EFFICIENT AND GENERIC REASONING FOR MODAL LOGICS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2008

By
Zhen Li
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Broadly speaking, existing modal logic theorem provers can be classified as belonging to two categories: the category of efficient and highly-optimized provers, and the category of generic, usually slow, provers. This thesis investigates and develops reasoning for modal logic which aims to maximize both efficiency and generality. For this purpose, an automated reasoning system, called MLTP, has been developed, which implements a labelled tableau algorithm.

Several optimization techniques for modal logic reasoning are incorporated in MLTP and discussed in this thesis. We describe a dynamic backtracking algorithm for modal logics. A mechanism of forward reasoning is proposed for modal logic reasoning. Forward reasoning consists of two techniques, forward checking and forward implication. Simplification is also implemented to remove redundancy. Among these optimization techniques advanced backtracking methods have proved to be most effective. Evaluation results show that dynamic backtracking improves the performance significantly. The effectiveness of forward checking is less marked however and needs to be investigated further.

To facilitate generic reasoning, MLTP incorporates a generic data structure, a low-level macro language, an event-handler framework and a high-level specification language. The generic data structure stores data and its relations in highly connected relation-based structures to provide different kinds of data processing in a uniform way. The low-level macro language uses a small set of standard instructions to perform complicated operations. The event-handler framework divides the system into loosely connected modules. Each module can be defined using the low-level macro language, whereas the pairing of events and handlers is controlled by the high-level specification language. The high-level specification language can be used to specify the syntax and grammar of the logics, the inference rules to be used and the strategies of the prover.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I gratefully acknowledge the help and support of my supervisor, Renate Schmidt. Without her assistance the work presented in this thesis would not have been possible.

I also benefitted from useful discussions with Alexandre Riazanov and Dmitri Chubarov.

Finally, I would like to thank my wife, Lan Su, for her great support during the course of this research project.

This work was supported financially by a grant from the School of Computer Science.

Chapter 1

Introduction

The aim of this thesis is to investigate the realizability of efficient and generic reasoning in modal logic. We focus on tableau calculi for the modal logic K. For this purpose, an automated reasoning system, MLTP, has been developed. A labelled tableau calculus [Fit83] is implemented in MLTP. Previous studies have shown that a variety of optimization techniques can be used to improve the performance of modal logic reasoning. Among these techniques, backtracking techniques, in particular, backjumping have been found to have the most impact on a theorem prover's performance. A more sophisticated backtracking technique, dynamic backtracking [GM94], reduces however the search space more than backjumping. Forward reasoning mechanisms have been extensively studied for propositional logic by researchers in the area of CSPs (Constraint Satisfiability Problems). But the value of both dynamic backtracking and forward reasoning in modal logic has not previously been investigated. Apart from efficiency, generality is also a very important property of theorem provers. A generic prover can not only handle a good range of different logics but also allows users to specify their own logics, inference rules, and strategies.

1.1 Modal logic reasoning

Modal logic is a logic that uses modalities to handle concepts like possibility, existence and necessity [HC96]. The language of modal logic is an extension of the language of propositional logic with additional unary modal operators. The standard semantics of modal logics is defined by Kripke semantics [Kri63] or possible world semantics, which consist of relational structures, known as the

frames. Based on these frames, the validity and satisfiability of a modal formula are defined. The basic modal logic is called modal logic K. A variety of modal logics can be obtained by extending modal logic K with axiom schemas like 4, 5, T, B and D. These modal logics feature different accessibility relations with different properties determined by their axiom schemas. Satisfiability problems in modal logic are decidable with PSPACE-complete complexity [Lad77].

Featuring both enhanced expressive ability and the finite model property, modal logics are widely used as a formal way of modelling and reasoning about notions of knowledge, belief and time. The applications of modal logic include agent-based systems, linguistics, system verification, and protocol analysis. Modal logics can be seen as variants of description logics. Systems designed for modal logics can be used for description logic reasoning with little change. Description logics are used in the applications of knowledge representation. They are currently popular in research on the semantic web and ontology reasoning. Description logics provide the basis for standard web ontology languages used in these two areas.

Tableaux is a refutation approach that can be used as the decision procedure for a variety of logics. It proves the validity of a formula by looking for the proof of the unsatisfiability of the formula's negation.

There are many existing tableau theorem provers for modal logics or description logics. Some of them like FaCT [Hor97], RACER [HM01], DLP [PS98] and LWB [HJSS96] implement complicated procedures, sophisticated data structures and optimization techniques to try to get the best performance possible. These provers can only work on the logics they are designed for. Others provers like Lotrec [CFG$^+$01], TWB [AG03], TABLEAUX [Cat91] incorporate flexible frameworks and generic data structures to be able to cope with various situations. These kinds of provers can handle new logics specified by the users. Their deduction procedure and strategies are also customizable.

1.2 Efficient theorem proving

Optimization techniques are used by the theorem provers to improve their efficiency. The optimization techniques discussed in this thesis are the following:

- Simplification: removing the redundancy in the formulae using tautologies.
- Unit propagation: reducing non-deterministic expansions to deterministic

expansions.

