Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Spartacus

A Tableau Prover for Hybrid Logic

submitted by

Daniel Norbert Götzmann

submitted
12 January 2009

Supervisor

Prof. Dr. Gert Smolka

Advisor

Mark Kaminski, M.Sc.

Reviewers

Prof. Dr. Gert Smolka
Prof. Bernd Finkbeiner, Ph.D.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement under Oath

I confirm under oath that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum / Date) (Unterschrift / Signature)

Abstract

This thesis presents SPARTACUS, a tableau-based reasoner for hybrid logic with global modalities and reflexive and transitive relations. To obtain termination in the presence of global modalities and transitive relations, SPARTACUS uses pattern-based blocking. To achieve a competitive performance on practical problems, we employ a range of optimization techniques.

After describing the architecture of SPARTACUS, we evaluate the impact of pattern-based blocking and the implemented optimization techniques and heuristics on the performance of the prover. We also compare our system with existing reasoners for modal and description logics.

From the evaluation we conclude that pattern-based blocking is a promising technique that can significantly improve the performance of modal reasoning.

Acknowledgements

I would like to thank Prof. Gert Smolka, who suggested this topic and, thus, gave me the opportunity to work on an interesting subject. I also thank Prof. Bernd Finkbeiner for reviewing this thesis.

I am deeply grateful to Mark Kaminski for dedicating a great amount of time and effort to reviewing and discussing the progress of this project. I very much appreciate the numerous comments and suggestions he has offered, which continually encouraged me to further develop both the implementation and this thesis. It was also him who suggested the name of Spartacus.

Contents

Chapter 1

Introduction

Hybrid logic is an expressive extension of modal logic. Like modal logic, hybrid logic is used to reason about transition systems. Its increased expressive power compared to basic modal logic lies in the ability to refer to individual states by the use of nominals, which makes possible reasoning about state equality.

A typical reasoning problem in modal logic is deciding the satisfiability of formulas. Commonly, this problem is tackled using tableau-based algorithms.

This thesis presents an implementation of a tableau-based decision procedure for hybrid logic with global modalities and reflexive and transitive relations. Our system employs pattern-based blocking in order to guarantee termination.

This chapter gives a brief overview of hybrid logic and existing approaches for deciding satisfiability. Following that, we mention the contributions of our work and outline the organization of the thesis.

1.1 Hybrid Logics

Modal logics [12] are used to reason about transition systems. The basic modal logic contains operators for reasoning about states that are accessible from a given state. References to states in basic modal logic are always relative to the unnamed "current" state. So we can talk about successors of the current state but not about any fixed named states.

Hybrid logic [4] enriches the expressiveness of modal logic by allowing to explicitly refer to individual states. The simplest hybrid logic is obtained by extending basic modal logic with nominals, which can be seen as predicates that are true in exactly one state. Nominals are used to name individual states and allow to reason about state equality.

Most hybrid logics also include a satisfaction operator that is used to specify which property holds on a certain named state. Expressiveness can be increased further by adding operators like global modalities. The universal global modality can express that a property holds on all states. Its dual, the existential modality,

expresses that at least one state exists on which a certain property holds. Further expressive extensions can be obtained by including difference, converse or graded modalities.

Hybrid logics are closely related to description logics [54, 11, 1], which have applications in many fields, including software engineering [66], medical informatics [53], web-based information systems [35], natural language processing [20] and database technology [15].

1.2 Deciding Satisfiability

A typical reasoning problem for modal logics is deciding the satisfiability of a formula, i.e., determining whether there exists a transition system that fulfills all constraints specified by the formula. For computationally deciding satisfiability, several approaches exist.

The most widely used approach are tableau-based algorithms [36]. Given a formula, a tableau algorithm works by inferring constraints on transition systems specified by the formula by applying so-called tableau rules, either returning a representation of a satisfying transition system or proving the formula unsatisfiable. Examples of systems based on the tableau method include the hybrid logic reasoner HTab [31] and description logic reasoners like DLP [52], FaCT++ [64], pellet [58] and RacerPro [28].

A related approach is based on hypertableaux [7], which aim at reducing the nondeterminism involved in tableau algorithms. A reasoner following this approach is HermiT [51]. There also exists the translation-based method, which works by transforming a modal formula into a decidable fragment of first-order logic (FOL) and using a reasoner for FOL in order to decide satisfiability. For instance, MSPASS [38] and Hoolet [8] are reasoners for modal and, respectively, description logics that follow this approach. It is also possible to apply resolution directly on a hybrid formula (without translation), as it is done by the hybrid logic reasoner HyLoRes [2].

Other existing computational approaches include sequent-based, game-based and automata-based methods [36].

Implementing an efficient decision procedure for hybrid logic is challenging. Even in the case of basic modal logic, a naïve implementation of a tableau algorithm will perform very badly. Hence, to get a system of practical value, it is crucial to employ various computational optimization techniques [36]. Further issues arise once support for nominals and global modalities is to be added. In basic modal logic, the transition system constructed by the tableau algorithm has the shape of a tree, enabling the tableau algorithm to proceed in a strictly top-down manner [45]. Once nominals are used, the shape of the transition system can be more complex and more sophisticated strategies are required. Furthermore, in the presence of global modalities or transitive relations, blocking must be used to guarantee termination of the tableau algorithm [44, 36, 40].

1.3 Contributions

In this thesis, we present SPARTACUS, an implementation of a tableau-based decision procedure for hybrid logic with global modalities and reflexive and transitive relations.

SPARTACUS implements pattern-based blocking, a technique proposed by Kaminski and Smolka [42]. They argued that pattern-based blocking can significantly reduce the worst-case size of tableau branches [41] compared to chain-based blocking as described in [13]. One of the key motivations for our work is to test whether pattern-based blocking has a positive effect on performance in practice. This thesis presents our implementation of pattern-based blocking and evaluates its impact on the performance of a practical decision procedure. Three data structures are compared regarding their effectiveness for storing and querying blocked patterns.

To improve the performance of SPARTACUS, several optimization techniques are implemented. In particular, we explore a new technique called *lazy branching*.

SPARTACUS allows for different strategies of tableau rule application, which enables us to evaluate their practical fitness.

1.4 Outline

The following chapter introduces syntax and semantics of hybrid logic with the satisfaction operator and global modalities. It also describes the tableau algorithm SPARTACUS is based on, including the blocking technique used to guarantee termination. Chapter 3 is concerned with the architecture of SPARTACUS. The implementation of pattern-based blocking is described in Chapter 4. Chapter 4 also gives an overview of the implemented optimization techniques. Chapter 5 presents the results of the evaluation of our system. Chapter 6 concludes the thesis.

Chapter 2

Hybrid Logic

This chapter describes the theoretical foundation for a tableau-based decision procedure for hybrid logic.

Section 2.1 explains the syntax we use for hybrid logic as well as its semantics. Section 2.2 describes the basic tableau algorithm SPARTACUS is based on. Section 2.3 explains blocking conditions required for termination. Section 2.4 is concerned with extensions of the tableau algorithm in order to support reflexivity, transitivity and seriality.

2.1 Hybrid Logic with E

This section provides a basic definition of the syntax and semantics of $\mathcal{H}(@, E)$, the hybrid logic with the satisfaction operator and global modalities.

Traditionally, modal logic is regarded as an isolated logic whose semantics is typically defined in terms of Kripke structures [12, 36]. However, modal logic is also known to be a fragment of higher-order logic [22, 23]. Modal operators can be seen as higher-order constants [9, 42]. Accordingly, modal decision procedures can be described using higher-order syntax [29, 42].

We represent modal logic in simple type theory, following the representation in [42]. Terms and types are defined as usual. Unless otherwise stated, we use the letters s, t to denote types.

For a term t and a type τ, the notation $t : \tau$ is used to express that the term t is of type τ.

For types τ_1, τ_2, the functional type $\tau_1\tau_2$ is interpreted as the set of all total functions from the interpretation of τ_1 to the interpretation of τ_2. The notation $\tau_1\tau_2\tau_3$ is used as an abbreviation for $\tau_1(\tau_2\tau_3)$.

We use $\lambda x.t_1$ and t_1t_2 to denote abstraction and application, respectively. The notation $t_1t_2t_3$ is used as an abbreviation for $(t_1t_2)t_3$.

The representation of modal logic in simple type theory makes use of two base types, B and S. The interpretation of B consists of the two truth values, *true*

and *false*. Terms of type B are called *formulas*. The base type S is interpreted as a nonempty set whose elements are called states or worlds.

We will use three kinds of variables. Variables of type S are called *nominal variables* or *nominals*. Variables of type SB are called *propositional variables* and variables of type SSB are called *relational variables*. Unless otherwise stated, the letters x, y, z will denote nominals, the letters p, q will denote propositional variables and the letter r will denote a relational variable.

Intuitively, the logic can be seen as describing properties of graphs. Nominals denote states, which correspond to nodes in the graph and relational variables denote relations between states, which correspond to labeled directed edges in the graph. Propositional variables denote properties on states or nodes.

The following logical constants are used with their usual meaning:

$$\begin{aligned} \top, \bot &: B & \qquad \exists, \forall &: (SB)B \\ \neg &: BB & \doteq &: SSB \\ \wedge, \vee, \rightarrow &: BBB \end{aligned}$$

For the boolean connectives $\wedge$, $\vee$, $\rightarrow$ and the identity $\doteq$, infix notation is used. For instance, $t_1 \wedge t_2$ is written instead of $(\wedge)t_1t_2$. For $\exists(\lambda x.t)$ and $\forall(\lambda x.t)$, the common abbreviations $\exists x.t$ and $\forall x.t$ are used. When convenient, $\dot{x}$ is used as an abbreviation for $(\doteq)x : SB$, i.e., the application of the equality constant $\doteq$ to a single argument x.

Modal syntax is embedded into higher-order syntax by representing modal terms as terms of type SB [9, 42]. This representation requires lifted versions of the boolean connectives that take terms of type SB as arguments. They are defined as follows:

Definition 2.1 (Lifted boolean connectives).

$$\begin{aligned} \dot{\neg}px &= \neg(px) & \qquad \dot{\neg} &: (SB)SB \\ (p \mathbin{\dot{\wedge}} q)x &= px \wedge qx & \dot{\wedge} &: (SB)(SB)SB \\ (p \mathbin{\dot{\vee}} q)x &= px \vee qx & \dot{\vee} &: (SB)(SB)SB \end{aligned}$$

In addition, the modal constants $\langle\_\rangle$, $[\_]$, E, A, and @ are used, which are defined as follows:

Definition 2.2 (Modal constants).

$$\begin{aligned} \langle r\rangle px &= \exists y.rxy \wedge py & \qquad \langle\_\rangle &: (SSB)(SB)SB \\ [r]px &= \forall y.rxy \rightarrow py & [\_] &: (SSB)(SB)SB \\ Epx &= \exists p & E &: (SB)SB \\ Apx &= \forall p & A &: (SB)SB \\ @ypx &= py & @ &: S(SB)SB \end{aligned}$$

The diamond ($\langle\_\rangle$) and box ($[\_]$) operators are used to reason about the successors of a state with respect to some accessibility relation r. A formula of the form rxy is called an *r-edge* from x to y. If rxy holds, y is called an *r-successor*

of x. The diamond operator asserts the existence of an r-successor that has a certain property. Its dual, the box operator, asserts that all r-successors have a certain property.

The global modalities E (existential modality) and A (universal modality) are used to assert that a certain property must hold on some state or on all states, respectively. The satisfaction operator (@) is used to express that a certain property must hold on a particular named state.

Modal terms are defined as follows:

Definition 2.3 (Modal terms). A term $t : SB$ is *modal* if it has the form

$$t ::= p \mid \dot{x} \mid \dot{\neg} t \mid t \mathbin{\dot{\wedge}} t \mid t \mathbin{\dot{\vee}} t \mid \langle r \rangle t \mid [r]t \mid Et \mid At \mid @xt$$

We apply the convention that $\dot{\vee}$ binds weaker than $\dot{\wedge}$ and $\dot{\wedge}$ is preceded by $\dot{\neg}$, $\langle r \rangle$, $[r]$, E, A and $@x$. For instance, we write $\langle r \rangle p \mathbin{\dot{\vee}} p \mathbin{\dot{\wedge}} q$ for $(\langle r \rangle p) \mathbin{\dot{\vee}} (p \mathbin{\dot{\wedge}} q)$ and $[r]\dot{\neg} p \mathbin{\dot{\wedge}} q$ for $([r](\dot{\neg} p)) \mathbin{\dot{\wedge}} q$.

In the following, it is convenient to assume that modal terms are normalized such that negations occur only immediately before propositional variables and nominals.

Definition 2.4 (Negation-normal form (NNF)). A modal term $t : SB$ is in *negation-normal form (NNF)* if it has the form

$$t ::= p \mid \dot{\neg} p \mid \dot{x} \mid \dot{\neg} \dot{x} \mid t \mathbin{\dot{\wedge}} t \mid t \mathbin{\dot{\vee}} t \mid \langle r \rangle t \mid [r]t \mid Et \mid At \mid @xt$$

Each modal term can be transformed into an equivalent term in NNF in linear time by pushing negations inwards, applying de Morgan's laws and using the duality between the modal operators.

Definition 2.5 (Modal interpretation). A *modal interpretation* $\mathcal{M}$ is an interpretation that

- interprets B as the set $\{0, 1\}$,
- interprets S as a non-empty set,
- interprets $\top$ as 1 (i.e., true) and $\bot$ as 0 (i.e., false),
- gives the logical constants $\top$, $\bot$, $\wedge$, $\vee$, $\rightarrow$, $\exists$, $\forall$, $\doteq$ their usual meaning, and
- satisfies the equations of Definition 2.1 and Definition 2.2.

We use the following semantic notions:

Definition 2.6 (Models, satisfiability and validity). Let $t : SB$ be a modal term.

- A modal interpretation $\mathcal{M}$ *satisfies* t iff there exists a nominal $x \in \mathcal{M}S$ such that $\mathcal{M}tx = 1$
- A modal interpretation $\mathcal{M}$ that satisfies t is called a *model* for t.

- t is *satisfiable* iff there exists a model for t.
- t is *unsatisfiable* iff t is not satisfiable.
- t is *valid* iff for all modal interpretations $\mathcal{M}$ and for all nominals $x \in \mathcal{M}S$ it holds that $\mathcal{M}tx = 1$.

Typical reasoning problems of modal logics include deciding satisfiability or validity of a modal term. Validitity can be reduced to satisfiability since a formula t is valid iff $\dot{\neg}t$ is unsatisfiable.

The set of variables that occur free in a term t is denoted by $\mathcal{V}t$. For a set X of terms, $\mathcal{V}X$ is defined as $\bigcup\{\mathcal{V}t \mid t \in X\}$.

2.2 Tableaux for $\mathcal{H}(@, E)$

One method to decide modal satisfiability is to use a tableau algorithm. The tableau method was invented by Beth [10] and Hintikka [30], with Beth [10] being the first to use the term *tableau*. The general idea behind tableau algorithms is to gradually infer new terms by applying inference rules, called *tableau rules*, to terms that have already been derived. Satisfiability of a modal term t is proven if this approach yields a model for t.

This section explains the tableau algorithm for $\mathcal{H}(@, E)$ as described in [42].

The set of terms that have been inferred at a certain point during tableau construction is called a *tableau branch*, which is defined as follows:

Definition 2.7 (Tableau branch). A *tableau branch* $\Gamma = \{t_1, \ldots, t_n\}$ is a set of terms such that each t_i $(1 \leq i \leq n)$ is either

- a formula of the form tx for some negation-normal modal term t, or
- an edge of the form rxy.

The nominal x occurring in a formula tx refers to the individual state on which the term t holds. Thus, its function is analogous to a prefix used in related calculi [13, 14, 46].

Since, in the following, our formulas containing modal terms will always have the form tx, we will assume the modal operators in t to have a higher precedence than the final application of t to x. For instance, $\langle r\rangle tx$ will be seen as abbreviation for $(\langle r\rangle t)x$.

For each branch Γ, we define a state equivalence relation $\sim_\Gamma$ representing the equational constraints that hold on Γ:

Definition 2.8 (State equivalence relation). Let Γ be a branch. The *state equivalence relation* $\sim_\Gamma$ is defined as the equivalence closure of the relation

$$\{(x, y) \mid \dot{x}y \in \Gamma\} \ \cup \ \{(x, x) \mid x \in \mathcal{V}\Gamma\}$$

$$\mathcal{R}_{\dot\wedge}\ \frac{(t_1 \mathbin{\dot\wedge} t_2)x}{t_1x \qquad t_2x} \qquad \mathcal{R}_{\dot\Diamond}\ \frac{\langle r\rangle tx}{rxy \qquad ty}\ y \notin \mathcal{V}\Gamma \qquad \mathcal{R}_E\ \frac{Etx}{ty}\ y \notin \mathcal{V}\Gamma$$

$$\mathcal{R}_{\dot\vee}\ \frac{(t_1 \mathbin{\dot\vee} t_2)x}{t_1x \mid t_2x} \qquad \mathcal{R}_{\dot\Box}\ \frac{[r]tx \qquad rxy}{ty} \qquad \mathcal{R}_A\ \frac{Atx}{ty}\ y \in \mathcal{V}\Gamma$$

$$\mathcal{R}_{@}\ \frac{@ytx}{ty} \qquad \mathcal{R}_{\doteq}\ \frac{tx}{ty}\ x \sim_\Gamma y,\ t \text{ modal}$$

Γ is the tableau branch to which a rule is applied.

Figure 2.1: Tableau Rules

A tableau calculus for $\mathcal{H}(@, E)$ is defined by the rules in Figure 2.1. A tableau rule specifies which formulas can be inferred from the formulas already contained on a branch Γ. For instance, for a branch $\Gamma \supseteq \{[r]tx, rxy\}$, according to the rule $\mathcal{R}_{\dot\Box}$ the term ty can be derived, yielding an extended branch $\Gamma' = \Gamma \cup \{ty\}$ containing the derived term in addition.

Definition 2.9 (Extensions). A branch Γ' is called an *extension* of a branch Γ if Γ' can be obtained from Γ by finitely many applications of tableau rules defined in Figure 2.1. If additionally $\Gamma' \supsetneq \Gamma$, then Γ' is called a *proper extension* of Γ.

A branch Γ is called *maximal* if there is no branch Γ' such that Γ' is a proper extension of Γ.

The application of a tableau rule to a branch Γ yields one or, in the case of $\mathcal{R}_{\dot\vee}$, two extended branches. Thus, a tableau can be visualized by a tree where the root is labeled with the terms of the initial branch, the edges are labeled with the rules applied and the other nodes are labeled with formulas derived by the corresponding rule applications. Figure 2.2 shows an example of such a visualization.

An important property of a tableau branch is whether it contains obvious contradictions. This notion is formalized as follows:

Definition 2.10 (Closed and open branches). A tableau branch Γ is called *closed* if, for some x, y and p, either

- $px \in \Gamma$ and $\dot\neg px \in \Gamma$, or
- $\dot\neg \dot x y \in \Gamma$ and $x \sim_\Gamma y$.

Otherwise, Γ is called *open*.

Definition 2.11 (Satisfiability of branches). A modal interpretation $\mathcal{M}$ *satisfies* a branch Γ if it satisfies every formula $t \in \Gamma$, i.e., from $t \in \Gamma$ it follows that $\mathcal{M}t = 1$. For a branch Γ, the notion of *satisfiability* is analogous to the notion of satisfiability for single terms.

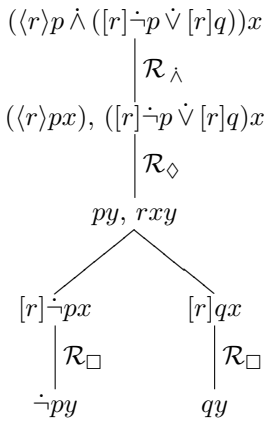

Figure 2.2: Example of a tableau

In particular, closed branches are always unsatisfiable as there exists no modal interpretation that satisfies both a term and its negation.

The tableau rules of Figure 2.1 are *sound* in the sense that a branch Γ is unsatisfiable if and only if all extensions of Γ are unsatisfiable. Furthermore, the tableau rules are *complete* in the sense that repeated application of the rules can reduce every unsatisfiable branch to a set of closed branches [42]. Together with the blocking conditions stated in the following section, which are required for termination, the tableau rules of Figure 2.1 thus yield a decision procedure for satisfiability in $\mathcal{H}(@, E)$. The unsatisfiability of a modal term t in negation-normal form is proven by starting with a branch $\Gamma = \{tx\}$ for some $x \notin \mathcal{V}t$ and successively applying all tableau rules that lead to proper extensions. The term t is satisfiable if tableau construction yields a maximal open branch and unsatisfiable otherwise.

