$$

BCP deterministically expands the constraint $x : (A \sqcup (B \sqcap C))$ because $x : \neg A \in S$:

$$S \longrightarrow S \cup \{x : (B \sqcap C)\}$$

After $x : (B \sqcap C)$ is expanded to give $S \longrightarrow S \cup \{x : B,\ x : C\}$, BCP also identifies $x : (\neg B \sqcup \neg C)$ as a clash because $x : B \in S$ and $x : C \in S$.

6.3.2 Heuristic Guided Search

A heuristic technique is used to guide the search in a way which tries to minimise the size of the search tree by maximising the pruning effect of the constraint added at each branching point. When expanding the constraints on a variable x, Grail examines all the unexpanded disjunctive constraints on x and branches on the disjunct which has the Maximum number of Occurrences in disjunctions of Minimum Size—the well known MOMS heuristic [16]. The order in which the two branches are searched is also heuristically determined based on the ratio of positive and negated occurrences of the chosen disjunct.

Example 6.4 When applied to the constraint system from Example 6.2 on page 16 the GRAIL algorithm would use the MOMS heuristic to select A as the disjunct on which to branch and search the two possible expansions:

$$S \longrightarrow S \cup \{x : A\} \quad \text{or} \quad S \longrightarrow S \cup \{x : \neg A\}$$

resulting in the search pattern illustrated in Figure 2 on the current page. Note that when $x : \neg A$ is added to S, BCP will lead deterministically to the further expansion $S \longrightarrow S \cup \{x : B,\ x : C\}$.

$$S$$

$$S \cup \{x : A\} \Rightarrow \textit{\textbf{clash}} \qquad S \cup \{x : \neg A\}$$

$$S \cup \{x : B,\ x : C\}$$

Figure 2: semantic branching

6.3.3 The GRAIL Search Algorithm

In general, for a variable x in a constraint system S, the GRAIL search algorithm proceeds as follows:

1. Perform all deterministic expansions; *e.g.* apply the $\sqcap$, $\exists$, $\forall$ and $\forall x$ rules from Table 5 on page 9 and Table 6 on page 10.
2. Perform Boolean Constraint Propagation (BCP):
   (a) Search for constraints $x : (C_1 \sqcup \ldots \sqcup C_n)$ where:
   $$\begin{aligned} \{x : \neg C_1, \ldots, x : \neg C_{i-1}\} &\subseteq S \\ \{x : \neg C_{i+1}, \ldots, x : \neg C_n\} &\subseteq S \\ x : C_i &\notin S \end{aligned}$$
   (b) If such a constraint is found and $x : \neg C_i \in S$ return ***clash***.
   (c) If such a constraint is found and $x : \neg C_i \notin S$, expand the constraint:
   $$S \longrightarrow S \cup \{x : C_i\}$$
   and return to step 1.

3. Perform heuristic guided search if there are unexpanded disjunctions $x : (C_{11} \sqcup \ldots \sqcup C_{1n}), \ldots, x : (C_{m1} \sqcup \ldots \sqcup C_{mn})$ in S.
   (a) Use MOMS heuristic to select C_{ij} such that $x : \neg C_{ij} \notin S$.
   (b) Search:
   $$S \longrightarrow S \cup \{x : C_{ij}\}$$
   or
   $$S \longrightarrow S \cup \{x : \neg C_{ij}\}$$
   in a heuristically determined order.

6.4 Dependency Directed Backtracking

Inherent unsatisfiability concealed in sub-problems can lead to large amounts of unproductive backtracking search known as thrashing.

Example 6.5 A constraint system S, where:

$$S = \{x : (C_1 \sqcup D_1), \ldots, x : (C_n \sqcup D_n),\ x : \exists R.(A \sqcup B),\ x : \forall R.E\}$$

and $(A \sqcup B) \sqcap E \Rightarrow$ ***clash***, could lead to the fruitless exploration of 2^n truth assignments for $C_1, \ldots, C_n$ before the inherent unsatisfiability is discovered.

The GRAIL classifier addresses this problem by employing a form of dependency directed backtracking called *backjumping*. This works by labeling constraints with a dependency set indicating the truth assignments which led to their introduction. When a clash is discovered the algorithm can jump back over irrelevant assignments without exploring the alternative branches.

For example when expanding the constraint system from Example 6.5 on this page, the first set of truth assignments for $C_1, \ldots, C_n$ will lead to the discovery of the clash with respect to $y : (A \sqcup B)$ and $y : E$. As neither of these constraints has the truth assignments for $C_1, \ldots, C_n$ in their dependency sets it is clear that the clash did not depend on any of these assignments. The algorithm can therefore either return *unsatisfiable* immediately (if the dependency sets of the clashing constraints were empty) or jump directly back to the most recent truth assignment on which the clash did depend *without* exploring the alternative truth assignments for $C_1, \ldots, C_n$.

In more general terms backjumping works as follows:

1. The initial constraint(s) in a constraint system S have their dependency sets initialised to $\emptyset$.

2. Constraints added by deterministic expansion rules ($\sqcap$, $\exists$, $\forall$ and $\forall x$) are labelled with the union of the label(s) from the constraint(s) which triggered the expansion.

