Enhancing a hybrid tableaux algorithm

A Dissertation submitted to
Faculty of Computer Science,
Free University of Bozen-Bolzano
in fullfillment to the Degree of

Masters in Computer Science
as part of the
European Masters Program in Language
and Communication Technologies

Submitted by:
Alejandra Lorenzo

Supervisors:
Diego Calvanese
Patrick Blackburn

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Fakultät für Informatik | Facoltà di Scienze e Tecnologie informatiche | Faculty of Computer Science

October 2009

Contents

1 Logical background **9**
- 1.1 Two flavors: Hybrid and Description Logics . . . 9
- 1.2 Introducing Hybrid Logics . . . 9
  - 1.2.1 Hybrid Details: . . . 10
  - 1.2.2 Translations: . . . 11
- 1.3 Introducing Description Logics . . . 12
  - 1.3.1 Description Logic details: . . . 12
  - 1.3.2 Reasoning Tasks: . . . 13

2 Tableaux Algorithm **17**
- 2.1 A little bit of tableau history . . . 17
- 2.2 A prefixed tableau . . . 18
  - 2.2.1 Prefixed Tableau Rules . . . 19
  - 2.2.2 A procedural view of the Prefixed Tableau Algorithm . . . 20
- 2.3 HTab: A terminating tableau System for Hybrid Logic . . . 21
  - 2.3.1 About HTab's implementation . . . 22
  - 2.3.2 Optimizations . . . 22

3 Caching **27**
- 3.1 What to "cache" . . . 27
  - 3.1.1 Caching Unsatisfiable Concepts: . . . 27
  - 3.1.2 Caching Satisfiable Concepts: . . . 28
- 3.2 Existing approaches to Caching . . . 28
- 3.3 Some implementations of Caching . . . 30

4 UNSAT Caching **31**
- 4.1 What to cache for H(@,A) . . . 31
- 4.2 Definitions and algorithms . . . 32
- 4.3 Integrating UNSAT Caching in the Tableaux Algorithm . . . 37
- 4.4 Some examples . . . 38

5 Implementing the UNSAT Cache **41**
- 5.1 About the subterms representation . . . 41
- 5.2 Bit matrices approach . . . 42
  - 5.2.1 Lists vector approach . . . 43
  - 5.2.2 Tree based approach . . . 44
- 5.3 Enabling Backjumping with UNSAT Caching . . . 45
- 5.4 Source Code . . . 46

6 Other Approaches to Caching **47**
6.1 MIXED Caching . . . 47
6.1.1 So, what else could be done? . . . 48
6.2 GLOBAL Caching . . . 48
6.2.1 How does it work for Description Logics? . . . 48
6.2.2 How would it work for Hybrid Logics? . . . 50
6.2.3 Conclusion and direction for further work . . . 52

7 Evaluation **55**
7.1 The chosen testing framework: GridTest . . . 55
7.2 Evaluations . . . 56
7.2.1 First Part: General Evaluation . . . 56
7.2.2 Second Part: Cache System Evaluation . . . 59
7.2.3 An example . . . 60

8 Conclusion **63**
8.1 What we have learned so far: . . . 64
8.2 What is left for future work: . . . 64

A Source Code Summary **65**
A.1 Branch.hs module . . . 65
A.2 UnsatCache.hs module . . . 66
A.2.1 Update Functions . . . 66
A.2.2 Search Functions . . . 68
A.2.3 Helper functions used for both the UNSAT cache update and search . . . 70
A.2.4 Functions related to the maintenance of the Mapping Structure . . . 71

Acknowledgements

"Hay una gran diferencia entre ser instruido y ser sabio.
La instrucción nos la da la escuela, la universidad.
La sabiduría nos la da la edad, nos la da la vida ...".

Ramiro Garza

I couldn't have made this far without the help of many people. To all of them I would like to thank.

First of all, thanks mum and dad for all your love, understanding and unconditional support. Thanks for being there when I needed you the most and for encouraging me to go on when I felt like giving up. Thanks for teaching me that everything can be learned and that, in the end, every effort gets a reward. Thanks Sole, Silvi y Ariel for sharing with me a wonderful childhood, for being always my friends and for my lovely nephews and nieces.

Thanks Lautaro, mi "Enano" for giving me the strength and enthusiasm to go on. Thanks sun for your support, for being always with me, whenever I was, and fill my life with happiness. I love you so much and I couldn't have done anything I did without you. Thanks Humberto, for your love, for being with us and making our every day so special. Thanks for your patience and advises, and especially for this new family that makes me so happy and that gives me the strength to carry on.

To all my friends, those who are here with me and those who are far away but that are with me in thoughts. Thanks "Lu" for your support, for the time shared and for your friendship.

I would like to especially thank my supervisors Diego Calvanese and Patrick Blackburn for their expert guidance, advises and trust. Thanks to both of you for introducing me in this fascinating field of logics. Finally, this work would not have been done without the constant help of Guillaume Hoffmann. For that, many thanks Guillaume for your generous help and numerous advises during all this thesis. It was really nice to work with you.

Alejandra Lorenzo.

Description of the project

Hybrid Logic is a formalism used to represent knowledge and enable inference on it. The term covers a number of logical systems living between modal and classical logic. As it offers a good trade-off between complexity and expressivity, its theory has been investigated for more than fifteen years. However theorem provers (and model builders) for this logic have only been developed relatively recently.

One of the most recent theorem provers for Hybrid Logic is HTab ([Hoffmann and Areces, 2003]). HTab is one of the few theorem provers (and model builders) for Hybrid Logic that support not only basic Hybrid Logic but also the universal and difference modalities and the down-arrow binder. It was developed by the TALARIS team (INRIA), and is based on a prefixed tableaux method adapted from the terminating tableaux introduced by T. Bolander and P. Blackburn in [Bolander and Blackburn., 2007].

HTab's performance has been compared with a number of other theorems provers and the results are good. But although a number of optimizations have already been incorporated into HTab, a potentially interesting one has not, namely caching. Caching can be defined as the storing of intermediate results in order to avoid re-computing them again.

Caching has been applied to a family of logics closely related to Hybrid Logic, namely the Description Logic family. So it is natural to ask: will caching help in the case of Hybrid Logic, including Hybrid Logic with the global modality? The aim of this thesis is to find out whether this optimization will be useful in this setting or not. We will restrict our attention to basic Hybrid Logic and the logic enriched with the universal modality, as Hybrid Logics containing the down-arrow binder are known to be undecidable.

This thesis is organized into eight chapters. In the first chapter, we briefly introduce the two families of languages: Hybrid Logic and Description Logic, and provide the main definitions needed for the rest of the report. The aim of this section is to provide the theoretical background needed in order to understand the similarities and differences between these two families.

In Chapter 2 we present the tableau algorithm. After providing a brief history of tableaux in different logics, we introduce the specific algorithm covered in this thesis: the *prefixed tableau* calculus for Hybrid Logic. Finally we present HTab, an implementation of this calculus for Hybrid Logic.

Chapter 3 provides a general introduction to the caching optimization, and discusses the approaches for caching already existing for the case of Description Logics.

Chapters 4 to 6 are the core of this work, as they explain in more detail each of the caching approaches, and how they can — and cannot — be implemented in Hybrid Logic. Chapter 4 introduces the main ideas needed to carry out UNSAT caching for the Hybrid Logic $\mathcal{H}(@, \mathsf{A})$, and Chapter 5 discusses its implementation. In Chapter 6 we discuss two other methods: MIXED caching and Global caching.

We show that MIXED caching cannot be used with Hybrid Logic, and discuss what GLOBAL caching for Hybrid Logic might look like.

In Chapter 7 we provide some testing examples, and some early results on the UNSAT caching implementation. This testing stage is extremely important in order to know effectively if the improvement was significant, or even successful. The testing was carried out using an automated testing environment developed at INRIA called GridTest.

Finally, Chapter 8 contains our conclusions: what we have learned during this work and what we would like to do next.

Chapter 1

Logical background

What is logic and why is it important? Intuitively, we can see logic as a tool that helps us to reason things out and to act rationally; thinking clearly is important to everyone in their everyday lives. But from a mathematical perspective,"logics" are formal languages for representing information in such a way that conclusions can be drawn. And there are many different logical formalisms: they offer different expressiveness (that is, they differ in what kinds of information they can represent) and also differ in the efficiency of their reasoning procedures.

In this section we introduce two families of logics which offer a good compromise between expressiveness and efficiency, and which are closely related to one another: Hybrid Logic and Description Logic ([Areces., 2003]).

1.1 Two flavors: Hybrid and Description Logics

Hybrid Logics were first investigated in the work of Arthur Prior, a philosophical logician who invented and used them in his investigations of the logic of time and tense in the mid 1960s. The earliest published reference is [Prior, 1967, Chapter 5 and Appendix B3]. Hybrid Logics have been intensively investigated, and these investigations for the most part have been theoretical: the main results concern expressive power (model theory), axiomatizations, completeness and complexity.

The history of Description Logics (or terminological languages, as they were initially called) started with the development of the KL-ONE system of Brachman and Schmolze [1985]. Because of their applicability to problems as varied as deductive databases, image retrieval, system modeling and information classification, they flourished rapidly. Applications and effective inference algorithms, together with complexity results, are the main results in the field. In particular, many existing tableau algorithms for Description Logics already implement various types of caching optimization, and most of the work of this thesis is based on adapting these implementations (when possible, which it isn't always) to the case of Hybrid Logic.

1.2 Introducing Hybrid Logics

"Hybrid Logic" is a loose term covering a number of logical systems living between modal and classical logic.

The simplest hybrid languages use formulas to refer to specific points in a model. To build a simple hybrid language, take an ordinary language of propositional modal

logic (built over some collection of propositional variables p, q, r and so on), and add a second type of atomic formula. These new atoms are called nominals, and are typically written i, j and k. Both types of atoms can be freely combined to form more complex formulas in the usual way; for example

$$\Diamond(i \wedge p) \wedge \Diamond(i \wedge q) \rightarrow \Diamond(p \wedge q)$$

is a well formed formula.

Now for the key idea: we insist that each nominal must be true at exactly one point in any model. In this way a nominal names a point by being true there and nowhere else. This simple idea gives rise to richer logics. Once we have nominals, we can think of other interesting ideas: why not introduce an operator that allows us to jump to the point named by a nominal? This is what the $@_i\varphi$ (read "at i, φ") formula does: it moves the point of evaluation to the point named by i and checks whether φ is true there. $@_i\varphi$ is Prior's $T(i, \varphi)$ construct ("φ is true at time i"), which he used to define his "third grade tense logics" [Prior, 1967]. It is also related to the $\mathsf{Holds}(i, \varphi)$ operator introduced in [Allen, 1984] for temporal representation in AI.

1.2.1 Hybrid Details:

The basic hybrid language is $\mathcal{H}$, which corresponds to the basic modal logic extended with nominals. Further extensions will be named by listing the added operators. The most expressive system we will discuss in this thesis is $\mathcal{H}(@, \mathsf{A})$, the basic hybrid system extended with the satisfaction operator @ and the universal modality A.
The following two definitions give the syntax and semantics of $\mathcal{H}(@, \mathsf{A})$.

Definition 1 *Suppose we are given the countably infinite sets of symbols* $\mathsf{REL} = \{R_1, R_2, \ldots\}$ *(the* relational symbols*),* $\mathsf{PROP} = \{p_1, p_2, \ldots\}$ *(the* propositional variables*) and* $\mathsf{NOM} = \{i_1, i_2, \ldots\}$ *(the* nominals*). Then the well-formed formulas of the hybrid language* $\mathcal{H}(@, \mathsf{A})$ *in the signature* $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM} \rangle$ *are given by the following recursive definition:*

$$\mathsf{FORMS} := \top \mid p \mid i \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle R \rangle \varphi \mid @_i\varphi \mid \mathsf{A}\varphi$$

where $p \in \mathsf{PROP}$*,* $i \in \mathsf{NOM}$*,* $R \in \mathsf{REL}$ *and* $\varphi, \varphi_1, \varphi_2 \in \mathsf{FORMS}$*. For* $T \subseteq \mathsf{FORMS}$*,* $\mathsf{PROP}(T)$ *and* $\mathsf{NOM}(T)$ *denote, respectively, the set of propositional variables and nominals which occur in formulas in* T*.*

Now for the semantics, in the rest of the section we assume fixed a signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{NOM} \rangle$.

Definition 2 *A hybrid* model $\mathcal{M}$ *is a triple* $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ *such that* M *is a non-empty set,* $\{R_i\}$ *is a set of binary relations on* M*, and* $V : \mathsf{PROP} \cup \mathsf{NOM} \rightarrow Pow(M)$ *is such that for all nominals* $i \in \mathsf{NOM}$*,* $V(i)$ *is a singleton subset of* M*. We usually call the elements of* M states *or* worlds*,* R_i *the* accessibility relations*, and* V *the* valuation*.*

Let $\mathcal{M} = \langle M, \{R_i\}, V \rangle$ *be a model,* $m \in M$*. Then the* satisfiability relation *is defined as follows*

$$
\begin{array}{rll}
\mathcal{M}, m \models \top & & \textit{always} \\
\mathcal{M}, m \models p & \textit{iff} & m \in V(p),\ P \in \mathsf{PROP} \\
\mathcal{M}, m \models i & \textit{iff} & m \in V[i],\ i \in \mathsf{NOM} \\
\mathcal{M}, m \models \neg\varphi & \textit{iff} & \mathcal{M}, m \not\models \varphi \\
\mathcal{M}, m \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{M}, m \models \varphi_1 \textit{ and } \mathcal{M}, m \models \varphi_2 \\
\mathcal{M}, m \models \langle R\rangle\varphi & \textit{iff} & \exists m'.(R(m, m') \ \&\ \mathcal{M}, m' \models \varphi) \\
\mathcal{M}, m \models \mathsf{A}\varphi & \textit{iff} & \forall m'.(\mathcal{M}, m' \models \varphi) \\
\mathcal{M}, m \models @_i\varphi & \textit{iff} & \mathcal{M}, m' \models \varphi, \textit{ where } V(i) = \{m'\},\ i \in \mathsf{NOM}
\end{array}
$$

A formula φ is satisfiable *if there is a model $\mathcal{M}$ and a world $m \in M$ such that $\mathcal{M}, m \models \varphi$. A formula φ is* valid *if for all models $\mathcal{M}$, $\mathcal{M} \models \varphi$.*

1.2.2 Translations:

We said earlier that "Hybrid Logic" is a loose term covering a number of logical systems living between modal and classical logic. This is not a metaphor, it is a simple mathematical fact: it is straightforward to translate Hybrid Logic into first-order logic.

Following [Areces., 2003], let us describe the first-order correspondence language for Hybrid Logic. This is the first-order language with equality that contains a set UREL of unary predicates P, one for each propositional variable $p \in \mathsf{NOM}$, and a binary predicate R for each modal operator $\langle R\rangle$.

Hence, the *hybrid* signature is of the form $\langle \{R_i\} \cup \mathsf{UREL}, \{\}, \mathsf{CONS}, \mathsf{VAR}\rangle$.

Any hybrid model $\mathcal{M} = \langle M, \{R_i\}, V\rangle$ can be seen as a first-order model over the hybrid signature, for the accessibility relations R_i can be used to interpret the binary predicates R_i, unary predicates can be interpreted by the subsets that V assigns to propositional variables, and constants can be interpreted by the worlds that nominals name. We let the context determine whether we are thinking of first-order or hybrid models, and continue to use the notation $\mathcal{M} = \langle M, \{R_i\}, V\rangle$.

The standard translation ST for $\mathcal{H}(@, \mathsf{A})$, is given in the following definition.

Definition 3 *The translation ST_x from the hybrid language $\mathcal{H}_S(@, \mathsf{A})$ over $\langle$*REL, PROP, NOM$\rangle$ *into first-order logic over the signature $\langle \mathsf{REL} \cup \{P_j \mid p_j \in \mathsf{PROP}\}, \{\}, \mathsf{NOM}, \{x, y\}\rangle$ is defined as follows*

$$
\begin{array}{rcl}
ST_x(i_j) & = & (x = i_j),\ i_j \in \mathsf{NOM} \\
ST_x(p_j) & = & P_j(x),\ p_j \in \mathsf{PROP} \\
ST_x(\neg\varphi) & = & \neg ST_x(\varphi) \\
ST_x(\varphi \wedge \psi) & = & ST_x(\varphi) \wedge ST_x(\psi) \\
ST_x(\langle R\rangle\varphi) & = & \exists y.(R(x, y) \wedge ST_y(\varphi)) \\
ST_x(\mathsf{A}\varphi) & = & \forall x.ST_x(\varphi) \\
ST_x(@_i\varphi) & = & (ST_x(\varphi))[x/i]
\end{array}
$$

where y is a new variable not yet used.

The translation given above is truth-preserving. To state this formally, one makes use of the observation that models and assignments for Hybrid Logic can be considered as models and assignments for first-order logic and vice versa.

Proposition 1 *Let φ be a hybrid formula, then for all hybrid models $\mathcal{M}$, $m \in M$ $\mathcal{M}, m \Vdash \varphi$ iff $\mathcal{M} \models ST_x(\varphi)[x \leftarrow m]$.*

Now, we introduce the *satisfiability* problem for Hybrid Logic as the task of showing that a given formula is *satisfiable*. That is, given an hybrid formula φ, find a model $\mathcal{M}$ and a world $m \in M$ such that $\mathcal{M}, m \models \varphi$. The complexity of the *satisfiability* problem for the Hybrid Logic $\mathcal{H}(@)$ is PSPACE-complete, while for $\mathcal{H}(@, \mathsf{A})$ is EXPTIME-complete.

1.3 Introducing Description Logics

Description Logics (DLs) are a family of formal languages with a clearly specified semantics, usually in terms of first-order models, together with specialized inference mechanisms to account for knowledge classification.

As this thesis is about Hybrid Logic, why discuss Description Logic. There are a number of reasons. One has already been mentioned: we are interested in caching mechanisms, and these have been extensively investigated by the Description Logic community, so there of relevance to this that we can learn from Description Logic.

But there are other reasons for looking at Description Logic. One is that the basic systems of Description Logic and Hybrid Logic really are very closely related (as will become clear below). In particular, if you take the logic $\mathcal{ALC}$ plus Nominals ($\mathcal{O}$) (this is defined below) it is very closely related to $\mathcal{H}(@, \mathsf{A})$. Thus knowing both approaches gives two perspectives on the same set of ideas.

But, as we shall see in the course of the thesis, although they are closely related, it is not always straightforward, or even possible to directly transfer methods from Description Logic to Hybrid Logic.

1.3.1 Description Logic details:

Most description languages model information as a pair $\langle T, A\rangle$, where T is a set of formulas concerning "terminological" information (the T-Box) and A is a set of formulas concerning "assertional" information (the A-Box).

Another way to look at this separation of information is from a database point of view: the T-Box is a general schema concerning the classes of individuals to be represented, their general properties and mutual relationships, while the A-Box is a partial instantiation of this schema, containing assertions relating either individuals to classes, or individuals to each other.

Definition 4 *Let* $\mathsf{CON} = \{C_1, C_2, \ldots\}$ *be a countable set of* atomic concepts*,* $\mathsf{ROL} = \{R_1, R_2, \ldots\}$ *be a countable set of* atomic roles *and* $\mathsf{IND} = \{a_1, a_2, \ldots\}$ *be a countable set of* individuals*.* $\mathcal{S} = \langle \mathsf{CON}, \mathsf{ROL}, \mathsf{IND}\rangle$ *is a* signature*. Once a signature* $\mathcal{S}$ *is fixed, an* interpretation $\mathcal{I}$ *for* $\mathcal{S}$ *is a tuple* $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\rangle$*, where*

- $\Delta^{\mathcal{I}}$ *is a non empty set.*
- $\cdot^{\mathcal{I}}$ *is a function assigning an element* $a_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ *to each constant* a_i*; a subset* $C_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ *to each atomic concept* C_i*; and a relation* $R_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ *to each atomic role* R_i*.*

In other words, a Description Logic interpretation is a model for a particular kind of first-order signature, where only unary and binary predicate symbols are allowed and the set of function symbols is empty.

The atomic symbols in a Description Logic signature can be combined by means of *concept and roles constructors*, to form complex concept and role expressions. Each Description Logic is characterized by the set of concept and roles constructors they allow. Figure 1.1 defines the roles and concepts constructors for some Description Logics, together with their semantics.

We will not discuss in detail all possible languages which can be obtained by combining constructors from Figure 1.1.

The language $\mathcal{AL}$ (Attributive language) is defined as the Description Logics allowing universal quantification, conjunction, unqualified existential quantifications of the form $\exists R.T$, and negation of atomic concept (negation of concepts that do not appear on the left hand side of axioms). $\mathcal{ALC}$ is $\mathcal{AL}$ extended with full negation.

Constructor	Syntax	Semantics
concept name	C	$C^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
disjunction ($\mathcal{U}$)	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
universal quant.	$\forall R.C$	$\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \rightarrow d_2 \in C^{\mathcal{I}})\}$
existential quant. ($\mathcal{E}$)	$\exists R.C$	$\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}})\}$
number restr. ($\mathcal{Q}$)	$(\geq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \text{ and } d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$
	$(\leq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \text{ and } d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$
one-of ($\mathcal{O}$)	$\{a_1, \ldots, a_n\}$	$\{d \mid d = a_i^{\mathcal{I}} \text{ for some } a_i\}$
role name	R	$R^{\mathcal{I}}$
inverse roles ($\mathcal{I}$)	R^{-1}	$\{(d_1, d_2) \mid R^{\mathcal{I}}(d_2, d_1)\}$

Figure 1.1: Common operators of Description Logics

We will be interested in languages having full Boolean expressivity and usually consider $\mathcal{ALC}$ and its extensions. In particular, we will be interested in the logic $\mathcal{ALC}$ plus nominals ($\mathcal{O}$).