2.3 Blocking

The tableau calculus presented in Section 2.2 is not terminating. For terms of the shape Atx where t contains a diamond or an existential modality, the tableau construction may not terminate. Figure 2.3 shows two examples of non-terminating tableau constructions caused by alternating applications of $\mathcal{R}_A$ and the nominal generating rules $\mathcal{R}_\Diamond$ or, respectively, $\mathcal{R}_E$.

Termination of tableau expansion can be ensured by introducing blocking conditions that restrict the applicability of the rules that introduce new nominals, $\mathcal{R}_E$ and $\mathcal{R}_\Diamond$.

The blocking condition for $\mathcal{R}_E$ is simple. It states that the rule $\mathcal{R}_E$ must not be applied to a formula $Etx \in \Gamma$ if there exists a witness for t, i.e., some y such that $ty \in \Gamma$.

For $\mathcal{R}_\Diamond$, a more complex blocking condition is needed. We use a technique called *pattern-based blocking*, which was introduced by Kaminski and Smolka [42]. Unlike the traditional chain-based techniques [36, 13] it does not require

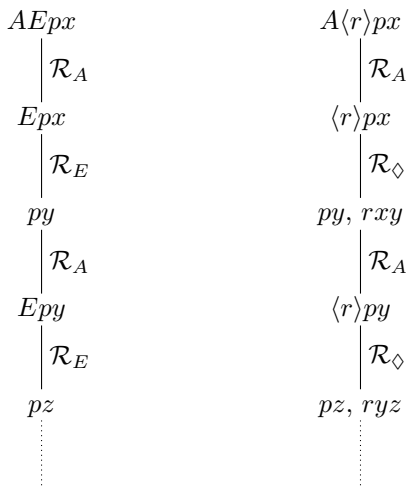

Figure 2.3: Non-terminating tableau constructions

any information about the order in which nominals are introduced to a branch.

Roughly speaking, the idea behind pattern-based blocking is not to apply $\mathcal{R}_\Diamond$ to $\langle r\rangle sx \in \Gamma$ if there is a z such that $\langle r\rangle sx$ can be satisfied by adding to Γ the edge rxz such that this addition does not trigger any box propagations.

The blocking condition described in [42] achieves this by introducing the notion of a *pattern*:

Definition 2.12 (Diamond pattern). The *diamond pattern* of $\langle r\rangle sx$ on a branch Γ is defined as

$$P_\Gamma(\langle r\rangle sx) := \{\langle r\rangle s\} \cup \{[r]t \mid [r]tx \in \Gamma\}$$

Using this definition, the blocking condition for $\mathcal{R}_\Diamond$ can be formulated as follows: The rule $\mathcal{R}_\Diamond$ must not be applied to a formula $\langle r\rangle sx \in \Gamma$ if there are y, z such that $ryz, sz \in \Gamma$ and $P_\Gamma(\langle r\rangle sx) \subseteq P_\Gamma(\langle r\rangle sy)$. In practice, this check can be performed by remembering, at the time we do diamond expansion, the corresponding pattern as expanded.

This blocking condition is subsumed by the blocking condition introduced in [40], which states that $\mathcal{R}_\Diamond$ must not be applied to $\langle r\rangle sx \in \Gamma$ if there is a z such that $sz \in \Gamma$ and for all $\{t|[r]tx \in \Gamma\}$, it holds that $tz \in \Gamma$. Note that, in the latter approach, termination is only guaranteed if the rule $\mathcal{R}_\Box$ is prioritized before $\mathcal{R}_\Diamond$.

In contrast to the blocking condition in [42], it remains open whether the stronger condition in [40] can be implemented efficiently. Hence, we decided to implement the weaker blocking condition described in [42].

Pattern-based blocking does not only guarantee termination in the presence of global modalities but can also reduce search space. For instance, if the tableau algorithm without pattern-based blocking is applied to the branch $\Gamma = \{\langle r\rangle(p \dot\wedge \langle r\rangle t) \dot\wedge \langle r\rangle(q \dot\wedge \langle r\rangle t)\}$, the formula $\langle r\rangle t$ might be tested for satisfiability

$$\mathcal{R}_{refl}\ \frac{[r]tx}{tx}\ r \text{ reflexive} \qquad \mathcal{R}_{trans}\ \frac{[r]tx \qquad rxy}{[r]ty}\ r \text{ transitive}$$

Figure 2.4: Tableau rules for reflexivity and transitivity

twice, depending on the tableau construction strategy. When pattern-based blocking is applied, this is prevented by the blocking condition.

For practical purposes it is beneficial to work with reduced diamond patterns.

Definition 2.13 (Reduced diamond pattern). The *reduced diamond pattern* of $\langle r\rangle sx$ on a branch Γ is defined as

$$P^R_\Gamma(\langle r\rangle sx) := \{s\} \cup \{t \mid [r]tx \in \Gamma\}$$

The blocking condition for $\mathcal{R}_\Diamond$ with reduced diamond patterns states that the rule $\mathcal{R}_\Diamond$ must not be applied to a formula $\langle r\rangle sx \in \Gamma$ if there are y, z such that $ryz, sz \in \Gamma$ and $P^R_\Gamma(\langle r\rangle sx) \subseteq P^R_\Gamma(\langle r\rangle sy)$.

The advantage of using the reduced blocking condition is that it may be applicable more often than the original blocking condition. The reason is that, using reduced diamond patterns, the relational symbols are not relevant and the bodies of diamonds and boxes can be exchanged. For example, unlike the original blocking condition, the reduced blocking condition is applicable for $\langle r_1\rangle tx$ on a branch $\Gamma = \{\langle r_1\rangle tx, \langle r_2\rangle ty, ryz, tz\}$ and for $\langle r\rangle sx$ on $\Gamma = \{\langle r\rangle sx, [r]tx, \langle r\rangle ty, [r]sy, ryz, sz, tz\}$.

2.4 Reflexivity, Transitivity and Seriality

The tableau algorithm presented in Section 2.2 with blocking conditions described in Section 2.3 can easily be extended to enforce reflexivity, transitivity or seriality of relations [19, 42, 46].

Definition 2.14 (Reflexivity, transitivity and seriality). A relation r is

- reflexive iff every state is an r-successor of itself,
- transitive iff for every state x it holds that if y is an r-successor of x and z is an r-successor of y, then z is an r-successor of x, and
- serial iff for every state x there exists an r-successor of x.

Reflexivity and transitivity can be enforced by the additional tableau rules $\mathcal{R}_{refl}$ and $\mathcal{R}_{trans}$ shown in Figure 2.4. Seriality of a relation r can be enforced by conjoining the term $A\langle r\rangle\dot{\top}$ (where $\dot{\top} = \lambda x.\top$) to the term that is to be tested for satisfiability.

While the original definition of pattern-based blocking is applicable for transitive relations without modifications, the reduced blocking condition must be adapted. The reason is that, if r is transitive, a box $[r]t$ that holds on x also holds on all r-successors of x. So, for instance, if r is transitive, $[r]\langle r\rangle p \mathbin{\dot{\wedge}} \langle r\rangle\dot{\neg}p$ is satisfiable but $\langle r\rangle\langle r\rangle p \mathbin{\dot{\wedge}} [r]\dot{\neg}p$ is unsatisfiable, while both terms have the reduced pattern

$\{\langle r\rangle p, \dot{\neg} p\}$. A similar problem occurs when relational variables are exchanged. If r is transitive, $\langle r'\rangle\langle r\rangle p \mathbin{\dot{\wedge}} [r']\dot{\neg} p$ is satisfiable but $\langle r\rangle\langle r\rangle p \mathbin{\dot{\wedge}} [r]\dot{\neg} p$ is unsatisfiable.

To cope with the above problems, we adapt the definition of reduced patterns as follows:

Definition 2.15 (Reduced transitive diamond pattern). For a transitive relation r, the *reduced diamond pattern* of $\langle r\rangle sx$ on a branch Γ is defined as

$$P^R_\Gamma(\langle r\rangle sx) := \{s\} \cup \{t,\ [r]t \mid [r]tx \in \Gamma\}$$

Chapter 3

Architecture of SPARTACUS

This chapter explains the basic architecture of SPARTACUS.

Section 3.1 gives an overview of the implementation and a brief description of practical considerations required to implement the tableau algorithm efficiently. The subsequent sections give a more detailed explanation of the core components needed for deciding basic modal logic. Section 3.5 explains additional infrastructure to support global modalities, state equality, the satisfaction operator as well as reflexive, transitive and serial relations.

3.1 Overview

The implementation is based on the tableau algorithm presented in Section 2.2. It attempts to compute, for any given modal term t, whether t is satisfiable. In order to do this, it tries to find a model for t by constructing a maximal open branch that is an extension of the branch $\Gamma = \{tx\}$ for some $x \notin \mathcal{V}\Gamma$.

The transition from the theoretical description of the tableau algorithm to an efficient decision procedure requires some practical considerations.

One of those considerations concerns the practical application of the rule $\mathcal{R}_{\dot{\vee}}$. It is necessary to check whether at least one of the branches obtained by its application has a maximal open extension. We chose to perform depth first backtracking search, a technique that considers only one branch at a time, as it is done in many logic reasoners [33, 52, 31]. This means, a disjunction $(t_1 \,\dot{\vee}\, t_2)x$ on a branch Γ is handled by selecting one of the disjuncts t_i. Only after the algorithm fails to find a maximal open extension of $\Gamma \cup \{t_i x\}$, the other branch $\Gamma \cup \{t_j x\}$ $(j \neq i)$ is considered.

Another issue concerns the representation of the information that holds on a branch. For theoretical purposes, it suffices to regard a branch as a set of terms. For practical purposes, it is desirable to structure the terms in such a way that they can be accessed efficiently. The general idea is to store terms that pertain to a particular node in the same place and separate from terms that pertain to other nodes [36]. A so-called *node store* (cf. Section 3.3) stores, for each node,

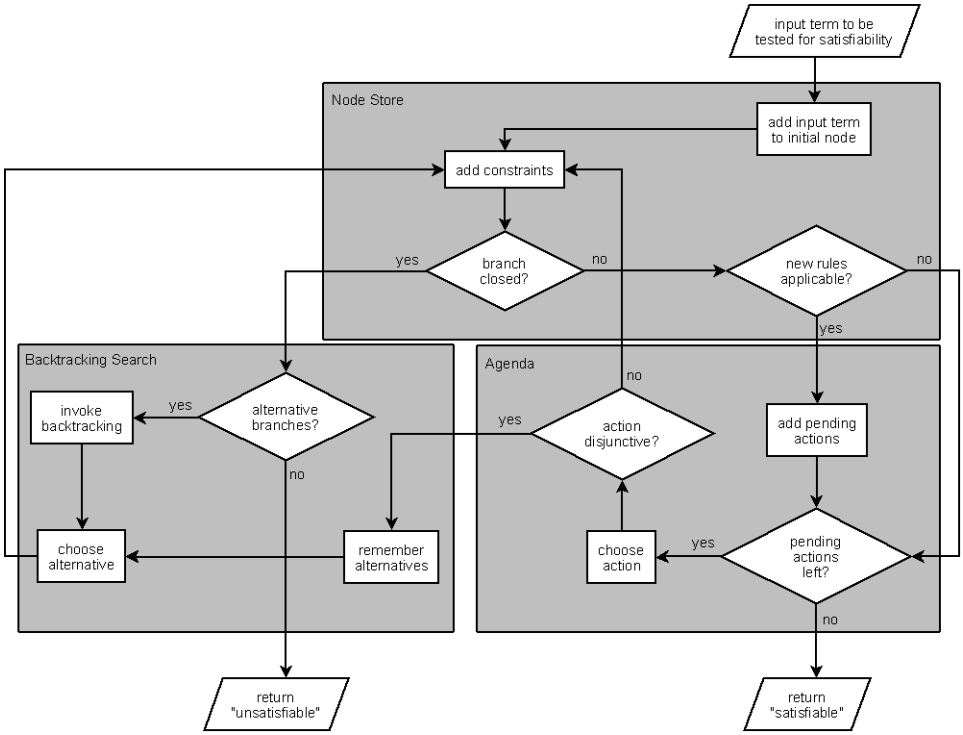

Figure 3.1: Architecture of SPARTACUS

the terms that have been inferred as constraints on that node and the list of its successors with respect to the accessibility relations. Intuitively, the node store stores the transition system determined by the constraints that have been derived by tableau expansion. When a new constraint is inferred, the contents of the node store are changed accordingly. The corresponding operations on the node store include creating new nodes and edges between nodes (e.g., when applying $\mathcal{R}_{\Diamond}$) and adding newly derived terms to existing nodes. When a negation of a term already contained in a node is added to that node, the algorithm continues with the exploration of the next branch.

Another consideration involves the applicability of tableau rules. In theory, a tableau rule is applicable on a branch Γ if it leads to a proper extension of Γ. In practice, one needs an efficient way to determine when a rule becomes applicable. This may happen when a new term is added to a node or when a new node or a new edge is created. In order to keep track of new terms that have been added to a node and for which a rule must still be applied, a technique similar to the *ToDo list architecture* of FACT++ [64] is implemented. The general idea is to use an *agenda* (cf. Section 3.2) to keep track of terms for which a rule must still be applied.

Figure 3.1 sketches how the agenda, the node store and backtracking search interact in order to determine whether a modal term of basic modal logic is satisfiable. Initially, the node store contains a single node. The input term that is to be tested for satisfiability is added to that node and terms for which tableau rules must be applied are added to the agenda. The agenda is processed until

either it becomes empty or there are no alternative branches to be explored after a branch has become closed.

In the latter case, the input is proven to be unsatisfiable as all branches are closed. If, however, the agenda becomes empty, all applicable tableau rules have been applied. Hence, a maximal open branch has been found and a model for the input can be obtained from the contents of the node store.

In order to support global modalities, equality and the satisfaction operator, some extensions to the infrastructure are needed as described in Section 3.5.

3.2 Agenda

The *agenda* is one of the key parts of SPARTACUS. It is used to keep track of pending actions. In particular, it stores disjunctions that have not been considered for branching yet, diamonds and existential modalities not yet expanded, boxes, universal modalities and satisfaction operators whose contents have not been propagated yet and equational constraints between nodes (resulting from positive nominals) that have not been satisfied yet and might later require nodes to be merged.

In general, if the agenda is empty, all tableau rules that lead to a proper extension have been applied. As long as the agenda contains terms, tableau construction continues by selecting a term on the agenda, removing it from the agenda and inferring new terms by applying corresponding tableau rules. For instance, if a box expression $[r]t$ pertaining to a node n is considered, the term t is added to all r-successors of n, corresponding to the application of $\mathcal{R}_{\Box}$.

Note that most terms can safely be removed from the agenda after the corresponding actions have been performed, as additional tableau rules do not become applicable for them later during tableau construction. The only exceptions are box expressions and universal modalities, for which $\mathcal{R}_{\Box}$ or, respectively, $\mathcal{R}_A$ might become applicable if a new node is created after they have been processed and removed from the agenda. Hence, when a new node is created, actions corresponding to $\mathcal{R}_{\Box}$ or $\mathcal{R}_A$[1] are performed immediately. For instance, when processing a diamond expression $\langle r \rangle s$ pertaining to a node n, a new r-successor m of n is created and s is added to m, corresponding to the application of $\mathcal{R}_{\Diamond}$. In this case, if n already contains the boxes $[r]t_1, \ldots, [r]t_k$, their bodies $t_1, \ldots, t_k$ are also added to m immediately, corresponding to the application of $\mathcal{R}_{\Box}$.

The agenda is implemented as a priority queue [16] with the different types of actions having different priorities. The prioritization is flexible. For instance, one can freely choose whether diamond expansions have a higher priority than processing disjunctions. For diamonds and disjunctions the expansion strategy can be refined further, e.g., one can choose whether they should be processed in a FIFO or in a LIFO way.

Since the priority queue is based on a balanced tree map that is ordered according

[1] cf. Section 3.5.1 for a detailed description on how universal modalities are handled.

to the priority of the actions, both adding a new action and finding the next action to be performed can be done in logarithmic time in the size of the priority queue [43].

3.3 Node Store

Another key part of SPARTACUS is the *node store*. It is motivated by the idea of storing information about a node bundled in the same place and separate from information about other nodes.

The node store consists of an array that stores, for each node, a list of successors and a store containing all terms that have been added to the node as constraints during tableau construction. Each node is assigned a unique identifier, which corresponds to its position in the array.

Operations on the node store include creating new nodes and edges (e.g., when applying $\mathcal{R}_{\Diamond}$) and adding newly derived terms to existing nodes. A new node is created in the intuitive way, by extending the array with an additional element. A new r-edge from a node n to a node m is created by inserting the node m into the list of r-successors of n.

For each individual node n, a special data structure is maintained that stores terms that have been added as constraints to n as well as additional information needed for optimizations described in Chapter 4. In the following, we use $\mathcal{L}n$ to denote the set of terms that have been added to n during tableau construction according to the store of n. The event that a term t is found to hold in a node n when $\dot{\neg}t \in \mathcal{L}n$ is called a *clash*.

When a new term t is added to a node, it must be checked whether adding t leads to a clash. In many cases, it is also necessary to add t to the agenda in order to remember that actions corresponding to t must be performed. In particular, when a new term t is added to a node n, the following effects will result:

- If $\dot{\neg}t \in \mathcal{L}n$, backtracking is invoked.
- If t is a propositional literal or a negated nominal, it is added to $\mathcal{L}n$.
- If t is a diamond, box, disjunction or positive nominal, it is added to $\mathcal{L}n$ and inserted into the agenda together with the information that it pertains to the node n.
- If t is a global modality or a satisfaction term, it is inserted into the agenda.
- If $t = t_1 \dot{\wedge} \ldots \dot{\wedge} t_k$ is a conjunction, the procedure is repeated for each t_i $(1 \leq i \leq k)$.

In order to store terms, a special data structure, *term store*, is used. Terms of different syntactic categories are stored in separate stores. For instance, each node contains separate term stores for diamonds and boxes. Thus, terms of a given syntactic category can be retrieved independently from terms of other categories.

A term store is implemented as ordered and balanced binary tree map that stores, for each term, dependency information required for our implementation of dependency-directed backtracking (*backjumping*, cf. Section 4.3). Therefore, adding new terms to a term store and determining whether a term store contains a given term can be done in logarithmic time in the size of the store. If a term that leads to a clash is found, its dependency information, which is necessary to invoke backjumping, can be determined without additional costs.

3.4 Backtracking Search

The typical mechanism for dealing with the non-determinism of the rule $\mathcal{R}_{\dot\vee}$ is to apply a *depth first backtracking search*, exploring one branch at a time [36]. This means, a disjunction $(t_1 \dot\vee t_2)x \in \Gamma$ is processed by selecting one of the disjuncts, let's say $t_1 x$, and continuing with tableau expansion on the branch $\Gamma \cup \{t_1 x\}$. If that leads to a clash, *backtracking* is invoked in order to restore the branch Γ. Then, the tableau expansion continues on the branch $\Gamma \cup \{t_2 x\}$, which contains the alternative disjunct $t_2 x$.

An implementation of backtracking search must keep track of unexplored branches. In order to perform backtracking efficiently, we also need a mechanism to restore the contents of the agenda and the node store as they were before a branching decision.

In the following, we describe how the unoptimized variant of backtracking search is implemented. Many of the optimizations integrated into SPARTACUS aim at improving backtracking search by reducing the search space. Those optimizations are explained in Chapter 4.

3.4.1 Branching Stack

A *branching stack* is maintained in order to keep track of unexplored branches. The branching stack contains, for each branching point, a list of remaining alternatives.

When, according to the agenda, a branching action has to be performed for $t = (t_1 \dot\vee \ldots \dot\vee t_k)$ located in a node n, one disjunct t_i $(1 \leq i \leq k)$ of t is selected and added to n. An item containing the remaining disjuncts $\{t_j \mid 1 \leq j \leq k,\ i \neq j\}$ is added to the branching stack. The so-called *branching depth* is equal to the number of items contained in the branching stack.

If a clash is detected, backtracking is invoked as described in the following subsection. After backtracking is completed, the next alternative is chosen from the list of remaining alternatives on top of the branching stack. This alternative is then added to the corresponding node and removed from the list. If that disjunct is the last alternative of the branching point, the topmost item of the branching stack is removed.

3.4.2 Backtracking

When a clash has been detected, all actions depending on the last branching point must be undone. For this task, two different approaches exist, copying and trailing.

The copying approach works by storing a copy of the state immediately before a branching alternative is chosen. Backtracking is done by restoring the saved state. However, if the branching depth becomes high, many copies have to be stored, leading to high memory consumption and computational overhead for copying states.