3. A constraint added by the nth truth assignment is labelled $\{n\}$.

4. After a clash, return a dependency set $\mathcal{D}$ consisting of the (union of the) label(s) from the clashing constraint(s).
5. If the first branch of the nth truth assignment 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 branch of the nth truth assignment returns a dependency set $\mathcal{D}_2$, backtrack returning the dependency set $(\mathcal{D}_1 \cup \mathcal{D}_2) - \{n\}$.

6.5 Caching

The combination of normalisation, encoding and lazy expansion [8] facilitates the rapid detection of 'obvious' unsatisfiability (subsumption) but detecting 'obvious' satisfiability (non-subsumption) is more difficult for tableaux algorithms. This is unfortunate as:

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

The Grail classifier tackles this problem by trying to use cached results from previous tableaux tests to demonstrate the satisfiability of a concept expression. A considerable amount of work can be saved when large or complex models are re-used in this way.

Example 6.6 Given two concepts:

$$\begin{aligned} A &= C \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2 \\ \neg B &= D \sqcap \exists R_3.C_3 \end{aligned}$$

the satisfiability of $A \sqcap \neg B$ (and thus the non-subsumption $A \not\preceq B$) can be demonstrated by a model consisting of models of A and $\neg B$ joined at their roots, as shown in Figure 3 on the next page.

To demonstrate that two models joined at their roots result in a valid (non-clashing) model, it is only necessary to examine their root constraints (constraints on their first variables). Two models represented by constraint systems S^A and S^B can be joined in this way *unless*:

1. The union of their root constraints contains an immediate contradiction; *e.g.*:

$$x_0 : C \in S^A \wedge x_0 : \neg C \in S^B$$

2. An additional tableaux expansion rule may be applicable when the root constraints are combined, *e.g.*:

$$x_0 R x_n \in S^A \wedge x_0 : \forall R.C \in S^B \wedge x_0 : \forall R.C \notin S^A \qquad (1)$$

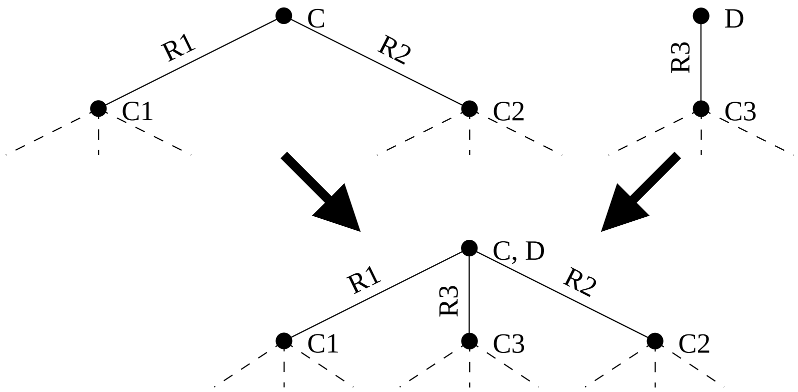

Figure 3: Merging models of $C \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2$ and $D \sqcap \exists R_3.C_3$

Note that condition 2 states that an additional rule *may* be applicable—it would be possible to merge models in a wider range of cases by considering additional variables: *e.g.* in Equation 1 on the preceding page, if $x_n : C \in S^A$. However caching only root constraints has the advantage of simplifying the merging procedure and minimising space requirements. The space required for caching can be further reduced by storing only:

$$\begin{aligned} S_C &= \{C \mid x_0 : C \in S\} \\ S_{\neg C} &= \{C \mid x_0 : \neg C \in S\} \\ S_R &= \{R \mid x_0 R y \in S\} \\ S_\forall &= \{\langle R, C\rangle \mid x_0 : \forall R.C \in S\} \end{aligned}$$

In more general terms, when testing if A is subsumed by B ($A \preceq B$?), caching works as follows:

1. If models for A, $\neg A$, B or $\neg B$ have not been cached, perform the satisfiability test(s) and cache S_C, $S_{\neg C}$, S_R and $S_\forall$ from the fully expanded constraint system(s). If a satisfiability test fails the concept is equal to $\bot$ and its negation to $\top$.

2. Return *true* or *false* if obvious subsumption or non-subsumption is detected:

$$\begin{aligned} \neg B = \bot &\;\Rightarrow\; \textit{true}\ (A \preceq B) \\ A = \bot &\;\Rightarrow\; \textit{true}\ (A \preceq B) \\ B = \bot \wedge A \neq \bot &\;\Rightarrow\; \textit{false}\ (A \not\preceq B) \\ \neg A = \bot \wedge \neg B \neq \bot &\;\Rightarrow\; \textit{false}\ (A \not\preceq B) \end{aligned}$$

3. Return *false* ($A \not\preceq B$) if the root constraints from A (S^A) and $\neg B$ ($S^{\neg B}$) can be merged. S^A and $S^{\neg B}$ can be merged *unless*:

   (a) $S^A_C \cap S^{\neg B}_{\neg C} \neq \emptyset$

(b) $S^A_{\neg C} \cap S^{\neg B}_C \neq \emptyset$

(c) $\exists R, C.(R \in S^A_R \wedge \langle R, C\rangle \in S^{\neg B}_\forall \wedge \langle R, C\rangle \notin S^A_\forall)$

(d) $\exists R, C.(R \in S^{\neg B}_R \wedge \langle R, C\rangle \in S^A_\forall \wedge \langle R, C\rangle \notin S^{\neg B}_\forall))$

(e) $\exists R.(R \in S^A_R \wedge R \in S^{\neg B}_R \wedge R$ is functional)

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

It would be possible to extend this technique in order to avoid constructing obviously satisfiable sub-models during tableaux expansions but this has not been implemented in the current version of the Grail classifier.

6.5.1 Interactions

The behaviour of the optimisation techniques is not orthogonal; in particular, the normalisation and encoding of concept expressions interacts with both semantic branching and caching.

The semantic branching optimisation uses a heuristic function to select the concept on which to branch with the objective of maximising pruning by selecting the most constraining concept; in the current implementation this is based on frequency of occurrence. Efficiency is an essential characteristic of the heuristic function if its effectiveness is to exceed its cost and the required efficiency would be difficult to achieve without normalisation and encoding due to the cost of repeatedly comparing arbitrarily complex concept expressions. Normalisation and encoding effectively performs all such comparisons once, replacing concept expressions with concept names and leaving the heuristic function only needing to compare the names. In the current implementation neither the heuristic function nor the subsequent branching procedure are able to deal with non-atomic concept expressions.

The caching optimisation interacts with normalisation and encoding in a similar way. When testing if two cached models can be merged the algorithm must check for interactions between value restriction constraints ($x_0 : \forall R.C$) which occur only in one model and relation constraints ($x_0 R x_n$) which occur in the other model. This means comparing the concept expressions in value restrictions, a process which is made much more efficient by normalisation and encoding so that it is only necessary to compare concept names.

7 Preliminary Results

The new classifier has been tested in a number of ways:

- by classifying the GALEN core knowledge base (the high level ontology which contains 1,794 distinct concepts, 207 roles and 277 GCIs) with various combinations of optimisations enabled and disabled;

- by comparing its performance with that of the KRIS DL [6] using a modified version of the full GALEN knowledge base (the modified knowledge base contains 413 roles and 3,917 concepts);

- by comparing its performance with the KSAT algorithm [20].

The comparisons with KRIS and KSAT also provided useful correctness tests: the concept hierarchies computed by KRIS and GRAIL were identical, as were the results of 44,000 satisfiability tests performed by both KSAT and GRAIL.

The current version of the classifier is written in Lisp and no attempt has been made to tune or optimise the Lisp code beyond that performed by the Lisp compiler with 'optimization' settings speed=3 and safety=0. All tests have been performed using Allegro Lisp on a Sun SPARCstation 20/61 equipped with a 60MHz superSPARC processor, a 1Mbyte off-chip cache and 128Mbytes of RAM.

Testing work is still at an early stage but some interesting and encouraging results have already emerged:

- Without optimisation, attempting to classify the GALEN core knowledge base results in effective non-termination—single satisfiability tests have been run for more than $> 10^6 s$ without producing a result. With optimisation, the GALEN core terminology is classified in approximately 547s (an average of $0.3s$ per concept).

- KRIS takes over 1.4 times as long as GRAIL to classify (an $\mathcal{ALCF}$ version of) the GALEN knowledge base, performs over 3.6 times as many satisfiability tests and exhibits a higher rate of increase in the number of satisfiability tests required as the knowledge base size is increased.

- GRAIL out-performs KSAT when testing the satisfiability of single randomly generated $\mathcal{ALC}$ concept expressions, a test method devised and used by KSAT's developers [21]. For the hardest problems generated, KSAT's median solution time is more than 3 times greater than GRAIL's and its mean solution time is more than 10 times greater; KSAT also takes almost 25 times longer to solve the complete problem set.

  With this kind of problem GRAIL also exhibits easy-hard-easy behavior as constrainedness is varied to give problems with probabilities of satisfiability in the range 1–0, a behavior which has been shown to be characteristic of many (NP-Complete) search problems [25].

7.1 Classifying the GALEN Knowledge Base

The performance of the new GRAIL classifier was first tested with respect to classification of the GALEN core knowledge base. In Figure 4 on the current page the classification time per concept is plotted against knowledge base size. Each data point shows the mean classification time per concept for a set of 100 concepts. The complete classification process required 21,610 subsumption tests of which only 8,376 resulted in tableaux satisfiability tests—the remaining tests were avoided, largely by the caching optimisation which detected obvious subsumptions and non-subsumptions.