- Lexical normalization: transforming a formula so that it only contains basic operators, which makes it easier to identify repeated occurrences of formulae.
- Complement splitting: simulating the DPLL [DP60] approach by adding a formula's positive occurrence and negative occurrence into two alternative branches respectively to guarantee that a certain model is explored only once.
- Backjumping and dynamic backtracking: using dependency information to avoid unnecessary backtracking.
- Caching: keeping the results of earlier deductions and reusing them when possible.
- Heuristics: utilizing strategies to handle a non-deterministic procedure in order to minimize the proof.

Dynamic backtracking improves backjumping by trying to avoid undoing the inference steps that have nothing to do with the clash during backtracking. Though backjumping has been studied and implemented in several modal logic or description logic theorem provers, there is hardly any published work that investigates applying dynamic backtracking to modal logics. In this thesis, we propose a dynamic backtracking algorithm for modal logics, which has been implemented in MLTP. We also provide the proofs for the algorithm's correctness and benchmarking results.

A mechanism of forward reasoning is introduced in this thesis for modal logic deduction. Using pre-computed connections, forward reasoning tries to prune the search space that has yet to be explored. The mechanism of forward reasoning consists of two techniques, forward checking and forward implication. Forward checking utilizes contradictory subformulae to find a clash earlier and therefore saves unnecessary inference steps. Forward implication uses the implication relation to remove redundant non-deterministic formulae. Forward checking has been implemented in MLTP and empirically compared against backward checking.

1.3 Generic theorem proving

Generic theorem provers are highly customizable. They can help the users to create their own prover for a given logic. Lotrec and TWB have a high-level language for users to specify the syntax of the logic, the deduction calculus and the strategies.

However, some highly operational techniques involve a lot of low-level operations and therefore cannot be specified using the high-level language mentioned above. These techniques include various advanced backtracking techniques and some heuristics like MOMS. These techniques have great impact on the performance of a theorem prover. Without them, a prover is not likely to be able to solve a real life problem within reasonable time.

MLTP intends to offer a solution so that users can also specify highly operational techniques like dynamic backtracking. This solution consists of a generic data structure, a low-level macro language, an event-handler framework and a high-level specification language.

The generic data structure developed in MLTP uses a highly connected relation-based structure to store data and its relations. In this way various forms of data operations can be processed in a uniform way.

The low-level macro language has a syntax similar to C++ but is much easier to program. Supported by the generic data structure, it can use a small set of standard instructions to perform complicated operations.

The event-handler framework divides the system into loosely connected modules. Whenever an event is triggered, its corresponding handler is evoked. Each module can be defined using the low-level macro language, whereas the pairing of events and handlers is controlled by the high-level specification language.

The high-level specification language can be used to specify the syntax and grammar of the logics, inference rules and strategies. Most of these are achieved by manipulating the events and their handlers in the event-handler framework.

1.4 Thesis overview

Chapter 2 introduces modal logic and tableau calculus. Chapter 3 describes several optimization techniques. The techniques described include simplification, unit propagation, lexical normalization, complement splitting, backjumping,

caching and heuristics. Chapter 4 proposes a dynamic backtracking algorithm for modal logics. Dynamic backtracking improves backjumping by trying to avoid undoing the inference steps that have nothing to do with the clash during backtracking. The algorithm's correctness has been proved. Chapter 5 introduces two forward reasoning mechanism, namely forward checking and forward implication. Forward checking utilizes contradictory information to find a clash earlier and therefore save unnecessary inference steps. Forward implication uses the implication relation to remove non-deterministic formulae. Forward checking has been implemented in MLTP. Chapter 6 describes the infrastructure of MLTP. It features a generic event-handler framework, a powerful macro language and a proof generation mechanism. Chapter 7 presents benchmarking results of tests on two classes of problems. The effectiveness of simplification is evaluated. Different backtracking techniques including chronological backtracking, backjumping and dynamic backtracking are compared against each other. The results using forward checking are compared with results using backward checking. The overall performance of MLTP is compared against MSPASS and RACER. In this chapter, we also describe the a generic data structure used in MLTP and investigate memory cache efficiency.

Chapter 2

Modal logic theorem proving

Boasting an extremely long history (from Aristotle), modal logic is well known as a logic of possibility and necessity [HC96]. Its consideration about the truth and falsity of a statement is not only restricted on the issue itself. Modal logic is put into some context and considers how things would be if the context is different. The context can be represented by a relational frame constructed using worlds and links. Based on this frame, the validity and satisfiability of a formula are defined. According to the different sets of valid formulae, modal logics are categorized into various systems. Each system features some relational property.

The rest of this chapter is organised as follows: Section 2.1 briefly introduces modal logic. Section 2.2 and Section 2.3 define the syntax and semantics of modal logic. Section 2.4 introduces tableau calculus for propositional logic. Section 2.5 describes labelled tableau algorithm for modal logic. Section 2.6 presents existing tableau systems for modal logic.

2.1 Introduction

Nowadays, applications of computer systems often face the need of formalising human language into logics [MB04]. It is desirable that the logic we use is capable of handling some important notions of natural language. It is very common in natural language that the conclusion of a statement depends on its context. Given a statement like "It will rain tomorrow", we do not know if it is true until tomorrow. Given another example, "Manchester United is the best football team in the world", then someone, especially a fan of United will believe that it is true, while some others may object. These examples illustrate that, a statement

being true or false depends on the context where it is happening. These contexts require different modes of truth.