The other approach, trailing, has a lower memory consumption because backtracking is performed by undoing those actions that have been performed since the most recent branching point. Therefore, in order to employ trailing, the data structures used during tableau construction must allow to efficiently undo changes.

SPARTACUS uses trailing as the backtracking technique to reduce memory consumption and computational overhead. When backtracking is invoked, terms that have been added to a term store or to the agenda after the branching point to which the system is backtracked have to be removed. Similarly, nodes and edges that have been added to the node store after the corresponding branching point must be deleted.

For most data structures, trailing can be implemented by maintaining a stack that stores, for each branching depth, the information that has been added at that branching depth. Backtracking on a store is performed by removing elements according to the stack. This approach is possible because the branching depth at which an action is performed is always greater or equal than the branching depth of all previous actions.

The agenda, however, needs special consideration because it is the only data structure for which backtracking requires more than removal of information added after a certain branching point. If the agenda is backtracked to a branching depth that lies between the depth of insertion of a term t and the depth of execution of the action corresponding to t, the effects of that action are backtracked as well. However, t, which is the cause of that action, is still present on the branch but, since an action for t has been performed, t is no longer on the agenda. Hence, t must be reinserted into the agenda so that the action corresponding to t is rescheduled. In order to do this, the agenda maintains a *reinsertion stack*. If an action for a term t is executed at a branching depth that is greater than the depth at which t has been inserted, t is added to the reinsertion stack. Thus, it can be reinserted if required after backtracking.

3.5 Extensions

The infrastructure described in the previous sections can handle terms of basic modal logic. In this section, we present how this system is extended to support

global modalities (Section 3.5.1), nominals and the satisfaction operator (Section 3.5.2) as well as reflexivity, transitivity and seriality (Section 3.5.3).

3.5.1 Global Modalities

In order to support global modalities, some extensions are required.

An existential modality Et is expanded by creating a new node n and propagating t to n. In addition, t is added to a global term store that stores all expanded existential modalities. If this store contains the term t at the time Et is scheduled for expansion, a new node for t is not created according to the blocking condition for E described in Section 2.3.

If a universal modality At must be satisfied according to the agenda, t is propagated to all nodes that are contained in the node store at the time of the propagation. In addition, t is added to a global term store that contains all terms that must hold universally. When a new node n is created, all terms contained in this store are propagated to n in order to satisfy the universal modality.

In order to guarantee termination in the presence of global modalities, a blocking technique is needed as described in Section 2.3. We employ pattern-based blocking, whose implementation details are described in Section 4.2.

3.5.2 Nominals

In order to support nominals, the node store contains a *nominal map*, which maps nominals to nodes. The nominal map is initialized by mapping each nominal x occurring in the input term to a node n where $\mathcal{L}n$ initially only contains $\dot{x}$.

During tableau construction, the nominal map is used to determine which node corresponds to a given nominal when satisfaction terms must be handled or equational constraints must be satisfied. A satisfaction term $@xt$ is handled by propagating t to the node corresponding to x according to the nominal map. The remainder of this section is concerned with the more involved task of satisfying equational constraints.

During tableau construction, it can happen that $\dot{x}$ is found to hold on multiple nodes $n_1, \ldots, n_k$. A naïve implementation of the tableau rule $R_{\doteq}$ shown in Figure 2.1 would copy each term t that holds on n_i to hold on n_j ($1 \leq j \leq k$, $i \neq j$), eventually yielding k nodes containing the same information. This redundancy is avoided by merging the information contained in $n_1, \ldots, n_k$ into a single representative node.

Equivalence classes of nodes are represented by disjoint set forests [16]. More precisely, if two nodes m and n are found to be equal, their contents are merged into one of the nodes, let's say m, which becomes the representative of the equivalence class. A forward pointer to m is inserted at the position of n in the node store. This way, the items on the agenda referring to n do not need to be changed because the representative m can be determined by following the forward pointer of n.

Note that, for the procedure to be correct, it is not necessary to merge two nodes immediately after inferring that they contain the same nominal. It might be beneficial to postpone the costly merge operation until further constraints have been derived. Therefore, when the term $\dot{x}$ is added to a node n, the information that $\dot{x}$ must be satisfied in n is added to the agenda in order to schedule merging of nodes depending on the prioritization of the agenda.

How an action for $\dot{x}$ located in n is performed depends on whether n is the representative node for the nominal x. If this is the case, the equational constraint is already satisfied and no further action is required. Otherwise the nodes m and n must be merged in order to satisfy the equational constraint.

In the latter case, one of the nodes, let's say m, is selected to become the representative of the equivalence class. The entry in the node store pertaining to n is replaced by a forward pointer to m. Terms stored in $\mathcal{L}n$ are copied to $\mathcal{L}m$. If a clash is detected in the process, backtracking is invoked. Otherwise, the diamond expressions copied from $\mathcal{L}n$ to $\mathcal{L}m$ must be handled properly. This could be done by adding them to the agenda as usual. However, we chose a different approach that works by adding all successors of n to the list of successors of m. This way, we avoid creating another node for a diamond for which a successor of n has already been created. For a diamond in $\mathcal{L}n$ for which a node has not been created at the time of merging, a successor of m will eventually be created, as it is already on the agenda and m can be determined as representative of n as described above. In order to make sure that all boxes are propagated to all successors, each box contained in $\mathcal{L}m$ after merging is added to the agenda.

3.5.3 Reflexivity, Transitivity and Seriality

Support for reflexivity, transitivity and seriality requires only minor changes.

For reflexivity, the rule $\mathcal{R}_{refl}$ shown in Figure 2.4 is implemented. This means, when a box expression $[r]t$ is added to a node n and r is a reflexive relation, the term t is added to n as well.

In order to handle transitivity, propagation of boxes must be adapted according to the rule $\mathcal{R}_{trans}$ shown in Figure 2.4. When a box $[r]t$ located in n must be propagated to an r-successor m of n and r is a transitive relation, $[r]t$ is added to m in addition to t.

Seriality is handled by initially adding, for each serial relation r, the universal modality $A(\langle r\rangle\dot{\top})$, where $\dot{\top}$ denotes a valid term that causes no effects when it is propagated to a node[2].

[2] $\dot{\top}$ is the term represented by the integer 1 as described in Section 4.1.

Chapter 4

Blocking and Optimizations

This chapter provides information about the optimization techniques implemented in SPARTACUS and describes the implementation of pattern-based blocking.

Section 4.1 explains the internal representation of terms as well as the initial simplification of input terms. Section 4.2 provides detailed information about the implementation of pattern-based blocking.

Sections 4.3-4.6 describe several optimization techniques that aim at reducing the size of the search space, i.e., the amount of branches that have to be examined. Section 4.7 explains caching of unsatisfiable sets of terms.

SPARTACUS allows optimizations to be turned off, except for normalization of terms as described in Section 4.1. If desired, pattern-based blocking can also be disabled. However, as explained in Section 2.3, termination is not guaranteed on all inputs of $\mathcal{H}(@, E)$ if pattern-based blocking is disabled.

4.1 Representing and Normalizing Terms

Much of the data that has to be stored and dealt with during tableau construction pertains to terms. Hence, the efficiency of an implementation depends on a suitable internal representation of terms. In particular, the following properties are desirable:

- Terms should have a compact representation so that they can be stored cheaply.
- Retrieving additional information about a term must be efficient.
- Conjunctions that contain the same conjuncts should have the same representation independent of the order in which they occur. The same applies to disjunctions.
- Terms containing subterms that are obviously valid or unsatisfiable should be simplified accordingly.

$$\begin{array}{lll} \bigwedge\{t, \dot{\neg}t, \ldots\} \mapsto 0 & \langle r\rangle 0 \mapsto 0 & \mathrm{E}0 \mapsto 0 \\ \bigvee\{t, \dot{\neg}t, \ldots\} \mapsto 1 & [r]1 \mapsto 1 & \mathrm{A}1 \mapsto 1 \\ \bigwedge\{0, \ldots\} \mapsto 0 & @x0 \mapsto 0 & \mathrm{A}0 \mapsto 0 \\ \bigvee\{1, \ldots\} \mapsto 1 & @x1 \mapsto 1 & \mathrm{E}1 \mapsto 1 \end{array}$$

Figure 4.1: Simplification of terms

- It should be easy to determine whether two terms are negations of each other.

In general, a representation of hybrid terms that fulfills those properties can be obtained using techniques similar to those described by Horrocks [33] for description logic.

The general idea is to represent terms as integers, which we call *term indices*. This way, less space is required and efficient data structures for integers can be used. The tableau system is analytic, i.e., the tableau branches contain only subterms of the input expression. Hence, it suffices to generate a mapping assigning an integer to each subterm of the input term. The mapping allows for subterm sharing. Different occurrences of the same subterm are always assigned the same integer. The mapping from integers to compact terms, whose subterms have been replaced by the corresponding integers, is stored in an array A. Hence, information about a term can be retrieved efficiently.

In order to be able to quickly determine whether two terms are negations of each other, the negation of a term t is always encoded as an integer that differs from the index of t only in the least significant bit. This means, if i is even then $A[i+1]$ is the negation of $A[i]$. This way, it is possible to detect a clash earlier (e.g., if $\{\langle r\rangle p, [r]\dot{\neg}p\} \subseteq \mathcal{L}n$) and not only on the atomic level (i.e., if $\{p, \dot{\neg}p\} \subseteq \mathcal{L}n$ or $\{\dot{x}, \dot{\neg}\dot{x}\} \subseteq \mathcal{L}n$).

Furthermore, conjunctions and disjunctions are normalized. The conjuncts in conjunctions and the disjuncts in disjunctions are ordered by syntactic category (e.g., propositional variable, diamond, box, ...) and, within the same category, according to the order of their indices. Double occurrences of the same term are eliminated. Therefore, conjunctions and disjunctions containing the same subterms are always assigned the same integer.

In addition, terms are simplified by detecting obvious validity and unsatisfiability. The integers 0 and 1 are assigned to obviously unsatisfiable and obviously valid terms, respectively. If a conjunction contains both a term and its negation, it is obviously unsatisfiable. Therefore, 0 is assigned to it. Similarly, if a disjunction contains a term and its negation, then 1 is assigned to it. In addition, if the immediate subterm of a diamond term is obviously unsatisfiable, the diamond term is also obviously unsatisfiable and 0 is assigned to it. Similar rules are applied to other operators as shown in Figure 4.1. When 0 is propagated to a node during tableau construction, backtracking is invoked. Propagating 1 to a node has no effects.

4.2 Pattern-based Blocking

Pattern-based blocking is a blocking technique for diamond expressions. While the theory behind pattern-based blocking is described in Section 2.3, this section is concerned with the implementation details.

When, according to the agenda, a diamond is due to be considered, we have to check whether the blocking condition is applicable. To do so, a *pattern store* is maintained, which stores the patterns for which a witnessing node exists. Furthermore, for each node, a term store of blocked diamonds is maintained.

Before a diamond $\langle r \rangle s$ located in n is expanded by creating a new node, we check whether a witness already exists for the corresponding pattern. Provided that $\{t_1, \ldots, t_k\} = \{t | [r]t \in \mathcal{L}n\}$, this is the case if the pattern $\{s, t_1, \ldots, t_k\}$ (or, if r is transitive, $\{s, t_1, \ldots, t_k, [r]t_1, \ldots, [r]t_k\}$, cf. Section 2.4) is contained in the pattern store. If so, $\langle r \rangle s$ is not expanded. Instead, $\langle r \rangle s$ is stored in the term store of blocked diamonds of the node n. Otherwise, $\langle r \rangle s$ is expanded as usual and the pattern $s, t_1, \ldots, t_k$ (if r is transitive: $\{s, t_1, \ldots, t_k, [r]t_1, \ldots, [r]t_k\}$) is added to the pattern store, as a witness for this pattern is generated by the diamond expansion.

When a new box $[r]t$ is found to hold in a node n, for each diamond contained in the term store of blocked diamonds of n, we check whether the blocking condition is still applicable. Those diamonds for which this is not the case are removed from the store of blocked diamonds and reinserted into the agenda.

If the input is satisfiable, the information about blocked diamonds is used to compute the complete model. In order to do so, for each blocked diamond $\langle r \rangle s$ in n, a node m is searched such that $s \in \mathcal{L}m$ and $\{t | [r]t \in \mathcal{L}n\} \subseteq \mathcal{L}m$. Then, an r-edge from n to m is inserted.

While the implementation of pattern-based blocking as described so far is sufficient to guarantee termination, a stronger variant of pattern-based blocking is also implemented, which is possibly more effective as an optimization technique. The stronger variant does not only store a pattern as blocked when a node is created but also when the body of a box is propagated to a node later during tableau construction. This means, when the body t of a box $[r]t \in \mathcal{L}n$ is propagated to an r-successor m of n that was created for a diamond $\langle r \rangle s \in \mathcal{L}n$, the pattern $\{s\} \cup \{t | [r]t \in \mathcal{L}n\}$ (if r is transitive: $\{s\} \cup \{t, [r]t | [r]t \in \mathcal{L}n\}$) is stored in the pattern store as m is a witness for it.

In order to be efficient, the pattern store must be based on a data structure that provides an efficient operation for determining whether a superset of a given set is stored. Since backtracking can reset the status of a pattern from expanded to not expanded, the pattern store must also be capable of being backtracked. In the following, we describe three versions of the pattern store. We have implemented a version based on bit matrices as proposed by Giunchiglia and Tacchella [25] and a version based on an array of lists, which is derived from the bit-matrix-based pattern store. A third version of the pattern store is based on a tree-based data structure as proposed by Hoffmann and Koehler [32].

	0	1	2	3	4	5	...	31
$M[t_1]$	1	0	1	1	0	0	...	0
$M[t_2]$	1	1	0	1	1	0	...	0
$M[t_3]$	1	1	1	0	1	0	...	0
$M[t_4]$	0	0	1	1	1	0	...	0
$M[t_5]$	0	1	1	1	0	0	...	0

Figure 4.2: Example of the contents of a bit matrix after five patterns have been inserted.

4.2.1 Bit-Matrix-based Pattern Store

One of the implementations of the pattern store uses a bit matrix as proposed by Giunchiglia and Tacchella [25].

The bit matrix of the pattern store contains, for each subterm of the input, a bit vector. Each stored pattern corresponds to a bit. The bit matrix is maintained in such a way that a pattern $t_1, \ldots, t_n$ is stored iff the bitwise $\bigwedge$ of all bit vectors corresponding to $t_1, \ldots, t_n$ is a bit vector that contains at least one bit that is not zero.

For example, figure 4.2 shows the contents of the bit matrix after the insertion of the patterns $\{t_1, t_2, t_3\}$, $\{t_2, t_3, t_5\}$, $\{t_1, t_3, t_4, t_5\}$, $\{t_1, t_2, t_4, t_5\}$ and $\{t_2, t_3, t_4\}$. For instance, since $\{t_1, t_2, t_4, t_5\}$ has been inserted as the fourth pattern, $M[t_i][3]$ is set for $t_i \in \{t_1, t_2, t_4, t_5\}$.

In the following, let M denote the bit matrix of the pattern store, let $M[t]$ denote the bit vector corresponding to the term t and let $M[t][i]$ denote the i-th bit of the bit vector corresponding to the term t.

Initially, each $M[t]$ is the empty bit vector. A new pattern $t_1, \ldots, t_n$ is inserted to the store by setting $M[t_i][c]$ to one for $1 \leq i \leq n$, where c is the next bit not yet used for storing a pattern. A query for whether a superset of $\{t_1, \ldots, t_n\}$ is stored is performed by testing whether the bitwise $\bigwedge$ of $M[t_1], \ldots, M[t_n]$ is non-zero.

In order to store all blocked patterns, the size of the bit vectors must be larger than or equal to the number of blocked patterns. Once the number of patterns exceeds the size of the bit vectors, which is initially 32, the length of the bit vectors is doubled.

Since it is typically the case that most of the subterms of the input are not part of any pattern to be stored, a bit vector for a term t is only allocated when t is part of a blocked pattern for the first time. In the meantime, the empty bit vector in $M[t]$ is represented by a special symbol in order to reduce the size of the bit matrix.

4.2.2 List-based Pattern Store

A potential drawback of the bit-matrix-based pattern store is that, if many patterns with different terms are stored, the bit matrix is large and sparse, i.e., most bits in the bit matrix are zero. The list-based pattern store is a modification of the bit-matrix-based approach that addresses this problem. The general idea is to store, for each term, a list of integers such that the intersection of all lists associated with a pattern is empty iff that pattern is not blocked. These lists thus correspond to the bit vectors in the bit matrix. The length of a list is equal to the number of stored patterns that contain the corresponding term, whereas the length of a bit vector depends on the total number of stored patterns. Hence, the list-based approach may significantly reduce the amount of memory required.

Initialization. Once the encoding of terms as described in Section 4.1 has been completed, the total number n of subterms of the input is known. An array A of size n is generated such that each element $A[i]$ $(0 \leq i < n)$ of A is initially an empty list. The counter c is set to zero.

Insert operation. When a pattern $\{t_1, \ldots, t_m\}$ $(m > 0)$ is added to the *pattern store*, the current value of the counter c is appended to the lists $A[t_i]$ $(1 \leq i \leq m)$. Hence, the lists associated with the pattern share the same element. Formally, the array is updated as follows:

$$A'[t_j] = \begin{cases} c :: (A[t_j]) & \text{if } t_j \in \{t_i | 1 \leq i \leq m\} \\ A[t_j] & \text{otherwise} \end{cases}$$

After the array has been updated, the counter c is incremented by one. Therefore, the lists stored in A are always sorted in descending order.

For instance, after the insertion of the patterns $\{t_1, t_2, t_3\}$, $\{t_2, t_3, t_5\}$, $\{t_1, t_3, t_4, t_5\}$, $\{t_1, t_2, t_4, t_5\}$ and $\{t_2, t_3, t_4\}$, the contents of the lists are as follows:

$$\begin{array}{ll} A[t_1]: & [3, 2, 0] \\ A[t_2]: & [4, 3, 1, 0] \\ A[t_3]: & [4, 2, 1, 0] \\ A[t_4]: & [4, 3, 2] \\ A[t_5]: & [3, 2, 1] \end{array}$$

Since, e.g., $\{t_1, t_2, t_4, t_5\}$ has been inserted as the fourth pattern, 3 is contained in the lists for t_1, t_2, t_4 and t_5. Note the similarity between the contents of the bit matrix in Figure 4.2 and the contents of the lists. An integer is contained in a list iff the corresponding bit is set in the bit matrix.

Querying. A query pattern $\{t_1, \ldots, t_m\}$ contains $m \geq 1$ terms. If $m = 1$, a superset of the query pattern $\{t_1\}$ is stored iff $A[t_1]$ is not the empty list. If the pattern to be queried contains more than one element, i.e., $m > 1$, it must be determined whether the intersection of the lists associated with the query pattern is empty. In other words, it must be checked whether the lists contain at least one common element. Since the lists are ordered, an obvious way to do this is to try to find the largest common element.

fun *query* $\{t_1, \ldots, t_m\}$
  Initialization:
  for $1 \leq i \leq m$**:** $l_i := A[t_i]$
  Apply drop to pairs of lists until the heads of all lists are equal:
  while $\exists i, j$**:**$hd(l_i) \neq hd(l_j)$**:**
    for $2 \leq i \leq m$**:**
      $(l_1, l_i) := drop(l_1, l_i)$
      if $l_1 = \mathsf{nil}$ **then return** false
  return true

Figure 4.3: Algorithm for computing whether a superset is stored

Let *drop* be a procedure that, given a pair of two lists l_1 and l_2, removes all elements from l_1 and l_2 that are larger than the largest common element of the two lists, or returns $(\mathsf{nil},\ \mathsf{nil})$ if l_1 and l_2 have no common element. In other words, *drop* satisfies the following recursive equations:

$$drop(\mathsf{nil}, ys) = drop(xs, \mathsf{nil}) = (\mathsf{nil}, \mathsf{nil})$$
$$drop(x :: xr, y :: yr) = \begin{cases} (x :: xr, y :: yr) & \text{if } x = y \\ drop(xr, y :: yr) & \text{if } x > y \\ drop(x :: xr, yr) & \text{if } x < y \end{cases}$$

Hence, if the query pattern contains exactly two elements t_1, t_2, a superset of $\{t_1, t_2\}$ is stored iff *drop* $(A[t_1],\ A[t_2])$ does not yield $(\mathsf{nil},\ \mathsf{nil})$. Using *drop*, a procedure for query patterns containing more than two elements is obtained in a straightforward way. A superset of $\{t_1, \ldots, t_m\}$ $(m > 2)$ is stored iff the function sketched in Figure 4.3 returns true.

4.2.3 Tree-based Pattern Store

An alternative implementation of the pattern store is based on a data structure of unlimited branching trees proposed by Hoffmann and Koehler [32].