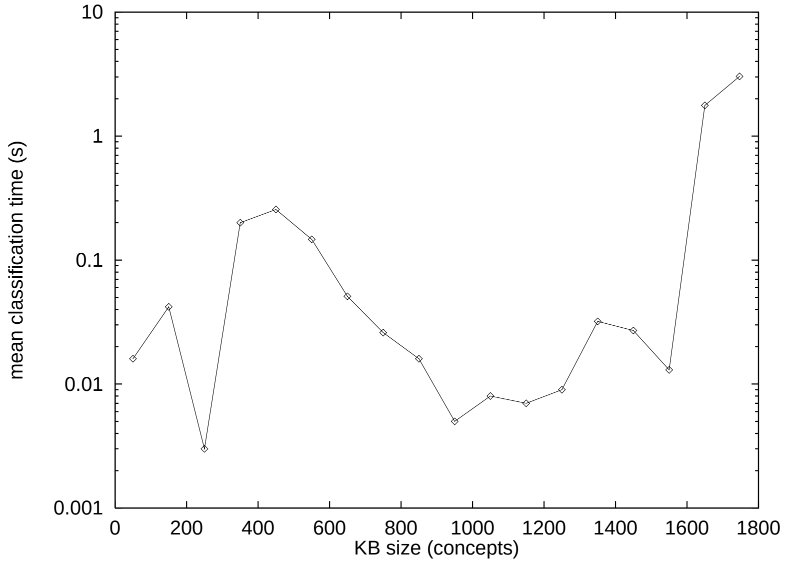

Figure 4: mean classification time

In order to measure the effectiveness of some of the optimisation techniques, the above test was repeated several times with one of the optimisations disabled in each case.

7.1.1 Normalisation and Encoding

To measure the effectiveness of normalisation and encoding, the GALEN core knowledge base classification test was repeated with the optimisation partially disabled—it is not possible to fully disable encoding as the semantic branching technique cannot deal with non-atomic concept expressions (see Section 6.5.1 on page 22). However in order to measure the direct benefit of normalisation and encoding—the detection of lexically obvious unsatisfiability—the matching process was disabled so that every expression and sub-expression was uniquely encoded.

In Figure 5 on the next page the classification time per concept, both with and without normalisation and encoding, is plotted against knowledge base size Although total classification time increased from 547s to 1,122s this was largely due to interaction with the caching optimisation (see Section 6.5.1 on page 22) which resulted in the number of satisfiability tests performed increasing from 8,376 to 13,949.

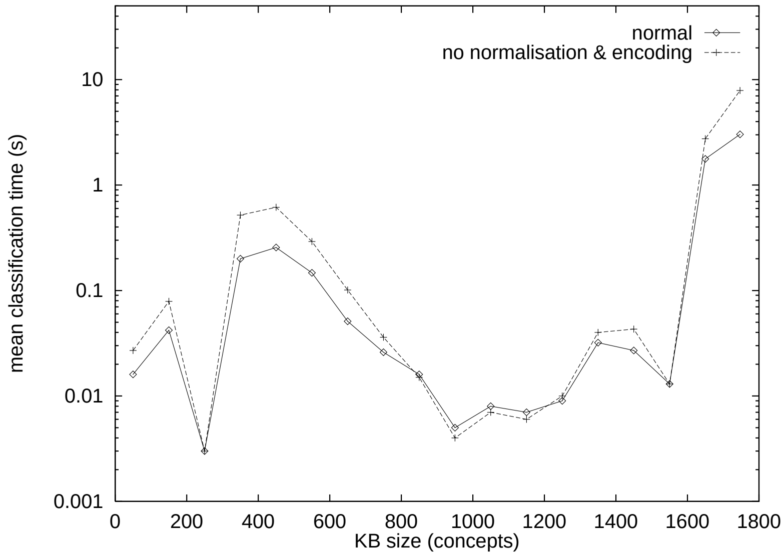

Figure 5: mean classification time with & without normalisation & encoding

7.1.2 Caching

The Galen core knowledge base classification test was repeated a second time with the caching optimisation disabled. In Figure 6 on the next page the mean classification time per concept, both with and without caching, is plotted against knowledge base size. With caching disabled disabled total classification time increased to 2,573s, a factor of more than 4.7, while the number of satisfiability tests performed increased to 22,961, a factor of more than 2.7.

7.2 Comparing GRAIL and KRIS

In order to compare the performance of Grail and Kris the Galen knowledge base was converted to an $\mathcal{ALCF}$ knowledge base [26] by:

1. Discarding all role inclusion axioms.
2. Converting all GCIs $A \sqsubseteq B$ into concept definitions $\mathsf{CN} \doteq A \sqcap B$ where CN is a unique system generated name.
3. Eliminating semantically equivalent concepts as these were not handled correctly by Kris.