Let's give some notation for the following definitions. Given a language $\mathcal{L}$, let $\mathsf{CON}(\mathcal{L})$ be the set of complex concept expressions and $\mathsf{ROL}(\mathcal{L})$ be the set of complex role expressions which can be formed by using the constructors of $\mathcal{L}$.

In Description Logics we want to perform inferences given certain background knowledge.

Definition 5 (Knowledge bases) *Fix a description language $\mathcal{L}$, a* knowledge base Σ *in $\mathcal{L}$ is a pair $\Sigma = \langle T, A\rangle$ such that*

- *T is the T(erminological)-Box, a finite, possibly empty, set of expressions of the form $C_1 \sqsubseteq C_2$ where C_1, C_2 are in $\mathsf{CON}(\mathcal{L})$. $C_1 \doteq C_2$ is notation for $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. Formulas in T are called* terminological axioms.
- *A is the A(ssertional)-Box, a finite, possibly empty, set of expressions of the forms $a\!:\!C$ or $(a, b)\!:\!R$ where C is in $\mathsf{CON}(\mathcal{L})$, R is in $\mathsf{ROL}(\mathcal{L})$ and a, b are individuals. Formulas in A are called* assertions.

It is time to define the appropriate notion of inference for Description Logics.

Definition 6 *Let $\mathcal{I}$ be an interpretation and φ a terminological axiom or an assertion. Then $\mathcal{I}$* models *φ (notation, $\mathcal{I} \models \varphi$) if*

- *$\varphi = C_1 \sqsubseteq C_2$ and $C_1^{\mathcal{I}} \subseteq {C_2}^{\mathcal{I}}$, or*
- *$\varphi = a\!:\!C$ and $a^{\mathcal{I}} \in C^{\mathcal{I}}$, or*
- *$\varphi = (a, b)\!:\!R$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.*

Let $\Sigma = \langle T, A\rangle$ be a knowledge base and $\mathcal{I}$ an interpretation, then $\mathcal{I}$ models Σ (notation, $\mathcal{I} \models \Sigma$) if for all $\varphi \in T \cup A, \mathcal{I} \models \varphi$. We say in this case that $\mathcal{I}$ is a model of the knowledge base Σ. *Given a knowledge base Σ and a terminological axiom or assertion φ, $\Sigma \models \varphi$ if for all models $\mathcal{I}$ of Σ we have $\mathcal{I} \models \varphi$.*

1.3.2 Reasoning Tasks:

In Description Logics the term *T-Box reasoning* is used for inferences from a knowledge base $\Sigma = \langle T, A\rangle$ where T is non-empty, and similarly, *A-Box reasoning* is inference for A non-empty. We can define a number of *reasoning tasks* or *reasoning*

services which can be provided by a knowledge representation system. The following are some of the standard reasoning tasks usually considered in Description Logics.

Definition 7 *Let* Σ *be a knowledge base,* $C_1, C_2 \in \mathsf{CON}(\mathcal{L})$*,* $R \in \mathsf{ROL}(\mathcal{L})$ *and* $a, b \in \mathsf{IND}$*, we define the following* reasoning tasks

- Subsumption, $\Sigma \models C_1 \sqsubseteq C_2$.
  Check whether for all interpretations $\mathcal{I}$ *such that* $\mathcal{I} \models \Sigma$ *we have* $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$.
- Instance Checking, $\Sigma \models a{:}C$.
  Check whether for all interpretations $\mathcal{I}$ *such that* $\mathcal{I} \models \Sigma$ *we have* $a^{\mathcal{I}} \in C^{\mathcal{I}}$.
- Concept Satisfiability, $\Sigma \not\models C \doteq \bot$.
  Check whether for some interpretation $\mathcal{I}$ *such that* $\mathcal{I} \models \Sigma$ *we have* $C^{\mathcal{I}} \neq \{\}$.

Research on Description Logics has focused mainly on understanding the relations between the reasoning tasks mentioned above, and on establishing their computational complexity. In this work, we are interested in the Concept Satisfiability checking reasoning task, particularly those involving tableau algorithms.

A full discussion of the relationship between Hybrid Logic and Description Logic, and in particular the relationship between $\mathcal{H}(@, \mathsf{A})$ and $\mathcal{ALC} + \mathcal{O}$ is not possible here; for a detailed account we refer the reader to the very clear account given in [Areces., 2003]. However it will be useful to point out the main points of contact, and the main points of difference, between these two formalisms.

First it should be clear that although Hybrid Logic talks of formulas ϕ, and Description Logic talks of concepts C, concepts and formulas correspond in a very direct way. Furthermore, it should also be clear that the boolean expressivity offered by both formalisms is the same and that $\langle R \rangle$ corresponds to $\exists R$ and $[R]$ corresponds to $\forall R$. These correspondences were first pointed out by Klaus Schild [Schild, 1991] in his well known paper that pointed out the link between the Description Logic $\mathcal{ALC}$ and the basic modal logic K.

Second, it should clear be that the nominals of Hybrid Logic play a similar role to used in the individuals in the A-Box and $\mathcal{O}$ (indeed, individuals are often called "nominals" in the description logic literature,). Furthermore, it should also be clear that A-Box statements of the form $a : C$ correspond to formulas of the form $@_i\varphi$. But here we also see a difference: in Description Logic, expressions of the form $a : C$ are only found in the A-Box; in Hybrid Logic they belong to the basic language.

What about A? Basically, a Description Logic subsumption relation of the form $C_1 \sqsubseteq C_2$ corresponds to a Hybrid Logic formula of the form $A(\phi \rightarrow \psi)$, as this asserts that the constraint $\phi \rightarrow \psi$ holds globally. The other way round, the hybrid formula $A(\phi \rightarrow \psi)$ can not be expressed in Description Logics. We can see this kind of formulas as "boolean T-Boxes" ([Areces *et al.*, 2003])

Summing up, $\mathcal{H}(@, \mathsf{A})$ and $\mathcal{ALC} + \mathcal{O}$ have a lot in common, and from the above description it should be clear that the two languages have similar expressivity. It should come as no surprise that the satisfiability problems for $\mathcal{H}(@, \mathsf{A})$ and the subsumption problem for $\mathcal{ALC} + \mathcal{O}$ (with non-empty T-Boxes) have identical computational complexity: both are EXPTIME complete.

But the differences are also clear. Hybrid Logic is "flat" in the sense that there is only one level of language: nominals are simply formulas that can be combined with any other formula, and A and @ are simply treated as additional modalities. Description Logic is more structured; it makes a distinction between the A-Box and the T-Box.

To conclude, there is a clear conceptual and theoretical link between hybrid and Description Logic. But the differences between the two formalisms can have unexpected consequences when it comes to implementing caching algorithms.

Chapter 2

Tableaux Algorithm

"*A tableaux algorithm is a formal proof procedure, existing in many varieties and for different logics, but always with certain characteristics.*" ([D'Agostino *et al.*, 1999]).

More specifically, we can see a tableau algorithm as a decision procedure that aims to determine the satisfiability of an input formula in a given logic. Intuitively, it tries to construct, for the input formula φ, a model of φ. It does so by "breaking the formula down" into its sub-formulas and deducing constraints on the model it is going to build. This "breaking down" is realized through "tableau expansion rules" (or just "tableau rules").

As we mentioned in the previous section, the work done in this thesis consist in adding the caching optimization to a tableau algorithm for satisfiability. In this section we present the basic *prefixed tableau* algorithm for the Hybrid Logic $\mathcal{H}(@, \mathsf{A})$ described in [Bolander and Blackburn., 2007], where a terminating decision procedure for Hybrid Logics up to $\mathcal{H}(@, \mathsf{A}, \Diamond^{-1})$ is introduced. Then we provide a procedural view of the algorithm. Finally we present HTab ([Hoffmann and Areces, 2003]), the tableau based reasoner for Hybrid Logics covered in this thesis, together with a brief explanation of the optimizations already present in HTab.

But first let us have a brief look at the history of tableau algorithms.

2.1 A little bit of tableau history

The history of tableau essentially begins with Gentzen (1935), for classical logics ([Gentzen, 1969]). He introduced the sequent calculus, the motivation for which was proof-theoretic. In 1955, Beth motivated by semantic concerns (that is, opposite to Gentzen's syntactical motivation), introduced the terminology "semantic tableau" as a method for constructing a counter-example. Quoted from Beth ([Beth, 1955]):

"*If such a counter-example is found, then we have a negative answer to our problem. And if it turns out that no suitable counter-example can be found, then we have an affirmative answer...*".

It is clear from this quotation that Beth's idea of the tableau method was to use it as a refutation procedure.

However, there is a more semantic way of thinking about tableau: as a search method for models meeting certain characteristics. This approach for tableau was first introduced by Smullyan in 1968, in the First-Order Logic book ([Smullyan, 1995]), through which the tableau method became more widely known.

There is a connection between these two views of the tableau method (i.e. tableau as a proof procedure and as a model search procedure): if we search for a

model in which a given formula φ is false (i.e. we search for a model of $\neg\varphi$), and we reach a closed tableau, then it means that there is no such model, and so φ is valid.

Smullayan's work was extended by Fitting (1972) to the case of modal logic ([Fitting, 1972]). Fitting also gave a tableau system using "prefixes" in which the idea was to designate possible worlds in such a way that syntactical rules determined accessibility. Prefixed tableau systems are the ones we use in this thesis, as such tableaus are also used in Hybrid Logic.

For Description Logics, the first tableau algorithm was proposed by Schmidt-SchauB and Smolka in 1991 ([Schmidt-Schauß and Smolka, 1991]), for the Description Logic $\mathcal{ALC}$, and the approach was quickly extended to other descriptions logics. There were implementations of the these algorithms, which behaved well in practice. But it was only after the implementation of FaCT, by Ian Horrocks ([Horrocks, 1999]), using a highly optimised implementation of a tableau algorithm, that other highly efficient systems started to be designed. Nowadays, thanks the high applicability of descriptions logic, its tableau algorithms make use of many highly sophisticated optimizations.

2.2 A prefixed tableau

A *prefixed tableau* for the Hybrid Logic $\mathcal{H}(@, \mathsf{A})$ ([Bolander and Blackburn., 2007]) is a tableau where the formulas occurring in its rules are *prefixed formulas* of the form $\sigma{:}\varphi$, where σ is a *prefix* and φ is a formula in $\mathcal{H}(@, \mathsf{A})$. The intended interpretation of $\sigma{:}\varphi$ is that σ denotes a world at which φ holds.

The set of *prefixes* will be denoted by $PREF$. We require that $NOM \cap PREF = \emptyset$.

Definition 8 *Let's write $\sigma\varphi \in \theta$ for any prefixed formula $\sigma\varphi$ occurring in a tableau branch θ. This expression means that φ is true at σ on θ.*

The presence of nominals enables Hybrid Logics to express equality of words. In order to handle this notion of equality, we define the notion of *equivalence classes* of prefixes and *representative* of a class:

Definition 9 *Define a binary relation $\sim_\theta$ on the prefixes of a branch θ by $\{(\sigma, \tau) \in \sim_\theta | \sigma a, \tau a \in \theta, a \in NOM\}$.*

Definition 10 *Let θ be a branch of a tableau, and let σ be a prefix occurring on θ. The **nominal urfather** of σ on θ, written $s_\theta(\sigma)$, is defined to be the earliest introduced prefix τ on θ for which $\tau \sim_\theta \sigma$. A prefix σ is called a **nominal urfather** on θ if $\sigma = s_\theta(\tau)$ for some prefix τ.*

We also need to introduce a concept that enables us to have terminating tableaus in the presence of the universal modality. We withdraw from some prefixes the privilege of firing the rule ($\Diamond$) if the information that they have is already included in the information of another prefix.

Definition 11 *For a prefix σ, let $L^\theta(\sigma)$ be the set of formulas true at $s_\theta(\sigma)$, of the shape $\Diamond\theta, \neg\Diamond\theta, s$ and $\neg s$, with s being a propositional symbol or a nominal. We call these formulas **local formulas**.*

Definition 12 *Let θ be a branch of a tableau. We define the **inclusion urfather** of a prefix σ on θ, written $u_\theta(\sigma)$, to be the smallest prefix τ for which: $L^\theta(\sigma) \subseteq L^\theta(\tau)$. A prefix σ is called an **inclusion urfather** on θ if $\sigma = u_\theta(\tau)$ for some prefix τ.*

$$\frac{\sigma{:}(\varphi \wedge \psi)}{\sigma{:}\varphi,\ \sigma{:}\psi}\ (\wedge) \qquad \frac{\sigma{:}(\varphi \vee \psi)}{\sigma{:}\varphi \mid \sigma{:}\psi}\ (\vee)$$

$$\frac{\sigma{:}\Diamond\varphi}{\sigma{:}\Diamond\tau,\ \tau{:}\varphi}\ (\Diamond)^1 \qquad \frac{\sigma{:}\Box\varphi,\ \sigma{:}\Diamond\tau}{\tau{:}\varphi}\ (\Box)$$

$$\frac{\sigma{:}\mathsf{E}\varphi}{\tau{:}\varphi}\ (\mathsf{E})^1 \qquad \frac{\sigma{:}\mathsf{A}\varphi}{\tau{:}\varphi}\ (\mathsf{A})^2$$

$$\frac{\sigma{:}\varphi,\ \sigma{:}a,\ \tau{:}a}{\tau{:}\varphi}\ (\nu Id)^3 \qquad \frac{\sigma{:}@_a\varphi}{\tau{:}\varphi}\ (@)^3$$

[1] The prefix τ is new on the branch.
[2] The prefix τ is already on the branch. [3] τ is the earliest introduced prefix in the branch making a true.

Figure 2.1: Rules of the prefixed tableaux method for $\mathcal{H}(@, \mathsf{A})$

Then a *prefixed tableau* is simply a tree with finite number of branches where the edges represent applications of *tableau rules* and the nodes are labelled by the set of *prefixed formulas* obtained by the rule application corresponding to the incoming edge.

2.2.1 Prefixed Tableau Rules

The *tableaux rules* for $\mathcal{H}(@, \mathsf{A})$ are given in Figure 2.1. Note that in addition to the prefixed formulas, these rules include "accessibility statements" of the form $\sigma\Diamond\tau$, for σ and τ prefixes. The intended interpretation of $\sigma\Diamond\tau$ is that the world denoted by τ is accessible from the word denoted by σ, by means of the "accessibility relation" $\Diamond$.

We explain the rules by dividing them into groups:

Propositional rules:

- The $(\wedge)$ rule is just a rewriting rule. When applied, the denominator of the rule break down the conjunction into its conjuncts.
- The $(\vee)$ rule, on the other hand, matches a disjunction and creates new branches corresponding to the disjuncts that will be explored by the tableau algorithm.

Global rules:

- The (A) rule copies the universally quantified formula to every prefix in the branch.
- The (E) rule generates a new world where the quantified formula holds.

Modal rules:

- The ($\Diamond$) rule generates new accessible worlds, while
- The ($\Box$) rule extends the label of the accessible worlds.

Rules for dealing with nominals:

- The satisfaction rule (@), when applied to a prefixed formula σ:@$_a\varphi$, the denominator of the rule copies the formula φ in the earliest prefix of the branch for which a holds.
- The (νId) rule enforces the representative of an equivalence class of prefixes to receive all true formulas of the class.

The rules ($\Diamond$) and (E) are called prefix generating rules. They choose the smallest prefix that is not present in the branch. Their saturation constraint is that they can't be applied twice on the same prefixed formula on the same branch:

- ($\Diamond$) can not be applied to $\sigma\varphi$ on θ if it has been applied to $\tau\varphi$ with $\sigma \sim_\theta \tau$
- (E) is never applied to $\sigma\mathsf{E}\varphi$ if there is a prefix τ such that $\tau\varphi$

Another saturation constraint is that a formula is never added to a tableau branch where it already occurs. Thus:

- (@) is never applied to σ if it has already been applied to $\tau\varphi$

Also, given that the hybrid language incorporates the universal modality ($\mathcal{A}$), additional restrictions are required by blocking ($\Diamond$) to ensure termination: the *loopcheck*.

- The rule ($\Diamond$) is only applied to a formula $\sigma\theta$ on a branch if σ is an inclusion urfather on that branch

A *saturated tableau* is a tableau in which no more rules can be applied that satisfies the saturation constraints. A *saturated branch* is a branch of a saturated tableau. A branch of tableau is called *closed* if it contains *clashing formulas* (of the form $\sigma\varphi$ $\sigma\neg\varphi$, for some prefix σ). Otherwise the branch is *open*. If all branches of a tableau are closed then the tableau is called *closed*, otherwise, if at least one branch is open, the tableau is called *open*.

2.2.2 A procedural view of the Prefixed Tableau Algorithm

In this section we provide a procedural view of the tableau algorithm in this calculus. This approach will be useful later to understand where to include the caching optimization inside this tableau algorithm.

So, suppose that we want to apply tableaux to a formula φ, containing the set of nominals NOM_φ. At the beginning of the tableaux, we create the root branch θ_{root}, which contains:

- A prefixed formula σ_0:φ, called root formula, where $\sigma_0 \in PREF$ is a new prefix.
- A prefixed formula σ_n:n, for each nominal $n \in NOM_\varphi$, with σ_n a new prefix.

Then the *tableau algorithm* (Algorithm 1) is called with the root branch as input parameter. The algorithm starts by looking for a clash in the current branch (lines 2 to 4). If it finds a clash, it stops and returns the value CLOSED. Otherwise, it goes on.

The next step is to find a rule to apply. This is done by the function choose-rule-to-apply (in line 5). As we said before, the possible rules to apply in the tableau algorithm are shown in figure 2.1, and the rule application should respect the saturation and loop checking constraints.

Once a rule is chosen, it is applied. This is done by the function apply-rule in the algorithm (in line 7). The application of a rule results in a list of new branches, which can be of size one in the case of non-branching rules.

Finally, the tableau algorithm is applied to every branch obtained when applying rules. It stops when we reach an open branch or when all the branches are closed, in which cases it returns as result OPEN or CLOSED, respectively.

Algorithm 1 Tableau Algorithm

```
1:  function TABLEAU(Θ_current)
2:      if σ:p, σ:¬p ∈ Θ_current then
3:          return CLOSED
4:      else
5:          rule := choose-rule-to-apply
6:          if rule ≠ null then
7:              list-branches := apply-rule (rule)
8:              index := 0
9:              max-index := length(list-branches)
10:             repeat
11:                 Θ_current := list-branches[index]
12:                 res := tableau(Θ_current)
13:                 index := index + 1
14:             until res = OPEN or index = max-index
15:             return res
16:         else
17:             return OPEN
18:         end if
19:     end if
20: end function
```

2.3 HTab: A terminating tableau System for Hybrid Logic

As we mentioned previously, the aim of this thesis is to find out whether or not the caching optimization could be useful for a tableau procedure in Hybrid Logics. In order to do this, we implemented the optimization in HTab ([Hoffmann and Areces, 2003]), a tableau based theorem prover for Hybrid Logics. We now introduce the basic details of HTab, and then we present the optimizations already implemented in HTab.

HTab is a theorem prover for Hybrid Logic. It implements an adaptation of the tableau algorithm introduced in [Bolander and Blackburn., 2007]. Currently, HTab handles Hybrid Logics up to $\mathcal{H}(@, \mathsf{A}, \mathsf{D}, \downarrow)$. That is, the basic modal logic K, nominals, the satisfaction operator @, the universal modality A, the difference modality D, and the down-arrow binder $\downarrow$. It guarantees termination only for all inputs of the Hybrid logic $\mathcal{H}(@, \mathsf{A}, \mathsf{D})$. Its aim is to provide a range of inference tasks beyond satisfiability checking. For instance, currently it provides model building (i.e. generates a model from a saturated open branch in the tableau).

However, as we mentioned previously, for the scope of this thesis we will cover only the Hybrid logic $\mathcal{H}(@, \mathsf{A})$.

2.3.1 About HTab's implementation

HTab is implemented in the functional programming language Haskell, and the executable is generated using the Glasgow Haskell Compiler (GHC). The code is released under the GNU GPL and can be downloaded from:
`http://code.google.com/p/intohylo/`.

It is executed from the command line. The simplest way to execute it is:

```
htab -f file
```

Where the -f flag specifies the file htab takes as input, and is also mandatory. We can also ask HTab to generate a model, in which case we should add the argument -m *filename*, and it will generate (in case the evaluated formula is satisfiable), a model and write it into the file *filename*.

Some other options can also be established via command line. For example, we can set a timeout for its execution, configure the rules application strategy, or enable/disable the optimizations.