The pattern store consists of a set of trees whose nodes are labeled with terms (more precisely, term indices as described in Section 4.1). The general idea is to represent a pattern $\{t_1, \ldots, t_n\}$ by a path in a tree containing the labels $t_1, \ldots, t_n$. Terms in the path are stored in increasing order as given by their indices, i.e., if a node is labeled with t, the label of its children is always strictly larger than t. A compact forest is obtained as, if two patterns $\{t_1, \ldots, t_k, t_{k+1}, \ldots, t_n\}$ and $\{t_1, \ldots, t_k, t'_{k+1}, \ldots, t'_n\}$ share their smallest k elements, their representation in the forest shares the corresponding nodes. Hence, different children of a node always have a different label.

An example of the forest storing the patterns $\{t_1, t_2, t_3\}$, $\{t_2, t_3, t_5\}$, $\{t_1, t_3, t_4, t_5\}$, $\{t_1, t_2, t_4, t_5\}$ and $\{t_2, t_3, t_4\}$ is shown in Figure 4.4.

In order to store a pattern $\{t_1, \ldots, t_n\}$, where $t_1 < \ldots < t_n$, we first search the largest prefix $t_1, \ldots, t_k$ $(k \leq n)$, such that a tree exists whose root is labeled with t_1 and which contains $t_1, \ldots, t_k$ as a path. If such a tree exists, we add the path

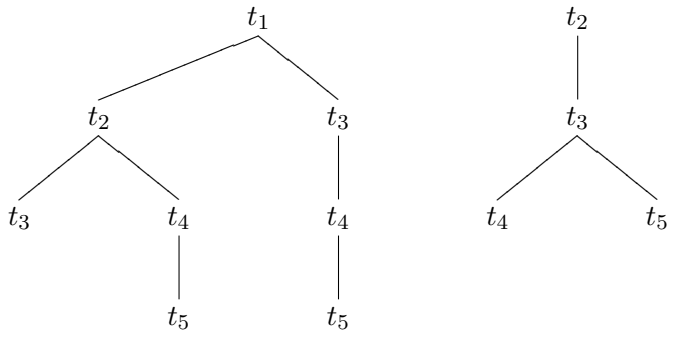

Figure 4.4: Example of a forest storing five patterns

fun *query* $(\{t_1, \ldots, t_m\},\ N)$
  for $x \in N$ such that $label(x) \leq t_1$ **:**
    if $label(x) = t_1$
    then
      if $m = 1$
      then return true
      else *query* $((t_2, \ldots, t_m), children(x))$
    else *query* $((t_1, \ldots, t_m), children(x))$
  return false

Figure 4.5: Algorithm for computing whether a superset is stored in the forest

$t_{k+1}, \ldots, t_n$ as a subtree to the node labelled t_k. Otherwise, we add a new tree consisting of the path $t_1, \ldots, t_n$ to the forest.

Figure 4.5 sketches the *query* function, which is used to determine whether a superset of $\{t_1, \ldots, t_n\}$ is stored in the forest. Provided that $t_1 < \ldots < t_n$ and N is the set of root nodes of the forest, this is the case iff *query* $(\{t_1, \ldots, t_n\}, N)$ returns true.

Since removing patterns from a forest is non-trivial, the tree-based pattern store is the only data structure in our implementation that uses copying instead of trailing for backtracking.

4.3 Backjumping

Backjumping [24] is a variant of dependency-directed backtracking [59] that makes use of information about the cause of a clash in order to prune the search space. The technique has been used in constraint solvers [5] and SAT solvers [21, 56] and adapted to modal and description logic reasoners [33, 52, 60].

Backjumping aims at reducing the search space by avoiding *thrashing*, i.e., the exploration of branches that differ only in inessential features from branches that have been explored previously.

Figure 4.6 illustrates an example where thrashing occurs. Branching search starts by choosing the alternative $p_1 x$ of $(p_1 \dot{\vee} q_1)x$. Both branches that are then gen-

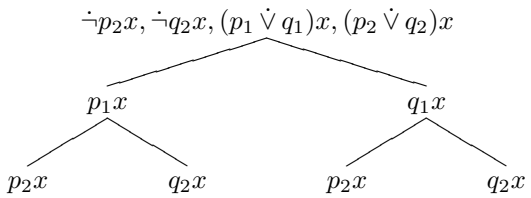

Figure 4.6: Example of thrashing

erated by the disjunction $(p_2 \dot{\vee} q_2)x$ are closed as $\dot{\neg}p_2x$ and $\dot{\neg}q_2x$ are already on the branch. Naïve backtracking, as described in Section 3.4, does not recognize that the cause for the branches becoming closed is independent from the choice made for $(p_1 \dot{\vee} q_1)x$. Therefore, the useless exploration of the alternative branch containing q_1x is not avoided.

Backjumping reduces the amount of thrashing as it makes use of information about the cause of a clash. While naïve backtracking always goes back to the most recent branching point, backjumping prunes the search space by ignoring branching points where a different choice can obviously not result in an open branch.

One way to implement backjumping is to label each term with a *dependency set* that contains the branching points on which the term depends [33]. When a clash is detected, the dependency sets of the clashing terms are used to determine the most recent branching point that introduced one of the clashing terms. Then, backtracking directly jumps back to that branching point.

For backjumping to be correct, it is crucial that each dependency set contains all the branching points on which the corresponding term depends. It is thus important to understand, which branching points must be added to the dependency set of a term. In general, a branching point b must be included in the dependency set of a term t if t has been introduced to the branch by an action that depends on b.

Assume that b is a branching point containing the alternatives $t_1, \ldots, t_n$. The dependency set of the first alternative t_1 of a branching point b is the dependency set of the corresponding disjunction extended with b. When, after a clash, a subsequent alternative t_i $(1 < i \leq n)$ of the branching point is chosen, the dependency set of t_i additionally contains each element contained in at least one of the dependency sets of the clashing terms. This is necessary because the clash could be avoided by a different choice at one of those branching points. Note that b is removed from the dependency set of the last alternative of b, because, in that case, jumping to b after a clash would be useless as there are no unexplored alternatives on b.

The dependency set of a term introduced by an action other than branching is the union of the dependency sets of the terms on which the action depends. For instance, the conjuncts of a conjunction t inherit the dependency set of t. When the body of a box $tx = [r]sx$ is propagated to an r-successor y of x, then the dependency set of sy is the union of the dependency sets of tx and rxy. If terms are copied from a node n to a node m to satisfy an equational constraint, their

dependency sets and the dependency sets of boxes to be propagated to successors of m must be unioned with the dependency sets of the corresponding nominals.

When a clash is detected, backtracking is invoked with the dependency sets of the clashing terms. If the union of those dependency sets is empty, it is certain that there exists no alternative branch that might lead to an open maximal branch. Therefore, the result *unsatisfiable* can be returned immediately. Otherwise, backtracking jumps back to the most recent branching point contained in the union of the dependency sets of the clashing terms. Since a branching point is only contained in a dependency set if there are unexplored alternatives, the next alternative can be chosen from this branching point and added to the branch.

Dependency-directed backtracking as proposed in [59] could prune the search space even further. By maintaining assumption sets containing all formulas that have contributed to the closure of a branch, the investigation of any branch that contains the formulas in one of the assumption sets is avoided. However, it is not obvious how this approach could be adapted to modal logics.

4.4 Boolean Constraint Propagation

Boolean constraint propagation (BCP) is an optimization technique that helps reducing the number of branches and lowering the average branching depth [21].

In our realization of BCP, before branching on a disjuction $t = t_1 \dot{\vee} \ldots \dot{\vee} t_k$ located in a node n, we check, for each t_i $(1 \leq i \leq k)$, whether the negation of t_i is already in $\mathcal{L}n$. If so, t_i does not have to be considered as an alternative because adding t_i would immediately lead to a clash. If, after BCP is applied, there are at least two alternatives left, branching must be performed on them as usual, but not on disregarded alternatives.

The greatest gain, however, is achieved if at most one alternative remains. If there is no remaining alternative, backtracking is invoked immediately. If exactly one alternative remains, it is added to the branch deterministically.

Note that, if BCP is used in combination with backjumping, eliminated disjuncts must be handled as if they had actually caused a clash. This means, the union of the dependency sets of terms with which eliminated disjuncts would have clashed must be added to the dependency set of each remaining alternative.

We also implemented a more eager version of BCP. Instead of just looking at the first disjunction on the agenda, it tries to find a disjunction on the agenda that can be simplified to the point where it contains at most one alternative. If such a disjunction is found, the remaining alternative is added deterministically, or, if no alternative remains, backtracking is invoked immediately. Only if no disjunction can be simplified that far, branching is applied with the weaker version of BCP as described above.

Thus, the eager version of BCP helps reducing the amount of search by allowing more deterministic steps to be made before we need to branch. Our implemen-

$$\mathcal{R}^{d}_{\dot\vee}\ \frac{(t_1 \mathbin{\dot\vee} t_2)x}{t_1x \mid t_2x,\ (\dot\neg t_1)x}$$

Figure 4.7: Tableau rule for disjoint branching

tation of eager BCP can require all disjunctions on the agenda to be examined every time the propagation routine is invoked. Therefore, eager BCP can also have a noticeable adverse effect on the performance of the implementation, especially if many disjunctions are examined without actually detecting cases that can be simplified far enough. A more sophisticated implementation that reduces work, e.g., by using watched literals [50, 49], may solve that problem.

4.5 Disjoint Branching

The search technique described so far uses purely syntactic branching. A disjunction $(t_1 \mathbin{\dot\vee} \ldots \mathbin{\dot\vee} t_k)x$ is processed by generating a branching point b and selecting one of the disjuncts, let's say t_1x, to be added to the branch. When backtracking to b, the disjunct t_1x is discarded and tableau construction continues with an alternative branch containing an alternative disjunct, let's say t_2x. From backtracking to b and discarding t_1x, we know that the alternative branch will eventually become closed as well if t_1x is added to it. However, purely syntactic branching does not make use of this information.

The problem that arises when branching syntactically is that different branches are not necessarily semantically disjoint. Therefore, tableau-based algorithms using syntactic branching cannot polynomially simulate truth-tables [17].

However, the tableau algorithm can be changed to enforce semantically disjoint branches by replacing $\mathcal{R}_{\dot\vee}$ by the tableau rule $\mathcal{R}^{d}_{\dot\vee}$ shown in Figure 4.7 [65]. In our system, this is implemented as follows: When a disjunct t is discarded, its negation $\dot\neg t$ is added to the corresponding node as an additional constraint. A possible disadvantage of this approach is that adding $(\dot\neg t)x$ to the branch might result in a significantly larger search space. For example, if disjoint branching is applied on $\langle r\rangle(p \mathbin{\dot\wedge} q) \mathbin{\dot\wedge} ([r]\dot\neg p \mathbin{\dot\vee} [r]t)$ as shown in Figure 4.8, the negation of $[r]\dot\neg p$ (i.e., $\langle r\rangle p$) is added to the initial node after the first branch has become closed. Hence, a second r-successor must be created and the subproblem t must be solved twice.

To avoid this problem, discarded alternatives can also be stored separately in so-called *no-good lists* [36]. The general idea is that, if a term t is added to a node whose no-good list contains t, backtracking can be invoked immediately because we have already discovered that adding t will eventually lead to a clash. In general, using no-good lists does not enforce that different branches are semantically disjoint. For instance, if $[r](p \mathbin{\dot\wedge} q)$ is added to a node whose no-good list contains only $[r]p$, backtracking is not invoked immediately.

We have implemented both the disjoint branching rule in Figure 4.7 and no-good lists.

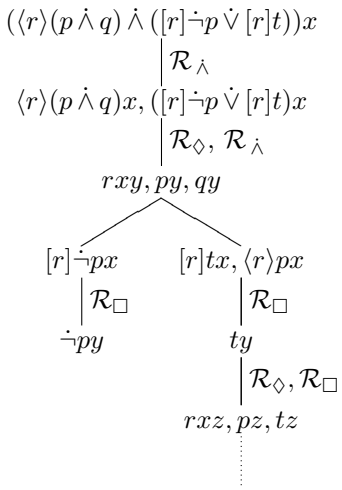

Figure 4.8: Example of disjoint branching causing a larger search space

In our system, no-good lists are implemented by maintaining an additional term store for each node containing the discarded alternatives. When a new term is added to a node, it is checked whether it is already contained in the corresponding store of discarded alternatives. If this is the case, backtracking is invoked as usual.

4.6 Lazy Branching

Lazy branching is an extension of an optimization technique known as lazy unfolding [33]. It aims at reducing the search space by postponing the application of the tableau rule $\mathcal{R}_{\dot\vee}$ to disjunctions containing propositional disjuncts.

In order to describe lazy branching, we use the letter l to denote a propositional literal that corresponds to a possibly negated propositional variable p. The negation of l is denoted by $\bar{l}$. In addition, we use the function σ that maps a propositional literal l to $\bot$ iff l is a negated propositional variable. Formally, σ is defined as follows:

$$\sigma l = \begin{cases} \bot & l \text{ has the form } \dot\neg p \\ \top & \text{otherwise} \end{cases}$$

The general idea behind lazy branching is that a propositional literal l is a witness for the satisfiability of a disjunction $(l \dot\vee \ldots) \in \mathcal{L}n$ as long as $\bar{l} \notin \mathcal{L}n$ and provided that $\bar{l}$ is not already used as a witness for the satisfiability of another disjunction $(\bar{l} \dot\vee \ldots) \in \mathcal{L}n$. As long as a disjunction contains such a witness, branching is not applied. This preserves completeness of the decision procedure because the addition of a propositional constraint to a branch not containing its negation does neither lead to a clash nor to another tableau rule becoming applicable.

In order to implement lazy branching, for each node, a special *lazy branching store* is maintained that stores disjunctions together with their respective witnesses.

When a new disjunction t is added to a node, we check whether t contains a witness for its satisfiability. If so, t is added to the lazy branching store. Only if t has no witness, t is added to the agenda.

Each lazy branching store contains a map from propositional variables to pairs of the shape $(\top,\ \{t_1, \ldots, t_k\})$ or $(\bot,\ \{t_1, \ldots, t_k\})$. A propositional variable p is mapped to $(\top,\ \{t_1, \ldots, t_k\})$ if $t_1, \ldots, t_k$ are disjunctions that can be delayed because they contain p as a witness. Similarly, p is mapped to $(\bot,\ \{t_1, \ldots, t_k\})$ if $t_1, \ldots, t_k$ are disjunctions that can be delayed because they contain $\dot{\neg}p$ as a witness.

In order to determine whether a disjunction $t \in \mathcal{L}n$ can be delayed, we first check whether t contains a propositional literal l such that $l \in \mathcal{L}n$. If so, t is already satisfied and additional considerations for t are not necessary. Otherwise, we check whether t contains a propositional literal l, which is a possibly negated propositional variable p, such that its negation $\bar{l} \notin \mathcal{L}n$ and

- p is mapped to $(\sigma l,\ \{t_1, \ldots, t_k\})$, in which case t can be added to the list, thus mapping p to $(\sigma l,\ \{t, t_1, \ldots, t_k\})$, or,
- p is not mapped to anything, in which case an entry $(\sigma l,\ \{t\})$ is created to which p is mapped.

Only if no such l exists, t is added to the agenda.

When a propositional literal l, which is a possibly negated propositional variable p, is added to n during tableau construction, we check whether p is mapped to $(\sigma\bar{l},\ \{t_1, \ldots, t_k\})$, in which case $t_1, \ldots, t_k$ may no longer be delayed due to their witness becoming false. Consequently, we try to find a different witness for $t_1, \ldots, t_k$. The terms t_i for which no witnesses can be found are added to the agenda.

If the input term is satisfiable, propositional constraints on the propositional variable p corresponding to l are obtained by checking whether the corresponding lazy branching store maps the propositional variable p to $(\top,\ \{\ldots\})$ or $(\bot,\ \{\ldots\})$.

Besides lazy branching on propositional literals as described above, SPARTACUS also implements lazy branching on boxes, which delays a disjunction $([r]t \mathbin{\dot{\vee}} \ldots) \in \mathcal{L}n$ if $\mathcal{L}n$ contains no diamonds and r is not reflexive.

Lazy branching could be extended further by postponing disjunctions containing a negated nominal $\dot{\neg}\dot{x}$ provided that $\dot{x} \notin \mathcal{L}n$.

4.7 Caching

Several reasoners for modal logic or description logic cache the satisfiability results of sets of terms [25, 27, 34] in order to avoid solving the same subproblem twice. If the set of terms that hold in a node is cached as unsatisfiable, backtracking can be invoked immediately. If the set of terms that hold in a node is cached as satisfiable, the recomputation of the result already known is avoided.

Reasoners that apply caching typically work top-down, looking at one node at a time [34, 25]. Using this strategy, a node can be cached as unsatisfiable if it is deleted during backtracking. Furthermore, a node can be cached as satisfiable when it and all of its successors are fully expanded.

However, in the presence of global modalities or nominals, the top-down expansion strategy cannot be applied naïvely because constraints can be propagated upwards. For example, when the procedure is applied to $\dot{x} \dot{\wedge} \langle r \rangle (\langle r \rangle \dot{x})$, a loop is created, making it impossible to follow the top-down strategy.

SPARTACUS features a caching technique that allows to cache unsatisfiable sets of nominal-free terms regardless of the expansion strategy. For a node n, let T_n denote the set of terms that have been propagated to n from outside, including bodies of diamonds, boxes and global modalities. The general idea is to cache T_n once it is found to be unsatisfiable, provided that T_n is nominal-free.

In order to determine whether T_n is unsatisfiable, we employ a technique that makes use of the dependency sets used for backjumping (cf. Section 4.3). When a clash is detected in a node n and T_n is nominal-free, it is checked whether T_n can be cached. This is the case iff the union of the dependency sets of the clashing terms does not contain a branching point that pertains to n. If T_n is cached as unsatisfiable and n has been generated as a successor[1] of m, we recursively try to cache T_m.

A set of formulas T is certainly unsatisfiable if a subset of T is unsatisfiable. Hence, it is beneficial to use for the cache a data structure that supports efficient querying for whether a subset of a given set is already cached. For this purpose, we have implemented both a data structure based on bit matrices as proposed by Giunchiglia and Tacchella [25] and a tree-based data structure as proposed by Hoffmann and Koehler [32]. The former approach has the advantage that the cache can be size bounded [25].

When a cache hit occurs, the dependency set needed for backjumping is approximated as described by Horrocks and Patel-Schneider [34]. Since the approximated dependency set can contain irrelevant branching points, caching can possibly limit the effectiveness of backjumping. The negative interaction of caching with backjumping may be reduced if, for each cached set of terms, the terms that caused the clash were stored. This way, a better approximation of the dependency sets would be possible.

When caching is enabled, we typically do not encounter any cache hits and, therefore, no positive effect on performance. In the few cases were cache hits occur, the number of branches that are explored often increases when caching is enabled and a positive effect on the decision time is not encountered.

We have not implemented caching for satisfiable sets of terms because, in the presence of nominals and global modalities, it is not clear how to efficiently determine whether a node and all of its successors are fully expanded as the models are not necessarily tree-shaped. However, pattern-based blocking (cf.

[1] Note that a node has at most one predecessor unless it contains a nominal and that caching is not applied to nodes containing nominals.

Sections 2.3, 4.2) subsumes some implementations of satisfiability caches (e.g., the satisfiability cache of Donini and Massacci [18], which is implemented as *mixed caching* by Goré and Postniece [27]).

Chapter 5

Evaluation

This chapter presents the results of the evaluation.

Section 5.1 provides general information about the tests performed and describes the sets of terms we used for the evaluation.

We are particularly interested in the effects of pattern-based blocking on performance. Section 5.2 compares the cases in which blocking is enabled with those in which it is disabled. It also shows the results obtained with different data structures for pattern-based blocking.

Section 5.3 shows the results for different rule application strategies.

Sections 5.4-5.6 deal with the effects of disjoint branching, lazy branching and boolean constraint propagation.

A comparison of Spartacus with other systems can be found in Section 5.7.

5.1 Approach

In order to evaluate the effect of different optimizations and ordering heuristics on performance, we use several problem sets from different sources:

- Modal terms generated with the K-CNF-generator written by Tacchella and Sebastiani [62]. Propositional literals only occur on the lowest level.
  - mCNF-c*v03d*: Sets of terms that contain three propositional variables with the number of clauses ranging from 30 to 300 and the modal depth ranging from two to six. Each problem set contains nine terms.
  - mCNF-c075v04d01: 16 terms that contain 75 clauses, have a modal depth of one and contain four propositional variables.
  - mCNF-c110v05d01: 16 terms that contain 110 clauses, have a modal depth of one and contain five propositional variables.