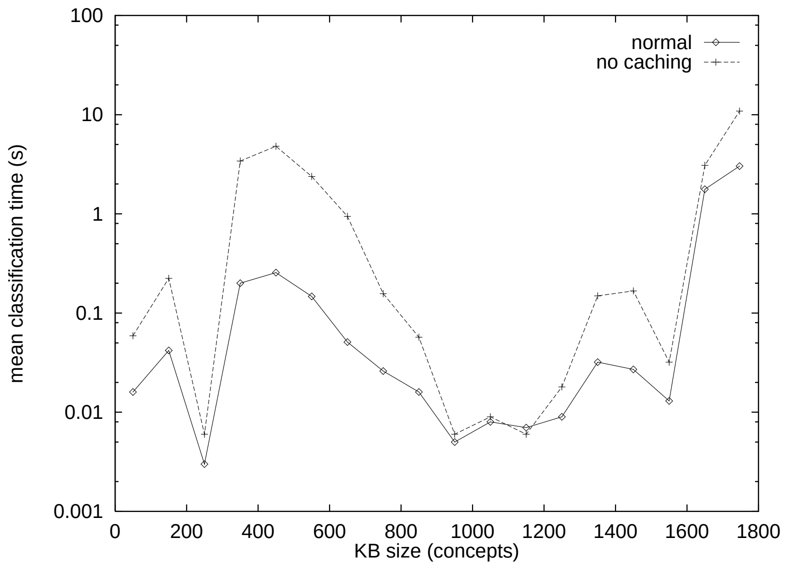

Figure 6: mean classification time with & without caching

The method of converting GCIs was chosen to produce large numbers of realistic non-primitive concepts—if the knowledge base includes the assertion that A implies B it is reasonable to assume that concepts of type $A \sqcap B$ are realistic. Non-primitive concepts were prefered as their classification on the basis of subsumption reasoning is a key characteristic of DLs.

The performance of GRAIL and KRIS was compared as the size of the knowledge base was increased from 2,719 concepts to 3,917 concepts by adding the non-primitive converted GCI concepts. In Figure 7 on the following page the mean classification time per concept is plotted against knowledge base size while in Figure 8 on the next page the mean numbers of subsumption and satisfiability tests performed per classification is also plotted against knowledge base size. Classifying all 3,917 concepts took GRAIL 670s while it performed 149,150 subsumption tests and 30,717 satisfiability tests. The same process took KRIS 955s, 256,555 subsumption tests and 113,296 satisfiability tests.

7.3 Comparing GRAIL and KSAT

As well as classifying the GALEN terminology, GRAIL was also compared with KSAT, a decision procedure for the propositional modal logic $\mathbf{K_{(m)}}$ which has been shown to be equivalent to $\mathcal{ALC}$ [43, 20]. The GRAIL and KSAT decision procedures were compared using randomly generated $N\text{-}CNF_{K(m)}$ concepts, a test method devised by KSAT's developers [21]. An $N\text{-}CNF_{K(m)}$ concept is defined as follows:

- $N\text{-}CNF_{K(m)}$ concept = conjunction of clauses
- clause = disjunction of literals