Traditional propositional or predicate logics are incapable of dealing with different modes of truth. First order logic is sophisticated enough to handle modes of truth, but it is not decidable [GG02]. It means that no first order logic theorem prover can guarantee that it will solve every problem within limited time. In contrast, modal logics are designed to reason about modes of truth and are used in many applications as a formal means to model and reason about notions of knowledge, belief and time. The modal logics of knowledge are called *epistemic logics* [MvdH95], logics of belief are called *doxastic logics* [Mey03], and those of time are called *temporal logics* [RU71]. In addition, many modal logics, including those considered in this chapter, are decidable and have the finite model property [BDRV01].

2.2 Syntax

The language of normal modal logics is that of propositional logic with two extra unary modal operators, *box* ($\Box$) and *diamond* ($\Diamond$).

Formally, let $\mathcal{P} = \{p, q, r, \ldots\}$ be a countable infinite set of propositional variables. Modal formulae of the basic modal logic are defined inductively by:

1. $\top$ (true) and $\bot$ (false) are modal formulae.
2. Every propositional variable in $\mathcal{P}$ is a modal formula.
3. If ϕ and φ are modal formulae, then so are
   - $\neg\phi$ (not ϕ)
   - $(\phi \wedge \varphi)$ (ϕ and φ)
   - $(\phi \vee \varphi)$ (ϕ or φ)
   - $(\phi \rightarrow \varphi)$ (ϕ implies φ)
   - $(\phi \leftrightarrow \varphi)$ (ϕ is equivalent to φ)
   - $\Box\phi$ (box ϕ)
   - $\Diamond\phi$ (diamond ϕ)

The connectives are listed in descending order of priorities: $\neg$, $\Box$, $\Diamond$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

The unary connective $\Box$ is a universal quantifier and $\Diamond$ is a existential quantifier of modal logic. They are like the universal and existential quantifiers in first-order logic.

2.3 Semantics

The semantics of modal logic is given by the well known *possible worlds semantics*, or *Kripke semantics*, in which Kripke frame is the basic notion [Kri63]. A *Kripke frame* $\mathcal{F}$ is specified by an tuple $(\mathcal{W}, \mathcal{R})$ including a set of worlds $\mathcal{W}$ and a relation $\mathcal{R}$ ($\mathcal{R} \subseteq \mathcal{W} \times \mathcal{W}$) which is called the accessibility relation.

Kripke frames can be depicted by directed graphs. For example, Figure 2.1 depicts the Kripke frame $\mathcal{F} = (\{1, 2, 3\}, \{(1, 2), (2, 1), (1, 3), (2, 3)\})$.

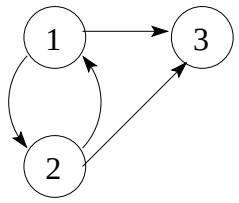


Figure 2.1: Graph of the Kripke frame $\mathcal{F} = (\{1, 2, 3\}, (1, 2), (2, 1), (1, 3), (2, 3))$

A *Kripke model* $\mathcal{M}$ over $\mathcal{P}$ is a tuple ($\mathcal{F}$,$\mathcal{L}$), in which $\mathcal{F}$ is a Kripke frame and $\mathcal{L}$ is a labelling function, $\mathcal{P} \rightarrow 2^{\mathcal{W}}$. The labelling function $\mathcal{L}$ gives the assignment of the truth values of propositional variables on the worlds. If $p \in \mathcal{P}$, $w \in \mathcal{L}(p)$ denotes that p is true in world w. A Kripke model can be depicted by labelling the worlds in a Kripke frame with the corresponding true propositional variables.

Rigorously, given a frame $(\mathcal{W}, \mathcal{R})$, a Kripke model should be written $\mathcal{M} = ((\mathcal{W}, \mathcal{R}), \mathcal{L})$. But for convenience, we usually write it as $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{L})$.

Given a Kripke model $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{L})$, The notion "formula F is true in the

world w" is denoted by $\mathcal{M}, w \vDash F$ and defined as follows:

$$\begin{aligned}
\mathcal{M}, w &\vDash p \text{ iff } w \in \mathcal{L}(p) \\
\mathcal{M}, w &\vDash \top \\
\mathcal{M}, w &\nvDash \bot \\
\mathcal{M}, w &\vDash \neg\phi \text{ iff } \mathcal{M}, w \nvDash \phi \\
\mathcal{M}, w &\vDash (\phi \wedge \varphi) \text{ iff } (\mathcal{M}, w \vDash \phi \text{ and } \mathcal{M}, w \vDash \varphi) \\
\mathcal{M}, w &\vDash (\phi \vee \varphi) \text{ iff } (\mathcal{M}, w \vDash \phi \text{ or } \mathcal{M}, w \vDash \varphi) \\
\mathcal{M}, w &\vDash (\phi \rightarrow \varphi) \text{ iff } (\text{if } \mathcal{M}, w \vDash \phi \text{ then } \mathcal{M}, w \vDash \varphi) \\
\mathcal{M}, w &\vDash (\phi \leftrightarrow \varphi) \text{ iff } (\mathcal{M}, w \vDash \phi \text{ iff } \mathcal{M}, w \vDash \varphi) \\
\mathcal{M}, w &\vDash \Box\phi \text{ iff for every } v \in \mathcal{W}, \text{ if } (w, v) \in \mathcal{R} \text{ then } \mathcal{M}, v \vDash \phi \\
\mathcal{M}, w &\vDash \Diamond\phi \text{ iff for some } v \in \mathcal{W}, (w, v) \in \mathcal{R} \text{ and } \mathcal{M}, v \vDash \phi
\end{aligned}$$