The latter two sets are interesting because approximately half of their terms are unsatisfiable, whereas randomly generated terms with a modal depth larger than one are typically satisfiable for a reasonable number of clauses.

- Hybrid terms with global modalities generated by HGEN [3]. hCNF-C* denotes terms of this class with the number of clauses ranging from 20 to 200. Each term contains three propositional variables and five nominals. The global depth is two, propositional and nominal literals occur with a probability of 40 percent on lower levels. The ratio of propositional variables to nominals is 4:1. Each disjunction consists of three disjuncts, the probability of a literal being a modal subformula is 80 percent, global modalities and satisfaction operators each occur with a probability of ten percent. Each set consists of 20 terms.

- A subset of the *Unbounded Modal QBF* benchmarks used for TANCS-2000 [48]. We use QBFs encoded as modal formulas using approaches based on [55] (qbfS-C*-V*-D*) and [45] (qbfL-C*-V*-D*). We also use *modalized* variants (qbfMS-C*-V*-D* and qbfML-C*-V*-D*) where different propositional variables are encoded as modal formulas with a single variable (see [47] for details). Each set consists of eight terms.

- Benchmarks from the Logics Workbench (LWB) [6] for the propositional modal logic K, KT (with reflexivity) and S4 (with reflexivity and transitivity). For each of these logics there exist 18 sets of terms. Each of these sets consists of 21 terms of increasing complexity.

Results of the evaluation are presented in tables that show, for each set of terms, how many problems from that set were solved within 60 seconds. In addition, we show the average of the decision times.

Results for LWB benchmarks are presented differently because they consist of sets of terms of increasing complexity. Thus, the tables show, for each setting, an integer between 1 and 21 that indicates the most complex problem solved within 60 seconds. The decision time in seconds is shown in addition unless there is another setting with which a more complex problem can be solved within the given time-out.

If optimizations do not have a significant effect on a class of formulas, we omit the corresponding rows in the tables.

Note that, for the evaluation of different settings of SPARTACUS, we focus on the decision times, excluding the time needed for parsing and preprocessing since it does not depend on the settings. However, for the comparison of SPARTACUS with other systems in Section 5.7, the full running times are shown.

We start with the following initial configuration, which is adjusted after each experiment according to the insights gained from it:

- disjuncts are processed in the following order: negated nominals, propositional literals, diamonds, boxes, satisfaction terms, existential modalities, nominals, universal modalities, conjunctions

- backjumping is enabled

	eager list		eager matrix		eager tree		on list		off	
mCNF-c030v03d02	9	0.00	9	0.00	9	0.00	9	0.00	9	0.00
mCNF-c060v03d02	9	0.01	9	0.01	9	0.01	9	0.01	9	0.01
mCNF-c090v03d02	9	0.16	9	0.18	9	0.18	9	0.16	9	0.16
mCNF-c120v03d02	8	4.90	8	5.77	8	5.31	8	8.02	8	3.48
mCNF-c030v03d04	9	0.03	9	0.04	9	0.05	9	0.03	9	0.05
mCNF-c060v03d04	9	0.13	9	0.16	9	0.20	9	0.13	9	0.27
mCNF-c090v03d04	9	0.29	9	0.41	9	0.54	9	0.30	9	0.67
mCNF-c120v03d04	9	0.64	9	1.07	9	1.37	9	0.65	9	1.54
mCNF-c150v03d04	9	1.33	9	2.53	9	2.83	9	1.32	9	4.09
mCNF-c180v03d04	9	1.97	9	4.30	9	4.69	9	2.02	9	7.20
mCNF-c210v03d04	9	3.01	9	7.04	9	7.86	9	3.01	9	10.66
mCNF-c240v03d04	9	4.46	9	11.25	9	12.77	9	4.50	9	15.58
mCNF-c270v03d04	9	9.85	8	16.30	8	17.87	9	9.88	8	23.40
mCNF-c300v03d04	9	12.77	8	25.20	8	27.16	9	12.93	6	22.24
mCNF-c030v03d06	9	0.17	9	0.20	9	0.35	9	0.17	9	0.52
mCNF-c060v03d06	9	0.59	9	0.91	9	2.39	9	0.60	9	2.57
mCNF-c090v03d06	9	1.01	9	2.14	9	8.55	9	1.04	9	7.50
mCNF-c120v03d06	9	2.33	9	5.66	9	24.44	9	2.39	7	12.71
mCNF-c150v03d06	9	4.16	9	12.10	6	42.59	9	4.24	6	35.15
mCNF-c180v03d06	9	7.27	9	21.53	0		9	7.42	2	51.54
mCNF-c210v03d06	9	9.69	9	37.89	0		9	10.10	0	
mCNF-c240v03d06	9	13.70	5	41.70	0		9	14.11	0	
mCNF-c270v03d06	9	17.09	2	51.22	0		9	17.74	0	
mCNF-c300v03d06	9	27.15	0		0		9	27.88	0	
mCNF-c075v04d01	16	0.15	16	0.18	16	0.17	16	0.14	16	0.13
mCNF-c110v05d01	16	9.77	16	12.83	16	12.02	16	8.33	16	8.03

Table 5.1: Evaluation results for blocking on mCNF

- the non-eager variant of boolean constraint propagation is employed
- no-good lists are enabled (except for mCNF-*d01, where disjoint branching is enabled)
- lazy branching on propositional literals is enabled
- caching is disabled
- terms on the agenda are prioritized as follows:
  - for mCNF-*d04, mCNF-*d06, mCNF-*d01 and LWB benchmarks: heuristic A as described in Section 5.3
  - for mCNF-*d02, hCNF, qbfMS and qbfML: heuristic B as described in Section 5.3
  - for qbfS and qbfL: heuristic D as described in Section 5.3

We ran all experiments on a PC with 2.8 GHz Intel Pentium 4 CPU (Hyper-threading disabled) and 1 GB RAM, running Linux.

	eager list		eager matrix		eager tree		on list	
hCNF-C20	20	0.00	20	0.00	20	0.00	20	0.00
hCNF-C40	20	0.02	20	0.02	20	0.02	20	0.02
hCNF-C60	20	0.56	20	0.59	20	0.60	20	0.78
hCNF-C80	16	4.99	16	5.97	16	5.24	16	7.26
hCNF-C100	13	6.77	12	3.47	13	7.48	12	7.25
hCNF-C120	13	14.07	13	16.73	13	15.65	13	11.83
hCNF-C140	5	6.67	5	7.62	5	7.92	5	5.06
hCNF-C160	7	14.44	7	16.55	7	17.43	7	10.50
hCNF-C180	3	5.37	3	6.06	3	5.92	4	15.80
hCNF-C200	8	28.38	7	28.83	7	29.83	8	20.14

Table 5.2: Evaluation results for blocking on hCNF

5.2 Pattern-based Blocking

This section evaluates the effects of pattern-based blocking in general and compares the different data structures used for storing patterns. Tables 5.1-5.5 show the results of the evaluation for different pattern stores when eager blocking is applied. An additional column shows the results for the case when the non-eager variant of pattern-based blocking is employed (using the best of the three data structures). If blocking is not crucial to achieve termination, we also show the results obtained without blocking.

Table 5.1 shows the results on the class mCNF. While pattern-based blocking has only a small effect on performance if the modal depth is small, it leads to an immense speedup for larger modal depths if the list-based data structure is used. The matrix-based data structure performs worse on this problem class. Using the tree-based data structure can even worsen the performance compared to the case where pattern-based blocking is disabled.

The results for blocking on terms of the class hCNF is shown in Table 5.2. On this class, the list-based pattern store is only slightly superior to pattern stores based on other data structures. While the eager variant is advantageous on terms with up to 100 clauses, the non-eager variant performs better on terms with more clauses.

Table 5.3 compares the results obtained for non-modalized QBFs. On this class of terms, the tree-based data structure generally outperforms the other data structures. Enabling pattern-based blocking leads to immense speedups on terms of the class qbfS. On the class qbfL, performance slightly degrades if blocking is employed. Note that, according to the heuristic D that is used here, disjunctions are processed with a higher priority than diamonds. In the case of the basic modal logic K, this has the consequence that the eager and the non-eager variants of pattern-based blocking coincide. Hence, there is no need to compare the two variants and we can exclude the corresponding column from the table.

As can be seen in Table 5.4, pattern-based blocking is crucial for the performance on modalized QBFs. Enabling the eager variant is beneficial. The tree-based pattern store is superior to the other two data structures.

	eager list		eager matrix		eager tree		off	
qbfS-C10-V4-D4	8	0.00	8	0.00	8	0.00	8	0.03
qbfS-C20-V4-D4	8	0.01	8	0.02	8	0.02	8	0.06
qbfS-C30-V4-D4	8	0.04	8	0.04	8	0.04	8	0.07
qbfS-C40-V4-D4	8	0.03	8	0.03	8	0.03	8	0.03
qbfS-C50-V4-D4	8	0.02	8	0.03	8	0.03	8	0.02
qbfS-C10-V4-D6	8	0.01	8	0.01	8	0.01	8	0.34
qbfS-C20-V4-D6	8	0.03	8	0.03	8	0.03	8	1.28
qbfS-C30-V4-D6	8	0.07	8	0.08	8	0.08	8	1.52
qbfS-C40-V4-D6	8	0.06	8	0.08	8	0.07	8	0.87
qbfS-C50-V4-D6	8	0.07	8	0.08	8	0.07	8	0.15
qbfS-C10-V8-D4	8	0.01	8	0.01	8	0.01	8	1.39
qbfS-C20-V8-D4	8	0.03	8	0.03	8	0.03	8	13.08
qbfS-C30-V8-D4	8	0.09	8	0.10	8	0.10	8	23.23
qbfS-C40-V8-D4	8	0.26	8	0.32	8	0.29	6	13.09
qbfS-C50-V8-D4	8	0.44	8	0.50	8	0.43	6	7.11
qbfS-C10-V8-D6	8	0.01	8	0.01	8	0.01	8	6.73
qbfS-C20-V8-D6	8	0.07	8	0.08	8	0.08	1	7.35
qbfS-C30-V8-D6	8	0.36	8	0.47	8	0.37	0	
qbfS-C40-V8-D6	8	1.13	8	1.49	8	1.10	0	
qbfS-C50-V8-D6	8	5.62	8	11.47	8	4.91	0	
qbfS-C10-V16-D4	8	0.01	8	0.01	8	0.01	4	9.63
qbfS-C20-V16-D4	8	0.04	8	0.04	8	0.05	0	
qbfS-C30-V16-D4	8	0.63	8	0.92	8	0.75	0	
qbfS-C40-V16-D4	8	0.46	8	0.55	8	0.49	0	
qbfS-C50-V16-D4	8	10.19	7	9.63	8	8.47	0	
qbfS-C10-V16-D6	8	0.01	8	0.01	8	0.01	3	9.94
qbfS-C20-V16-D6	8	0.10	8	0.11	8	0.12	0	
qbfS-C30-V16-D6	8	0.28	8	0.33	8	0.30	0	
qbfS-C40-V16-D6	8	1.87	8	2.76	8	1.80	0	
qbfS-C50-V16-D6	8	5.40	8	6.96	8	4.06	0	
qbfL-C10-V4-D4	8	0.97	8	0.99	8	0.96	8	0.75
qbfL-C20-V4-D4	8	2.59	8	2.69	8	2.56	8	2.01
qbfL-C30-V4-D4	8	2.34	8	2.49	8	2.28	8	1.76
qbfL-C40-V4-D4	8	1.10	8	1.12	8	1.12	8	0.81
qbfL-C50-V4-D4	8	1.11	8	1.16	8	1.12	8	0.82
qbfL-C10-V4-D6	8	13.82	8	15.48	8	10.74	8	9.60
qbfL-C20-V4-D6	4	32.28	2	24.24	6	32.53	6	29.57
qbfL-C30-V4-D6	1	13.91	1	15.68	3	38.81	5	43.56
qbfL-C40-V4-D6	6	13.71	5	5.72	7	19.07	7	16.10
qbfL-C50-V4-D6	8	7.52	8	8.34	8	7.38	8	5.59

Table 5.3: Evaluation results for blocking on non-modalized QBFs

	eager list		eager matrix		eager tree		on tree		off	
qbfMS-C10-V4-D4	8	0.01	8	0.01	8	0.02	8	0.02	5	27.58
qbfMS-C20-V4-D4	8	0.13	8	0.15	8	0.16	8	0.59	1	44.47
qbfMS-C30-V4-D4	8	0.34	8	0.45	8	0.40	8	7.62	1	49.98
qbfMS-C40-V4-D4	8	4.91	8	8.17	8	5.02	6	8.02	0	
qbfMS-C50-V4-D4	8	3.68	8	5.24	8	3.76	6	5.63	0	
qbfMS-C10-V4-D6	8	0.02	8	0.02	8	0.02	8	0.03	0	
qbfMS-C20-V4-D6	8	2.00	8	3.20	8	2.27	7	3.10	0	
qbfMS-C30-V4-D6	8	6.93	7	3.60	8	7.57	7	5.38	0	
qbfMS-C40-V4-D6	6	9.75	5	10.57	6	9.64	5	16.73	0	
qbfMS-C50-V4-D6	8	6.04	8	13.38	8	6.56	7	15.44	0	
qbfMS-C10-V8-D4	8	0.02	8	0.02	8	0.03	8	0.04	0	
qbfMS-C20-V8-D4	8	0.21	8	0.27	8	0.25	8	0.62	0	
qbfMS-C30-V8-D4	8	2.28	8	4.08	8	2.77	7	2.89	0	
qbfMS-C40-V8-D4	3	21.81	1	2.99	3	19.77	1	24.75	0	
qbfMS-C50-V8-D4	8	3.68	8	5.24	8	3.76	6	5.63	0	
qbfMS-C10-V8-D6	8	0.03	8	0.04	8	0.04	8	0.09	0	
qbfMS-C20-V8-D6	8	0.67	8	1.01	8	0.90	8	1.63	0	
qbfMS-C30-V8-D6	7	1.74	7	2.91	7	2.11	7	3.45	0	
qbfMS-C40-V8-D6	3	11.41	3	13.48	4	12.69	4	28.04	0	
qbfMS-C50-V8-D6	1	13.72	1	29.93	1	16.93	0		0	
qbfMS-C10-V16-D4	8	0.04	8	0.05	8	0.06	8	0.07	0	
qbfMS-C20-V16-D4	8	0.59	8	0.82	8	0.76	8	1.65	0	
qbfMS-C30-V16-D4	8	10.22	8	9.36	8	4.25	8	6.04	0	
qbfMS-C40-V16-D4	2	24.54	2	25.80	3	13.31	3	18.36	0	
qbfMS-C50-V16-D4	0		0		2	45.55	1	58.56	0	
qbfMS-C10-V16-D6	8	0.05	8	0.06	8	0.07	8	0.08	0	
qbfMS-C20-V16-D6	8	0.48	8	0.65	8	0.61	8	0.74	0	
qbfMS-C30-V16-D6	7	8.09	7	9.75	8	8.77	8	12.64	0	
qbfMS-C40-V16-D6	4	22.27	3	16.25	5	23.02	5	27.24	0	
qbfMS-C50-V16-D6	1	35.73	1	50.35	1	21.74	1	35.63	0	
qbfML-C10-V4-D4	8	1.13	8	1.43	8	1.20	8	3.72	0	
qbfML-C20-V4-D4	8	6.29	8	8.22	8	6.36	8	17.12	0	
qbfML-C30-V4-D4	8	7.69	8	9.23	8	7.69	8	20.22	0	
qbfML-C40-V4-D4	8	14.51	8	16.96	8	14.23	8	34.77	0	
qbfML-C50-V4-D4	6	21.01	6	24.45	6	20.55	4	40.95	0	
qbfML-C10-V4-D6	8	11.42	7	8.94	8	11.34	6	12.95	0	
qbfML-C20-V4-D6	8	20.27	8	28.81	8	19.90	2	16.86	0	
qbfML-C30-V4-D6	8	25.52	7	33.06	8	24.04	3	57.45	0	
qbfML-C40-V4-D6	3	42.05	1	55.08	4	45.25	0		0	
qbfML-C50-V4-D6	1	33.56	1	47.10	1	31.78	0		0	
qbfML-C10-V8-D4	8	10.30	8	14.56	8	9.74	6	27.04	0	
qbfML-C20-V8-D4	5	40.22	1	29.38	7	39.67	0		0	
qbfML-C30-V8-D4	0		0		0		0		0	
qbfML-C40-V8-D4	0		0		0		0		0	
qbfML-C50-V8-D4	0		0		0		0		0	
qbfML-C10-V8-D6	5	12.43	4	4.87	7	22.10	3	6.42	0	
qbfML-C20-V8-D6	0		0		0		0		0	
qbfML-C30-V8-D6	0		0		0		0		0	
qbfML-C40-V8-D6	0		0		0		0		0	
qbfML-C50-V8-D6	0		0		0		0		0	

Table 5.4: Evaluation results for blocking on modalized QBFs

(a) Results on K and KT benchmarks

	eager list		eager matrix		eager tree		on tree		off	
k_d4_n	21	0.71	21	0.72	21	0.72	21	4.53	5	
k_d4_p	21	0.14	21	0.15	21	0.15	21	0.17	9	
k_dum_n	21	0.02	21	0.02	21	0.02	21	0.03	19	
k_dum_p	21	0.01	21	0.01	21	0.01	21	0.01	21	0.19
k_path_n	21	0.55	21	0.54	21	0.56	21	0.57	11	
k_path_p	21	0.51	21	0.52	21	0.54	21	0.55	13	
k_poly_n	21	0.18	21	0.18	21	0.20	21	0.20	21	4.50
k_poly_p	21	0.17	21	0.17	21	0.18	21	0.18	21	4.35
k_t4p_n	21	0.09	21	0.09	21	0.09	21	0.09	7	
kt_45_n	21	0.28	21	0.27	21	0.30	21	0.30	6	
kt_45_p	21	0.06	21	0.05	21	0.06	21	0.06	7	
kt_dum_n	21	0.03	21	0.03	21	0.03	21	0.03	15	
kt_md_n	7	17.09	7	11.55	7	9.59	7	21.80	5	
kt_md_p	5	20.24	5	8.41	5	10.51	5	10.47	4	
kt_path_n	21	35.76	21	35.87	21	36.29	21	36.43	11	
kt_path_p	21	28.94	21	28.88	21	29.73	21	29.72	12	
kt_ph_n	21	28.52	21	29.05	21	29.28	21	29.81	21	38.88
kt_ph_p	7	43.83	7	46.26	7	46.61	7	44.65	6	
kt_poly_n	8	48.66	8	48.67	8	48.65	8	48.74	2	
kt_poly_p	5	56.69	5	55.97	5	56.72	5	56.62	1	
kt_t4p_n	21	0.23	21	0.25	21	0.24	21	0.24	7	
kt_t4p_p	21	0.04	21	0.05	21	0.05	21	0.06	6	

(b) Results on S4 benchmarks

	eager list		eager matrix		eager tree		on tree	
s4_45_n	21	0.40	21	0.53	21	1.86	21	19.84
s4_branch_n	13		14	27.47	14	42.29	14	27.32
s4_md_n	21	19.48	21	24.36	21	20.34	21	26.39
s4_md_p	9	32.96	9	35.64	9	33.81	9	35.71
s4_path_n	16	49.25	16	48.80	16	47.96	16	49.34
s4_path_p	17	57.52	17	58.51	17	56.02	17	57.10
s4_ph_n	9	26.86	9	28.53	9	26.67	8	
s4_ph_p	4	1.28	4	1.68	4	1.46	4	11.01
s4_s5_n	16	53.57	16	53.71	16	57.89	16	54.11

Table 5.5: Evaluation results for blocking on LWB benchmarks

	A		B		C		D		E	
mCNF-c030v03d02	9	0.00	9	0.00	9	0.00	9	0.00	9	0.00
mCNF-c060v03d02	9	0.10	9	0.01	9	0.03	9	0.01	9	0.09
mCNF-c090v03d02	5	0.08	9	0.17	9	4.67	8	2.95	7	0.40
mCNF-c120v03d02	4	19.44	8	5.05	6	11.86	3	7.31	2	30.18
mCNF-c030v03d04	9	0.03	9	0.05	9	0.06	9	0.03	9	0.07
mCNF-c060v03d04	9	0.13	9	0.27	9	0.40	9	0.12	9	0.45
mCNF-c090v03d04	9	0.29	9	0.74	9	1.08	9	0.28	9	1.20
mCNF-c120v03d04	9	0.64	9	1.83	9	2.97	9	0.62	9	3.53
mCNF-c150v03d04	9	1.30	8	4.26	8	9.51	9	1.13	7	9.90
mCNF-c180v03d04	9	1.97	8	8.88	8	21.27	9	1.95	7	22.51
mCNF-c210v03d04	9	3.00	8	17.24	5	27.55	9	2.84	5	34.65
mCNF-c240v03d04	9	4.42	5	22.78	3	42.25	9	4.22	0	
mCNF-c270v03d04	9	9.73	4	36.76	1	55.62	9	8.13	0	
mCNF-c300v03d04	9	12.69	1	52.87	0		9	10.97	0	
mCNF-c030v03d06	9	0.17	9	0.20	9	0.30	9	0.17	9	0.41
mCNF-c060v03d06	9	0.58	9	0.97	9	1.62	9	0.57	9	2.03
mCNF-c090v03d06	9	1.01	9	2.52	9	5.10	9	0.97	9	6.68
mCNF-c120v03d06	9	2.32	9	7.42	9	17.67	9	2.23	8	17.85
mCNF-c150v03d06	9	4.17	9	22.60	7	37.49	9	3.93	7	48.07
mCNF-c180v03d06	9	7.25	6	30.74	3	50.19	9	6.97	0	
mCNF-c210v03d06	9	9.82	4	40.09	0		9	9.16	0	
mCNF-c240v03d06	9	13.66	3	56.68	0		9	12.84	0	
mCNF-c270v03d06	9	16.99	0		0		9	15.87	0	
mCNF-c300v03d06	9	26.94	0		0		9	25.63	0	
mCNF-c075v04d01	16	0.15	16	1.27	16	0.84	5	9.16	15	2.26
mCNF-c110v05d01	16	9.84	3	22.92	13	27.05	0		0	

Table 5.6: Evaluation results for ordering heuristics on mCNF

Table 5.5 shows the results obtained for LWB benchmarks. We omitted benchmarks on which pattern-based blocking did not have any effect. On many terms, we encounter a significantly increased performance if pattern-based blocking is used. If there is a significant difference between the different data structures, the tree-based data structure is mostly better.