The main data structure in HTab's implementation is a Branch. It is defined as a record containing:

- The clashable formulas: these are the atomic formulas true at the branch, and the idea of keeping them apart is to make it easier to detect a clash.
- Pending formulas: sets of prefixed formulas still not applied.
- Accessibility relations: to keep the accessible prefixes from a given prefix and for a given relation.
- $\Box$-constraints: for each prefix and relation
- Universal constraints: the set of formulas that are true at all prefixes of the branch.
- Some charts to handle the saturation of rules requiring book keeping (i.e. the $\Diamond$, E and @ rules)
- Sets of formulas true a a given prefix
- The Equivalent classes of prefixes and the representative of each class.
- The rules, which can be classified as:
  - Immediate rules that are applied as soon as a formula of the corresponding type is added to the branch: ($\Box$), (A) and (νId)
  - Delayed rules which application order can be specified: ($\wedge$), ($\vee$), ($\Diamond$), (@) and (E)

This way, we can classify the information contained in a branch in HTab as into two groups: the history and the todo lists. The history is everything that is a result of rules applications: clashable formulas, $\Box$ and A-constraints, saturation information of rules $\Diamond$, E and @, etc. The todo lists are lists of formulas yet to be processed by the tableau rules. The way the formulas are picked is the result of the strategy chosen for rules applications.

2.3.2 Optimizations

Currently, HTab makes use of the following optimizations:

Semantic Branching Semantic Branching ([Horrocks and Patel-Schneider, 1999]) addresses an inherent inefficiency of the tableau algorithms, namely that they use a search technique based on syntactic branching. When applying a disjunction $\varphi_1 \vee \varphi_2 \vee ... \vee \varphi_n$, syntactic branching works by choosing unexpanded disjuncts one at a time, and expanding them by searching different models. This creates different branches in the tableau tree that might "overlap" as they are not disjoint. As a result, there might be wasted expansions which could be costly. For instance, consider the formula ([Horrocks and Patel-Schneider, 1999]):

$$(\varphi \vee q_1) \wedge (\varphi \vee q_2)$$

where φ is unsatisfiable. The tableau expansion for this formula requires that the unsatisfiability of φ is demonstrated twice.

Semantic branching allows tableau to avoid such a situation, by trying to avoid repeating "failed" choices when expanding disjunctions. For that, we add to the second explored branch the negation of the formula added to the first branch (which should be closed). Then, the disjunction rule is replaced by:

$$\frac{\sigma{:}(\varphi \vee \psi)}{\sigma{:}\varphi \mid \sigma{:}\neg\varphi \wedge \psi}\ (\textit{semantic branching})$$

Unit Propagation Also known as "simplification" ([Horrocks and Patel-Schneider, 1999]) or "boolean constant propagation" ([Freeman, 1995]). Unit Propagation is a technique used to reduce the number of ($\vee$)-rule applications and lowering the average branching depth. The basic idea is to identify disjunctions $\varphi_1 \vee \varphi_2 \vee ... \vee \varphi_n$ in the branch, such that the negation of one (or more) of the disjuncts is already a formula in the branch. Those disjunctions are reduced and the disjuncts which have a negated counterpart in the branch are not considered as an alternative because adding them would anyway lead to a clash. That is, we apply the following rule:

$$\frac{\sigma{:}\neg\varphi, \sigma{:}\varphi \vee \psi}{\sigma{:}\psi}\ (\textit{unit propagation})$$

Note that as we are working with a prefixed tableau one extra constraint is added to the application of this rule: that the negation of the disjuncts in the branch occurs at the same world (prefix) as the disjunction.

Note that if Unit propagation is used in combination with backjumping (discussed below), eliminated disjuncts have to be handled as if they had actually caused a clash. That is, we should add the union of the dependency sets to each of the eliminated disjuncts (those that would have caused the clash).

Backjumping Backjumping ([Horrocks and Patel-Schneider, 1999]), is an optimisation that aims to reduce the search space by using dependency directed backtracking instead of the usual one-level backtracking. It tries to avoid "trashing", i.e. the exploration of branches differing only in inessential features from branches that have been previously explored.

In order to prune the search space, backjumping uses the information about the cause of the clash. That is, while naive backtracking always go back one level, backjumping ignores those branching points where a different choice can not result in an open branch.

To be able to determine exactly up to which branching point we can backtrack, backjumping needs further information to be attached to the prefixed formulas,

namely the branching points of each formula (i.e., the applications of the $\vee$ rule), which are called "dependency points". Thus, we can label each formula with a "dependency set" and then, when a clash is detected, use the "dependency set" of the clashing formulas for determining the most recent branching point that introduced one of the clashing formulas and backtracking directly to this branching point.

For instance, consider the following example from [Tsarkov *et al.*, 2007]:

$$(A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge ... \wedge (A_n \vee B_n) \wedge \Diamond(A \wedge B) \wedge \Box\neg A$$

Figure 2.2 shows the search tree obtained with the application of tableau to this formula. In the example, the branches in red represent the trashing that can be avoided with backjumping. We can see that without backjumping we need to explore every disjunction, although the cause of the clash is in the last two conjuncts ($\Diamond(A \wedge B) \wedge \Box\neg A$), and does not depend on the rest of the formula. As a result, we have a useless exploration of branches, as we have to explore and close every possible branch. Although the cause of each branch becoming closed is always the same, the last two conjuncts of the formula, and is independent from the choices made by the disjunctions, naive backtracking does not recognize it.

On the other hand, with backjumping we avoid exploring the branches in red as it recognizes that the cause for the branch becoming closed is independent of any disjunction. For that, each formula is labeled with the corresponding dependency set (the information between curly brackets). When the first clash (clash between A and $\neg A$) is detected, the union of the dependency sets of the clashing formulas is used to determine the branching point where we should backtrack. In this particular example, this union of dependency sets is empty, i.e. there exists no alternative branch that might lead to an open branch. Therefore, the result unsatisfiable can be immediately returned.

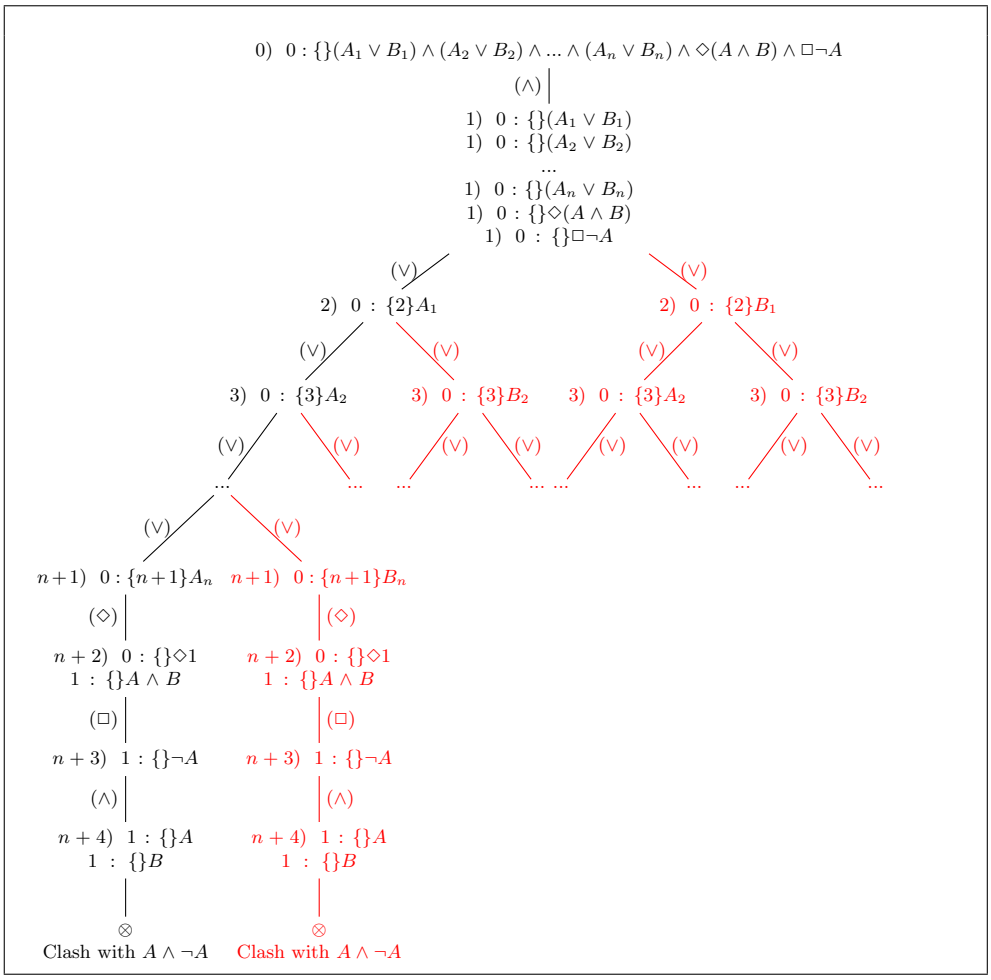


Figure 2.2: Example of Trashing and how it can be avoided with backjumping. Trashing is marked in red and the information between curly brackets corresponds to the "dependency sets". With backjumping we can avoid exploring the red branches, as we can use the union of the dependency sets of the clashing formulas to backtrack directly to the root branch.

Chapter 3

Caching

When applying the tableau algorithm to a formula, there may be many branches created. Suppose that a subbranch has been fully expanded until reaching a clash. Now suppose that in the same formula we find another subbranch which subterms contain the subterms of the closed branch. This will result in the same subproblem being solved again. As the satisfiability check cares only whether a branch is satisfiable or not, this recomputation is time wasted. To avoid this, the usual technique is to store in a cache the intermediate results, and query the cache before applying any rules to a branch, so we can re-use the results of already solved sub-problems. This is caching.

Caching has been widely investigated in Description Logics. However, it is not yet known whether it will help in the case of Hybrid Logic; nominals and the universal modality may cause problems. In this chapter we explain caching optimizations in general terms and briefly introduce the approaches already existing for the case of Description Logics. Finally, we mention some implementations of caching in the field of Description Logics, and the recently one in Hybrid Logic. In the next three chapters we will investigate whether these approaches can be adapted to Hybrid Logic.

3.1 What to "cache"

We already know that storing intermediate results in order to avoid recomputing sub-problems could help reducing considerable time. But now the question is which information should be stored. As we said, the satisfiability check cares only whether a branch is satisfiable or unsatisfiable. So, the intermediate results we are interested in are those sets of formulas/concepts that are already known to be satisfiable or unsatisfiable. Below, we discuss separately each case.

3.1.1 Caching Unsatisfiable Concepts:

A set of concepts can be stored as unsatisfiable when it contains a clash or when it produces a clash (i.e. when we reach a clash after expanding it). Then, when we find the same set of concepts (or a superset), we can claim it is unsatisfiable. The stored unsatisfiable sets of concepts can be used through all the tableau process, and it has been shown that it ensures soundness ([Massacci and Donini, 1999]).

3.1.2 Caching Satisfiable Concepts:

A set of concepts can be claimed to be satisfiable when it has been fully expanded. However, when doing satisfiability check by means of tableau, a branch which has been fully expanded is an open branch. And the tableau algorithm stops when it reaches an open branch, making it worthless to cache these formulas.

Still, we could try to store sets of formulas which can't be further expanded as satisfiable, and reuse these intermediate results later during the tableau. But this "satisfiable concepts caching" must be done only at the level of the branch. That is, the stored sets of satisfiable concepts should be discarded when passing from one branch to another. In [Massacci and Donini, 1999] and [Nguyen and Goré, 2007] the authors remark that caching "permanently" (i.e. "globally"), potentially satisfiable sets of concepts might lead to an unsound calculus.

Besides, caching satisfiable concepts would only be possible in a logic where the top-down strategy is applied naively (that is, where constraints can't be applied upwards), which is not the case of the logics covered in this thesis. In the presence of nominals, new concepts can be propagated to already cached information, making the hole set unsatisfiable.

Even in the absence of nominals, when the tableau algorithm uses blocking to ensure termination, we have to be careful when determining the satisfiability of a concept (which may depend on the satisfiability of an ancestor) [Haarslev and Möller, 2000].

The example provided in Figure 3.1 illustrates two problems associated with caching of satisfiable formulas.

So, we have to take into account two things here: first, we can not claim that the formulas at a blocked prefix are satisfiable; and second, we can not use the satisfiable values added to the cache at a given branch in other different branch.

3.2 Existing approaches to Caching

In this section we explain briefly the various caching methods already implemented for Description Logics [Nguyen and Goré, 2007].

1. UNSAT Caching using Depth First Search (DFS) expansion strategy: In this caching approach, there is an explicit separate data structure that stores only the unsatisfiable sets of concepts found. So, before any expansion, the procedure searches in the cache, if it finds a coincidence, then the expansion does not occur and the current set of concepts is set to unsatisfiable; otherwise, the algorithm continues with the DFS procedure. Something important to note in this approach, is the fact that the algorithm requires a blocking condition to be evaluated in order to avoid infinite loops.

2. MIXED Caching using DFS: In this approach, the explicit UNSAT cache is maintained in the same way as the previous approach, but here we also maintain a local sat cache, which scope will be only the current branch. That is, we maintain the local cache as long as the DFS procedure moves in the same branch, and, when it moves to another branch the cache is emptied. This method also requires the evaluation of a blocking condition to guarantee termination.

3. GLOBAL Caching DFS: This approach uses a DFS strategy also, but it maintains a different kind of structure. It keeps a graph whose nodes' status can be

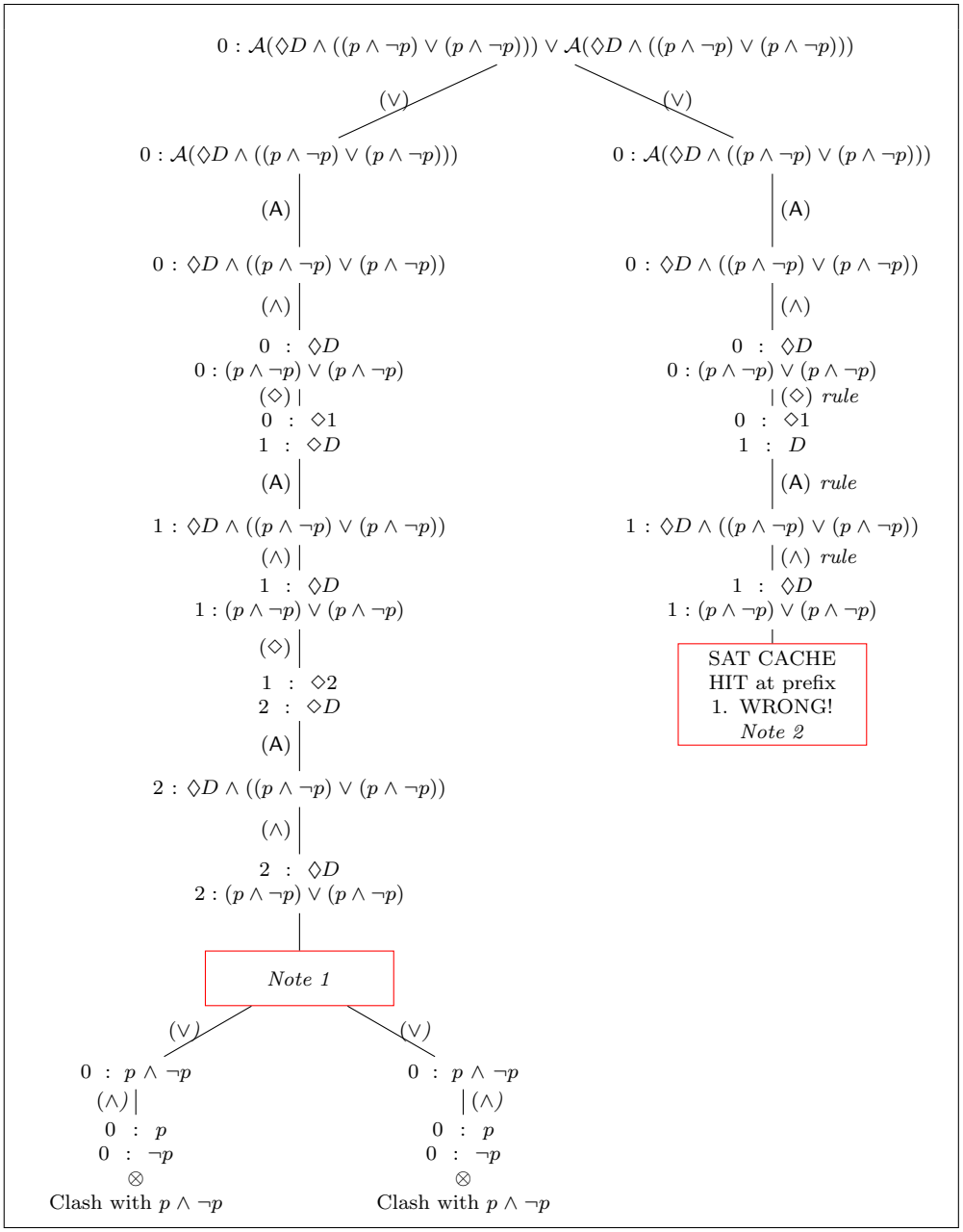


Figure 3.1: Problems with caching satisfiable sets of formulas in Hybrid Logics. *Note 1*: Prefix 2 is blocked by prefix 1, so, following the sat caching strategy used in the DL approach, we would cache the set of formulas true at prefix 2 as satisfiable. *Note 2*: If we use the satisfiable cache globally, we find a cache hit at this point. However, this formula is unsatisfiable.

either unexpanded, expanded, UNSAT or sat; and it propagates the sat/UNSAT status of the nodes through the graph when it knows it. In this approach there is no need of extra blocking condition evaluation, as it guarantees termination.

4. Unrestricted GLOBAL Caching (i.e. GLOBAL Caching, but non-DFS): This approach does not use a DFS strategy. In the DFS strategy, when the applied rule gives k denominators $y_1, ..., y_k$, we just create (or look) the first successor, and put it in the queue for expansion. Then we continue with this denominator. In the non-DFS approach instead, for the same example, we create and put in the queue each denominator y_i immediately after we have applied the rule. Moreover, whenever we find a cache hit to an unexpanded node z, where z is a node already in the queue, z is brought to the front of the queue, indicating that z should be processed sooner (as we know that at least two nodes rely on its result).

3.3 Some implementations of Caching

As we said, Caching has been widely investigated in the field of Description Logics. And even when the main interests of the field seemed to be to have a really bounded ExpTime algorithm, several reasoners implemented it. In this section we briefly mention some of them and how they implemented the optimization.

- Description Logic Prover (DLP) [Patel-Schneider, 1998], is an experimental Description Logic knowledge representation system. Although it is an experimental system, it provides a fast satisfiability checker. DLP performs caching by storing in a concept store just sets of concepts (in the form of a concept expression obtained with the conjunction of the concepts in the set), and their satisfiability status (satisfiable, unsatisfiable, unknown). New concept expressions are added to the store when they are not there. Otherwise, their satisfiability status is used/updated as appropriate. We can say that there is much in common between this approach and GLOBAL caching.
- FaCT++ [Tsarkov, 2003], implements MIXED caching, but for logics not involving nominals or inverse roles.
- Pellet [Sirin *et al.*, 2007] also caches the satisfiability status of internal nodes when no inverse properties or nominals are used in the input ontology.
- CWB [Goré and Postniece, 2008], is a prototype developed in C++ in order to evaluate the different approaches to Caching. It implements all four approaches, i.e. UNSAT Caching, MIXED Caching, GLOBAL Caching and the variation of GLOBAL Caching that does not necessarily uses a DFS strategy.

For the case of Hybrid Logics, currently the only implementation of Caching available is in Spartacus [Götzmann *et al.*, 2009], an Hybrid Logic theorem prover recently developed in the University of Saarland, Saarbrücken, Germany. It features a number of optimizations, among which is a "restricted" way of UNSAT Caching that only allows to cache nominal-free unsatisfiable sets of formulas.

In the next chapters we explain how each of approaches to Caching could (or could not) be applied to the case of Hybrid Logics. In particular chapter 4 develops the UNSAT Caching approach, chapter 5 explains some issues about its implementation, and chapter 6 covers the case of the MIXED and GLOBAL Caching.

Chapter 4

UNSAT Caching

4.1 What to cache for H(@,A)

As we have seen, UNSAT caching has already been explored for Description Logics. The case of Description Logic $\mathcal{ALC}$ is well-known, and properties of the tableau calculus are used to guarantee that the right information is immediately available to check for cache hits during the calculus. For instance, as noted in [Horrocks and Patel-Schneider, 1999]:

> "If successors are created only when other possibilities at a node are exhausted, then the entire set of concept expressions that come into a node label can be generated at one time."

This quote can be transposed into the basic modal logic as: if we have a tableaux calculus in which the rule ($\Box$) has the highest priority and is instantly applied after applications of rule ($\Diamond$), then we know that the set of formulas of a prefix is constant. Therefore, a sound caching technique is to cache the set of formulas true at a prefix.

Now, if we move to Hybrid Logics: because of nominals and the satisfaction operator, new formulas can be added to a world after it is first created, even if the rule ($\Box$) is immediate. For instance, consider the formula

$$\begin{array}{ll} & p \\ \wedge & (n \vee \Box\neg n) \\ \wedge & \Box(p \vee \Diamond\neg p) \\ \wedge & \Diamond(\neg p \wedge \Box(n \vee (\neg p \wedge n))) \end{array} \tag{4.1}$$

That is why in the case of Hybrid Logic, we need also to add the true formulas corresponding to every nominal of the input formula.