The definitions for $\top$ and $\bot$ express that irrespective of the world considered, $\top$ is always true while $\bot$ is always false. The semantics of the connectives of $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ is the same as in propositional logic. However, there is something new for the modal operators. For $\Box\phi$ to be true in the world w, it is required that ϕ be true in all worlds reachable from w (in graphical notation, v is reachable from w iff there is a arrow from w to v). And for $\Diamond\phi$ to be true at w, it is required that there exist at least one world reachable from w in which ϕ is true.

A formula ϕ is *satisfiable* in a model $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{L})$ if and only if there is some world $w \in \mathcal{W}$ in $\mathcal{M}$ at which ϕ is true, denoted by $\mathcal{M}, w \vDash \phi$. A formula ϕ is *unsatisfiable* in a model $\mathcal{M}$ if it is not satisfiable in $\mathcal{M}$. A formula ϕ is satisfiable in a frame $\mathcal{F}$ iff there exists a model $\mathcal{M}$ based on $\mathcal{F}$ such that ϕ is satisfiable in $\mathcal{M}$. Finally, a formula ϕ is satisfiable in a class $\mathcal{C}$ of frames iff there exists a frame $\mathcal{F}$ in $\mathcal{C}$ such that ϕ is satisfiable in $\mathcal{F}$.

Validity can be defined using the concept of satisfiability. A modal formula ϕ is valid if and only if $\neg\phi$ is not satisfiable.

There are different kinds of validity in modal logic.

- A formula F is valid in a model $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{L})$ iff
  F is true in every world $w \in \mathcal{W}$ in $\mathcal{M}$

- A formula F is valid in a frame $\mathcal{F} = (\mathcal{W}, \mathcal{R})$ iff
  F is valid in every model based on $\mathcal{F}$.

- A formula F is valid in a class $\mathcal{C}$ of frames iff
  F is valid in every frame $\mathcal{F}$ in $\mathcal{C}$.

All the propositional tautologies and their substitution instances are valid in the class of all Kripke frames. For example, a propositional tautology $p \vee \neg p$ is valid. And so is its substitution instance $(\Box q \rightarrow q) \vee \neg(\Box q \rightarrow q)$ by replacing p with $\Box q \rightarrow q$.

There are also some other formulae, which are not propositional validities that are valid in modal logic.

1. $\Box(\phi \rightarrow \varphi) \rightarrow (\Box\phi \rightarrow \Box\varphi)$
2. $\Diamond\phi \leftrightarrow \neg\Box\neg\phi$
3. $\Diamond(\phi \vee \varphi) \leftrightarrow (\Diamond\phi \vee \Diamond\varphi)$
4. $\Box(\phi \wedge \varphi) \leftrightarrow (\Box\phi \wedge \Box\varphi)$

Formula 1 is valid in all of the Kripke frames. Formula 2 gives an important correspondence between the two modal operators. Formula 3 shows that $\Diamond$ distributes over $\vee$. Formula 4 shows that $\Box$ distributes over $\wedge$.

This basic modal logic is also called *modal logic K*. There are other modal logic systems which are extensions of modal logic K [BDRV01]. These modal logic systems feature different sets of formulae which are valid in their systems. These formulae are also called axioms. Most of the well-known axioms have corresponding accessibility properties. Which means that if is an axiom is valid in a class of Kripke frames, this class of Kripke frames satisfies the axiom's corresponding accessibility property.

The well known axioms and their corresponding accessibility properties are listed as follows:

Name	Property	Axiom	FOL definition
T	reflexivity	$\Box p \rightarrow p$	$\forall x.r(x,x)$
D	seriality	$\Box p \rightarrow \Diamond p$	$\forall x \exists y.r(x,y)$
B	symmetry	$p \rightarrow \Box\Diamond p$	$\forall x,y.r(x,y) \rightarrow r(y,x)$
4	transitivity	$\Box p \rightarrow \Box\Box p$	$\forall x,y,z.(r(x,y) \wedge r(y,z)) \rightarrow r(x,z)$
5	euclideanness	$\Diamond p \rightarrow \Box\Diamond p$	$\forall x,y,z.(r(x,y) \wedge r(x,z)) \rightarrow r(y,z)$

Table 2.1: Axioms and their correspondence properties

The meaning of Table 2.1 is that: given a frame $\mathcal{F} = (\mathcal{W}, \mathcal{R})$, an axiom is valid in $\mathcal{F}$ *if and only if* $\mathcal{F}$ has the axiom's corresponding property. For example, $\Box\phi \rightarrow \phi$ is valid in $\mathcal{F}$ if and only if $\mathcal{F}$ is reflexive.