For the following experiments, we use the eager variant of pattern-based blocking. We employ the tree-based data structure, except for the classes mCNF and hCNF, for which we use the list-based pattern store.

5.3 Ordering Heuristics

We selected five ordering heuristics for the terms on the agenda that performed well during preliminary tests and evaluated their performance.

All ordering heuristics we investigated have in common that boxes have the highest priority, followed by satisfaction terms, nominals, universal modalities and existential modalities. Diamonds and disjunctions always have the lowest priority. In the ordering heuristics we have chosen, the priorities of diamonds and disjunctions among each other are as follows:

A: Diamonds are processed before disjunctions. Diamonds on a new (more re-

	A		B		C		D		E	
hCNF-C20	20	0.00	20	0.00	20	0.00	20	0.00	20	0.00
hCNF-C40	20	0.04	20	0.02	20	0.02	17	0.28	18	0.33
hCNF-C60	16	0.87	20	0.56	20	1.84	4	0.26	9	1.90
hCNF-C80	9	3.70	14	0.91	16	4.12	0		3	1.96
hCNF-C100	5	0.95	12	2.83	8	3.18	0		2	5.25
hCNF-C120	1	16.71	11	8.61	10	5.84	0		0	
hCNF-C140	1	0.45	5	6.72	3	9.62	0		0	
hCNF-C160	0		6	11.78	5	12.45	0		0	
hCNF-C180	0		3	5.28	3	12.32	0		0	
hCNF-C200	0		4	12.44	3	16.00	0		0	

Table 5.7: Evaluation results for ordering heuristics on hCNF

cently generated) node precede diamonds on an old (less recently generated) node. Disjunctions on a new node have a higher priority than disjunctions on an old node.

B: Diamonds are processed before disjunctions. Diamonds on a new node precede diamonds on an old node. Disjunctions on an old node have a higher priority than disjunctions on a new node.

C: Diamonds are processed before disjunctions. Diamonds on a new node precede diamonds on an old node. Disjunctions are prioritized according to their dependency sets, i.e., disjunctions that depend on more recent branching points have a lower priority than disjunctions that only depend on less recent branching points.

D: Disjunctions are processed before diamonds. Diamonds on a new node precede diamonds on an old node. Disjunctions on a new node have a higher priority than disjunctions on an old node.

E: The priorities of diamonds and disjunctions are based on their dependency sets. The diamond or disjunction that does not depend on branching points that are more recent than the branching points other diamonds or disjunctions depend on has the highest priority.

Table 5.6 shows the results obtained on terms of the class mCNF. While heuristic B is superior on terms of modal depth 2, A and D are better on the terms with higher modal depths. Heuristic A performs best on terms of the class mCNF-*d01.

The results for hybrid terms with global modalities are summarized in Table 5.7. The heuristics B and C clearly outperform the other heuristics on that class.

Table 5.8 illustrates the results obtained for non-modalized QBFs. On this class of terms, heuristic D shows the best performance.

Table 5.9 shows the results for modalized QBFs. While heuristic D performs well on the class qbfMS, heuristic E is preferable on the class qbfML.

Table 5.10 summarizes the results obtained on the LWB benchmarks. For the most part, heuristic D shows a good performance on this class of terms.

In the following, we will use heuristic D with the following exceptions: For experiments on qbfML-*d01, we use heuristic A. Heuristic B is used on the classes

	A		B		C		D		E	
qbfS-C10-V4-D4	8	0.00	8	0.00	8	0.01	8	0.00	8	0.00
qbfS-C20-V4-D4	8	1.17	8	0.02	8	0.04	8	0.02	8	0.03
qbfS-C30-V4-D4	8	0.08	8	0.09	8	0.21	8	0.04	8	0.11
qbfS-C40-V4-D4	8	1.98	8	0.06	8	1.47	8	0.03	8	0.14
qbfS-C50-V4-D4	8	0.46	8	0.06	8	0.92	8	0.03	8	0.15
qbfS-C10-V4-D6	8	0.01	8	0.01	8	0.01	8	0.01	8	0.01
qbfS-C20-V4-D6	8	0.06	8	0.07	8	0.17	8	0.03	8	0.07
qbfS-C30-V4-D6	8	0.21	8	0.60	8	0.36	8	0.12	8	0.36
qbfS-C40-V4-D6	7	1.51	8	0.48	7	4.22	8	0.07	8	0.92
qbfS-C50-V4-D6	8	1.71	8	0.17	7	5.50	8	0.07	8	1.01
qbfS-C10-V8-D4	8	0.01	8	0.01	8	0.01	8	0.01	8	0.01
qbfS-C20-V8-D4	8	0.03	8	0.04	8	0.04	8	0.03	8	0.05
qbfS-C30-V8-D4	8	0.08	8	0.14	8	0.11	8	0.10	8	0.14
qbfS-C40-V8-D4	7	0.88	8	1.14	7	3.84	8	0.28	8	3.36
qbfS-C50-V8-D4	4	16.83	8	2.67	2	9.54	8	0.43	8	11.95
qbfS-C10-V8-D6	8	0.01	8	0.01	8	0.01	8	0.01	8	0.01
qbfS-C20-V8-D6	8	0.05	8	0.13	8	0.11	8	0.08	8	0.10
qbfS-C30-V8-D6	8	0.19	8	0.50	8	0.85	8	0.37	8	1.31
qbfS-C40-V8-D6	6	5.92	7	5.75	6	16.66	8	1.11	5	7.17
qbfS-C50-V8-D6	5	2.63	5	7.08	4	18.16	8	4.96	3	13.46
qbfS-C10-V16-D4	8	0.01	8	0.01	8	0.01	8	0.01	8	0.01
qbfS-C20-V16-D4	8	0.03	8	0.05	8	0.05	8	0.05	8	0.05
qbfS-C30-V16-D4	8	0.30	8	1.04	8	1.06	8	0.76	8	0.97
qbfS-C40-V16-D4	8	4.72	8	0.98	8	0.99	8	0.50	8	1.51
qbfS-C50-V16-D4	5	4.30	4	9.21	5	19.11	8	8.59	4	24.81
qbfS-C10-V16-D6	8	0.02	8	0.02	8	0.02	8	0.01	8	0.01
qbfS-C20-V16-D6	8	0.07	8	0.13	8	0.14	8	0.11	8	0.12
qbfS-C30-V16-D6	8	0.21	8	0.42	8	0.59	8	0.30	8	0.46
qbfS-C40-V16-D6	8	0.80	8	4.75	8	3.58	8	1.81	8	7.08
qbfS-C50-V16-D6	8	3.02	7	4.52	8	7.81	8	4.10	7	4.21
qbfL-C10-V4-D4	7	1.03	8	1.35	7	2.62	8	0.95	8	4.43
qbfL-C20-V4-D4	1	30.53	8	4.70	1	53.72	8	2.55	8	9.88
qbfL-C30-V4-D4	1	18.48	8	3.20	0		8	2.26	7	11.31
qbfL-C40-V4-D4	2	1.64	8	2.82	0		8	1.10	7	9.05
qbfL-C50-V4-D4	3	4.35	8	2.10	0		8	1.10	7	5.61
qbfL-C10-V4-D6	6	14.80	6	10.16	2	25.34	8	10.63	4	22.01
qbfL-C20-V4-D6	1	18.09	4	35.57	0		6	32.18	0	
qbfL-C30-V4-D6	0		1	29.30	0		3	38.38	0	
qbfL-C40-V4-D6	1	15.32	5	4.83	0		7	18.65	1	57.43
qbfL-C50-V4-D6	0		8	9.06	0		8	7.21	0	

Table 5.8: Evaluation results for ordering heuristics on non-modalized QBFs

	A		B		C		D		E	
qbfMS-C10-V4-D4	8	0.02	8	0.02	8	0.02	8	0.01	8	0.02
qbfMS-C20-V4-D4	4	0.25	8	0.16	8	7.05	8	0.07	8	6.21
qbfMS-C30-V4-D4	6	22.63	8	0.41	7	7.69	8	0.18	8	1.84
qbfMS-C40-V4-D4	5	3.50	8	5.10	6	2.53	8	0.46	8	2.42
qbfMS-C50-V4-D4	5	1.39	8	3.87	4	2.86	8	0.88	8	7.39
qbfMS-C10-V4-D6	8	0.02	8	0.02	8	0.02	8	0.02	8	0.03
qbfMS-C20-V4-D6	4	9.61	8	2.30	7	3.42	8	0.17	7	1.99
qbfMS-C30-V4-D6	3	6.18	8	7.66	4	4.56	8	0.49	6	18.05
qbfMS-C40-V4-D6	2	17.46	6	9.74	1	3.72	8	1.23	3	8.54
qbfMS-C50-V4-D6	3	3.16	8	6.86	3	6.93	8	2.91	5	12.36
qbfMS-C10-V8-D4	8	0.03	8	0.03	8	0.03	8	0.02	8	0.03
qbfMS-C20-V8-D4	7	0.47	8	0.26	8	0.46	8	0.15	8	0.52
qbfMS-C30-V8-D4	6	2.35	8	2.80	6	16.64	8	0.91	8	19.44
qbfMS-C40-V8-D4	2	22.00	3	19.80	1	8.19	8	14.26	0	
qbfMS-C50-V8-D4	1	9.15	7	26.06	1	2.33	8	10.75	1	2.36
qbfMS-C10-V8-D6	8	0.04	8	0.05	8	0.09	8	0.04	8	0.07
qbfMS-C20-V8-D6	8	0.70	8	0.91	7	1.65	8	0.45	8	2.07
qbfMS-C30-V8-D6	6	2.86	7	2.13	7	2.56	8	2.19	7	6.69
qbfMS-C40-V8-D6	1	2.21	4	12.72	1	6.31	8	16.81	1	7.06
qbfMS-C50-V8-D6	0		1	16.90	0		6	35.25	0	
qbfMS-C10-V16-D4	8	0.06	8	0.06	8	0.06	8	0.06	8	0.07
qbfMS-C20-V16-D4	8	0.36	8	0.77	8	0.65	8	0.57	8	0.58
qbfMS-C30-V16-D4	7	1.38	8	4.25	8	4.60	8	3.42	8	8.08
qbfMS-C40-V16-D4	3	9.31	3	13.32	2	7.45	4	13.08	2	11.41
qbfMS-C50-V16-D4	0		2	46.15	0		6	35.36	0	
qbfMS-C10-V16-D6	8	0.06	8	0.07	8	0.07	8	0.07	8	0.07
qbfMS-C20-V16-D6	8	0.24	8	0.62	8	0.56	8	0.53	8	1.00
qbfMS-C30-V16-D6	8	2.46	8	8.85	7	9.72	8	6.89	8	11.39
qbfMS-C40-V16-D6	5	9.55	5	23.09	2	21.77	5	15.09	3	18.76
qbfMS-C50-V16-D6	2	42.78	1	21.67	0		1	17.58	0	
qbfML-C10-V4-D4	0		8	1.24	8	1.84	3	23.26	8	0.23
qbfML-C20-V4-D4	0		8	6.61	8	6.98	0		8	0.47
qbfML-C30-V4-D4	0		8	7.89	8	10.75	0		8	0.77
qbfML-C40-V4-D4	0		8	14.83	8	17.62	0		8	1.06
qbfML-C50-V4-D4	0		6	21.42	2	31.52	0		8	1.11
qbfML-C10-V4-D6	1	35.58	8	11.81	7	3.00	0		6	1.00
qbfML-C20-V4-D6	0		8	20.91	7	11.98	0		7	4.65
qbfML-C30-V4-D6	0		8	25.12	7	17.21	0		8	1.72
qbfML-C40-V4-D6	0		3	42.60	5	31.41	0		6	7.78
qbfML-C50-V4-D6	0		1	33.06	2	38.81	0		8	9.20
qbfML-C10-V8-D4	1	21.24	8	10.22	7	8.47	0		7	2.40
qbfML-C20-V8-D4	0		7	41.46	6	38.05	0		8	7.47
qbfML-C30-V8-D4	0		0		0		0		7	7.00
qbfML-C40-V8-D4	0		0		0		0		8	9.17
qbfML-C50-V8-D4	0		0		0		0		8	16.26
qbfML-C10-V8-D6	0		7	23.26	6	22.42	0		4	6.69
qbfML-C20-V8-D6	0		0		0		0		7	25.05
qbfML-C30-V8-D6	0		0		0		0		6	9.43
qbfML-C40-V8-D6	0		0		0		0		7	12.35
qbfML-C50-V8-D6	0		0		0		0		7	15.42

Table 5.9: Evaluation results for ordering heuristics on modalized QBFs

	A		B		C		D		E	
k_branch_n	10	30.28	10	55.90	7		10	29.84	6	
k_branch_p	21	0.04	10		11		16		11	
k_d4_n	21	0.73	21	0.17	21	0.16	21	0.85	21	0.18
k_d4_p	21	0.15	21	0.07	21	0.08	21	0.06	21	0.04
k_dum_n	21	0.02	21	0.02	21	0.02	21	0.04	21	0.03
k_dum_p	21	0.01	21	0.01	21	0.01	21	0.03	21	0.01
k_grz_n	21	0.01	21	0.01	21	0.01	21	0.01	21	0.02
k_grz_p	21	0.00	21	0.00	21	0.00	21	0.00	21	0.00
k_lin_n	21	0.00	21	0.00	21	0.00	21	0.00	21	0.00
k_lin_p	21	0.00	21	0.00	21	0.00	21	0.00	21	0.00
k_path_n	21	0.58	21	0.58	21	0.59	21	0.59	21	0.61
k_path_p	21	0.56	21	0.54	21	0.55	21	0.50	21	0.55
k_ph_n	21	0.25	21	0.25	21	0.30	21	0.19	21	0.24
k_ph_p	7		7		7		8	59.12	7	
k_poly_n	21	0.20	21	0.20	21	0.20	21	0.16	21	0.16
k_poly_p	21	0.19	21	0.19	21	0.20	21	0.06	21	0.16
k_t4p_n	21	0.09	21	0.17	21	0.22	21	0.10	21	2.90
k_t4p_p	21	0.02	21	0.06	21	0.03	21	0.04	21	0.15
kt_45_n	21	0.30	21	0.05	21	0.06	21	0.11	21	0.06
kt_45_p	21	0.06	21	0.16	21	0.21	21	0.05	21	0.12
kt_branch_n	10	30.46	10	56.22	6		10	29.97	6	
kt_branch_p	21	0.02	21	0.65	21	0.65	16		21	0.66
kt_dum_n	21	0.03	21	0.01	21	0.01	21	0.01	21	0.01
kt_dum_p	21	0.01	21	0.01	21	0.01	21	0.01	21	0.01
kt_grz_n	21	0.01	21	0.01	21	0.11	21	0.01	21	0.14
kt_grz_p	21	0.00	21	0.00	21	0.00	21	0.00	21	0.00
kt_md_n	7	9.74	7	12.07	7	13.64	7	4.29	7	15.29
kt_md_p	5		5		5		6	8.15	4	
kt_path_n	21	36.99	21	36.09	21	36.09	21	36.03	21	36.25
kt_path_p	21	29.75	21	29.66	21	29.81	21	26.06	21	29.67
kt_ph_n	21	29.82	21	18.30	15		21	28.66	15	
kt_ph_p	7	46.71	7	36.87	6		7	9.55	7	11.41
kt_poly_n	8		8		4		10		16	53.16
kt_poly_p	5		5		2		21	1.51	10	
kt_t4p_n	21	0.23	21	0.05	21	0.05	21	0.09	21	0.04
kt_t4p_p	21	0.05	21	0.04	21	0.06	21	0.52	21	0.21
s4_45_n	21	0.53	21	0.87	21	0.36	21	0.22	21	0.67
s4_45_p	21	0.01	21	0.01	21	0.01	21	0.10	21	0.01
s4_branch_n	14	27.53	11		11		14	24.88	11	
s4_branch_p	21	0.01	21	0.29	21	0.32	21	21.53	21	0.32
s4_grz_n	21	0.03	21	0.04	21	0.37	21	0.02	21	0.65
s4_grz_p	21	0.00	21	0.00	21	0.00	21	0.00	21	0.00
s4_ipc_n	21	2.30	21	2.31	21	2.37	21	2.33	12	
s4_ipc_p	21	2.96	21	3.10	21	3.00	5		12	
s4_md_n	21	24.57	21	24.18	21	24.35	21	23.86	9	
s4_md_p	9	35.68	9	49.59	9	49.97	9	37.62	3	
s4_path_n	16	49.82	16	48.78	16	49.20	16	49.34	16	48.86
s4_path_p	17	57.12	17	58.00	17	57.17	17	54.42	17	56.48
s4_ph_n	9		10		10		9		11	49.67
s4_ph_p	4		5		5		7	10.53	5	
s4_s5_n	16	54.20	16	54.27	16	53.68	16	54.10	16	59.99
s4_s5_p	21	32.72	21	31.81	21	45.92	21	0.00	21	0.00
s4_t4p_n	21	0.28	21	0.30	21	0.30	21	0.26	21	0.28
s4_t4p_p	21	0.11	21	0.08	21	0.14	21	0.22	21	0.11

Table 5.10: Evaluation results for ordering heuristics on LWB benchmarks

(a) Results for randomly generated terms

	on		ngl		off	
mCNF-c075v04p00d01	16	0.15	16	1.43	15	11.76
mCNF-c110v05p00d01	16	9.72	2	32.93	0	
hCNF-C60	19	0.34	20	0.56	20	1.66
hCNF-C80	16	3.52	16	4.99	14	3.07
hCNF-C100	12	5.38	13	6.77	7	1.21
hCNF-C120	12	12.06	13	14.07	4	18.05
hCNF-C140	8	20.59	5	6.67	0	
hCNF-C160	9	12.34	7	14.44	1	32.53
hCNF-C180	4	14.99	3	5.37	1	13.99
hCNF-C200	9	13.58	8	28.38	0	
qbfMS-C50-V4-D4	8	0.47	8	0.88	6	26.85
qbfMS-C50-V4-D6	8	2.06	8	2.89	5	20.24
qbfMS-C50-V8-D4	8	10.10	8	10.78	6	19.09
qbfMS-C50-V8-D6	6	35.69	6	35.56	2	15.98
qbfMS-C50-V16-D4	6	35.34	6	35.62	5	34.68
qbfMS-C50-V16-D6	1	17.58	1	17.75	1	17.67

(b) Results for LWB benchmarks

	on		ngl		off
k_ph_p	8	43.70	8	58.79	6
kt_ph_p	8	31.12	7		5
s4_ipc_p	12	32.16	5		5
s4_ph_p	8	32.38	7		5

Table 5.11: Evaluation results for disjoint branching

mCNF-*d02 and hCNF. For experiments on qbfML, we employ heuristic E.

5.4 Disjoint Branching

Table 5.11 compares disjoint branching and no-good lists with the naïve approach.