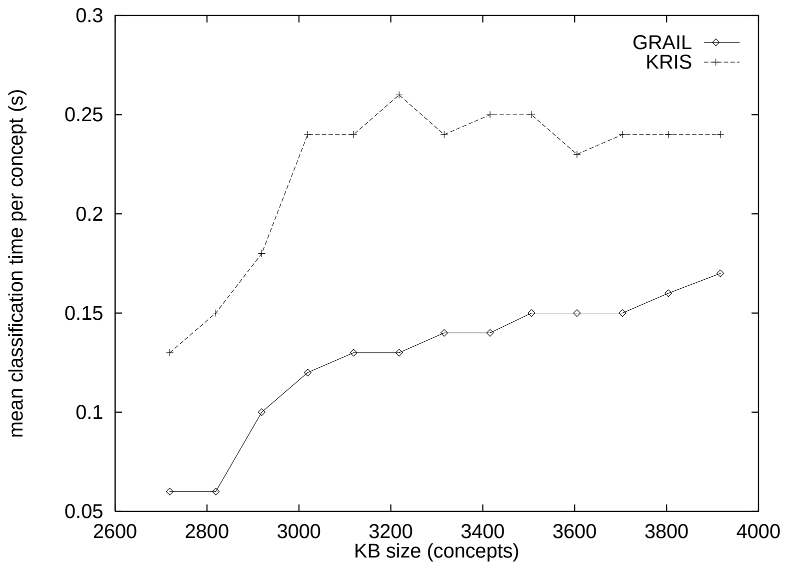

Figure 7: GRAIL -v- KRIS — classification time

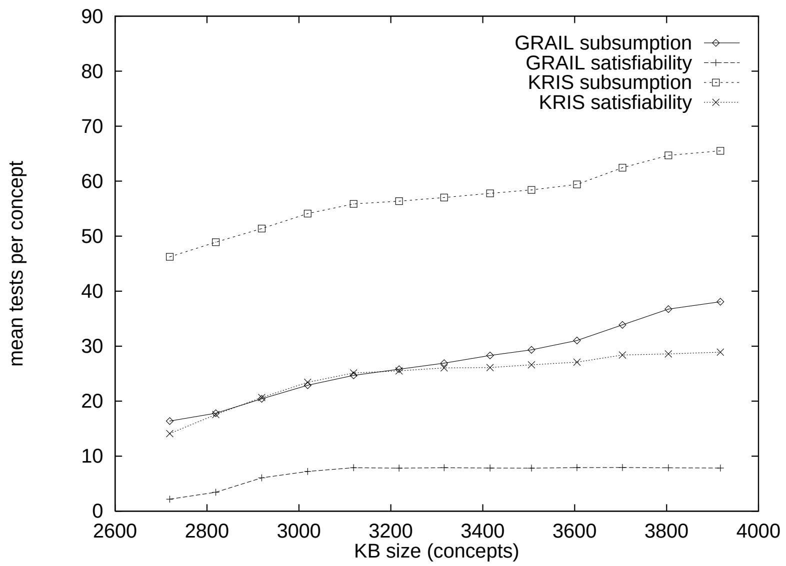

Figure 8: GRAIL -v- KRIS — subsumption and satisfiability tests

- literal = atom *or* ¬atom
- atom = primitive concept *or* $\forall R$.clause

Generation is controlled by parameters L, N, m, p and d where:

L	$=$	number of clauses
N	$=$	number of primitive concepts
m	$=$	number of different roles (modalities)
p	$=$	probability of atom being primitive
d	$=$	maximum role chain (modal depth)

For fixed N, m, p and d, concepts with a probability of satisfiability in the range 1–0 can be obtained by varying L, the number of clauses in the top level conjunction. The tests performed here follow the basic method from [21], using $N = 3$, $m = 1$, $p = 0.5$, $d = 3$ and L in the range 3–123.

The performance of KSAT and the new GRAIL satisfiability algorithm is compared in Figure 9 on this page. Each data-point on the graph shows the median time taken by KSAT and GRAIL to perform satisfiability tests on 1,000 randomly generated concepts. The probability of a concept being satisfiable (in the range 1–0) for each value of L is also shown for reference purposes.

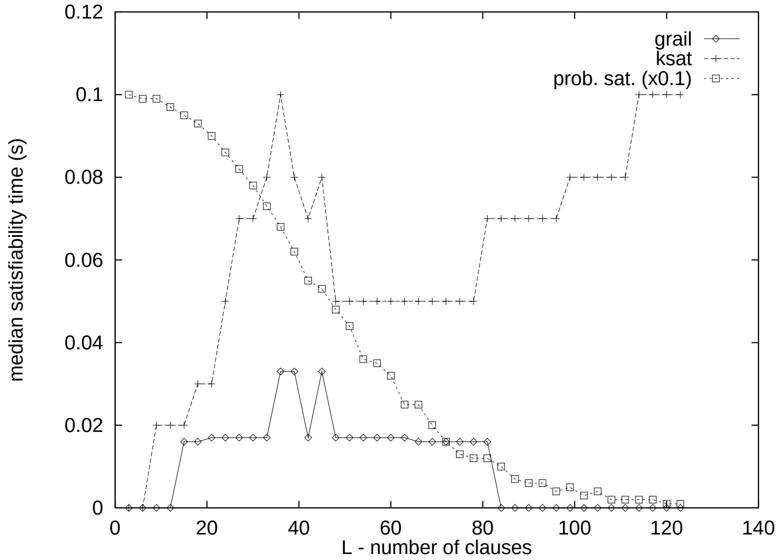

Figure 9: GRAIL -v- KSAT — median satisfiability times

In order to measure the effectiveness of some of the optimisation techniques with this kind of data the test was repeated several times with one of the GRAIL optimisations disabled in each case.

7.3.1 Normalisation and Encoding

Figure 10 on the next page shows the result of partially disabling GRAIL's normalisation and encoding as described in section 7.1.1. In this experiment only 100 satisfiability tests per data point were performed due to the length of time required to perform some of these tests.

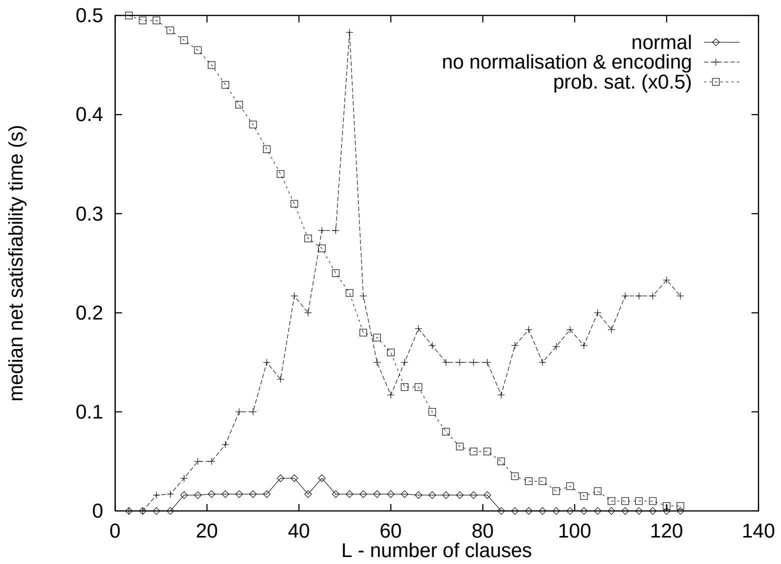

Figure 10: GRAIL with & without normalisation & encoding

7.3.2 Dependency Directed Backtracking

Figure 11 on the following page shows the result of disabling GRAIL's dependency directed backtracking. In this case mean satisfiability time has been plotted to show more clearly the effect of dependency directed backtracking on the small number of very hard problems which occur in the generally easy region where L is in the range 90–123. When dependency directed backtracking was disabled the test had to be curtailed for $L > 90$ due to the time taken to test some of the concepts in this region.