2.4 Tableau-based methods

Tableau-based methods are widely used deductive systems to verify satisfiability in modal logics [Fit83, Gor99]. Proofs of tableau-based methods are usually presented in the form of trees. The derived formulas are the nodes of the tree. Nodes of different level are connected by edges.

Tableau procedures are refutation procedures. To prove a formula ϕ is valid we begin with some syntactical expression intended to assert that its negation $\neg\phi$ is satisfiable. We attempt to show that the satisfaction of $\neg\phi$ leads to a contradiction. According to the correspondence of satisfiability and validity, if $\neg\phi$ is not satisfiable then ϕ is valid. During the derivation of the tableau method, the expression to assert the satisfiability of $\neg\phi$ is broken into subformulae by applying expansion rules. Basically there are two types of expansions. They are *deterministic expansion* and *non-deterministic expansion*. In a deterministic expansion, there is exactly one possible result. For example, given $\phi \wedge \varphi$ is true, we know that both ϕ and φ must be true. Whereas in a non-deterministic expansion there is more than one possible result. For example, given $\phi \vee \varphi$ is true, we have two possible cases that either ϕ or φ is true. After a non-deterministic expansion, the tableau makes two branches each of which represent a possible assignment. Satisfiability search will be performed along every branch. Finally there are rules for closing branches to identify impossible conditions. If each branch closes, the tableau is said to be closed.

From a semantic point of view, a tableau can be considered as a search procedure [MDAP99]. A *path* of a tableau contains all of the formulae collected along the edges from one point to another point of the tableau. Each full path of the tableau, starting from the initial expression at the root going down along the expansions and picking one of the branches till the ends of the tableau, is a model subjected to the satisfiability test. Each branch of a tableau can be looked as a possible description of that model, provided it contains no clash. If such a satisfiable model is found, this model is also a proof that the initial expression is satisfiable. Otherwise if after exhausting all of the possible models, no model is

found, we can say that the initial expression is unsatisfiable.

Here we first give the tableau procedure for propositional logic in a more formal way. Given a formula ϕ, to test its validity, let $\varphi = \neg\phi$ and transfer φ to a negation normal form which only includes connectives of $\neg$, $\wedge$ and $\vee$.

Sequences of tableaux derivation are laid out in the form of a tree, and every node of the tree contains a tuple

$$\langle \theta, s \rangle,$$

in which θ is the formula derived from the previous nodes on the same branch by applying expansion rules, and s denotes formula θ's status. The use of the formula's status s is to prevent a formula being reused and to guarantee the termination of the tableau procedure.

At the start of the derivation, the tuple $< \theta = \varphi, 0 >$ is put in the root of the tableau. Then we apply expansion rules as follows:

Rule ($\wedge$)	Rule ($\vee$)
$\dfrac{A \wedge B, 0}{\begin{array}{c} A \wedge B, 1 \\ A, 0 \\ B, 0 \end{array}}$	$\dfrac{A \vee B, 0}{\begin{array}{c} A \vee B, 1 \\ A, 0 \mid B, 0 \end{array}}$

Table 2.2: Tableau rules for propositional logic

Rule ($\wedge$) is the expansion rule for conjunctions. If a branch contains a node with a tuple $\langle A \wedge B, 0 \rangle$, then extend the branch by adding the tuples $\langle A, 0 \rangle$ and $\langle B, 0 \rangle$ to the end of the branch. Change the status of the tuple $\langle A \wedge B, 0 \rangle$ to 1. Status 0 means a formula is unexpanded and status 1 means that the formula has been expanded

Rule ($\vee$) is the expansion rule for disjunctions. If a tableau contains a node with a tuple $\langle A \vee B, 0 \rangle$, then extend the branch by adding two new branches, and add $\langle A, 0 \rangle$ to one branch, add $\langle B, 0 \rangle$ to the other. Change the status of the tuple to $\langle A \wedge B, 1 \rangle$.

If a branch of the tableau contains both $\langle A, 0 \rangle$ and $\langle \neg A, 0 \rangle$ then a *clash* is found, the branch is said to be *closed*. The branch is not to be further extended.

If none of the rules ($\wedge$, $\vee$) can be applied on a branch and no logic forms A and $\neg A$ are both found in this branch then the branch is said to be *open*.

If all the branches of the tableau are closed, then the tableau is said to be *closed*. If the tableau is closed, then φ is unsatisfiable, thus ϕ is valid. Otherwise

when an open branch is found, the tableau derivation can stop immediately, since this already establishes that φ is satisfiable.

In the tableau algorithm just presented, translation to normal form is performed before derivation. But that is not necessary and new expansion rules can be added to deal with other connectives such as $\rightarrow$ and $\leftrightarrow$.

The soundness and completeness can be demonstrated by proving that ϕ is satisfiable if and only if the tableau construction for ϕ returns satisfiable. The proofs of soundness and completeness can be found in Fitting's book [Fit90].