While we have not encountered a class of terms on which it is beneficial to use neither disjoint branching nor no-good lists, Table 5.11 shows several examples for cases where disjoint branching or no-good lists can lead to a noticeable increase in performance. Disjoint branching sporadically performs worse than the approach using no-good lists on the set of hybrid terms with global modalities. On the other hand, the results of disjoint branching are often significantly better than the results obtained with no-good lists.

On the other problem sets we investigated, there are almost never noticeable differences between the three settings.

Since, on the average, disjoint branching performs better than the approach using no-good lists, from now on, we will use disjoint branching.

5.5 Lazy Branching

Table 5.12 shows the effects of different variants of lazy branching.

For terms that have a large modal depth and contain propositions only on the lowest level, lazy branching on boxes can be beneficial. Lazy branching on propositional literals leads to a significantly better performance on terms that have a small modal depth. An improvement of the performance can also be observed for some terms that contain hard propositional subproblems.

However, Table 5.12(a) also shows several cases in which lazy branching has a negative effect on performance. On the hybrid terms with global modalities we used for testing, enabling the propositional variant of lazy branching often has an adverse effect. On one class of terms (qbL-C*-V4-D6), we also observe longer decision times in case both kinds of lazy branching are combined.

On the LWB benchmarks, the results are also mixed, as it can be seen in Table 5.12(b). Note that lazy branching on disjunctions with boxes has no effect on the benchmarks for KT and S4 due to reflexivity.

Despite the mixed results, we will use lazy branching on both propositional literals and boxes for the tests presented in the following sections.

5.6 Boolean Constraint Propagation

Table 5.13 shows the impact of boolean constraint propagation on performance.

In many cases, the use of the simple variant of boolean constraint propagation that considers only the first disjunction on the agenda leads to a significant speedup.

The eager variant often leads to a further reduction of the decision times, although, in the worst case, the current implementation of the eager variant must, at each branching step, examine all disjunctions on the agenda. Among the cases we investigated, negative effects of the eager variant are only encountered on the class qbfML and the problem kt_ph_n of the LWB benchmarks. In the case of qbfML, the performance degradation may be related to the ordering heuristic we use for this class, which might cause a large number of pending disjunctions on the agenda.

Since the eager variant of BCP performs better on average, we use the eager variant when comparing the performance of Spartacus with other systems.

5.7 Comparison with Other Systems

After we have evaluated the impact of ordering heuristics and optimizations on the performance of Spartacus, we can compare its performance with other implementations.

(a) Results for randomly generated terms

	on		prop		box		off	
mCNF-c120v03d02	8	3.63	8	4.43	8	4.04	8	4.99
mCNF-c150v03d04	9	0.87	9	1.13	9	0.88	9	1.13
mCNF-c180v03d04	9	1.48	9	1.92	9	1.52	9	1.94
mCNF-c210v03d04	9	2.09	9	2.83	9	2.10	9	2.82
mCNF-c240v03d04	9	3.21	9	4.23	9	3.21	9	4.24
mCNF-c270v03d04	9	6.72	9	8.16	9	6.70	9	8.19
mCNF-c300v03d04	9	8.86	9	11.04	9	8.86	9	11.01
mCNF-c150v03d06	9	2.43	9	4.02	9	2.39	9	4.01
mCNF-c180v03d06	9	4.08	9	7.01	9	4.09	9	6.97
mCNF-c210v03d06	9	6.09	9	9.16	9	6.09	9	9.12
mCNF-c240v03d06	9	8.66	9	12.82	9	8.67	9	12.82
mCNF-c270v03d06	9	10.23	9	15.85	9	10.23	9	15.91
mCNF-c300v03d06	9	15.51	9	25.78	9	15.52	9	25.72
mCNF-c075v04d01	16	0.14	16	0.15	16	0.22	16	0.23
mCNF-c110v05d01	16	9.19	16	9.72	4	30.78	3	25.69
qbfS-C50-V16-D4	8	8.42	8	8.37	7	5.89	7	5.84
qbfS-C50-V16-D6	8	3.89	8	3.85	7	6.39	7	6.50
qbfML-C50-V8-D4	8	16.38	8	16.34	8	20.39	8	20.32
qbfML-C50-V8-D6	7	15.57	7	15.64	7	16.94	7	16.96
hCNF-C60	17	0.11	19	0.33	19	0.67	20	0.68
hCNF-C80	15	6.54	16	3.54	16	4.05	15	3.21
hCNF-C100	9	1.43	12	5.40	12	3.36	14	12.18
hCNF-C120	10	6.62	12	12.05	14	9.80	14	10.98
hCNF-C140	8	10.80	8	20.73	9	16.90	8	15.53
hCNF-C160	8	11.12	9	12.48	11	13.15	9	7.18
hCNF-C180	6	19.57	4	14.90	8	22.98	6	15.25
hCNF-C200	11	21.39	9	13.58	13	20.49	13	22.99
qbfL-C10-V4-D6	8	23.47	8	10.50	8	13.32	8	13.09
qbfL-C20-V4-D6	1	19.89	6	32.04	7	28.75	7	28.50
qbfL-C30-V4-D6	1	17.50	3	39.25	2	19.88	2	19.58
qbfL-C40-V4-D6	5	6.88	6	12.08	6	12.59	6	12.43
qbfL-C50-V4-D6	8	7.59	8	7.26	8	5.17	8	5.16

(b) Results for LWB benchmarks

	on		prop		box		off	
k_branch_n	10		10		14	35.16	14	41.10
k_branch_p	16		16		21	30.41	21	34.82
k_ph_n	20		21	0.19	13		12	
k_ph_p	7		8	43.39	6		6	
kt_branch_n	10		10		14	41.70	14	41.83
kt_branch_p	16		16		21	35.83	21	35.75
kt_ph_n	21	28.52	21	28.59	17		17	
kt_ph_p	8	31.19	8	31.11	7		7	
s4_branch_n	14	24.86	14	24.91	14	41.11	14	41.20
s4_branch_p	21	21.55	21	21.59	21	35.00	21	35.21
s4_ph_p	8	32.22	8	32.29	7		7	
s4_s5_n	16		16		21	50.05	21	49.80

Table 5.12: Evaluation results for lazy branching

(a) Results for randomly generated terms

	eager		on		off	
mCNF-c090v03d02	9	0.28	9	0.15	9	0.68
mCNF-c120v03d02	8	3.14	8	3.63	6	1.33
mCNF-c120v03d04	9	0.52	9	0.50	9	4.27
mCNF-c150v03d04	9	0.89	9	0.87	9	8.95
mCNF-c180v03d04	9	1.44	9	1.48	9	16.74
mCNF-c210v03d04	9	2.02	9	2.09	9	26.72
mCNF-c240v03d04	9	2.97	9	3.21	8	37.89
mCNF-c270v03d04	9	5.74	9	6.72	5	50.82
mCNF-c300v03d04	9	7.72	9	8.86	0	
mCNF-c120v03d06	9	1.23	9	1.33	9	17.94
mCNF-c150v03d06	9	2.10	9	2.43	8	33.92
mCNF-c180v03d06	9	3.80	9	4.08	4	54.28
mCNF-c210v03d06	9	5.39	9	6.09	0	
mCNF-c240v03d06	9	7.62	9	8.66	0	
mCNF-c270v03d06	9	8.21	9	10.23	0	
mCNF-c300v03d06	9	12.65	9	15.51	0	
mCNF-c075v04d01	16	0.05	16	0.14	16	0.23
mCNF-c110v05d01	16	2.50	16	9.19	16	17.37
hCNF-C20	20	0.00	20	0.00	3	0.00
hCNF-C40	20	0.01	20	0.02	0	
hCNF-C60	19	0.51	17	0.11	0	
hCNF-C80	19	2.71	15	6.54	0	
hCNF-C100	15	3.55	9	1.43	0	
hCNF-C120	11	6.47	10	6.62	0	
hCNF-C140	7	10.49	8	10.80	0	
hCNF-C160	7	5.91	8	11.12	0	
hCNF-C180	9	9.97	6	19.57	0	
hCNF-C200	10	6.12	11	21.39	0	
qbfML-C10-V8-D6	4	7.58	4	6.70	4	6.82
qbfML-C20-V8-D6	6	23.07	7	25.26	7	20.71
qbfML-C30-V8-D6	6	14.64	6	9.47	6	9.92
qbfML-C40-V8-D6	7	23.43	7	12.45	8	12.70
qbfML-C50-V8-D6	7	33.99	7	15.57	7	14.47

(b) Results for LWB benchmarks

	eager		on		off
k_ph_n	21	0.28	20		2
k_ph_p	8	40.23	7		2
kt_ph_n	20		21	28.52	1

Table 5.13: Evaluation results for boolean constraint propagation

	Spartacus				CWB		FaCT++		HTab		*SAT	
	I		II									
mCNF-c030v03d02	9	0.0	9	0.0	9	1.6	9	0.1	9	1.4	9	0.0
mCNF-c060v03d02	9	0.0	9	0.0	0		7	4.6	0		9	12.3
mCNF-c090v03d02	9	0.3	8	1.8	0		1	58.6	0		6	19.4
mCNF-c120v03d02	8	3.3	6	11.3	0		0		0		1	19.2
mCNF-c030v03d04	9	0.3	9	0.1	0		9	3.2	9	6.2	9	19.9
mCNF-c060v03d04	9	2.6	9	0.3	0		0		0		0	
mCNF-c090v03d04	9	10.1	9	0.5	0		0		0		0	
mCNF-c120v03d04	8	26.6	9	0.9	0		0		0		0	
mCNF-c150v03d04	3	29.4	9	1.3	0		0		0		0	
mCNF-c180v03d04	2	38.2	9	2.0	0		0		0		0	
mCNF-c210v03d04	0		9	2.6	0		0		0		0	
mCNF-c240v03d04	0		9	3.7	0		0		0		0	
mCNF-c270v03d04	0		9	6.6	0		0		0		0	
mCNF-c300v03d04	0		9	8.6	0		0		0		0	
mCNF-c030v03d06	9	1.6	9	0.9	0		0		5	15.6	0	
mCNF-c060v03d06	9	10.0	9	1.9	0		0		0		0	
mCNF-c090v03d06	8	34.4	9	3.0	0		0		0		0	
mCNF-c120v03d06	2	51.6	9	4.3	0		0		0		0	
mCNF-c150v03d06	0		9	6.1	0		0		0		0	
mCNF-c180v03d06	0		9	8.6	0		0		0		0	
mCNF-c210v03d06	0		9	11.2	0		0		0		0	
mCNF-c240v03d06	0		9	14.4	0		0		0		0	
mCNF-c270v03d06	0		9	15.9	0		0		0		0	
mCNF-c300v03d06	0		9	21.4	0		0		0		0	
mCNF-c075v04d01	16	0.1	8	9.8	0		16	0.2	0		16	0.0
mCNF-c110v05d01	16	4.7	0		0		10	32.9	0		16	0.4

Table 5.14: Comparison on mCNF

	Spartacus				HTab	
	I		II			
hCNF-C20	20	0.0	20	0.3	20	0.1
hCNF-C40	20	0.0	14	2.4	20	0.1
hCNF-C60	19	0.5	7	0.9	19	0.7
hCNF-C80	19	2.8	0		13	1.5
hCNF-C100	15	3.9	0		7	2.7
hCNF-C120	11	6.9	0		4	6.0
hCNF-C140	7	11.5	0		4	2.2
hCNF-C160	7	6.4	0		7	3.1
hCNF-C180	9	10.9	0		7	3.9
hCNF-C200	10	6.8	0		9	2.0

Table 5.15: Comparison on hCNF

	Spartacus				CWB		FaCT++		HTab		*SAT	
	I		II									
qbfS-C10-V4-D4	8	0.0	8	0.0	8	0.4	8	0.0	8	0.2	8	0.0
qbfS-C20-V4-D4	8	0.0	8	0.0	8	3.2	8	0.1	8	0.5	8	0.1
qbfS-C30-V4-D4	8	0.1	8	0.0	5	8.4	8	0.7	7	1.6	8	0.2
qbfS-C40-V4-D4	8	0.1	8	0.0	4	13.4	8	0.8	8	1.9	8	0.2
qbfS-C50-V4-D4	8	0.1	8	0.0	2	18.3	8	0.3	8	1.4	8	0.2
qbfS-C10-V4-D6	8	0.0	8	0.0	8	1.3	8	0.0	8	1.0	8	0.0
qbfS-C20-V4-D6	8	0.0	8	0.0	7	7.0	8	0.1	7	5.7	8	0.4
qbfS-C30-V4-D6	8	0.2	8	0.1	3	14.8	5	11.3	4	27.8	8	1.4
qbfS-C40-V4-D6	8	0.2	8	0.1	1	6.8	6	2.2	2	39.1	8	1.2
qbfS-C50-V4-D6	8	0.2	8	0.1	0		6	6.7	1	36.8	8	1.0
qbfS-C10-V8-D4	8	0.0	8	0.0	8	2.6	8	0.0	7	7.3	8	0.1
qbfS-C20-V8-D4	8	0.1	8	0.0	4	16.9	7	0.2	1	39.6	8	0.7
qbfS-C30-V8-D4	8	0.2	8	0.1	0		2	24.7	0		8	8.3
qbfS-C40-V8-D4	8	0.6	8	0.3	0		2	30.9	0		8	13.4
qbfS-C50-V8-D4	8	2.0	8	0.4	0		2	9.4	0		8	21.5
qbfS-C10-V8-D6	8	0.0	8	0.0	8	6.5	8	0.2	4	26.4	8	0.1
qbfS-C20-V8-D6	8	0.1	8	0.1	0		7	8.0	0		8	0.9
qbfS-C30-V8-D6	8	0.6	8	0.4	0		0		0		8	18.1
qbfS-C40-V8-D6	8	4.0	8	1.1	0		0		0		2	34.0
qbfS-C50-V8-D6	7	15.3	8	5.1	0		0		0		0	
qbfS-C10-V16-D4	8	0.0	8	0.0	7	19.4	8	0.1	0		8	0.1
qbfS-C20-V16-D4	8	0.1	8	0.1	0		7	1.3	0		8	1.6
qbfS-C30-V16-D4	8	1.2	8	0.6	0		3	25.2	0		8	19.2
qbfS-C40-V16-D4	8	0.8	8	0.5	0		0		0		0	
qbfS-C50-V16-D4	6	14.3	8	9.9	0		0		0		0	
qbfS-C10-V16-D6	8	0.0	8	0.0	6	18.1	8	0.1	0		8	0.1
qbfS-C20-V16-D6	8	0.2	8	0.1	0		7	7.4	0		8	1.5
qbfS-C30-V16-D6	8	0.5	8	0.3	0		0		0		7	25.1
qbfS-C40-V16-D6	8	3.4	8	1.8	0		1	3.3	0		1	46.6
qbfS-C50-V16-D6	7	5.9	8	5.3	0		0		0		0	
qbfL-C10-V4-D4	8	2.5	8	1.4	8	19.9	8	0.5	7	3.7	3	43.1
qbfL-C20-V4-D4	8	5.4	8	4.0	5	27.9	8	3.3	1	18.4	0	
qbfL-C30-V4-D4	8	5.9	8	3.6	5	25.3	8	1.6	1	36.1	1	1.3
qbfL-C40-V4-D4	8	4.9	8	1.2	6	40.1	8	0.3	1	28.4	7	28.9
qbfL-C50-V4-D4	8	6.0	8	1.5	3	54.9	8	0.6	2	38.8	4	6.7
qbfL-C10-V4-D6	4	30.2	8	27.1	1	16.6	6	22.4	2	15.9	0	
qbfL-C20-V4-D6	0		1	21.7	0		1	13.9	0		0	
qbfL-C30-V4-D6	0		1	17.6	0		0		0		0	
qbfL-C40-V4-D6	0		5	6.9	0		6	6.8	0		0	
qbfL-C50-V4-D6	0		8	7.4	0		8	7.0	0		0	

Table 5.16: Comparison on non-modalized QBFs

	Spartacus				CWB		FaCT++		HTab		*SAT	
	I		II									
qbfMS-C10-V4-D4	8	0.0	8	0.0	8	3.1	8	0.0	0		8	0.1
qbfMS-C20-V4-D4	8	0.2	8	0.1	3	18.2	8	0.1	0		8	0.4
qbfMS-C30-V4-D4	8	0.4	8	0.2	0		8	8.0	0		8	0.7
qbfMS-C40-V4-D4	8	0.6	8	0.3	0		8	2.7	0		8	0.9
qbfMS-C50-V4-D4	8	0.7	8	0.5	2	15.7	7	1.1	0		8	0.6
qbfMS-C10-V4-D6	8	0.0	8	0.0	8	9.2	8	0.0	0		8	0.1
qbfMS-C20-V4-D6	8	0.9	8	0.2	0		7	12.7	0		8	1.4
qbfMS-C30-V4-D6	8	6.7	8	0.5	0		3	3.7	0		8	4.5
qbfMS-C40-V4-D6	8	12.0	8	1.1	0		4	8.0	0		8	12.3
qbfMS-C50-V4-D6	8	6.6	8	1.9	0		3	8.6	0		8	7.2
qbfMS-C10-V8-D4	8	0.1	8	0.0	6	14.1	8	0.1	0		8	0.1
qbfMS-C20-V8-D4	8	0.5	8	0.2	0		7	1.4	0		8	3.1
qbfMS-C30-V8-D4	8	6.5	8	0.8	0		2	1.3	0		5	22.5
qbfMS-C40-V8-D4	3	35.3	8	11.7	0		0		0		2	25.9
qbfMS-C50-V8-D4	6	27.5	8	9.0	0		2	41.8	0		3	31.8
qbfMS-C10-V8-D6	8	0.1	8	0.1	0		8	2.2	0		8	0.2
qbfMS-C20-V8-D6	8	1.5	8	0.4	0		6	8.1	0		8	6.4
qbfMS-C30-V8-D6	7	8.4	8	2.0	0		1	30.4	0		2	22.0
qbfMS-C40-V8-D6	1	17.7	7	11.8	0		1	25.7	0		0	
qbfMS-C50-V8-D6	0		6	34.3	0		0		0		0	
qbfMS-C10-V16-D4	8	0.1	8	0.1	1	23.4	8	0.1	0		8	0.3
qbfMS-C20-V16-D4	8	5.9	8	0.5	0		8	7.4	0		8	7.0
qbfMS-C30-V16-D4	6	18.9	8	5.2	0		0		0		1	29.3
qbfMS-C40-V16-D4	0		4	24.0	0		0		0		0	
qbfMS-C50-V16-D4	0		3	30.8	0		0		0		0	
qbfMS-C10-V16-D6	8	0.1	8	0.1	0		8	0.1	0		8	0.4
qbfMS-C20-V16-D6	8	1.6	8	0.5	0		7	6.8	0		8	8.3
qbfMS-C30-V16-D6	5	21.1	8	9.4	0		0		0		2	38.6
qbfMS-C40-V16-D6	2	14.1	5	14.7	0		0		0		0	
qbfMS-C50-V16-D6	0		1	24.4	0		0		0		0	
qbfML-C10-V4-D4	8	0.2	5	30.8	0		8	8.0	0		8	0.5
qbfML-C20-V4-D4	8	0.2	0		0		8	7.6	0		8	1.0
qbfML-C30-V4-D4	8	0.3	0		0		8	17.2	0		8	1.8
qbfML-C40-V4-D4	8	0.4	0		0		8	15.1	0		8	2.3
qbfML-C50-V4-D4	8	0.6	0		0		6	20.6	0		8	3.4
qbfML-C10-V4-D6	8	0.3	0		0		8	14.2	0		8	0.6
qbfML-C20-V4-D6	8	0.5	0		0		3	30.1	0		8	1.5
qbfML-C30-V4-D6	8	0.6	0		0		7	32.3	0		8	1.7
qbfML-C40-V4-D6	8	0.8	0		0		3	42.0	0		8	4.6
qbfML-C50-V4-D6	8	1.1	0		0		0		0		8	6.3
qbfML-C10-V8-D4	8	0.5	0		0		2	40.5	0		7	7.6
qbfML-C20-V8-D4	8	1.0	0		0		0		0		6	40.1
qbfML-C30-V8-D4	8	1.3	0		0		0		0		4	33.6
qbfML-C40-V8-D4	8	1.6	0		0		0		0		0	
qbfML-C50-V8-D4	8	2.0	0		0		0		0		1	56.9
qbfML-C10-V8-D6	8	0.9	0		0		0		0		8	15.7
qbfML-C20-V8-D6	8	2.0	0		0		0		0		7	33.6
qbfML-C30-V8-D6	8	2.8	0		0		0		0		3	34.4
qbfML-C40-V8-D6	8	3.4	0		0		0		0		0	
qbfML-C50-V8-D6	8	4.1	0		0		0		0		0	

Table 5.17: Comparison on modalized QBFs

	Spartacus				CWB		FaCT++		HTab		*SAT	
	I		II									
k_branch_n	9		10		12	37.2	10		4		12	48.2
k_branch_p	11		16		21	6.2	9		5		18	
k_d4_n	21	0.2	21	0.6	21	7.3	21	27.6	6		21	0.2
k_d4_p	21	0.1	21	0.1	21	3.7	21	0.0	21	50.1	21	0.0
k_dum_n	21	0.0	21	0.0	21	0.1	21	0.0	21	0.1	21	0.0
k_dum_p	21	0.0	21	0.0	21	0.1	21	0.0	21	0.1	21	0.0
k_grz_n	21	0.0	21	0.0	21	0.4	21	0.0	21	0.1	21	0.0
k_grz_p	21	0.0	21	0.0	21	0.5	21	0.0	21	0.1	21	0.0
k_lin_n	21	0.0	21	0.0	21	13.6	21	0.1	21	0.1	13	
k_lin_p	21	0.0	21	0.0	21	0.2	21	0.0	21	0.1	21	0.0
k_path_n	21	0.6	21	0.6	21	50.9	21	0.1	9		21	0.2
k_path_p	21	0.6	21	0.5	21	46.5	21	0.1	10		21	0.1
k_ph_n	21	1.2	21	1.2	9		13		16		21	1.9
k_ph_p	8	47.0	8	41.4	7		7		6		8	3.8
k_poly_n	21	0.2	21	0.2	21	9.6	21	0.1	21	2.1	21	0.1
k_poly_p	21	0.2	21	0.1	21	7.9	21	0.1	21	2.6	21	0.1
k_t4p_n	21	0.1	21	0.0	21	4.1	21	0.3	5		21	0.1
k_t4p_p	21	0.0	21	0.0	21	3.3	21	0.1	8		21	0.0

Table 5.18: Comparison on LWB benchmarks (K)

In particular, we compare the running time of Spartacus with the running times of the following systems:

- CWB [27] – A prototype reasoner for basic modal logic with reflexivity and transitivity, implemented in C++. CWB can use different caching techniques. For the comparison, we used the options `-oa` (all optimizations) and `-ogc` (global caching non-DFS).
- FaCT++ [64], version 1.2.1 – A reasoner for the very rich description logic $\mathcal{SROIQ}(\mathbf{D})$ [37]. FaCT++ is implemented in C++. We used the default settings.
- HTab [31], version 1.3.5 – A reasoner for the hybrid logic $\mathcal{H}(@, E, D)$, implemented in Haskell. In addition to global modalities supported by Spartacus, HTab supports the difference modality D. We used the default settings.
- *SAT [60], version 1.3 – A reasoner for the basic modal logic K, implemented in C. *SAT implements both satisfiability and unsatisfiability caching. We used the default compile-time and run-time settings.