Moreover, with existential and universal modalities, the same problem occurs. In modal (and therefore hybrid) logic, universal formulas can be nested deep into regular formulas. As a consequence, formulas of the shape $(\mathsf{A}\phi \vee \mathsf{A}\psi)$ create a situation where not the same universal modalities are "active" in every tableau branch. This is why we also need to cache this information. As an example consider the following formula:

$$(\Diamond\Diamond p \wedge \mathsf{A}\neg p) \vee (\Diamond\Diamond p \wedge \Box\neg p) \tag{4.2}$$

This does not happen in Description Logic $\mathcal{ALC}$, where the TBOX can be considered as a $\mathsf{A}-$subformula. Indeed, checking satisfiability of an ABOX in the

context of a TBOX is equivalent to checking the satisfiability of the Hybrid Logic formula:

$$\text{h(ABOX)} \wedge \mathsf{A}\ \text{h(TBOX)}$$

where h is the satisfiability-preserving function converting an $\mathcal{ALC}$ formula to a Hybrid Logic one. In that case, h(ABOX) does not contain any universal modality.

The formulas 4.1 and 4.2 shown above are developed later, in the examples of section 4.4. First we introduce the UNSAT caching approach developed for Hybrid Logics, in order to understand how does it work.

4.2 Definitions and algorithms

We now introduce the data structures and algorithms used to implement UNSAT caching. First, let us introduce a few definitions related to tableaux branches.

Definition 13 *We write pref(θ) for the set of prefixes occurring on θ, that is:* $pref(\theta) = \{\ \sigma\ \mid\ \sigma{:}\phi \in \theta\}$

Definition 14 *We define the set of* universally constrained formulas *of a branch* θ *as* $\mathcal{U}^\theta = \{\ \phi\ \mid\ \exists \tau \in pref(\theta)\ .\ \tau{:}\mathsf{A}\phi \in \theta\}$

We introduce the "son-of" relation on branches:

Definition 15 *If* θ_1 *is produced by the application of a rule to some formula in* θ_0*, we write* $\theta_0 \rightarrow \theta_1$*, and write* $\rightarrow^*$ *for its reflexive and transitive closure.*

We are in the context of a tableau algorithm using a depth-first search strategy. In this situation, we want to maximize the usefulness of caching, that is, when a clash occurs, we want to cache the earliest known sets of formulas that provokes a clash. Therefore, we should identify the earliest closed branch which produced the clash. For that, let us introduce the functions hclosed and lopen, that are illustrated on Figure 4.1.

Definition 16 *The function hclosed is defined by:*

$$hclosed(\theta) = \begin{cases} \theta' & \text{if there exists } \theta', \text{ the earliest closed branch such that } \theta' \rightarrow^* \theta \\ \emptyset & \text{otherwise} \end{cases}$$

We overload this function by defining it for a couple prefix-branch:

$$hclosed(\sigma, \theta) = \begin{cases} \theta' & \text{if there exists } \theta', \text{ the earliest closed branch such that } \theta' \rightarrow^* \theta \text{ and } \sigma \in pref(\theta') \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 17 *The function lopen is defined by:*

$$lopen(\theta) = \begin{cases} \theta' & \text{if there exists } \theta', \text{ the latest open branch such that } \theta' \rightarrow^* \theta \\ \emptyset & \text{otherwise} \end{cases}$$

Notice that when hclosed(θ) and lopen(θ) are both defined for a branch θ, we have $lopen(\theta) \rightarrow hclosed(\theta)$. We now turn to the definition of the UNSAT cache.

Definition 18 *We remind that FORMS is the set of well-formed formulas of Hybrid Logic. Let n be the number of nominals present in the input formula. We write* $\mathcal{C}_\mathcal{U}$ *for the UNSAT cache. It is such that:*

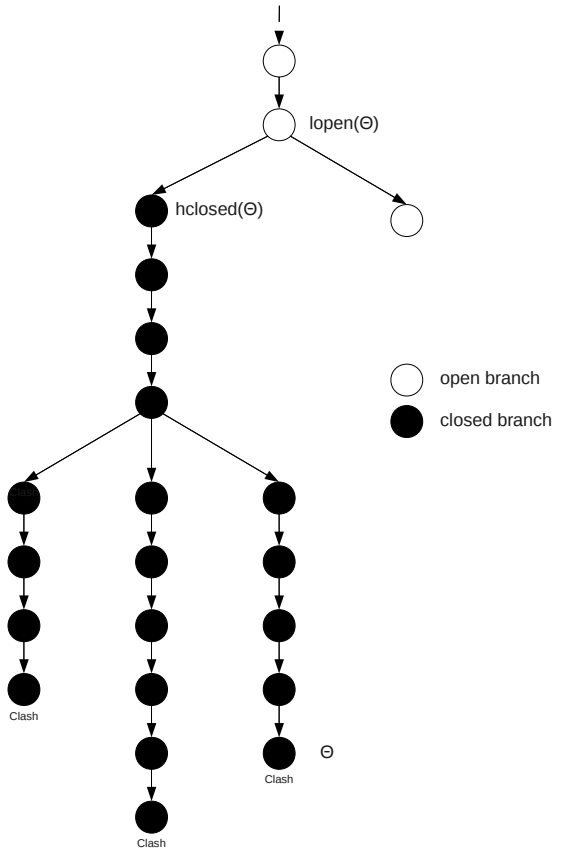


Figure 4.1: When θ is claimed closed, hclosed(θ) is the oldest ancestor of θ we can claim closed

$$\mathcal{C}_{\mathcal{U}} \subseteq \mathcal{P}(FORMS \times FORMS \times \overbrace{FORMS... \times FORMS}^{n\ times})$$

This structure reflects the fact that we are caching three main sets of formulas: formulas of a given prefix, universally constrained formulas, and formulas true at nominals.

Now, let us specify what formulas are to be added to the UNSAT cache during the tableau procedure.

Definition 19 *Given a branch θ and a prefix σ occurring on it, we define: $\mathcal{F}^{\theta}(\sigma) = \mathcal{T}^{\theta}(\sigma) \setminus \mathcal{U}^{\theta}$*

That is $\mathcal{F}^{\theta}(\sigma)$ is the set of formulas true at σ on θ minus the universally constrained formulas on θ.

Definition 20 *Let σ a prefix occurring on θ, and $N_1..N_i$ nominals of the input formula. We define the caching information of the prefix σ on θ by:*

$$c(\sigma, \theta) = (\mathcal{F}^{\theta}(\sigma), \mathcal{U}^{\theta}, \mathcal{F}^{\theta}(N_1), ..., \mathcal{F}^{\theta}(N_i))$$

We now describe how to use the UNSAT cache during the course of the calculus:

Initialization At the beginning of the tableaux algorithm, the cache is empty: $\mathcal{C}_{\mathcal{U}} := \emptyset$.

UNSAT Cache Update When a clash occurs at σ on θ (i.e. $\sigma{:}\varphi, \sigma{:}\neg\varphi \in \theta$ for σ a prefix on the branch θ and φ a formula), then for all τ such that:

1. $\tau \prec_\theta^* \sigma$, that is τ is a predecessor of σ
2. $hclosed(\tau, \theta) \neq \theta$, this condition is to avoid caching the formulas in the clashing prefix when it includes the clash, i.e. when the branch under consideration is θ. It allows adding to the cache formulas in the clashing prefix when we backtrack after all branches of a disjunction are closed
3. $\tau \in pref(\theta) \setminus pref(lopen(\theta))$, that is τ does not appear in the lowest open ancestor of θ
4. For all branch θ' such that $hclosed(\tau, \theta) \rightarrow^+ \theta'$ and a clash occurred on θ' at prefix σ': $\tau \prec_\theta^* \sigma'$. That is, τ has to be ancestor of every clashing prefix in the closed branch

update the cache:

$$\mathcal{C}_{\mathcal{U}} \quad := \quad \mathcal{C}_{\mathcal{U}} \cup c(\tau, hclosed(\tau, \theta))$$

Let us analyze each condition a little deeper. The first three conditions just state that for σ (in case that $hclosed(\sigma, \theta) \neq \theta$), and every predecessor τ of σ which does not appear in the lowest open ancestor of θ, we cache its cachable information in the context of the first closed branch where it appears. Why the first? Because later, its set of formulas only grows, so we are not interested into caching a superset of a set of formulas that we know as unsatisfiable.

Note here that we require that the prefix to be "cached" does not appear in the lowest open ancestor of θ. To understand why, consider the example shown in Figure 4.3. The formula shown in this example is satisfiable. However, when the clash occurs at the first closed branch, if we add to the cache the information valid at prefix 0, we are incorporating in the UNSAT cache sets of formulas which are satisfiable. As a result, the second branch (which is in fact open), is closed because of a cache hit.

Finally, let us analyze the last condition: "For each branch θ' such that *hclosed* $(\tau, \theta) \rightarrow^+ \theta'$ and a clash occurred on θ' at prefix σ, $\tau \prec_\theta^* \sigma$".

This condition is adding an extra restriction to the case of updating the cache after all branches of a disjunction are closed. In this case, we only update the cache with the formulas true at the prefix containing the disjunction formula, if this prefix is ancestor of all the clashing prefixes. In other words, we only update the cache with the information of the prefix containing the disjunction, if it is "responsible" for all the clashes.

Figure 4.2 illustrates an example showing why this condition is necessary.

UNSAT Cache Querying If for some prefix σ in the branch θ: [1]

$$c(\sigma, \theta) \supseteq L = (C_1, C_2, C_3, \ldots, C_{i+2}) \in \mathcal{C}_{\mathcal{U}}$$

That is: there is a tuple in $\mathcal{C}_{\mathcal{U}}$ such that every element is a subset of respectively each element of the caching information of σ.

Then, we can claim the branch is CLOSED.

[1] Note that after a rule application, we know precisely which prefixes have received formulas, so we can restrict the previous test to these "augmented" prefixes.

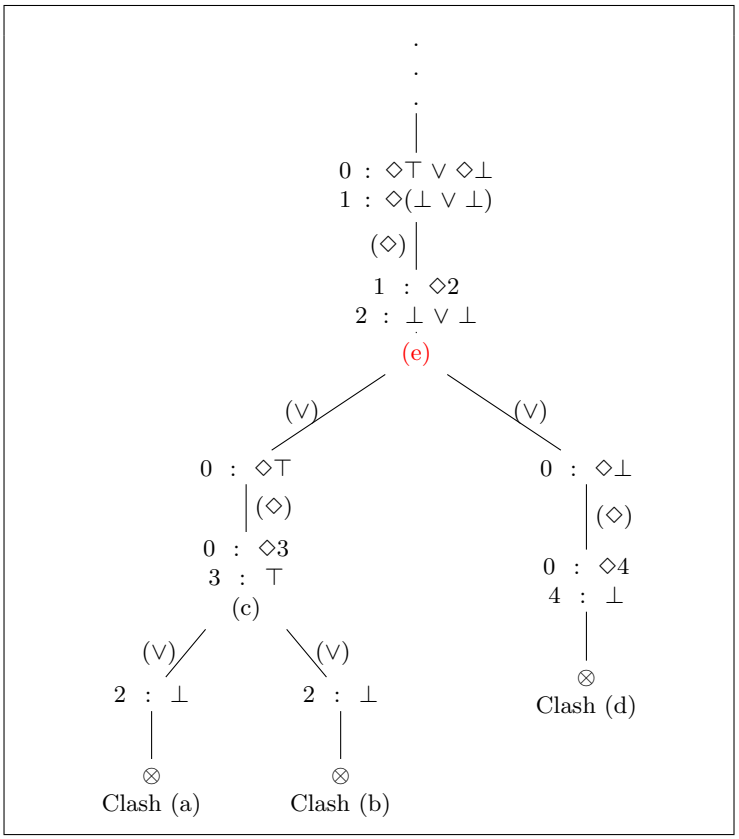
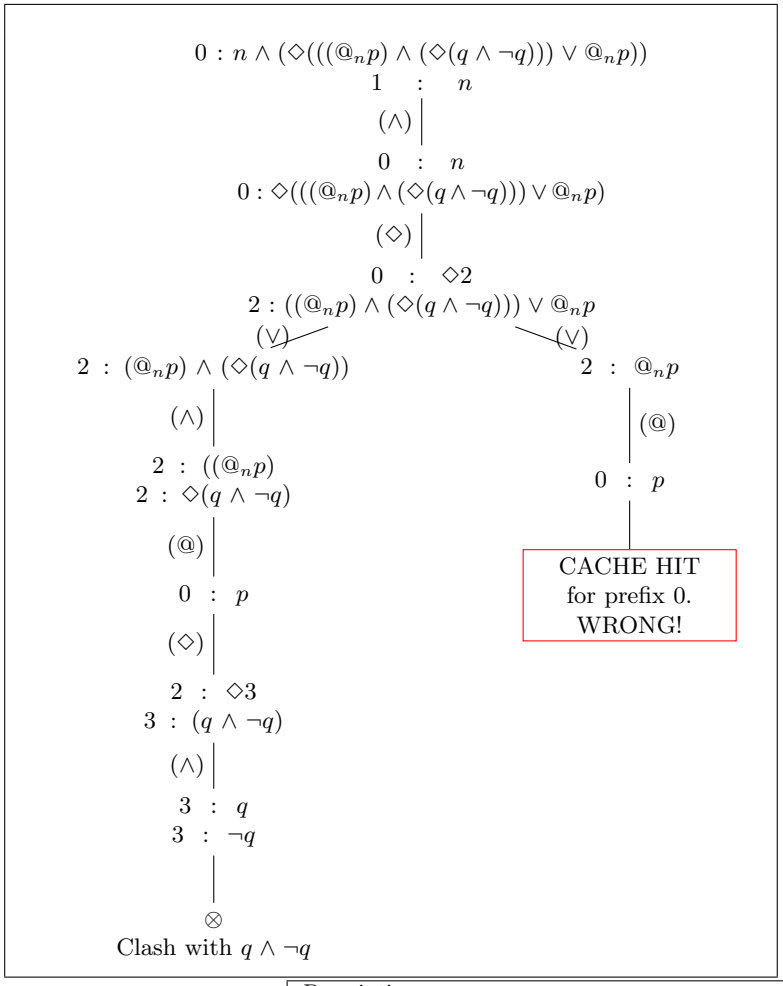


Figure 4.2: Part of a tableau search tree showing what happens if we don't add the fourth condition in the UNSAT Cache Update. After clash (a) and clash (b), the algorithm backtracks and adds to the cache the formulas in prefix 2 (c). This is OK, given that prefix 2 actually contains an unsatisfiable set of formulas that produced the clashes. However, after clash (d), when backtrack until point (e) we can not add to the UNSAT cache the formulas of the prefix that produced the disjunction (prefix 0), because in fact the formulas in prefix 0 at this point are satisfiable.



UNSAT Cache:

$M[T_1] \rightarrow [4, 5, 6, 7, 8, 9, 10, 11]$
$M[T_1] \rightarrow [0, 1, 2, 3, 4, 5, 6, 7]$

Description	Index
$@_n p \wedge \Diamond(q \wedge \neg q)$	0
$((@_n p) \wedge (\Diamond(q \wedge \neg q))) \vee @_n p$	1
$@_n p$	2
$\Diamond(q \wedge \neg q)$	3
Nominal n: n	4
Nominal n: p	5
Nominal n: $n \wedge (\Diamond(((@_n p) \wedge (\Diamond(q \wedge \neg q))) \vee @_n p))$	6
Nominal n: $\Diamond(((@_n p) \wedge (\Diamond(q \wedge \neg q))) \vee @_n p)$	7
n	8
p	9
$n \wedge (\Diamond(((@_n p) \wedge (\Diamond(q \wedge \neg q))) \vee @_n p))$	10
$\Diamond(((@_n p) \wedge (\Diamond(q \wedge \neg q))) \vee @_n p)$	11

Figure 4.3: Wrong Tableau applying UNSAT Caching and adding to the cache all the prefixes valid in the branch

Optimisation After the step of cache updating, we can add the following step to avoid redundant information in the cache:

$$\mathcal{C}_{\mathcal{U}} := \mathcal{C}_{\mathcal{U}} \setminus \{\ L \mid L \in \mathcal{C}_{\mathcal{U}}, \exists L' \in \mathcal{C}_{\mathcal{U}}, L \supsetneq L'\}$$

If a set of formulas is unsatisfiable, then any superset of this set will be unsatisfiable. For this reason, when we query for a cache hit, we check whether the set of formulas in the current prefix is a superset of some set of formulas in the cache. Based on this idea, we try to avoid redundant information in the cache by keeping only the smallest unsatisfiable sets of formulas. For that if the cache contains two sets of formulas L and L', and one is subset of the other, the subset remains in the cache while the superset is removed.

4.3 Integrating UNSAT Caching in the Tableaux Algorithm

Algorithm 2 integrates UNSAT caching (update and search operations), into the tableau algorithm presented in Section 3.

Algorithm 2 Tableau Algorithm, with UNSAT Caching

```
1: function TABLEAU(Θ_current)
2:     if σ:p, σ:¬p ∈ Θ_current then
3:         update-unsat-cache
4:         return CLOSED
5:     else
6:         if search-unsat-cache-hit then
7:             return CLOSED
8:         else
9:             rule := choose-rule-to-apply
10:            if rule ≠ null then
11:                list-branches := apply-rule (rule)
12:                index := 0
13:                max-index := list-branches.length
14:                repeat
15:                    Θ_current := list-branches[index]
16:                    res := tableau(Θ_current)
17:                    index := index + 1
18:                until res = OPEN or index = max-index
19:                if res = OPEN then
20:                    return OPEN
21:                else
22:                    update-unsat-cache
23:                    return CLOSED
24:                end if
25:            else
26:                return OPEN
27:            end if
28:        end if
29:    end if
30: end function
```

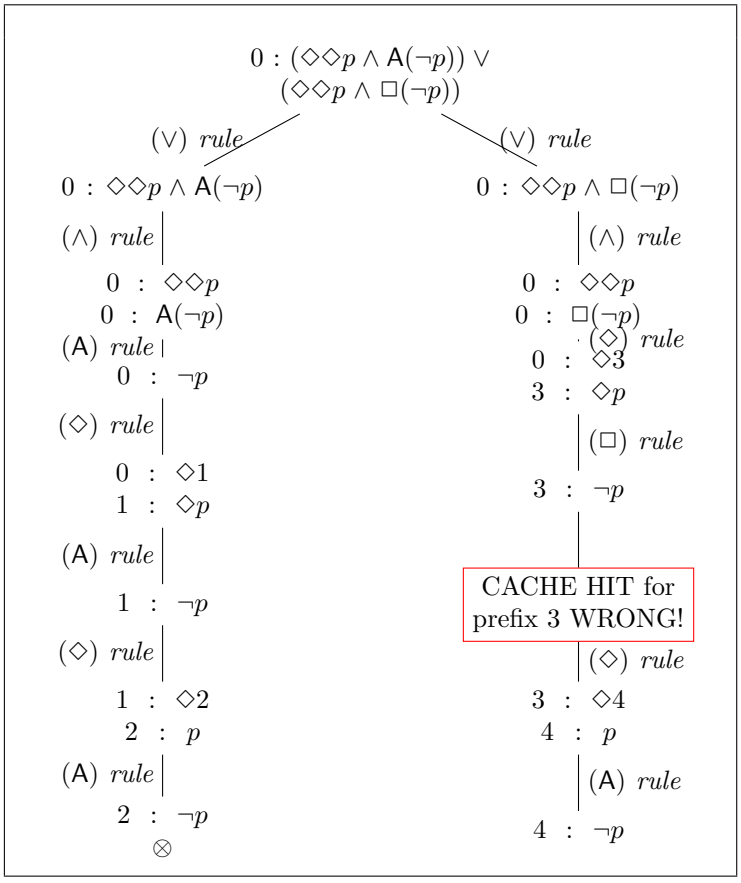
4.4 Some examples

Here we develop the formulas shown in Section 4.1. The first example illustrates the importance of adding information regarding the universal constraints true at the cached branch; while the second one covers a case where the lack of information regarding the nominals present in the cached formulas produce incorrect results.

Why adding universally constraints formulas? Figure 4.4 shows that the following example:

$$(\Diamond\Diamond p \wedge \mathsf{A}\neg p) \vee (\Diamond\Diamond p \wedge \Box\neg p)$$

leads to an incorrect result in the case of a tableau algorithm with UNSAT caching without caching the universally constrained formulas.



UNSAT Cache:

$M[T_1] \rightarrow [0, 1]$

Description	Index
$\neg p$	0
$\Diamond p$	1

Figure 4.4: Wrong Tableau applying UNSAT Caching without adding to the cache formulas universally constraint

Why adding extra information about nominals? Now suppose that we apply the tableau algorithm, again with UNSAT caching, but this time without caching the information true at nominals. In this case, it is also possible to introduce wrong answers, as can be seen on Figure 4.5 and Figure 4.6.