Theorem 2.4.1. *The tableau procedure is sound and complete.*

Theorem 2.4.2. *The tableau procedure always terminates and the derivation tree is finite.*

Proof. Let n be the number of the $\wedge$ and $\vee$ connectives in ϕ. If ϕ only consists of propositional variables and its negations then no expansion rules are to be used. The depth of the tree l is 1 (the level of the root is 1) and our theorem 2.4.2 is obviously true. Otherwise ϕ can be subjected to expansion rules. After applying the rule $(\wedge)$, the number of connectives n is decreased by 1 for the branch involved. After the rule $(\vee)$, n is also decreased by 1 for the branch involved and a new branch is generated. By changing a formula's status after it is expanded, we can guarantee that in each branch a formula can be expanded only once. Therefore in a given branch the number of expansions can be performed is at most n. Since an expansion increases the level of the tree by 1, the depth of the tree l is at most n. Considering that the tree is binary, the most nodes in the tree of level n is $1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$. The number of connectives n is a formula is finite, therefore the derivation tree is finite and the algorithm terminates in all cases. □

Derivations constructed by the tableau procedure above are always finite. This means:

Theorem 2.4.3. *The tableau procedure is a decision procedure for propositional logic.*

2.5 Labelled tableau system

The tableau system we introduced in the previous section cannot handle the modalities in modal logic. Therefore a more powerful system called *labelled*

tableau system is developed to test satisfiability in modal logics [Fit83].

Based on the Kripke semantics, the labelled tableau system consists of labelled formulae and tableau rules that operate on labelled formulae. A label denotes a world w of the Kripke model $\mathcal{M}$, to which $w \in \mathcal{W}$. A labelled formula is a structure of the form $w : \phi$ where w is a label and ϕ is a formula. It has the semantic meaning that the formula ϕ is true in the world w.

A branch of a labelled tableau is *closed* if it contains some labelled formula $w : \phi$ and also contains $w : \neg\phi$. If such pair is not found, the branch is *open*. A label w is *used* on a branch if there is some labelled formula $w : \phi$ on this branch. A label w is *new* to a branch if there is no labelled formula $w : \phi$ on this branch.

Table 2.3 classifies formulae into four groups based on their logical connectives. Their subformulae after normalization are also presented. The labelled tableau rules listed in Table 2.4 are categorised into three types: the α and β rules are the same as the ones for propositional logic and can be derived from the rules given in Section 2.4 by the addition of the new connectives. A necessary formula ν is true in world w if and only if ν is true in all the worlds w' reachable from w. So when $w : \nu$ appears on an open branch and there exist some world w' that is reachable from w, we can always add $w' : \nu_0$ at the end of that branch. A possible formula π is true in world w if and only if there exist a world w' reachable from w in which π is true. Therefore, when $w : \pi$ appears on an open branch, we can always add $w' : \pi_0$ to the end of that branch, where w' and its link $(w, w') \in \mathcal{R}$ are new. It is also known as a successor creator since it is the only rule to create new worlds.

Conjunction formulae

α	α_1	α_2
$\varphi \wedge \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$
$\neg(\varphi \rightarrow \psi)$	φ	$\neg\psi$
$\neg(\neg\varphi)$	φ	φ

Disjunctive formulae

β	β_1	β_2
$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$
$\varphi \vee \psi$	φ	ψ
$\varphi \rightarrow \psi$	$\neg\varphi$	ψ

Necessary formulae

ν	ν_0
$\Box\varphi$	φ
$\neg\Diamond\varphi$	$\neg\varphi$

Possible formulae

π	π_0
$\neg\Box\varphi$	$\neg\varphi$
$\Diamond\varphi$	φ

Table 2.3: Classification

Variants of modal logics can be obtained from modal logic K by addition of

α-rule	β-rule
$\dfrac{w : \alpha}{\begin{array}{l} w : \alpha_1 \\ w : \alpha_2 \end{array}}$	$\dfrac{w : \beta}{w : \beta_1 \mid w : \beta_2}$
ν-rule	**π-rule**
$\dfrac{w : \nu \ (w, w') \in \mathcal{R}}{w' : \nu_0}$	$\dfrac{w : \pi}{\begin{array}{c} w' : \pi_0 \\ (w, w') \in \mathcal{R} \\ \text{where } w' \text{ is new on the branch} \end{array}}$

Table 2.4: Tableau rules for modal logic

any combination of the axioms T, D, 4, 5, B, etc. As listed in Table 2.1, there is a accessibility property corresponding to each axiom. Any frame in which the axiom is valid has the corresponding property. To deal with modal logics with axioms, structural rules are applied on the frame created by the expansion rules ν and π in the tableau construction. The structural rules are given in Table 2.5 for a Kripke frame $\mathcal{F} = (\mathcal{W}, \mathcal{R})$.

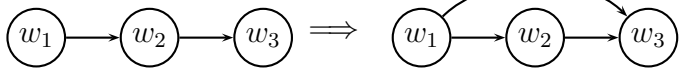
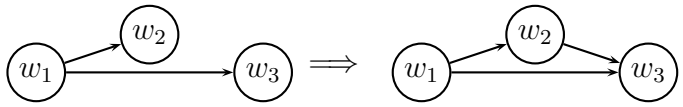
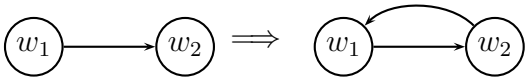
T (reflexity) $\Box p \rightarrow p$	
4 (transitivity) $\Box p \rightarrow \Box\Box p$	
5 (euclideanness) $\Diamond p \rightarrow \Box\Diamond p$	
B (symmetry) $p \rightarrow \Box\Diamond p$	
D (seriality) $\Box p \rightarrow \Diamond p$	

Table 2.5: Structural rules of T, 4, 5, B

The semantic explanation of the structural rules listed in Table 2.5 is:

1. For axiom T, for any world w on the branch add a link $(w, w) \in \mathcal{R}$ and $\mathcal{R} := \mathcal{R} \cup \{(w, w)\}$.