Spartacus is implemented in Standard ML and compiled with MLton[1].

Since we have not determined a single configuration that performs well on all sets of terms we investigated, we have selected two configurations of Spartacus to be compared to other systems. They differ in the used ordering heuristic and in the employed data structure for pattern-based blocking as follows:

I: Ordering heuristic B as described in Section 5.3 is employed. The tree-based pattern store is used for blocking.

[1] available from `http://mlton.org/`

	Spartacus				CWB	
	I		II			
kt_45_n	21	0.1	21	0.1	21	4.1
kt_45_p	21	0.1	21	0.1	21	2.3
kt_branch_n	9		10		12	28.2
kt_branch_p	21	0.1	16		21	0.6
kt_dum_n	21	0.0	21	0.0	21	0.2
kt_dum_p	21	0.0	21	0.0	21	0.3
kt_grz_n	21	0.0	21	0.0	21	0.5
kt_grz_p	21	0.0	21	0.0	21	0.5
kt_md_n	7	13.4	7	13.4	7	10.6
kt_md_p	5		6	25.4	5	
kt_path_n	21	36.9	21	36.8	5	
kt_path_p	21	30.3	21	25.7	6	
kt_ph_n	14		20	49.7	8	
kt_ph_p	7		8	36.1	6	
kt_poly_n	8		10	29.7	9	
kt_poly_p	5		21	1.6	18	
kt_t4p_n	21	0.1	21	0.1	21	40.5
kt_t4p_p	21	0.1	21	1.5	21	9.8
s4_45_n	21	1.3	21	0.3	21	30.9
s4_45_p	21	0.0	21	0.1	21	18.8
s4_branch_n	9		13	59.7	12	
s4_branch_p	21	0.6	19		21	0.4
s4_grz_n	21	0.1	21	0.0	13	
s4_grz_p	21	0.0	21	0.0	21	0.9
s4_ipc_n	21	2.7	21	2.1	11	
s4_ipc_p	21	3.0	12		21	0.6
s4_md_n	21	24.8	21	19.7	10	
s4_md_p	9	50.5	9	37.1	9	31.8
s4_path_n	16	51.1	16	48.9	2	
s4_path_p	17	57.4	17	52.6	4	
s4_ph_n	10	34.9	9		3	
s4_ph_p	5		8	37.4	6	
s4_s5_n	16	57.2	16	54.5	9	
s4_s5_p	21	32.1	21	0.0	21	1.1
s4_t4p_n	21	0.3	21	0.3	17	
s4_t4p_p	21	0.1	21	0.2	21	6.4

Table 5.19: Comparison on LWB benchmarks (KT and S4)

II: Ordering heuristic D as described in Section 5.3 is employed. The list-based pattern store is used for blocking.

Both configurations share the following settings:

- disjuncts are processed in the following order: negated nominals, propositional literals, diamonds, boxes, satisfaction terms, existential modalities, nominals, universal modalities, conjunctions
- backjumping is enabled
- the eager variant of pattern-based blocking is employed
- the eager variant of boolean constraint propagation is employed
- disjoint branching is enabled
- lazy branching is fully enabled
- caching is disabled

In the tests, shown in Tables 5.14-5.19, the two configurations we have chosen for comparison seem to have complementary strengths and weaknesses. So, configuration I performs significantly better on our hybrid samples and on modalized QBFs, while configuration II is more effective on non-modalized QBFs and on the class mCNF-*v03*. Unfortunately, we were not able to find a single setting that would combine the strengths of the two configurations.

This said, the results of the comparison suggest that SPARTACUS is very competitive with the other systems in so far as reasoning performance is concerned. As shown in Section 5.2, an important factor contributing to this performance is pattern-based blocking. Since pattern-based blocking can be seen as a particular form of satisfiability caching, we expected it to have similar effects on performance as observed for *SAT [61] and CWB [27]. The comparisons with *SAT on QBFs and with CWB on the LWB benchmarks fully confirm this expectation.

Chapter 6

Conclusion

This chapter concludes the thesis with a summary and an outlook on possible future work.

6.1 Summary

SPARTACUS is a tableau-based prover for hybrid logic with global modalities, reflexive, transitive and serial relations. Termination in the presence of global modalities and transitivity is achieved through pattern-based blocking.

One of the key motivations for our work was to evaluate the effectiveness of pattern-based blocking in practice. The evaluation shows that the performance can improve significantly when pattern-based blocking is used. Performance of pattern-based blocking depends on the data structure used for storing patterns. We have implemented three data structures that seemed to be promising for that purpose, which are based on lists, bit matrices and trees, respectively. While the tree-based data structure outperforms the data structures based on bit matrices and lists in many cases, we have also encountered several classes of hybrid terms where the list-based data structure performs significantly better.

We have also integrated other optimization techniques into SPARTACUS, including normalization of terms, backjumping, boolean constraint propagation, disjoint branching, lazy branching and caching of unsatisfiable terms.

We have investigated a novel optimization technique called lazy branching, which aims at reducing the search space by delaying branching on disjunctions that contain propositional literals or boxes. Although we have encountered several cases where lazy branching can have an adverse effect on performance, there are many cases where it seems to be an interesting optimization technique.

6.2 Outlook

This section discusses several issues that remained open during the work on this thesis but which might be interesting subjects for future investigation. It also mentions optimizations that may be integrated into future versions of SPARTACUS.

6.2.1 Difference Modality

It seems to be possible to add support for the difference modality D and its dual $\bar{D}$ to SPARTACUS with moderate effort.

Since the tableau rule for D is a nominal generating rule, a blocking condition is required to achieve termination in the presence of the difference modality [42]. Hence, some control mechanism must be implemented that keeps track of whether difference modalities are blocked. The operator $\bar{D}$, on the other hand, can be handled using the existing infrastructure for global modalities with only minor modifications.

6.2.2 Branching Heuristics

Currently, SPARTACUS allows for ordering heuristics that order disjunctions according to their dependency sets or the age of the corresponding nodes. Several more sophisticated heuristics have been proposed, which might have a positive effect on performance. For instance, learning-based disjunct selection [57] is an ordering heuristic that assigns priorities to disjuncts based on how often they have been involved in a clash. Other examples include ordering strategies based on the size or the maximum modal depth of the disjuncts [63].

For such heuristics, it may be reasonable to assign priorities to disjuncts instead of disjunctions. As a consequence, it would not be possible to handle disjunctions on the agenda in the same way as it is currently the case. In particular, an efficient mechanism to determine which disjunctions are satisfied by the selection of a certain disjunct seems necessary.

6.2.3 Optimization Techniques

There are several ideas how the optimization techniques implemented so far could be extended in order to improve their effectiveness.

The eager variant of boolean constraint propagation could be improved if watched literals were used to determine disjunctions that can be handled deterministically.

Lazy branching on disjunctions that contain boxes can be improved for terms that contain more than one relational variable. Currently, branching on such disjunctions is only delayed as long as no diamond has been derived as a constraint on the corresponding node. A more sophisticated version would delay disjunctions that contain r-boxes until an r-diamond is added to the corresponding

node. A variant of lazy branching that delays disjunctions that contain negated nominals is not implemented yet.

6.2.4 Evaluation

We have compared the practical fitness of several ordering heuristics and optimizations on different classes of formulas. However, we did not determine a single configuration that performs reasonably well on all classes of formulas investigated. This is unfortunate, as users may be required to perform the possibly daunting task of finding a good configuration themselves.

It would be desirable if, for each input term, a configuration that is likely to be good could be suggested automatically based on features of the input term like modal depth, number of variables or operator frequencies. Further evaluation might reveal relationships between features of input terms and promising configurations.

Appendix A

Input Syntax

File	:=	*Disjunction* | *Restrictions Disjunction*
Disjunction	:=	*Conjunction* ['|' *Disjunction*]
Conjunction	:=	*Literal* ['&' *Conjunction*]
Literal	:=	'0' | '1' | *Var* | '=' *Var* | '~' *Literal*
	|	'<' *Var* '>' *Literal* | '[' *Var* ']' *Literal*
	|	'A' *Literal* | 'E' *Literal* |
	|	'@' *Var Literal* | *Var* ':' *Literal* |
	|	'(' *Disjunction* ')'
Var	:=	$[a-z][a-z\ 0-9]^*$
Varlist	:=	*Var* [, *Varlist*]
Restrictions	:=	*Restriction* [*Restrictions*]
Restriction	:=	*Reflexivity* | *Transitivity* | *Seriality* |
Reflexivity	:=	'{reflexive: *}' | '{reflexive:' *Varlist* '}'
Transitivity	:=	'{transitive: *}' | '{transitive:' *Varlist* '}'
Seriality	:=	'{serial: *}' | '{serial:' *Varlist* '}'

Bibliography

[1] Carlos Areces and Maarten de Rijke. From description to hybrid logics, and back. In F. Wolter, H. Wansing, M. de Rijke, and M. Zakharyaschev, editors, *Advances in Modal Logic. Volume 3*. CSLI Publications, 2001.

[2] Carlos Areces and Juan Heguiabehere. Hylores 1.0: Direct resolution for hybrid logics. In A. Voronkov, editor, *Proceedings of CADE-18*, pages 156–160, Copenhagen, Denmark, July 2002.

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

[4] Carlos Areces and Balder ten Cate. Hybrid logics. In Patrick Blackburn, F. Wolter, and Johan van Benthem, editors, *Handbook of Modal Logics*. Elsevier, 2006.

[5] Andrew Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, U. of Oregon, 1995.

[6] Peter Balsiger, Alain Heuerding, and Stefan Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *J. Autom. Reasoning*, 24(3):297–317, 2000.

[7] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orłowska, editors, *Proc. European Workshop: Logics in Artificial Intelligence, JELIA*, volume 1126 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

[8] Sean Bechhofer. Hoolet, 2004. URL `http://owl.man.ac.uk/hoolet/`.

[9] Christoph Benzmüller and Lawrence Paulson. Festschrift in honour of Peter B. Andrews on his 70th birthday. Studies in Logic and the Foundations of Mathematics, chapter Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. IFCoLog, 2008. To appear.

[10] Evert W. Beth. Semantic entailment and formal derivability. *Medlingen der Koninklijke Nederlandse Akademie van Wetenschapen*, 18(13):309–342, 1955.

[11] Patrick Blackburn and Miroslava Tzakova. Hybrid completeness. *Logic Journal of the IGPL*, 6(4):625–650, 1998.

[12] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001. ISBN 0 521 52714 7 (pbk).

[13] Thomas Bolander and Patrick Blackburn. Termination for hybrid tableaus. *J. Log. Comput*, 17(3):517–554, 2007.

[14] Thomas Bolander and Torben Braüner. Tableau-based decision procedures for hybrid logic. *J. Log. Comput.*, 16(6):737–763, 2006.

[15] Alex Borgida, Maurizio Lenzerini, and Riccardo Rosati. Description logics for databases. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 462–484. Cambridge University Press, Cambridge, England, 2003.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2001.

[17] Marcello D'Agostino. Are tableaux an improvement on truth-tables? Cut-free proofs and bivalence. *Journal of Logic, Language, and Information*, 1(3):235–252, 1992.

[18] Francesco M. Donini and Fabio Massacci. EXPTIME tableaux for ALC. *Artif. Intell*, 124(1):87–138, 2000.

[19] Melvin Fitting. Tableau methods of proof for modal logics. *Notre Dame Journal of Formal Logic*, XIII(2), 1972.

[20] Enrico Franconi. Natural language processing. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 450–461. Cambridge University Press, Cambridge, England, 2003.

[21] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[22] Daniel Gallin. *Intensional and Higher-Order Modal Logic*, volume 19 of *Mathematics Studies.* North-Holland, Amsterdam, 1975.

[23] L. T. F. Gamut. *Logic, Language and Meaning*, volume 2. The University of Chicago Press, 1991.

[24] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh (PA), USA, 1979.

[25] Enrico Giunchiglia and Armando Tacchella. A subset-matching size-bounded cache for satisfiability in modal logics. In Roy Dyckhoff, editor, *TABLEAUX*, volume 1847 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2000. ISBN 3-540-67697-X.

[26] Fausto Giunchiglia and Roberto Sebastiani. Building decision procedures for modal logics from propositional decision procedures — the case study of modal K. *Lecture Notes in Computer Science*, 1104:583–??, 1996. ISSN 0302-9743.

[27] Rajeev Goré and Linda Postniece. An experimental evaluation of global caching for (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2008. ISBN 978-3-540-71069-1.

[28] Volker Haarslev and Ralf Möller. RACER system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, 2001. ISBN 3-540-42254-4.

[29] Moritz Hardt and Gert Smolka. Higher-order syntax and saturation algorithms for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 174(6):15–27, 2007. Proceedings of the International Workshop on Hybrid Logic (HyLo 2006).

[30] K. Jaakko J. Hintikka. Form and content in quantification theory. *Acta Philosophica Fennica*, 8:7–55, 1955.

[31] Guillaume Hoffmann and Carlos Areces. HTab : a terminating tableaux system for hybrid logic. In Stéphane Demri and Carlos Areces, editors, *Method For Modalities 5*, Cachan, France, 2007.

[32] Jörg Hoffmann and Jana Koehler. A new method to index and query sets. In Thomas Dean, editor, *IJCAI*, pages 462–467. Morgan Kaufmann, 1999. ISBN 1-55860-613-0.

[33] Ian Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[34] Ian Horrocks and Peter F. Patel-Schneider. Optimizing description logic subsumption. *J. of Logic and Computation*, 9(3):267–293, 1999.

[35] Ian Horrocks, Deborah McGuinness, and Christopher Welty. Digital libraries and web-based information systems. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter 14, pages 427–449. Cambridge University Press, 2003. ISBN 0-521-78176-0.

[36] Ian Horrocks, Ullrich Hustadt, Ulrike Sattler, and Renate Schmidt. Computational modal logic. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, chapter 4, pages 181–245. Elsevier, 2006.

[37] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006. ISBN 978-1-57735-271-6.

[38] Ullrich Hustadt and Renate A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In Roy Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *LNAI*, pages 67–71. Springer, 2000. ISBN 3-540-67697-X.

[39] Ullrich Hustadt and Renate A. Schmidt. Simplification and backjumping in modal tableau. In Harrie de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98)*, volume 1397 of *LNAI*, pages 187–201, Berlin, May 5–8 1998. Springer. ISBN 3-540-64406-7.

[40] Mark Kaminski and Gert Smolka. Terminating tableau systems for hybrid logic with difference and converse. Technical report, Saarland University, 2008.

[41] Mark Kaminski and Gert Smolka. A straightforward saturation-based decision procedure for hybrid logic. In Jörgen Villadsen, Thomas Bolander, and Torben Braüner, editors, *International Workshop on Hybrid Logic 2007 (HyLo 2007)*, Dublin, Ireland, 2007.

[42] Mark Kaminski and Gert Smolka. Hybrid tableaux for the difference modality. In Carlos Areces and Stéphane Demri, editors, *Proc. 5th Workshop on Methods for Modalities (M4M-5)*, Cachan, France, 2007. To appear in ENTCS.

[43] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading MA, second edition, 1998. ISBN 0-201-89685-0.

[44] Saul A. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9: 67–96, 1963.

[45] Richard E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. Comput*, 6(3):467–480, 1977.

[46] Fabio Massacci. Strongly analytic tableaux for normal modal logics. In Alan Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction (CADE'94)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 723–737, Berlin, 1994. Springer-Verlag.

[47] Fabio Massacci. Design and results of the Tableaux-99 Non-classical (Modal) Systems comparison. In Neil V. Murray, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-99)*, volume 1617 of *LNAI*, pages 14–18, Berlin, June 07–11 1999. Springer. ISBN 3-540-66086-0.

[48] Fabio Massacci and Francesco M. Donini. Design and results of TANCS-2000 non-classical (modal) systems comparison. In Roy Dyckhoff, editor, *TABLEAUX*, volume 1847 of *Lecture Notes in Computer Science*, pages 52–56. Springer, 2000. ISBN 3-540-67697-X.

[49] David G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[50] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[51] Boris Motik, Rob Shearer, and Ian Horrocks. Optimizing the Nominal Introduction Rule in (Hyper)Tableau Calculi. In *Proc. of the 21st Int. Workshop on Description Logics (DL 2008)*, Dresden, Germany, May 13–16 2008. To appear.

[52] Peter F. Patel-Schneider. System description: DLP. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 297–301. Springer, 2000. ISBN 3-540-67664-3.

[53] Alan L. Rector. Medical informatics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 406–426. Cambridge University Press, Cambridge, England, 2003.

[54] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *IJCAI*, pages 466–471, 1991.

[55] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[56] João P. Marques Silva and Karem A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *ICTAI*, pages 467–469, 1996.

[57] Evren Sirin, Bernardo Cuenca Grau, and Bijan Parsia. From wine to water: Optimizing description logic reasoning for nominals. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 90–99. AAAI Press, 2006. ISBN 978-1-57735-271-6.

[58] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem*, 5(2):51–53, 2007.

[59] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking. *Artificial Intelligence*, 9(2):135–196, 1977.

[60] Armando Tacchella. *SAT system description. In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *Description Logics*, volume 22 of *CEUR Workshop Proceedings*. CEUR-WS.org, 1999.

[61] Armando Tacchella. Evaluating *SAT on TANCS 2000 benchmarks. In Roy Dyckhoff, editor, *TABLEAUX*, volume 1847 of *Lecture Notes in Computer Science*, pages 77–81. Springer, 2000. ISBN 3-540-67697-X.

[62] Armando Tacchella and Roberto Sebastiani. K-CNF-generator, 1999. URL `http://www.mrg.dist.unige.it/~tac/StarSAT/getting_StarSAT.html`.

[63] Dmitry Tsarkov and Ian Horrocks. Ordering heuristics for description logic reasoning. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 609–614, 2005.

[64] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

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

[66] Christopher A. Welty. Software engineering. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 373–385. Cambridge University Press, Cambridge, England, 2003.