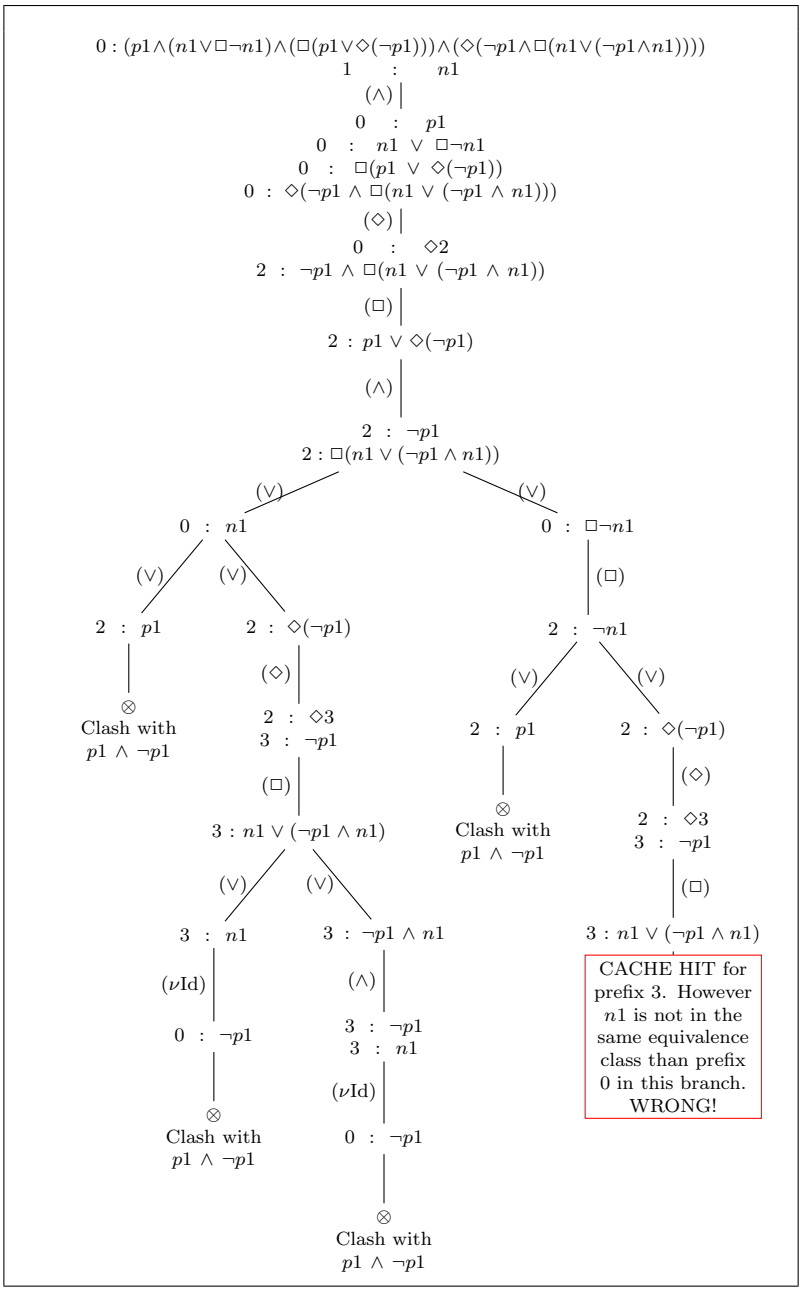


Figure 4.5: Wrong Tableau applying UNSAT caching without adding to the cache information regarding nominals

Next chapter explains some implementation issues about the UNSAT Caching approach.

UNSAT Cache:

$M[T_3] \rightarrow [2, 3, 4, 5, 6, 7]$
$M[T_1] \rightarrow [0, 1]$

Description	Index
$\neg p1$	0
$n1 \vee (\neg p1 \wedge n1)$	1
$n1$	2
$p1$	3
$(p1 \wedge (n1 \vee \Box\neg n1) \wedge (\Box(p1 \vee \Diamond(\neg p1))) \wedge$ $(\Diamond(\neg p1 \wedge \Box(n1 \vee (\neg p1 \wedge n1))))$	4
$n1 \vee \Box\neg n1$	5
$\Box(p1 \vee \Diamond(\neg p1))$	6
$\Diamond(\neg p1 \wedge \Box(n1 \vee (\neg p1 \wedge n1)))$	7

Figure 4.6: UNSAT cache representation for example in Figure 4.5

Chapter 5

Implementing the UNSAT Cache

It is important for efficient querying of the UNSAT cache to use a data structure enabling efficient subset checking. That is, the data structure should provide efficient operations for determining when a given set is superset or subset of any set stored in the cache.

In this report, we describe three approaches for implementing the UNSAT cache. First, we develop an approach based on bit matrices as proposed in [Giunchiglia and Tacchella, 2001]. Then, an approach based on a lists vector, which can be thought just as a derivation of bit matrices. And finally, we explain an approach based on tree based data structure as proposed in [Hoffmann and Koehler, 1999]. However, we just implemented the two former approaches.

Following, we explain in more detail each of this approaches, but first let's introduce the mapping structure used by the three approaches to map subterms to indexes.

5.1 About the subterms representation

In the three approaches for representing the UNSAT cache discussed below, we need to represent subterms of the input formula as indexes. For this reason, we need to keep an extra structure mapping each index with the corresponding subterm. There are two ways of initializing/updating this structure:

- The straight way: that is, to associate each possible subterm of the input formula to an index in the begging of the algorithm, when the UNSAT cache is initialized.
- The lazy way: that is, to add subterms to the structure as they are needed, i.e. each time we update the cache.

We adopted the second approach, i.e. we only increase the mapping when we try to add a new subterm to the UNSAT cache, in case it is not still present. So, for all three approaches to implement the UNSAT cache developed bellow, initially the mapping structure is empty.

5.2 Bit matrices approach

This approach was proposed in [Giunchiglia and Tacchella, 2001]. It uses Bit matrices to store the unsatisfiable sets of concepts. Bit matrices have three distinguishing advantages:

1. they can be queried for subsets and supersets;

2. they can be bounded in size;

3. when they are bounded in sized, they keep the latest obtained results.

We implemented the UNSAT cache as a bit matrix, where the columns represent subterms of the input formula, and each row represent a subset of unsatisfiable concepts. The mapping between each column index and the subterm it represents is maintained by the mapping structure.

Matrices have fixed number of columns and rows. Then the number of columns is set as the maximum number of subterms in the input formula, multiplied by two plus the number of nominals. That is, we have to be able to store in the cache not only the formulas true at a given prefix, but also those formulas universally constraint and also those formulas true at the nominals of the input formula. The number of rows is equal to the number of columns in a first approach. However, we know that it should be related to the size and other characteristics of the input formula.

When the matrix is created every bit is initialized to 0.

Insertion In order to insert a new set of unsatisfiable subterms in the UNSAT cache, we must first get its corresponding list of indexes from the mapping structure. As we said, if a subterm is not present in the mapping structure, it is inserted.

Then we proceed with the insertion, taking into account that:

- if the new set is a superset of an already existing set in the cache, we don't update the cache
- if the new set is a subset of any cache row, we replace the corresponding row in the cache
- as the size of the matrix is bounded, when the maximum number of rows is reached, we restart the row index to 0, and begin to reuse the rows starting from the first one

The first two items are to avoid keeping redundant information in the bit matrix, while the last one ensures that we maintain the latest results, as the first results discarded are the oldest ones.

Query During the tableau algorithm, before any expansion, we search for a cache hit in the UNSAT cache. We use subset checking (i.e. we search in the cache for a subset of the current set), because if a set of subterms is unsatisfiable, any superset of the current set is also unsatisfiable.

When searching in each row, we stop exploring each row as soon as we know the set stored in this row is not a subset (or superset, depending on the search), of the input subterms set.

	0	1	2	3	4	...	34
$M[T_1]$	0	0	0	0	0	...	0
$M[T_2]$	0	0	0	0	0	...	0
$M[T_3]$	0	0	0	0	0	...	0
$M[T_4]$	0	0	0	0	0	...	0
...	...	...	...	...	...	...	...

Description	
Index	

⇒

	0	1	2	3	4	...	34
$M[T_1]$	1	1	0	0	0	...	0
$M[T_2]$	1	0	1	0	0	...	0
$M[T_3]$	0	1	0	1	0	...	0
$M[T_4]$	1	0	0	0	1	...	0
...	...	...	...	...	...	...	...

Description	p	q	$\Diamond p$	$p \wedge q$	$\Diamond q$
Index	0	1	2	3	4

Figure 5.1: Bit Matrix with n rows and 34 columns, representing an UNSAT cache with actually 5 subterms and 5 sets of unsatisfiable subterms inserted.

An example Let's consider the bit matrix showed in Figure 5.1. Initially, every cell of the bit matrix is set to 0 and the mapping structure (Description Index) is empty. Now, suppose that we found out that the set $T_1 = \{p, q\}$is unsatisfiable. We first search in the mapping structure the indexes corresponding to p and to q. As it is empty, we add the pairs (p,0) and (q,1), and then use its indexes to update the cache. Something similar happens when inserting the second row in the cache.

Now, suppose that we want to insert $T_i = \{p, q, \Diamond p\}$ after we inserted the second row. Clearly, T_i is a superset of the set represented by the matrix row $M[T_1]$, thus we just don't insert it.

Finally, suppose that we try to insert T_i= $\{p \wedge q\}$ (once the rows shown in the figure are already inserted). In this case, T_i is a subset of the set represented by the matrix row $M[T_3]$, and so we rewrite this row by setting all columns to 0, excepting the one corresponding to $p \wedge q$.

5.2.1 Lists vector approach

The number of formulas in an set of the UNSAT cache is often small compared to the total number of formulas that can appear in the tableau. As a consequence, the bit matrix previously seen is scarce, that is, it is mostly filled with zeros. There can be more optimised ways of storing the UNSAT cache, and a list vector is a solution.[1]

The list vector is a list of lists. Each "sublist", or row, is a list of integers representing the indexes of the unsatisfiable subterms stored in this row. Thus, the length of each row is determined by the number of cached subterms, instead of being a constant determined by the total number of possible subterms to store.

Initially, the list is empty.

A possible implementation for this approach is a List of Sets of Indexes.

```
data ListVector = [IntSet]
data FormToInt = Map Formula Int
```

The advantage of this implementation is that the subset/superset checking for each row naively managed by the data Structure.

Insertion The insertion is easy, with, as previously, two precautions:

- if the new set is a superset of an already existing set in the cache, we don't update the cache

[1] Moreover, matrix handling in a functional programming language like Haskell is not easy nor optimal.

$$M[T_3] \rightarrow [1,3]$$
$$M[T_2] \rightarrow [0,2]$$
$$M[T_1] \rightarrow [0,1]$$

Description	p	q	$\Diamond p$	$p \wedge q$
Index	0	1	2	3

Figure 5.2: List approach for the UNSAT Cache, with 3 rows inserted.

– if the new set is a subset of any cache row, we replace the corresponding row in the cache.

However, there is a subtle implementation difference in this approach. Each time we update an existing row, what we do in fact is to remove the old row from the list and insert the new one in the top of the list. The insertion of a new row is also always in the top of the list. In this way, the first rows to be queried correspond to the more recently inserted ones.

Query Query is carried out almost in the same way as for the previous approach. During the tableau algorithm, before any expansion, we search in the UNSAT cache with the current set of subterms. Here, we also use subset checking to find a cache hit.

An advantage of this approach is that, as the first rows in the list correspond to the last updates, we search first the information more recently added to the cache.

An example Now, consider the list based approach of an UNSAT cache shown in Figure 5.2. Initially, the list and the mapping structure are empty. When we found out that the set T_1=p,q is unsatisfiable, as well as we do for the bit matrices approach, we first search in the mapping structure the indexes corresponding to p and to q. As the mapping structure is empty, we add it the pairs (p,0) and (q,1), and then we use their indexes to update the cache.

Again, if we wanted to insert $T_i = \{p, q, \Diamond p\}$ at any moment after having inserted $M[T_2]$, we don't do it because its index list is a superset of the row $M[T_1]$.

Finally, if we try to insert $T_i = \{p \wedge q\}$ (once the rows shown in the figure are already inserted), its indexes list is a subset of the row $M[T_3]$. So, in this approach, we delete the row, and insert a new row (at the top of the list) with new index list.

5.2.2 Tree based approach

Still another data structure allowing fast set query answering is the tree based approach developed in [Hoffmann and Koehler, 1999]. Although in this project this method was not implemented, we explain the main idea underling this data structure and how it could be applied to implement the UNSAT cache.

The UNSAT cache would consist in a forest of Unlimited Branching Trees (UB-Trees). Each node of an UBTree consists of three components:

– the element it represents,
– the children of the current node (a set of nodes), and
– an end-of-path marker to indicate the end of one possible path.

The element of a node in this approach would be its index in the mapping structure.

The idea is to represent each set of unsatisfiable subterms as a path in a tree, where the nodes are ordered according to their index number. This way, if two sets sharing the first elements are to be stored, we create only one UBTree with the first common elements and two branches, one for each set, resulting in a more compact representation.

The insertion operation To insert a set S into the UNSAT Cache implemented with this approach, first we get the indices of the elements of the set in the mapping structure, and then we try to insert them. The insertion creates a new path for these elements in the forest. For that, it tries to look up the elements of S (which are ordered) one after the other following already existing paths. If there is no node for some element in S, we create it.

Query The algorithm presented in [Hoffmann and Koehler, 1999] provides three search operations: "search-first", "search-subs" and "search-sups". Given a set S, and a set of trees T, the "search-first" operation consists in verifying if S is subset of T, "search-subs" retrieves all the subsets of S in T, and "search-sups" retrieves all the supersets of S in T. The former and second searching operations make use of a "money" method, which consists in finding all nodes in T corresponding to an element $e_i \in$ S, if any, or going on with the following element of S otherwise. The "search-sups" operation, instead, searches all trees starting with an element preceding the first element of S, and reducing the query as we find the elements in the query, until S is empty.

For searching a cache hit in the UNSAT cache we need only the "search-first" operation. So, for a given set of inconsistent subterms we must find in the cache a tree representing a subset of S.

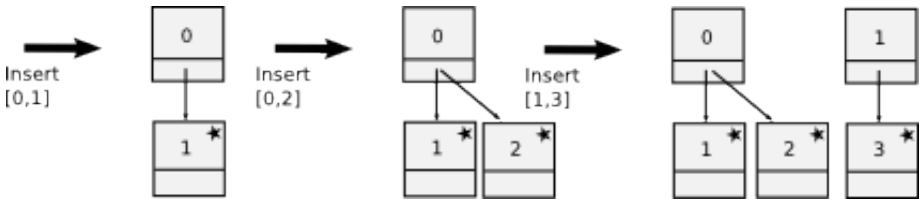


Figure 5.3: Tree approach for the UNSAT Cache, with 3 inconsistent sets inserted.

An example Following the last example, Figure 5.3 shows the insertion of the the sets p,q, p,$\Diamond p$ and q,$p \wedge q$ in this order. Now suppose that after inserting the first set p,q we want to insert p. This could be easily done by adding an end of path mark to the node corresponding to p.

5.3 Enabling Backjumping with UNSAT Caching

Backjumping, introduced in Chapter 2, is a variant of dependency-directed backtracking that makes use of information about the cause of the clash in order to prune the search space and avoid "trashing".

Although backjumping is an efficient optimisation, using it together with UNSAT caching is not trivial. The reason lies in the extra information required by

backjumping. In the case that we find out that a given set of formulas is unsatisfiable because of a cache hit, there is no information about the cause of the unsatisfiability to be used to derive the "dependency set" required for backjumping.

Still, Backjumping and UNSAT caching can be carried out together, by combining the dependency points of all the formulas in the matching set of formulas, as described in [Tsarkov *et al.*, 2007].

Then, when we find a cache hit with:

- $\mathcal{F}^\theta(\sigma) \supseteq C_1$,
- $\mathcal{U}^\theta \supseteq C_2$ and
- $\mathcal{N}^\theta(N_1) \supseteq C_3$, ..., $\mathcal{N}^\theta(N_i) \supseteq C_{i+2}$ for $N_1 ... N_i$ nominals appearing in the formulas of $\mathcal{F}^\theta(\sigma) \cup \mathcal{U}^\theta$.

for some cache element $(C_1, C_2, ..., C_i) \in \mathcal{C}_\mathcal{U}$

We can use the "dependency set" associated to the formulas in $C_1 \cup C_2 \cup ... \cup C_i$ for backjumping.

We implemented this approach although it may result in a set of dependency points containing irrelevant branching points, and could limit the effectiveness of backjumping.

5.4 Source Code

In chapter 4 we presented the theory associated with the implementation of the UNSAT Caching optimization for a hybrid prefixed tableaux. We also presented pseudocode in order to illustrate which parts of the tableau algorithm should be modified in order to include caching. Then we showed some particularities with respect to the implementation of the UNSAT cache. As we said, we implemented the optimization for the Hybrid Logic theorem prover HTab. It was done in the functional programming language Haskell, and the implemented structures were adapted to HTab. For a more detailed description of the implementation, the main algorithms of the source code are included in the final appendix.

Chapter 6

Other Approaches to Caching

In this chapter we develop the two remaining approaches to caching, and show how they could (or could not) be adapted for the case of the Hybrid Logic covered in this thesis. However, either because of time constraints (in the case of GLOBAL caching), or because it is in fact not possible (in the case of MIXED Caching), these approaches were not implemented.

6.1 MIXED Caching

In this approach, the UNSAT cache is maintained in the same way as in the previous approach, but here we also maintain a local sat cache, which scope will be only the current branch. That is, we maintain the local cache as long as the Depth Search First procedure (this approach assumes a Depth Search First strategy) moves in the same branch. When it moves to another branch the local cache is reinitialized.

The MIXED Caching approach is already implemented for Description Logics. It is first introduced in [Massacci and Donini, 1999], and then referred as MIXED caching in [Nguyen and Goré, 2007].

In both articles the UNSAT cache acts globally while the sat cache acts locally. Besides in the former article the sat and the UNSAT caches are handled by specific sets of rules. They also introduce the idea of a "witness" to refer a set of concepts previously processed and which is a subset of the set of concepts in the current node.

In the case of Hybrid Logics, we can say that the notion of local satisfiability caching developed for Description Logics is already subsumed by the notion of blocking condition presented for the hybrid tableaux. More precisely, the idea of a witness in [Massacci and Donini, 1999], is the same idea of a loop check introduced for the hybrid tableaux, where before expanding a branch containing a diamond rule, we check whether a witness already exists for the corresponding pattern.

However, the blocking condition does not cover all cases that the local satisfiability caching in [Massacci and Donini, 1999] and [Nguyen and Goré, 2007] does. In the MIXED caching approach implemented for Description Logics, when it happens that a set of concepts can't be further expanded (for example, the set $\{p, q\}$, where p and q are propositional symbols), these intermediate results can be cached as satisfiable. This does not happen for the blocking condition, given that it only "blocks" with respect to the ($\Diamond$) rule.

Nonetheless, in the presence of nominals, it would not be possible to cache these intermediate results ($\{p, q\}$), even if it was done in the same branch, as new formulas could be propagated to information already cached, making the hole set unsatisfiable. The example shown in Figure 6.1 illustrates one such instance. In the example, at the point marked as *Note 1*, prefix 2 can't be further expanded. According to the traditional MIXED Caching approach for Description Logics, the formulas true at prefix 2 at that moment could be cached as satisfiable. However, if we did so, we would have later a "sat-cache hit" for the formulas true at prefix 1 at the moment pointed by *Note 2*, making this set of formulas satisfiable. But it would not be right to say that the the set of formulas true at prefix 1 are satisfiable because it is at this prefix that the clash occurs. The reason is that because of the nominal n, new formulas are propagated later to this prefix, making it unsatisfiable.

6.1.1 So, what else could be done?

As we said in the previous section, we can say that, for satisfiability checking, the local SAT caching part of the approach is already implemented in the blocking condition (or at least the part that can be implemented for the covered Hybrid Logic). However, for inference tasks which involve retrieving all possible models, we could go a step further. We could try to use the results obtained for satisfiable set of concepts (only those obtained when we reach a saturated open branch) later in the procedure (even across branches), to avoid recomputing these set of nodes. That is, maintain a global cache for the satisfiable set of concepts (in the same way as we did for UNSAT caching).

Unluckily, this is not the case of HTab, which main goal is to test satisfiability of a formula, and which stops when it finds a possible model (i.e. when it encounters the first saturated open branch).

6.2 GLOBAL Caching

This approach was first introduced in [Goré and Nguyen, 2007a] and implemented in [Nguyen, 2008] and [Goré and Postniece, 2008]. It consists in a tableau based algorithm that uses "GLOBAL caching" and propagation of both satisfiability and unsatisfiability to achieve an EXPTIME procedure for the $\mathcal{ALC}$ Description logic. The algorithm builds an "and-or graph", where no two nodes of the graph contain the same formula set. That is, each node in the graph is "labelled" with an unique set of formulas, where a "label" is not a name for a possible word (i.e. is not the same as "prefix" in "prefixed tableau"), but a unique set of formulas contained in a node.

"GLOBAL caching" means that each possible set of concepts/formulas is expanded at most once.

This notion of "GLOBAL caching" can replace the notion of "blocking" which most tableau methods require for termination ("loopcheck" in the algorithm developed in chapter 2), and caching simultaneously, as it does not rely on knowing the satisfiability or unsatisfiability status of the stored nodes.