2. For axiom 4, for any worlds w_1, w_2, w_3 if $(w_1, w_2) \in \mathcal{R}$, $(w_2, w_3) \in \mathcal{R}$ then

add a link $(w_1, w_3) \in \mathcal{R}$ and $\mathcal{R} := \mathcal{R} \cup \{(w_1, w_3)\}$.

3. For axiom 5, for any worlds w_1, w_2, w_3 if $(w_1, w_2) \in \mathcal{R}$, $(w_1, w_3) \in \mathcal{R}$ then add a link $(w_1, w_3) \in \mathcal{R}$ and $\mathcal{R} := \mathcal{R} \cup \{(w_1, w_3)\}$.

4. For axiom B, for any worlds w_1, w_2 if $(w_1, w_2) \in \mathcal{R}$, then add a link $(w_2, w_1) \in \mathcal{R}$ and $\mathcal{R} := \mathcal{R} \cup \{(w_2, w_1)\}$.

5. For axiom D, for any world w_1 create a world w_2 and add a link $(w_1, w_2) \in \mathcal{R}$ and $\mathcal{W} := \mathcal{W} \cup \{w_2\}$, $\mathcal{R} := \mathcal{R} \cup \{(w_1, w_2)\}$.

Theorem 2.5.1. *The tableau procedure for the modal logic K and its extensions by any combination of the axioms T, D, 4, 5, B is sound and complete.*

The proof for Theorem 2.5.1 can be found in [Fit83]. Here is an example of a tableau construction in the modal logic KT.

Example 2.5.1. *The formula $\phi = \Box p \rightarrow (\neg\Diamond\neg p \wedge p)$ is valid in modal logic KT. Because the tableau method is a refutation procedure, we attempt to prove $\neg\phi$ is not satisfiable.*

1.	$w_0 : \neg(\Box p \rightarrow (\neg\Diamond\neg p \wedge p))$	
2.	$w_0 : \Box p$	$[\alpha : 1]$
3.	$w_0 : \neg(\neg\Diamond\neg p \wedge p)$	$[\alpha : 1]$

4.	$w_0 : \neg\neg\Diamond\neg p$	$[\beta : 3]$	10.	$w_0 : \neg p$	$[\beta : 3]$
5.	$w_0 : \Diamond\neg p$	$[\alpha : 4]$	11.	$(w_0, w_0) \in \mathcal{R}$	[axiom T]
6.	$w_1 : \neg p$	$[\pi : 5]$	12.	$w_0 : p$	$[\nu : 2, 11]$
7.	$(w_0, w_1) \in \mathcal{R}$	$[\pi : 5]$	13.	Closed	$[10, 12]$
8.	$w_1 : p$	$[\nu : 2, 7]$			
9.	Closed	$[6, 8]$			

In the tableau shown in the above example: 1. is the problem to test satisfiability; 2. is from 1 by applying an α rule; 3. is from 1 by an α rule; 4. is from 3 by a β rule. The derivation branches here, one branch goes to formula 4 the other to 10; 5. is from 4 by an α rule; 6. and 7. are from 5 by a π rule producing a new world and a link; 8. is from 2 and 7. Since w_1 is a successor of w_0 and $\Box p$ propagates p to all the successors of w_0 by a ν rule, 9. finds a clash between

6 and 8 since p and $\neg p$ can not both be true in the world w_1. This branch is closed. 10. is a branch together with 4 derived from 3; 11. generates a link from w_1 to itself by the property of axiom T making the frame reflexive; 12. is from 2 and 11 by a ν rule; 13. finds a clash between 10 and 12, this branch is closed. Since all the branches are closed, this tableau is closed and the KT-validity of $\Box p \rightarrow (\neg\Diamond\neg p \wedge p)$ is proved.

2.6 Existing modal tableau theorem provers

In recent years, we have seen many successful implementations of tableau-based algorithms for modal logics or description logics. Some of them were developed as test beds to study the inference procedures of the logics that they were designed for. Some of them were designed as reasoning engines used by applications of logics.

TABLEAUX [Cat91] was developed at IBM in Paris. It was implemented in Prolog. As an early attempt to build a generic theorem prover TABLEAUX can handle variants of modal logics including temporal, epistemic and dynamic logics. The semantic tableau method is used in TABLEAUX as a decision procedure.

The Logics Workbench (LWB) [HJSS96] was developed at the University of Bern in Switzerland. Written in C++, LWB provides separate libraries for well-known propositional decision procedures. LWB can handle a broad range of propositional logics including intuitionistic logic, modal logic and nonmonotonic logic. The LWB also features a user-friendly interface.