8 Conclusion

The results show that optimisation can dramatically improve the performance of a tableaux subsumption testing procedure for an expressive description logic: when applied to the new GRAIL classifier the techniques described in this report reduced classification time for the GALEN knowledge base by at least several orders of magnitude. The preliminary results have already lead to some interesting observations regarding the various techniques:

- Dependency directed backtracking and the indexing of GCIs are particularly important techniques: it proved impossible to classify the GALEN knowledge base without these optimisations.
- Techniques which work well with random data may not work as well with a real knowledge base: normalisation and encoding was highly effective with randomly

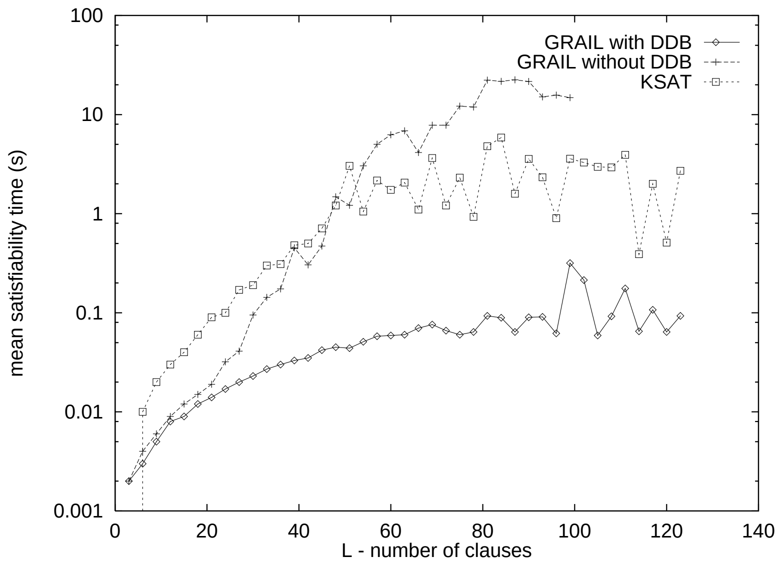

Figure 11: GRAIL with & without dependency directed backtracking

generated *3-CNF*$_{K(m)}$ concepts but was of little direct benefit when classifying the GALEN knowledge base. Conversely, techniques which work well with real knowledge bases may appear to be ineffective if tested with randomly generated data.

- Caching is a valuable technique which reduced by a factor of approximately 5 the number of satisfiability tests required to classify each concept. Moreover the number of tests per concept stabilised at less than 8 for knowledge bases larger than 3,120 concepts.

- The optimised satisfiability test displayed good 'pay as you go' characteristics: when tested with less expressive logics its performance compares favourably with that of more specific algorithms.

It may be that even a highly optimised classifier will be unable to provide acceptable performance as the size of the GALEN knowledge base increases, particular if many more GCIs are added. However, even if this is the case, the work described in this report will still be of value, since:

- Many of the techniques studied would also be useful with less expressive DLs and should become standard in tableaux satisfiability algorithms.

- A complete procedure could be used for 'background' processing after a quick answer has been provided by an incomplete procedure and to provide a benchmark against which the performance of an incomplete procedure could be judged [14]. In this context the criteria for acceptable performance are less stringent but still extant—the unoptimised GRAIL procedure would be of little value as single subsumption tests frequently take in excess of 10^6s.

- A complete procedure is a sensible starting point from which to retreat gracefully into limited and clearly characterised incompleteness.

The highly encouraging preliminary results justify further work on optimisation techniques. Future work will include refining, extending and adding to the described techniques as well as more extensive testing, preferably using large knowledge bases from a range of real applications.

References

[1] 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.

[2] 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.

[3] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *Proceedings of IJCAI'91*, pages 446–451, 1991.

[4] F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 1996. To appear.

[5] 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.

[6] F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3):8–14, 1991.

[7] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In *Processing declarative knowledge: Intermational workshop PDK'91*, number 567 in Lecture Notes in Artificial Intelligence, pages 67–86, Berlin, 1991. Springer-Verlag.

[8] 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.

[9] F. Baader and U. Sattler. Description logics with symbolic number restrictions. In Wolfgang Wahlster, editor, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96)*, pages 283–287. John Wiley & Sons Ltd., 1996.

[10] R. Baud, C. Lovis, L. Alpay, A.-M. Rassinoux, J.-R. Scherrer, A. W. Nowlan, and A. L. Rector. Modelling for natural language understanding. In Charles Safran, editor, *Seventeenth Annual Symposium on Computer Applications in Medical Care (SCAMC-93)*, pages 289–93, Washington DC, 1993. McGraw Hill.