6.2.1 How does it work for Description Logics?

Suppose that we want to prove the satisfiability of a formula φ with respect to a TBox Γ. The method searches fro a model which satisfies φ w.r.t. Γ by building a graph. During the graph construction, each node of the graph has three attributes:

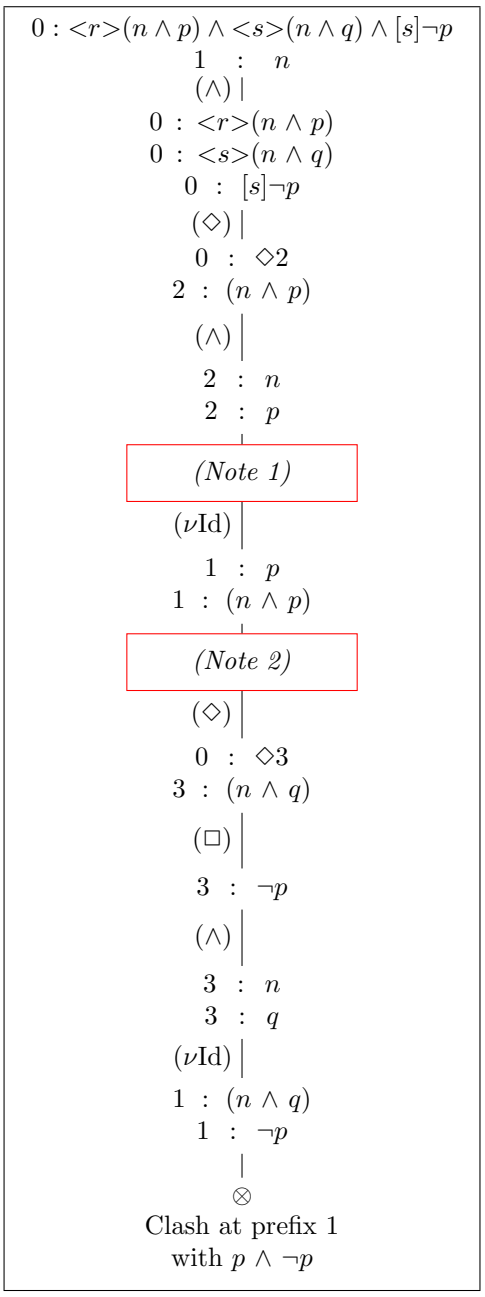


Figure 6.1: Local SAT caching failure in the presence of nominals *Note 1*: At this point we can't go on working with prefix 2, we could cache it as intermediate results, to be used later in the branch. *Note 2*: At this point we could use the intermediate results. However, it would produce a wrong answer.

- *content:* the set of formulas carried by the node
- *status:* unexpanded, expanded, sat or UNSAT (were the status sat/UNSAT represents the satisfiability/unsatisfiability of the input formula w.r.t. the TBox).
- *kind:* and-node, or-node, leaf-node

The graph construction for a basic tableau algorithm using "GLOBAL caching" proceeds as follows:

Initially, the graph consists only on a "root" node, which contains the input formula φ plus the TBox Γ, and which is assigned the unexpanded status.

The next step in the algorithm is to choose a node to expand (i.e. to apply a rule). When a rule is applied during a traditional tableau algorithm, the new formulas created result in a new node. However, with GLOBAL caching we first check whether a node already existing contains the new set of formulas, and if so, we do not create a new node, but just insert an edge from the current node to the one containing the set of formulas. Otherwise, we create a new node with status unexpanded, unless it contains an inconsistency, in which case it is assign UNSAT status, or it can not be further expanded (no rule can be applied), in which case it is assign sat status. Besides, each node in a graph is assign a types: if an or-rule was applied to it then the node is an "or-node", otherwise it is an "and-node". After the application of a rule to a node, if its status is not sat neither unsat, it is set to expanded.

The status of a sat or UNSAT node is "propagated" backwards (in most of the cases) to the other nodes in the graph. This propagation consists in checking if there is enough information to determine whether the satus of a parent node is sat or unsat, taking into account the kind of node (or-node or and-node) and the status of its children.

The algorithm continues the node expansion as long as there are unexpanded nodes to chose and as long as the status of the "root" node is not sat neither unsat. Let's assume a depth first expansion strategy (for comparison purposes, because HTab uses this one), but the algorithm supports other strategies. In the case of finishing because all nodes of the graph have been expanded, all nodes which status is different from UNSAT are assigned sat status (this way giving the open status to the tableau branches which loop).

Figure 6.2 taken form [Nguyen and Goré, 2007] illustrates how the GLOBAL Caching algorithm works when applied to the example given in [Haarslev and Möller, 2000].

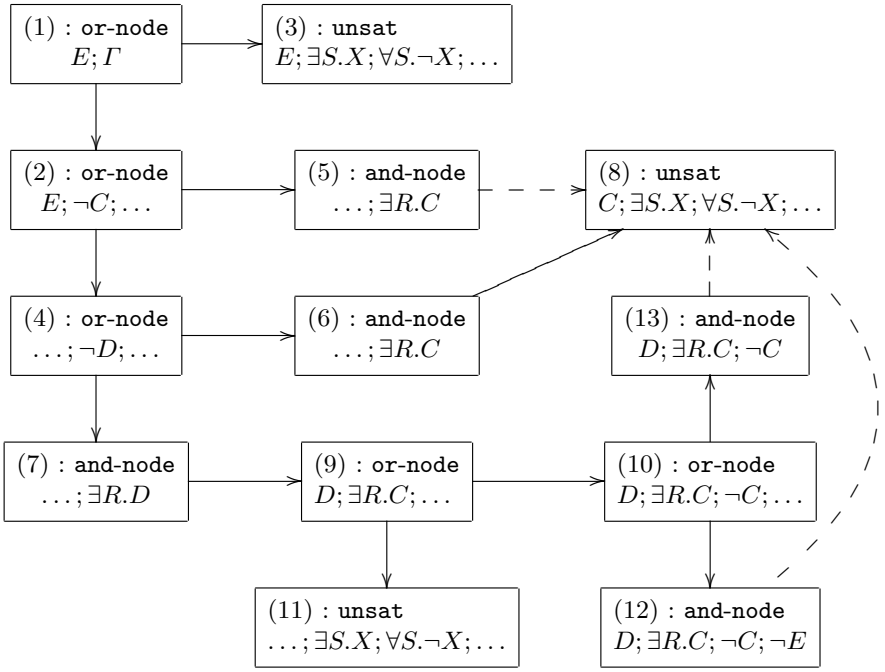
The main difference with other tableau methods is that with GLOBAL caching we create a child in the graph only if it does not already exists, and we search for existing nodes in any previous branch of the tableau.

There are variants and improvements to this algorithm, which cover different Description Logics (for example in [Goré and Nguyen, 2007b]).

It is already shown that caching satisfiable sets of concepts can help reducing the complexity of the tableau algorithms for expressive nominal free Description Logics to EXPTIME ([Goré and Nguyen, 2007b]). However it is still an open problem to find a corresponding technique working with nominals and universal modality.

6.2.2 How would it work for Hybrid Logics?

In this section we present the problems that could arise if we try to use the approach for the case of the Hybrid Logics $\mathcal{H}(@, \mathsf{A})$. More specifically, how to represent the information and deal with it in the same way as it is done for Description Logics, in order to obtain the same advantages.



Node	Content
(1)	$E; C \sqsubseteq^* (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A); D \sqsubseteq \exists R.C; (\exists R.C) \sqcup (\exists R.D)$
(2)	$E; \neg C; D \sqsubseteq^* \exists R.C; (\exists R.C) \sqcup (\exists R.D)$
(3)	$E; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; D \sqsubseteq \exists R.C$
(4)	$E; \neg C; \neg D; (\exists R.C) \sqcup^* (\exists R.D)$
(5)	$E; \neg C; \exists R^*.C$
(6)	$E; \neg C; \neg D; \exists R^*.C$
(7)	$E; \neg C; \neg D; \exists R^*.D$
(8)	$C; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; D \sqsubseteq \exists R.C; E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(9)	$D; \exists R.C; C \sqsubseteq^* (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A); E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(10)	$D; \exists R.C; \neg C; E \sqsubseteq^* (\exists R.C) \sqcup (\exists R.D)$
(11)	$D; \exists R.C; \exists R.D; \exists S.X; \forall S.\neg X; \forall S.A; E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)$
(12)	$D; \exists R^*.C; \neg C; \neg E$
(13)	$D; \exists R^*.C; \neg C$

Figure 6.2: Example taken from [Nguyen and Goré, 2007] illustrating the GLOBAL Caching algorithm. And-or graph created by the application of the algorithm to the concept E and the TBox $\Gamma = \{C \sqsubseteq (\exists R.D) \sqcap (\exists S.X) \sqcap \forall S.(\neg X \sqcap A); D \sqsubseteq \exists R.C; E \sqsubseteq (\exists R.C) \sqcup (\exists R.D)\}$. The main concepts are marked with superscript *. The nodes are numbered when created but explored using DFS: 1:(2,3), 2:(4,5), 4:(6,7), 6:8, 8, 7:9, 9:(10,11), 10:(12,13), 12:8, 13:8, 11, 5:8, 3. Dashed arrows are cache hits. The result is unsatisfiable.

The first question that arises when trying to adapt the approach for the case of a prefixed tableau algorithm for Hybrid Logics is what to consider as the *content* of a node. As we mentioned at the beginning of this chapter, the content of a node in the GLOBAL caching approach introduced by [Goré and Nguyen, 2007a] does not correspond to a possible world in modal logics. Instead, it is determined by the rule application, and corresponds to the denominator of the rule. In the prefixed tableau for Hybrid Logics, on the other hand, the main structure is a branch, instead of a node. So, in order to implement the GLOBAL caching optimization in Hybrid Logics, we can consider as node content the set of formulas determined by each rule application (this way following the ideas used in Description Logics). But, for the case of the prefixed tableaux for Hybrid Logics, we still have to decide whether to keep in each node formulas of a single world or of all the possible worlds.

If we take the first approach, i.e. if we consider that a node can only include formulas of a single world, then we face another problem: how to deal with nominals, as they introduce equality of worlds and upwards propagation. Consider the example in figure 6.3. It is not clear how to represent the node (or nodes) that would correspond to prefix 1, given that new formulas are propagated to this prefix after the expansion of prefixes 3 and 4. In the example we adopted an approach similar to the one used in Description Logics: each time the information of a world is augmented, we create a new node containing the old information about the node plus the change/new information. We can see that for this example the approach does not work because of the nominals: After the first branch is closed, the unsatisfiable status of node (11) is propagated to all nodes in the branch. In particular, node (10) will have the status unsatisfiable. Thus, once we expand the second branch, we find out that the formulas in node (12) are the same as those in node (10) and instead of going on with the expansion we create an edge between the two nodes. As a result, the unsatisfiable status of node (10) is propagated to node (12) and the tableau returns unsatisfiable, which obviously is not the correct answer.

Disregarding the way of representing nodes taken, in the case of the Hybrid Logic $\mathcal{H}(@, \mathsf{A})$ we face another problem: the Universal modality. Consider the formula $0 : (\Diamond\Diamond p \wedge \mathsf{A}(\neg p)) \vee (\Diamond\Diamond p \wedge \Box(\neg p))$, developed in the example of figure 6.4. In this example, the Universal modality is valid only in one branch (the branch corresponding $(\Diamond\Diamond p \wedge \mathsf{A}(\neg p))$), causing the branch to be closed. The other branch of the example $((\Diamond\Diamond p \wedge \Box(\neg p)))$, is in fact an open branch. As a result, the tableau should be open. However, if we apply GLOBAL caching directly, without taking care of the Universal Modality, the application of the $(\Diamond)$ and $(\Box)$ rules to formulas in node (7), produce the same set of formulas of node (5). This way, instead of creating a new node, we just add an edge from node (7) to node (5). Finally, given that node (5) has already the status unsatisfiable, after the propagation phase, the status of node (7) (and so node (3) and the root) is set to unsatisfiable, resulting in a closed tableaux.

6.2.3 Conclusion and direction for further work

The discussion of the above issues shows that implementing the GLOBAL caching optimization for the case of the Hybrid Logic $\mathcal{H}(@, \mathsf{A})$ is not an easy task. Special care should be taken for nominals and the Universal modality. Besides, the implementation of this approach for the hybrid theorem prover HTab, would be too hard to be done completly and satisfactorily within the scope of this work, because it requires a different core algorithm and main data structure. However, GLOBAL caching has shown to be a new very effective approach and it would be an interesting future work to implement this approach for Hybrid Logic.

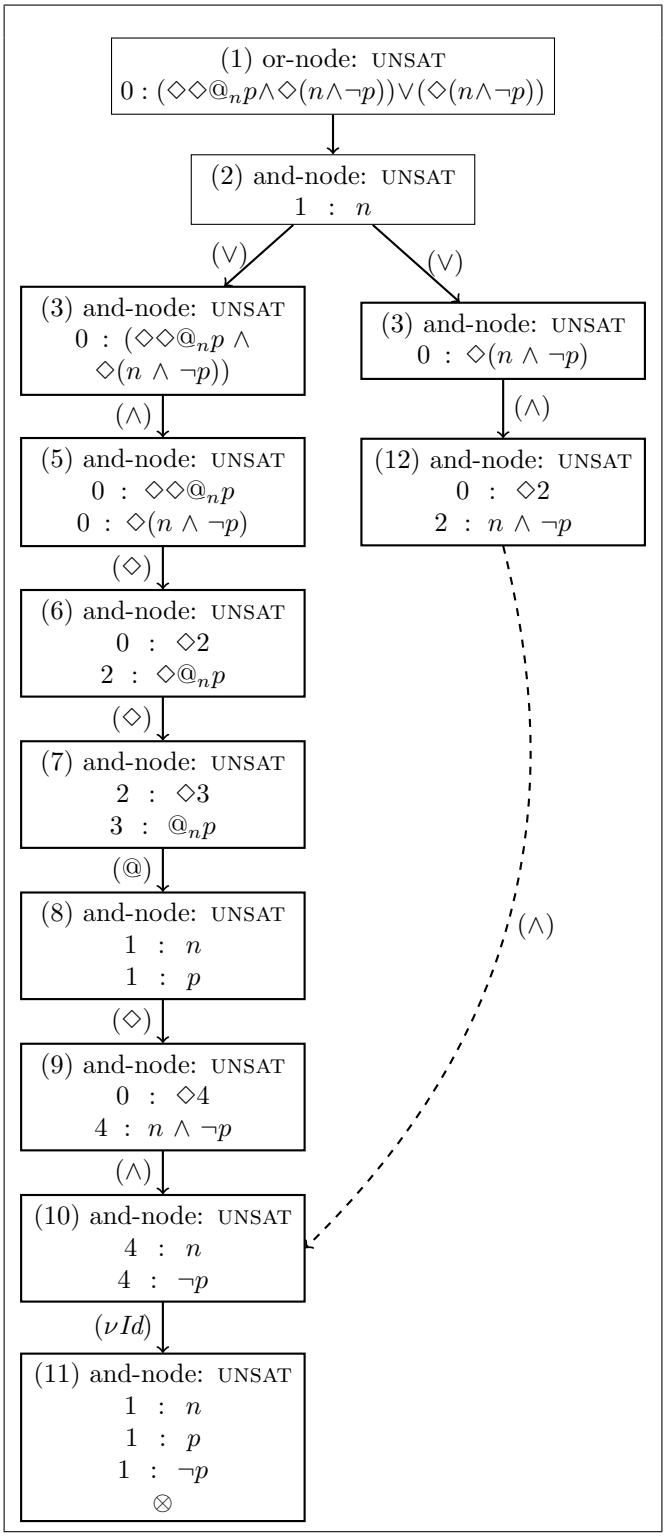


Figure 6.3: Wrong Tableau applying GLOBAL caching without taking care of the nominals

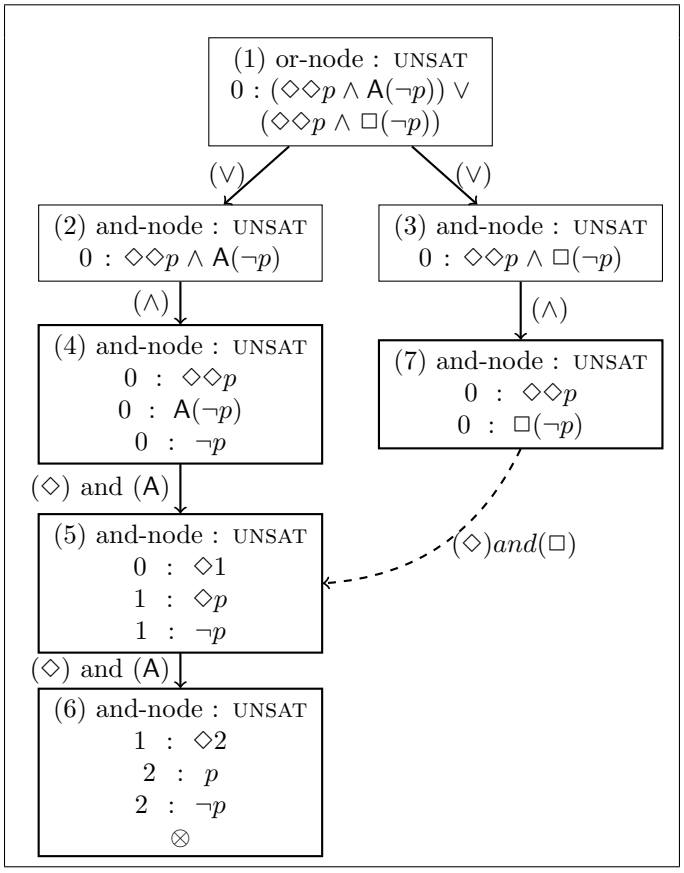


Figure 6.4: Wrong Tableau applying GLOBAL caching without taking care of the universal modality

Chapter 7

Evaluation

In the previous sections we explained the different approaches to caching existing for Description Logics. We also showed how some of them could be implemented for the Hybrid Logic covered in Thesis. In particular we developed an implementation of UNSAT Caching. In this section we provide an evaluation of the UNSAT caching implementation applied to HTab. The evaluation consists in two parts: First a comparison between different versions of the HTab theorem prover, where the versions differ in the optimizations incorporated:

- a first version incorporating all the default optimizations in HTab (i.e. semantic branching, unit propagation and backjumping),
- a second version using all the default optimizations plus UNSAT caching,
- a third one which does not incorporate any optimization at all,
- and finally, a fourth version without any other optimization than UNSAT caching.

The second part consists mainly in evaluating the cache system implemented for HTab, which is compared against the cache system used in Spartacus (the only hybrid theorem prover which implements this optimization).

7.1 The chosen testing framework: GridTest

To make the evaluations we use GridTest, a framework for testing automated theorem provers using randomly generated formulas. GridTest can be used to run tests locally, in a single computer, or in a computer grid. It automatically generates a report as a PostScript file. It can compile statistics provided by the provers (e.g., running time, number of applications of a particular rule, open/closed branches, etc.), and produce graphs generated using GnuPlot.

The framework currently uses hGen ([Areces and Heguiabehere, 2003]) to generate random formulas in a conjunctive normal form: each formula is a conjunction of disjunctive clauses. Since each disjunctive clause can be seen as an additional constraint on satisfying models, random formulas with a small number of clauses tend to be satisfiable while a large enough number of random clauses will be unsatisfiable. GridTest allows us to generate formulas with an increasing number of disjunctive clauses. This way generating tests that start with formulas with a high chance of being satisfiable, and progressively obtain formulas with a high chance of unsatisfiability, going through the point where the probabilities of the formulas being satisfiable and unsatisfiable are roughly the same. Formulas at this point tend to be difficult for most provers. Of course, the precise number of clauses needed to

reach this point varies depending on other parameters, such as the number of proposition symbols, the number and kind of modalities, the maximum modal depth, etc. hGen allows setting those parameters also.

Then a test run using GridTest would be as follows:

1. set the parameters, and generate the random formulas
2. run the provers under test on each of the random formulas generated, using a fixed time limit per formula
3. collect data of interest about each run (execution time, answer, number of rules fired if available, etc.) and plot it for comparison

Of course, this experiment is not statistically relevant by itself (because the input formula used in each data point has been generated randomly). However, by repeating it a sufficiently large number of times (or, equivalently, using batches of formulas sufficiently large for each data point) and using a statistical estimator on the sampled data (e.g., average, median, etc.) statistically relevant results can be obtained.

7.2 Evaluations

7.2.1 First Part: General Evaluation

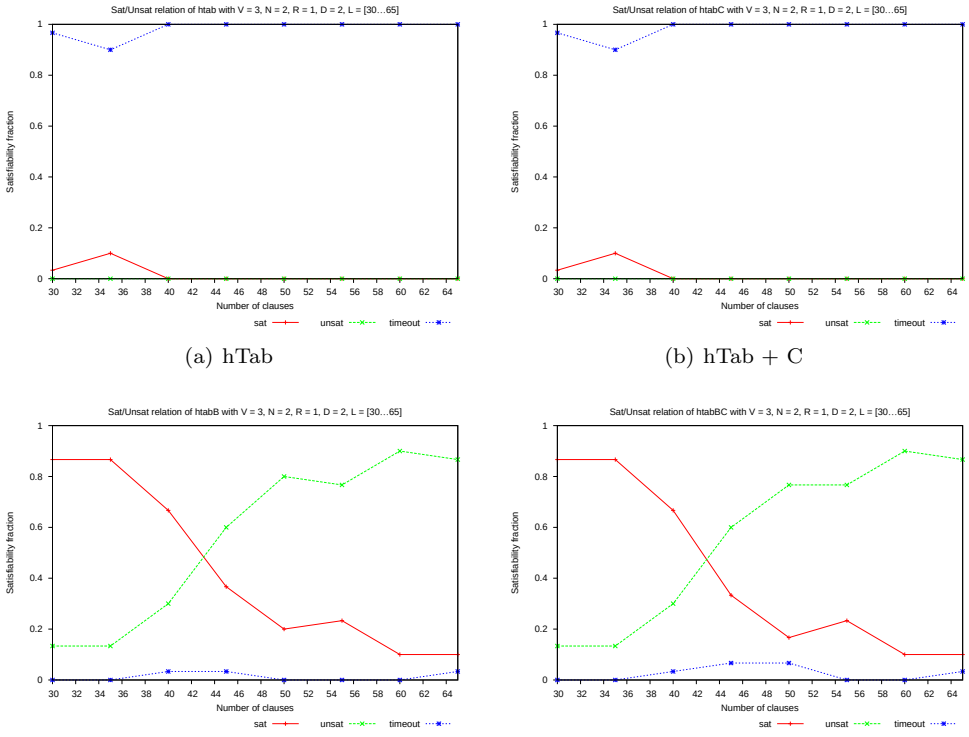
Test Description For the first part of the evaluation phase, we compared the performance of HTab 1.4.98.1 in four cases, on hybrid formulas that contain 3 propositional symbols, 2 nominals, 1 relational symbol, 1 global modality symbol, and modal depth of 2. We go from formulas of size 30 to formulas of size 65. The compared cases are:

- HTab with the default optimizations (i.e. semantic branching, unit propagation and backjumping),
- HTab with the default optimizations plus UNSAT caching,
- HTab without any optimization,
- HTab without the default optimizations, but using UNSAT caching.

Platform:	Architecure: i686 Operating system: Linux 2.6.22-14-generic
Parameters:	Test id: testHTabCaching Propositional symbols (#,freq): (3, 2) Nominals (#,freq): (2, 1) State variables (#,freq): (0, 0) Relations: 1 Max. (global) depth: 2 Diamonds (depth, freq): (2, 2) At (depth, freq): (0, 0) Downarrow (depth, freq): (0, 0) Inverse modality (depth, freq): (0, 0) Universal modality (depth, freq): (1, 1) Batch size: 40 Range of clauses: [30...65] Step: 5 Timeout: 60 seconds

Provers:

HTab	Version 1.4.98: without optimizations
HTabC	Version 1.4.98: without optimizations, plus UNSAT caching
HTabO	Version 1.4.98: with all default optimizations
HTabOC	Version 1.4.98: with all default optimizations, plus UNSAT caching



(a) hTab

(b) hTab + C

(c) hTab + O

(d) hTab + O + C

Figure 7.1: Satisfiability fraction for the various instances of hTab: with no optimizations, with no optimizations but with caching, with optimizations and with optimizations plus caching

Satisfiability fraction : The percentage of satisfiability of the input formulas can be seen on Figures 7.1. In all cases we see that we go from mostly satisfiable formulas to mostly unsatisfiable ones, and we can see that for the cases not including the default optimization there is a very high percentage of timeouts. As it in the general case, the hardest formulas are in the area of maximum uncertainly, where the percentage of satisfiable and unsatisfiable formulas is roughly the same.

Timings The cpu time (in seconds) for each version of the prover are shown in Figure 7.2.

Maximum number of Cache Hits: Figure 7.3 shows the maximum number of cache hits found in each version of the prover. Obviously, the only two versions which can be evaluated in this graphic are the ones including the caching optimiza-

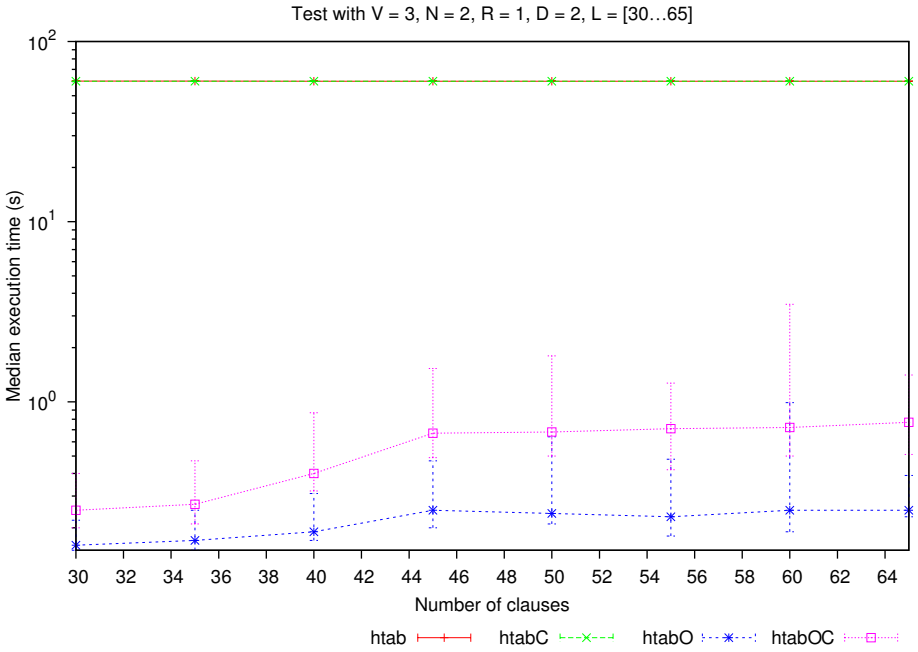


Figure 7.2: Compared execution time

tion. However, because most of the answers for the case of HTab with Caching and with no other optimizations were Timeout, it didn't report any cache hit.

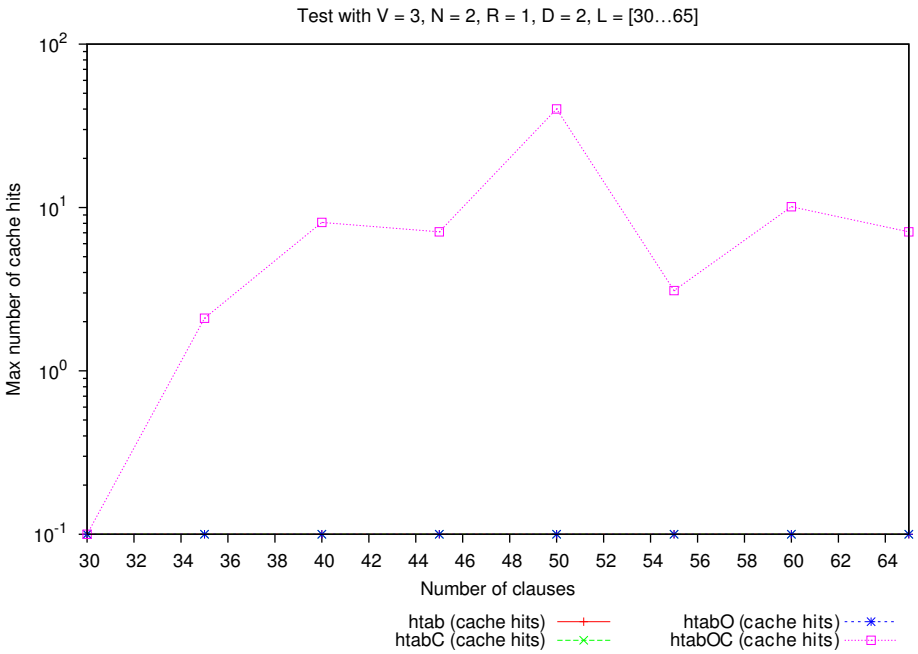


Figure 7.3: Maximum number of cache hits

Analysis of the provers responses : During the tests, no inconsistent answers were detected. That is, given a formula, the answer of the provers was either SAT or UNSAT in both cases, or SAT/UNSAT for one of them and TIMEOUT for the other.

With respect to the performance evaluation, we can see that there is no positive effect with the use of the UNSAT caching optimization. First of all, we should keep in mind that tests done are based on random formulas, and may be in a real world setting, some patterns are repeated more frequently, causing this kind of optimizations to be more significant.

Second, the storage and maintenance of the UNSAT cache are very expensive. If the proportion of cache hits found is not big enough, this can result in a significant overhead.

Finally, apart from the overhead problem, this lack of improvement may be because of a bad interaction between caching and backjumping: since the dependency set calculated when using caching can contain irrelevant branching points, caching can possible limit the effectiveness of backjumping.

7.2.2 Second Part: Cache System Evaluation

Test Description

For the second part of the testing phase, we focus the evaluation on the cache system performance of HTab. For that, we compared the number of cache hits found by HTab against the number of cache hits found by Spartacus. As we mention in chapter 3, Spartacus is the only hybrid theorem prover that currently does caching. It features a number of optimizations, among which is a restricted way of UNSAT Caching that only allows to cache nominal-free unsatisfiable sets of formulas.

For this tests the comparison was done on hybrid formulas that contain 2 propositional symbols, 3 nominals, 1 relational symbol and modal depth of 2. We go from formulas of size 10 to formulas of size 60. And the platform used is the same as for the previous tests.

Parameters:

Test id: testHTabCaching_Spartacus
Propositional symbols (#,freq): (2, 1)
Nominals (#,freq): (3, 1)
State variables (#,freq): (0, 0)
Relations: 1
Max. (global) depth: 2
Diamonds (depth, freq): (2, 1)
At (depth, freq): (0, 0)
Downarrow (depth, freq): (0, 0)
Inverse modality (depth, freq): (0, 0)
Universal modality (depth, freq): (0, 0)
Batch size: 99
Range of clauses: [10...60]
Step: 1
Timeout: 5 seconds

Provers:

HTab	Version 1.4.98: without optimizations
HTabC	Version 1.4.98: without optimizations, plus UNSAT caching
Spartacus	Version: 1.1 with UNSAT caching

Cache System Evaluation Figure 7.4 shows the maximum number of cache hits found in HTab (including all the optimizations plus UNSAT caching), and Spartacus.

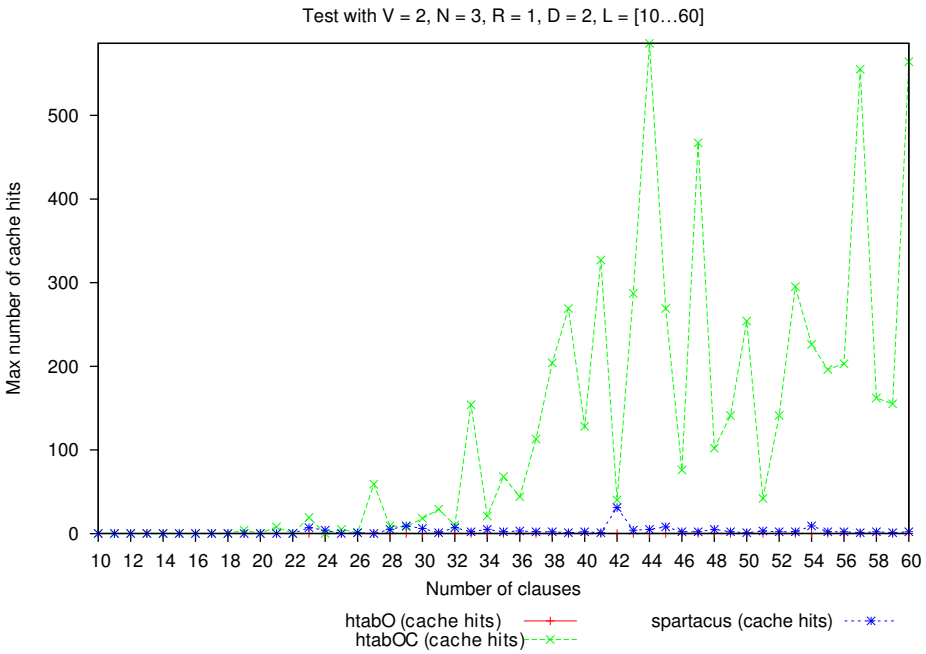


Figure 7.4: Maximum number of Cache hits found in HTab and Spartacus

Analysis of the provers responses Again, no inconsistent answers were detected during the tests.

As we said previously, the intention of this part of the tests is not to evaluate the provers' performance, but just to evaluate the performance of their cache systems (i.e. whether the provers actually find cache hits). For that, we compared the number of cache hits found by HTab against those found by Spartacus. Figure 7.4 shows this comparison.

We can see from this comparison that HTab's cache system behaves better than Spartacus' one: for complex formulas, with HTab we found an important maximum number of cache hits, while with Spartacus this number is almost zero.

However, this cache hit plot is not necessarily a performance measure: a lot of cache hits mean that the cache system works, but what we want is a faster prover. That Spartacus has less cache hits says nothing about the prover's performance. In fact, if we have a look at Figure 7.5, comparing the execution times for Spartacus and HTab (with and without caching[1]), we can notice that Spartacus' performance is best. The question is: does Spartacus actually need the optimization or does it take care of everything beforehand?

7.2.3 An example

We conclude this chapter by showing a very simple example where the optimization works for the case of HTab and not for the case of Spartacus. The example can be

[1] In this case both versions of HTab include all the default optimizations

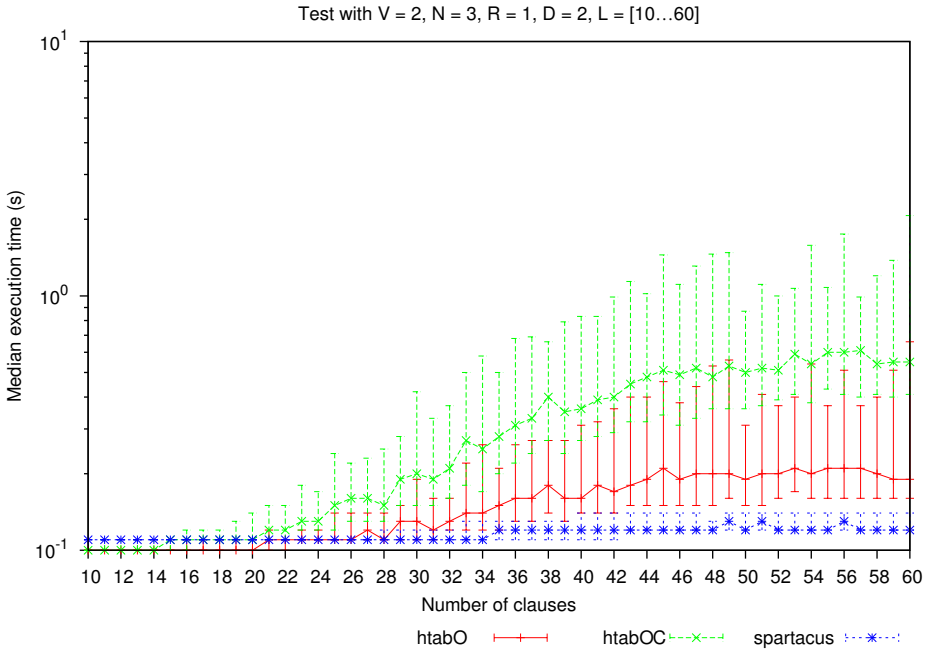
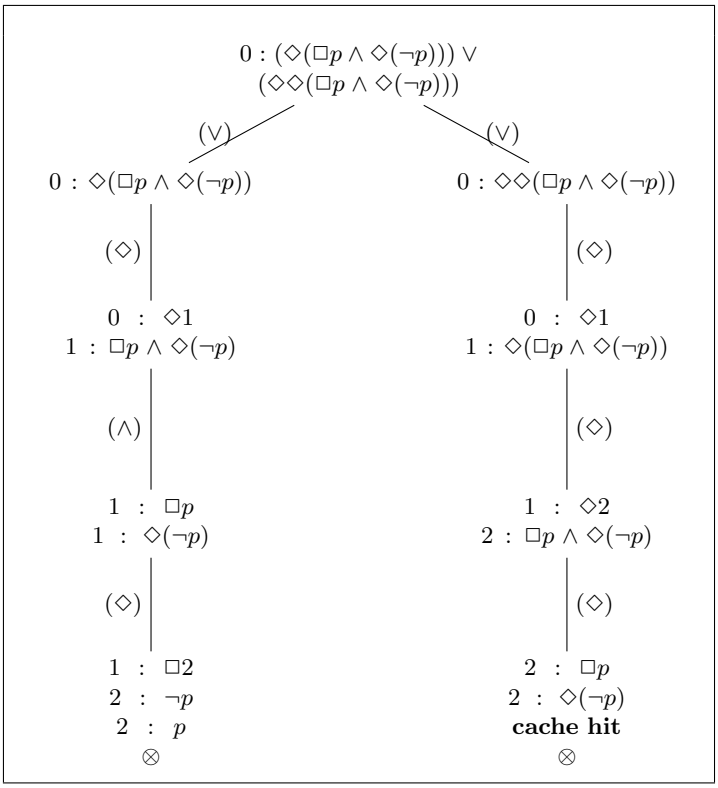


Figure 7.5: Compared execution time between HTab and Spartacus

seen in Figure 7.6.



UNSAT Cache:

$M[T_1] \rightarrow [0, 1, 2]$

Description	Index
$\Diamond\neg p$	0
$\Box p$	1
$2 : \Box p \wedge \Diamond(\neg p)$	2

Figure 7.6: Working example of UNSAT caching

Chapter 8

Conclusion

The goal of this thesis was to investigate the caching optimization in the context of Hybrid Logics. We restricted our attention to basic Hybrid Logic enriched with the universal modality, as Hybrid Logics containing the down-arrow binder are known to be undecidable. In particular, we wanted to see if the optimization could be useful for the hybrid theorem prover HTab.

For that, we first investigated the different approaches to caching already developed for the case of Description Logics, evaluating in each case whether or not they could be implemented for the case of Hybrid Logics.

We focused mainly in UNSAT caching, and we designed an approach able to work with formulas including nominals and the global modality. We also showed some examples which supported most of the decisions taken in the development of the approach.

Then we implemented the UNSAT caching approach for the hybrid theorem prover HTab. The implementation was done taking into account the developed theory and the already existing structures in the prover's implementation. This means that in order to implement the optimization, first we had to understand the main algorithms and structures of HTab, and then do the implementation taking into account these structures.

For the other approaches to caching, we provided examples showing why MIXED caching could not be implemented in the calculus covered in this thesis, and we introduced an idea of the problems that could arise in the case of GLOBAL caching.

Finally, once the implementation of the UNSAT caching approach for HTab was concluded, we evaluated not only its impact in the theorem prover, but also the effectiveness of the implemented cache system. For that we carried out two evaluations:

- For the first tests, we compared HTab's performance with and without the optimization;

- For the second tests, we compared the number of cache hits found by HTab against the number of cache hits found by Spartacus, another hybrid theorem prover implementing the optimization.

Although the first evaluation was not satisfactory and we only noticed negative effects on the performance of the prover with the use of this optimization; the second one showed that the implemented caching system actually works. In fact is the first implementation of the optimization for Hybrid Logics that caches nominal formulas.

8.1 What we have learned so far:

Summing up, this work involved:

- Understanding the prefixed tableaux algorithm for Hybrid Logics;
- Getting to know about the state of the art of the caching optimization in the fields of Description Logics and Hybrid Logics;
- Developing an approach for the optimization which works for the basic Hybrid Logic enriched with the global modality;
- Understanding the hole implementation of the hybrid theorem prover HTab;
- Learning how to program in the functional programing language Haskell (in which HTab is implemented);
- Implementing the UNSAT caching optimization for HTab;
- Getting to know how to run and use the Hybrid Logic test suite GridTest;

8.2 What is left for future work:

As we showed in chapter 7, the UNSAT caching approach introduced no positive effect on the performance of the hybrid theorem prover HTab. In fact, probably because of the high overhead required in order to handle the cache, the effect of this optimization tends to be negative. A first short term future work should investigate the exact reasons for this to happen. We could, for example, start by trying to reduce the negative interaction between backjumping and UNSAT caching, by calculating a better approximation for the dependency sets. A second one, could be to understand the reasons for the differences in number of cache hits between Spartacus and Htab. Obviously, this last work would involve understanding also exactly how does the cache system for Spartacus works.

There are also things to be done in the long term. To start with, we still did not implemented an approach to GLOBAL caching for the case of Hybrid Logic. With this respect, we just presented some of the problems that arise when dealing with nominals or the universal modality. But the presented work leaves room for further investigation along this topics, specially when it is a new topic that seems to work particularly well in the case of Description Logics, and that is becoming very important in the field.

Finally, future work could also investigate further optimizations that could be applied to HTab. Considering the evaluations done in chapter 7, we could try to investigate those optimizations implemented in Spartacus and in HTab, like pattern-based blocking, or lazy branching.

Appendix A

Source Code Summary

We implemented the UNSAT caching optimization for the hybrid theorem prover HTab. The implementation was done in functional programming language Haskell.

In this appendix we provide the source code for the main algorithms of our implementation. The code for the complete implementation can be accessed by downloading the development version of HTab, available at HTab's web page (`http://code.google.com/p/intohylo/`).

Following we will present HTab's relevant structures required for our implementation. Explaining all HTab modules is out of the scope of this thesis.

A.1 Branch.hs module

As explained in chapter 2, the main Structure used in HTab is a Branch, and can be find in the Branch.hs module. In order to implement the optimization we augmented this structure with two new fields:

```
data Branch = Branch
 {...
  branchTrueForms :: BranchTrueForms, --to keep track of all formulas
                                      --true in the branch
  prevPref :: PrevPrefixes,           --To keep track of the prefixes
                                      --true at b-b1, where b is the
                                      --current branch, and b1 is prev(b)
                                      --as explained in chapter 4
  ...
  }
```

Another important structure in this module is a State Monad used to keep track of the computations done on the branch. All the information involving computations that depend on and modify some internal state, is kept in the data structure BranchData. This is the UNSAT cache.

```
type BranchMonad a = StateT BranchData (StateT Statistics IO) a
data BranchData = BranchData { branch_info :: BranchInfo,
                               branch_clp :: CmdLineParams,
                               branch_path :: [Int],
                               timeout_signal :: TimeoutSignal,
                               -------UNSAT cache info--------
                               unsat_cache :: UCache,
                               disjunctPrefixes::DisjunctPrefixes}
```

The field *disjunctPrefixes* is required to implement the fourth conditon in the UNSAT cache operation (as explained in chapter 4).