[11] 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*, Saarbrücken, Germany, September 1994.

[12] 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*, 1995.

[13] P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive description logics: a preliminary report. In 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*, pages 28–39, 1995.

[14] M. Buchheit, F. M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[15] 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.

[16] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[17] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.

[18] G. De Giacomo and M. Lenzerini. Tbox and abox reasoning in expressive description logics. In Aiello et al. [1], pages 316–327.

[19] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[20] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures—the case study of modal K. In 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.

[21] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for $\mathcal{ALC}$. In Aiello et al. [1], pages 304–314.

[22] C. Goble, N. Paton, and S. Bechhofer, 1996. http://www.cs.man.ac.uk/mig/tambis/.

[23] 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}*, San Jose, California, USA, February 1996.

[24] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich. An empirical analysis of terminological representation systems. *Artificial Intelligence*, 68:367–397, 1994.

[25] T. Hogg, B. A. Huberman, and C. P. Williams. Phase transitions and the search problem. *Artificial Intelligence*, 81:1–15, 1996. Editorial.

[26] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Research Report RR-90-04, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), April 1990.

[27] I. Horrocks. A comparison of two terminological knowledge representation systems. Master's thesis, University of Manchester, 1995.

[28] J. Kirby and A. L. Rector. The PEN&PAD data entry system: From prototype to practical system. In *AMIA Fall Symposium*, 1996. to appear.

[29] D. B. Lenat and R. V. Guha. The evolution of cycl, the cyc representation language. *SIGART Bulletin*, 2(3):84–87, 1991.

[30] Alon Y. Levy and Marie-Christine Rousset. Using description logics to model and reason about views. In 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*, pages 48–49, 1996.

[31] R. M. MacGregor. Inside the LOOM description classifier. *SIGART Bulletin*, 2(3):88–92, 1991.

[32] E. Mays, R. Dionne, and R. Weida. K-rep system overview. *SIGART Bulletin*, 2(3):93–97, 1991.

[33] B. Nebel. Terminological cycles: Semantics and computational properties. In Sowa [46], chapter 11, pages 331–361.

[34] W. A. Nowlan. *Structured methods of information management for medical records.* PhD thesis, University of Manchester, 1993.

[35] W. A. Nowlan and A. L. Rector. Medical knowledge representation and predictive data entry. In M. Stefanelle, A. Hasman, M. Fieschi, and J. Talmon, editors, *Proceedings of AIME 91*, number 44 in Lecture notes in Medical Informatics, pages 105–116. Springer-Verlag, 1991.

[36] P. F. Patel-Schneider. The CLASSIC knowledge representation system: Guiding principals and implementation rationale. *SIGART Bulletin*, 2(3):108–113, 1991.

[37] A.-M. Rasinoux, C. Juge, P.-A. Michel, R. H. Baud, D. Lemaitre, F.-C. Jean, P. Degoulet, and J.-R. Scherrer. Analysis of medical jargon: The RECIT system. In P. Barahona, M. Stefanelli, J. Wyatt, and J. Sickmann, editors, *Fifth conference on Artificial Intelligence in Medicine Europe (AIME '95)*, number 934 in Lecture Notes in Artificial Intelligence, pages 42–42. Springer, 1995.

[38] A. Rector, S. Bechhofer, C. A. Goble, I. Horrocks, W. A. Nowlan, and W. D. Solomon. The GALEN concept modelling language for medical terminology. *AI in Medicine*, 1996. To appear.

[39] A. Rector and I. Horrocks. Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *Workshop on Ontological Engineering, AAAI Spring Symposium, Stanford, CA*. AAAI Press, Menlo Park, California, 1997. To appear.

[40] A. L. Rector, P. Zanstra, D. Solomon, and The GALEN Consortium. GALEN: Terminology services for clinical information systems. In M. F. Laires, M. J. Ladeira, and J. P. Christensen, editors, *Health in the New Communications Age*, pages 90–100. IOS Press, Amsterdam, 1995.

[41] 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.

[42] 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. Springer Verlag, 1996.

[43] K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of IJCAI'91*, pages 466–471, 1991.

[44] K. Schild. Terminological cycles and the propositional μ-calculus. Technical Report RR-93-18, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), April 1993.

[45] M. Schmidt-Schaußand G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[46] J. F. Sowa, editor. *Principals of Semantic Networks: Explorations in the representation of knowledge*. Morgan-Kaufmann, 1991.

[47] A. Tarski. *Logic, Semantics, Mathemetics: Papers from 1923 to 1938.* Oxford University Press, 1956.

[48] J. C. Wagner, A.-M. Rassinoux, R. H. Baud, and J.-R. Scherrer. Generating noun phrases from a medical knowledge representation. *Twelfth International Congress of the European Federation for Medical Informatics, MIE-94*, pages 218–223, 1994.

[49] W. A. Woods. Understanding subsumption and taxonomy: a framework for progress. In Sowa [46], chapter 1, pages 45–94.

[50] 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.