Bellow we present the structure used to represent the UNSAT cache. As we implemented two approaches for the cache (bit matrices and list based approach),

both approaches are included in the structure. However, when running the reasoner, if we want to use the caching optimization, we have to choose only one them (the other will be deactivated).

```
data UCache = UCache { matrix :: UCMatrix,      --the bit matrix
                       listsList :: UCList,     --list approach
                       descrip_matrix :: UCMap, --the mapping structure
                       current_index :: Int,
                      --fields used for the bit matrix approach--
                       current_row :: Int,
                       max_row :: Int}
      deriving (Show)
```

The other fields in the structure are:

- The mapping structure used by both approaches to map each subterm into an index
- The current index, also used by both approaches to update the mapping structure.
- Fields required by the bit matrix approach in order to optimize the implementation.

A.2 UnsatCache.hs module

This module contains the main functions implemented for the optimization. In order to provide the reader with a clearer idea of the implementation, we won't include the hole module, but only the main functions implemented.

A.2.1 Update Functions

update_cache is the function in charge of updating the cache. Given the chosen approach (bit matrix or list based), an input prefix and the point in the tableau algorithm where this function was called from [1], it updates the cache accordingly. This last parameter received by the function, called *CachingInstance*, is used to implement the second condition of the update UNSAT cache operation (see chapter 4). The update works as follows: first it verifies if the prefix is a valid prefix to be cached, i.e. if it doesn't appear in the lowest open ancestor of the clashing branch (third condition of the update UNSAT cache operation), and if it is ancestor of each clashing prefix (fourth condition of the update UNSAT cache operation). If it verifies both conditions then the formulas of this prefix are added to the cache. Then the function calls recursively with the ancestor of the input prefix, until we find an invalid prefix or it has no ancestor.

```
update_cache :: Caching -> Prefix -> Branch -> CachingInstance
                -> BranchMonad BranchData
------------------------------------------------------------------
--When called from a clash, don't add into the cache the info
--from the clashing prefix
------------------------------------------------------------------
update_cache approach pr1 br Cclash =
 do bd <- get
    let invalid_prefixes = nub $ prevPref br
    let new_disPr =
        del_pref_disjunctPrefixes br pr1 (disjunctPrefixes bd)
    case Map.lookup pr1 (prefParent br) of
          Nothing -> do put    bd{disjunctPrefixes = new_disPr}
                        return bd{disjunctPrefixes = new_disPr}
          Just p -> if not $ elem p invalid_prefixes
                    then do let new_uc =
```

[1] which can be either after a clash or after backtracking from a disjunction, when all disjuncts are closed branches

```
                                    update_cache_prefixes approach
                                                          p
                                                          br
                                                          unsat_cache bd
                                                          invalid_prefixes
                               put    bd{unsat_cache = new_uc,
                                         disjunctPrefixes = new_disPr}
                               return bd{unsat_cache = new_uc,
                                         disjunctPrefixes = new_disPr}
                        else do put    bd{disjunctPrefixes = new_disPr}
                                return bd{disjunctPrefixes = new_disPr}

-------------------------------------------------------------------
--When called after backtracking from the application of a disjunct
--rule, add the info from the first prefix
-------------------------------------------------------------------
update_cache approach pr1  br Cdisjunct =
 do bd <- get
    let pr = getUrfather br $ DS.Prefix pr1
    let add_cache = search_disjunctPrefixes pr1 (disjunctPrefixes bd)
    let invalid_prefixes = nub (prevPref br)
    if add_cache && (not $ elem pr1 invalid_prefixes )
     then
       do let n_uc = update_cache_ approach pr br (unsat_cache bd)
          case Map.lookup pr1 (prefParent br) of
                Nothing -> return bd{unsat_cache = n_uc}
                Just p -> if not $ elem p invalid_prefixes
                             then do let new_new_uc =
                                           update_cache_prefixes approach
                                                                 p
                                                                 br
                                                                 n_uc
                                                        invalid_prefixes
                                     put    bd{unsat_cache = new_new_uc}
                                     return bd{unsat_cache = new_new_uc}
                             else do put    bd{unsat_cache = n_uc}
                                     return bd{unsat_cache = n_uc}
      else return bd
```

In order to obtain the UNSAT cache modifications, the *update_cache* function makes use of the following functions:

The function *update_cache_prefixes* goes through all the input prefix ancestors, calling in each step the function *update_cache_*, which, given an input prefix, updates the UNSAT cache accordingly. It returns the UNSAT cache with all the modifications.

```
-------------------------------------------------------------------
--Calls the function to update the cache with the input prefix and
--calls recursively to all the ancestors of the input prefix.
--Returns the UNSAT cache with all the modifications
-------------------------------------------------------------------
update_cache_prefixes:: Caching -> Prefix -> Branch -> UCache-> [Prefix]
                        -> UCache
update_cache_prefixes approach pr br uc notvps=
 let u_pr   = getUrfather br $ DS.Prefix pr
     new_uc = update_cache_ approach u_pr br uc
 in case Map.lookup pr (prefParent br) of
       Nothing -> new_uc
       Just p -> if not $ elem p notvps
                    then update_cache_prefixes approach
                                               p
                                               br
                                               new_uc notvps
                    else new_uc


-------------------------------------------------------------------
--This function that actually computes the modifications that will
--be done to the UNSAT cache for a given prefix.
-------------------------------------------------------------------
update_cache_ :: Caching -> Prefix -> Branch -> UCache-> UCache
update_cache_ approach pr br uc =
 let -- See what formulas we cache
     trueForms     = DMap.lookupInter pr $ branchTrueForms br
```

```
    univForms1   = fst $ get_univ_forms (univCons br)
    univForms    = map UniversalC     univForms1
    nonUnivForms =
               map NonUniversalC (remove_univ trueForms univForms1)

    noms         = getNoms (trueForms ++ univForms1)
    nominalForms = fst $ get_nominal_forms noms br

    -- formulas to be cached
    cacheForms = nonUnivForms ++ univForms ++ nominalForms

    -- Update the Formula <-> Int BiMap
    (maxIdx, indexes,newMapDesc) =
                           update_ucmap ( descrip_matrix uc )
                                        cacheForms
                                        (current_index uc)

    -- common part for the following
    sorted_indexes = sort $ nub indexes
in
    case approach of
      MatrixCaching ->
       let (n_cr,new_matrix) =
                UCMatrix.update maxIdx
                                (current_row uc)
                                (max_row uc)
                                sorted_indexes $ matrix uc
       in uc{ descrip_matrix = newMapDesc,
              current_index  = maxIdx,
              current_row    = n_cr,
              matrix         = new_matrix}

      ListCaching ->
        uc{ descrip_matrix = newMapDesc,
            current_index  = maxIdx,
            listsList = UCList.update sorted_indexes $ listsList uc}
```

A.2.2 Search Functions

As mentioned in chapter 4, instead of searching for cache hits all over the tableau tree, we can restrict our search to the sets of formulas derived from each rule application. For that, there is a field in the Branch structure which stores all the new prefixes:

```
        incrPrs :: AugmentedPrefixes
```

The function *search_cache* does the search for cache hits and works as follows: First we create, for each prefix in the "AugmentedPrefixes" list, a list of indexes consisting in the non universal formulas true at the prefix, plus the universal formulas true in the branch, plus the formulas true at the nominals present in the previous formulas. Then we compare this list of indexes with the elements of the cache. If the indexes list is superset of some row in the UNSAT cache, we have a cache hit, and the status of the branch is set to unsatisfiable.

The *superset-matching* depends on the approach chosen for the UNSAT cache: bit matrix or list based approach. Each approach is implemented in a separate module. Thus, we implemented two further modules, UCList.hs and UCMatrix.hs, which include the particular implementations of the *superset-matching* and *subset-matching* functions. Those modules are not shown in this appendix because the most important aspects of each case are already explained in chapter 5.

```
-----------------------------------------------------------
--Calls the recursive function search_cache_
--returns an element BranchInfo, with the Clash in case it finds a
--cache hit
-----------------------------------------------------------
search_cache :: Caching -> Branch -> BranchData -> BranchInfo
search_cache caching br bd  =
```

```
                    let not_repeted_incrPrs = nub (incrPrs br)
                    in search_cache_ caching not_repeted_incrPrs br bd

search_cache_ :: Caching -> AugmentedPrefixes -> Branch -> BranchData
                 -> BranchInfo
search_cache_ caching (pr:tail_pr) br bd =
 case search_cache_pr caching pr br bd  of
     b@(BranchClash _ _ _ _) -> b
     BranchOK _              -> search_cache_ caching tail_pr br bd

search_cache_ _ [] _ bd = branch_info bd

-----------------------------------------------------------------
--This function actually does the search for a cache hit for a
--given prefix
-----------------------------------------------------------------
search_cache_pr :: Caching -> Prefix -> Branch -> BranchData
                   -> BranchInfo
search_cache_pr approach pr br bd =
 let  trueForms    = DMap.lookupInter pr $ branchTrueForms br

      univForms1 = fst $ get_univ_forms (univCons br)
      univForms  = map UniversalC univForms1
      nonUnivForms =
               map NonUniversalC (remove_univ trueForms univForms1)

      noms         = getNoms (trueForms ++ univForms1)
      nominalForms = fst $ get_nominal_forms noms br

      -- formulas to be cached
      cacheForms = nonUnivForms ++ univForms ++ nominalForms

      uc  = unsat_cache bd
      c_i = current_index uc
      m_r = max_row uc
      mat = matrix uc
      li = listsList uc
      de  = descrip_matrix uc

  in
     case ( do indexes <- sort . nub <$> lookup_ucmap de cacheForms
               case approach of
                 MatrixCaching ->
                       UCMatrix.superset_matching 0 m_r c_i indexes mat
                 ListCaching   ->
                       UCList.superset_matching  0  indexes li   ) of
      Nothing     -> branch_info bd
      Just newIdx ->
               let new_form_list = get_new_formula_list de newIdx
                   dps           = get_dps new_form_list br pr
               in
                 BranchClash br pr dps (neg taut)


-----------------------------------------------------------------
-- Receives the list of indexes of formulas in the cache,
-- and the bidirectional map, and
-- returns the list of formulas corresponding to this list of indexes
-----------------------------------------------------------------
get_new_formula_list :: UCMap -> [Int] -> [UCFormula]
get_new_formula_list inv_desc =
                map (\i -> fromJust $ Bimap.lookupR i inv_desc)

-----------------------------------------------------------------
--Gets the dependency set corresponding to a list of UCFormulas
-----------------------------------------------------------------
get_dps::[UCFormula] -> Branch -> Prefix -> DependencySet
get_dps (UniversalC form: restForms) br pr=
  let dps = get_dps_fU (univCons br) form
  in dsUnion dps (get_dps restForms br pr)

get_dps (NominalC n form: restForms) br pr
 = let ds_n =  (DS.Nominal (showNom n))
       uf_n = getUrfather br ds_n
       dps1 = DMap.lookup uf_n form (branchTrueForms br)
       dps = case dps1  of
                Nothing -> dsEmpty
```

```
                Just d -> d
   in dsUnion dps (get_dps restForms br pr)

get_dps (NonUniversalC form: restForms) br pr
 = let dps1 = DMap.lookup pr form (branchTrueForms br)
       dps = case dps1  of
                Nothing -> dsEmpty
                Just d -> d
   in dsUnion dps (get_dps restForms br pr)

get_dps [] _ _ = dsEmpty

get_dps_fU :: Univ_constraints -> Formula -> DependencySet
get_dps_fU ucs fo
 = case lookup fo $ map switch ucs of
      Nothing -> dsEmpty
      Just dps -> dps
    where switch (x,y) = (y,x)
```

A.2.3 Helper functions used for both the UNSAT cache update and search

```
-------------------------------------------------------------------
--Gets the set of formulas true at the nominals appearing in the
--formulas to be cached
-------------------------------------------------------------------
getNoms :: [Formula]-> [NomSymbol]
getNoms fs = Set.toList $ Set.unions $ map (fst . extractNominals ) fs

-------------------------------------------------------------------
--Gets the formulas true at each of the nominals in the input list
-------------------------------------------------------------------
get_nominal_forms:: [NomSymbol] -> Branch -> ([UCFormula],DependencySet)
get_nominal_forms noms br =
 foldr merge ([],dsEmpty) $ map get_nominal_forms_one noms
 where merge (fs1,ds1) (fs2,ds2) = (fs1 ++ fs2, dsUnion ds1 ds2)

       get_nominal_forms_one n =
        let ur  = getUrfather br $ DS.Nominal (showNom n)
            btf = Map.lookup ur $ DMap.toMap $ branchTrueForms br
            (btf_list,dps) = case btf of
                               Nothing     -> ([], dsEmpty)
                               Just btfSet -> (Map.keys btfSet,
                                   dsUnions $ Map.elems btfSet)
        in (map (NominalC n) btf_list, dps)

-------------------------------------------------------------------
--Gets the universally constrained formulas
-------------------------------------------------------------------
get_univ_forms :: Univ_constraints -> ([Formula],DependencySet)
get_univ_forms ucs = ( map snd ucs, dsUnions $ map fst ucs )


type TrueForms = [Formula]
type UnivForms = [Formula]

-------------------------------------------------------------------
--Removes the universally constraints formulas from the non universal
--formulas true at the prefix being cached
-------------------------------------------------------------------
remove_univ :: TrueForms -> UnivForms -> [Formula]
remove_univ trueForms univ_forms
 = filter (\f -> not $ is_universal f univ_forms) trueForms

is_universal :: Formula -> UnivForms -> Bool
is_universal (A _) _  = True
is_universal form ufs = any (== form) ufs
```

A.2.4 Functions related to the maintenance of the Mapping Structure

The following functions are used in order to access/update the structure used to map subterms into indexes.

```
-----------------------------------------------------------------------
--This function is called only during the update of the UNSAT cache
--It updates the mapping structure when necessary.
--It returns the tuple (current_index, indexes, newMapDesc),
--where current_index and newMapDesc are the updated current_index
--and descrip_matrix fields from the UNSAT cache, and indexes is
--the list of indexes correspondig to the input list [UCFormula]
-----------------------------------------------------------------------
update_ucmap :: UCMap -> [UCFormula] -> Int -> (Int,[Int],UCMap)
update_ucmap descMat fs maxIdx =
 foldr  (\f (currentMaxIdx,currentIdxs,currentMap)
            -> let (newMaxIdx,idx,newMap) =
                                get_index currentMap f currentMaxIdx
               in  (newMaxIdx, (idx:currentIdxs), newMap)
        )
        (maxIdx, [], descMat) fs

-----------------------------------------------------------------------
--This function is used by update_ucmap in order to get the index
--of an input UCFormula. In case the formula is not in mapping
--structure, it is added.
-----------------------------------------------------------------------
get_index :: UCMap -> UCFormula -> Int -> (Int,Int,UCMap)
get_index mapDes f maxIdx =
 case Bimap.lookup f mapDes of
    Just i  -> (maxIdx, i, mapDes)
    Nothing -> let (n_i,new_mapDes) = updateBiMap mapDes f maxIdx
               in  (n_i, n_i, new_mapDes)

-----------------------------------------------------------------------
--Updates the mapping structure in case it is necessary
-----------------------------------------------------------------------
updateBiMap :: UCMap -> UCFormula -> Int -> (Int,UCMap)
updateBiMap mapDes f maxIdx =
  let newMaxIdx  = maxIdx + 1
      new_mapDes = Bimap.insert f newMaxIdx mapDes
  in ( newMaxIdx, new_mapDes )

-----------------------------------------------------------------------
--This function is called only during the search operation of the
--UNSAT cache. Is used just to lookup indexes in the mapping
--structure given a list of formulas. In the case that one of the
--formulas of the list is not in the mapping structure, the function
--returns Nothing
-----------------------------------------------------------------------
lookup_ucmap :: UCMap -> [UCFormula] -> Maybe [Int]
lookup_ucmap descMat fs =
 foldr  (\f mList
            -> case mList of
                 Nothing -> Nothing
                 Just is -> case Bimap.lookup f descMat of
                              Just i -> Just (i:is)
                              Nothing -> Nothing
        )
        (Just []) fs
```

Bibliography

[Allen, 1984] J. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, July 1984.

[Areces and Heguiabehere, 2003] C. Areces and J. Heguiabehere. hgen: A random cnf formula generator for hybrid languages. In *Methods for Modalities 3 - M4M-3*, Nancy, France, September 2003.

[Areces *et al.*, 2003] Carlos Areces, Patrick Blackburn, Bernadette Martinez Hernandez, and Maarten Marx. Handling boolean aboxes. In *In Proc. of the 2003 Description Logic Workshop (DL 2003) (2003), CEUR (http://ceur-ws.org*, 2003.

[Areces., 2003] Carlos Areces. *Logic Engineering: The Case of Description and Hybrid Logics.* PhD thesis, 2003.

[Beth, 1955] Evert W. Beth. Semantic entailment and formal derivability. *Koninklijke Nederlandse Akademie van Wentenschappen, Proceedings of the Section of Sciences*, 18:309–342, 1955.

[Bolander and Blackburn., 2007] T. Bolander and P. Blackburn. Termination for hybrid tableaus. *Journal of Logic and Computation.*, (11):517–554, 2007.

[Brachman and Schmolze, 1985] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[D'Agostino *et al.*, 1999] Marcello D'Agostino, Dov M. Gabbay, Reiner Hhnle, and Joachim Posegga, editors. *Handbook of Tableau Methods.* Springer, 1999.

[Fitting, 1972] Melvin Fitting. Tableau methods of proof for modal logics. 1972.

[Freeman, 1995] Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Philadelphia, PA, USA, 1995.

[Gentzen, 1969] G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969.

[Giunchiglia and Tacchella, 2001] Enrico Giunchiglia and Armando Tacchella. A subset-matching size-bounded cache for testing satisfiability in modal logics. *Annals of Mathematics and Artificial Intelligence*, 33(1):39–67, 2001.

[Goré and Nguyen, 2007a] Rajeev Goré and Linh Anh Nguyen. Exptime tableaux for alc using sound global caching. In Diego Calvanese, Enrico Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Anni-Yasmin Turhan, and Sergio Tessaris, editors, *Description Logics*, volume 250 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2007.

[Goré and Nguyen, 2007b] Rajeev Goré and Linh Anh Nguyen. Exptime tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In *TABLEAUX '07: Proceedings of the 16th international conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 133–148, Berlin, Heidelberg, 2007. Springer-Verlag.

[Goré and Postniece, 2008] Rajeev Goré and Linda Postniece. An experimental evaluation of global caching for $\mathcal{ALC}$ (system description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 299–305, Berlin, Heidelberg, 2008. Springer-Verlag.

[Götzmann *et al.*, 2009] Daniel Götzmann, Mark Kaminski, and Gert Smolka. Spartacus: A tableau prover for hybrid logic. Technical report, Saarland University, 2009.

[Haarslev and Möller, 2000] Volker Haarslev and Ralf Möller. Consistency testing: The race experience. In *Proceedings, Automated Reasoning with Analytic*, pages 57–61. Springer-Verlag, 2000.

[Hoffmann and Areces, 2003] Guillaume Hoffmann and Carlos Areces. Htab: A terminating tableaux system for hybrid logic. System description, TALARIS. INRIA Lorraine., 2003.

[Hoffmann and Koehler, 1999] Jörg Hoffmann and Jana Koehler. A new method to index and query sets. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 462–467, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Horrocks and Patel-Schneider, 1999] Ian Horrocks and Peter F. Patel-Schneider. Optimising description logic subsumption. *Journal of Logic and Computation*, 9:9–3, 1999.

[Horrocks, 1999] I. Horrocks. FaCT and iFaCT. pages 133–135, 1999. FACT is available under the GNU public license at `http://www.cs.man.ac.uk/~horrocks`.

[Massacci and Donini, 1999] Fabio Massacci and Francesco M. Donini. Exptime tableaux for $\mathcal{ALC}$ // rapporto tecnico 32/99. Technical report, Dip Ingegneria Dell'informazione, Siena, Italy, 1999.

[Nguyen and Goré, 2007] Linh A. Nguyen and Rajeev Goré. Optimised exptime tableaux for alc using sound global caching, propagation and cutoffs. Technical report, 2007.

[Nguyen, 2008] Linh Anh Nguyen. An efficient tableau prover using global caching for the description logic ALC. In Gabriela Lindemann et al., editor, *Proceedings of CS&P'2008*, pages 362–373, 2008.

[Patel-Schneider, 1998] Peter F. Patel-Schneider. Dlp system description. In *Collected Papers from the International Description Logics Workshop (DL'98*, pages 87–89, 1998.

[Prior, 1967] A. Prior. *Past, Present and Future.* Oxford University Press, 1967.

[Schild, 1991] Klaus Schild. A correspondence theory for terminological logics. In *Proceedings of the 12th IJCAI*, pages 466–471. Springer-Verlag, 1991.

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

[Sirin *et al.*, 2007] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. volume 2, 2007. Available at `http://pellet.owldl.com/`.

[Smullyan, 1995] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, second corrected edition, 1995. First published in 1968 by.

[Tsarkov *et al.*, 2007] Dmitry Tsarkov, Ian Horrocks, and Peter F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *J. Autom. Reason.*, 39(3):277–316, 2007.

[Tsarkov, 2003] Dmitry Tsarkov. FaCT++ software. 2003. FACT++ is available under the GNU public license at `http://owl.man.ac.uk/factplusplus/`.