Lotrec [CFG+01] was developed in Toulouse, France. It was written in JAVA. Being able to handle modal and description logics, Lotrec features a high level of flexibility. It not only has a user-friendly interface but also uses a high-level language to define inference rules and strategies. Users can use the language offered to define and customize their own decision procedures.

The Tableaux Work Bench [AG03] was developed at the Autralian National University. It is written in Ocaml. The Tableaux Work Bench offers a high-level language for users to experiment with tableau calculi and even optimisation techniques.

FaCT [Hor97] was developed at the University of Manchester. It was first written in LISP and then reimplemented in C++ and became FaCT++ [TH06].

FaCT was designed to be an efficient theorem prover for TBox reasoning in description logics. It successfully used several optimisation techniques from classical logics to improve the performance of description logic reasoning.

DLP [PS98] was developed at AT&T and was written in ML. Like FaCT, DLP also implemented many optimisation techniques to offer a better performance. The logics that DLP can handle include description logics, modal logics and test-free propositional dynamic logic, the later only if certain optimisations are disabled.

Racer [HM01] was developed at the University of Hamburg, Germany. The latest version is called RacerPro. It is implemented in LISP. Inspired by FaCT and DLP, Racer incorporates many optimisation techniques for description logics. It can offer efficient reasoning services for ABoxes and TBoxes.

The theorem provers mentioned above fall into two categories. One category includes TABLEAUX, Lotrec and the Tableaux Work Bench. They are generic theorem provers which feature great specifiability. These provers incorporate flexible frameworks and generic data structures to be able to cope with various situations. These kinds of provers can handle new logics specified by the users. Their deduction procedure and strategies are also customisable. However some useful optimization techniques are too complicated to be defined by high-level specification languages. The lack of these optimization techniques makes these generic provers unsuited to solve real life problems.

The other category includes FaCT, DLP, RACER and LWB. They are highly optimized theorem provers. Complicated procedures, sophisticated data structures and optimization techniques are used in these provers to get better performance. They are very efficient and can handle real life problems. Their logic, deduction procedure and strategies are hard-coded in their system. Therefore, these provers can only work on the logics they are designed for.

Chapter 3

Optimisation techniques

In the previous chapter we briefly introduced modal logic and the tableau procedure. With the basic tableau procedure described in the previous chapter, it is in theory possible to solve all of the problems in the modal logic K. However, in reality, the resources we have are limited. It is desirable that the reasoning can be completed in reasonable time. A real world problem usually has hundreds or even thousands of formulae. It would take the basic tableau procedure several hours or even days to solve a problem on that scale. Therefore to fully take advantage of modal logic theorem provers, we have to find a way to refine the basic tableau algorithm.

Many optimization techniques have been developed to improve the performance of theorem provers[BHN$^+$92, Fre95, Hor97]. Many of them can be incorporated in a tableau-based modal logic system. In this chapter we will discuss some of these techniques including simplification, unit-propagation, lexical normalization and dependency-directed backtracking.

The rest of this chapter is organized as follows: Section 3.1 presents an optimization technique called unit propagation. Section 3.2 introduces simplification. Section 3.3 describes lexical normalization. Section 3.4 explains complement splitting. Section 3.5 presents a backtracking technique named backjumping. Section 3.6 introduces caching. Section 3.7 describes various selection heuristics.

3.1 Unit propagation

Unit propagation is also called *Boolean constraint propagation*. It was first introduced by Davis and Putnam [DP60] and then fully discussed by Freeman

in [Fre95]. Unit propagation is a technique used to minimize non-deterministic expansion (branching).

The idea of unit propagation is to perform clash detection before branching and prune a part of the search tree that is estimated to cause collision. Branching leads to backtracking and backtracking is an expensive operation that usually involves a lot of memory releasing. The number of backtracking steps can have a significant impact on the performance of a theorem prover. Therefore, it has become a common principle while choosing strategies, to reduce the number of backtracking steps as much as possible.

Now we are in the position to give a formal description of the procedure of unit propagation. Given a set B including all of the formulae on the current branch and a disjunction $(w : \phi_1 \vee \phi_2 \ldots \vee \phi_n) \in B$. Only the $w : \phi_i$ such that $w : \neg\phi_i \notin B$ $(i = 1, 2, \ldots, n)$ will be propagated. We can summarize the effect of unit propagation into a tableaux expansion rule:

$$\frac{x : \neg\phi,\ x : \phi \vee \psi}{x : \psi}$$

Example 3.1.1. *Let B be the set of formulae on the current branch and suppose*

$$\{w_1 : \phi \vee (\varphi \wedge \psi), w_1 : \neg\varphi \vee \neg\psi, w_1 : \neg\phi\} \subseteq B.$$

Unit propagation is applicable to $w_1 : \phi \vee (\varphi \wedge \psi)$ and $w_1 : \neg\phi$. The branch is extended with $w_1 : \varphi \wedge \psi$. Expanding $w_1 : \varphi \wedge \psi$, we get $B' = B \cup \{w_1 : \varphi \wedge \psi, w_1 : \varphi, w_1 : \psi\}$. Expanding $w_1 : \neg\varphi \vee \neg\psi)$ creates two branches containing $w_1 : \neg\varphi$ and $w_1 : \neg\psi$ respectively. Both can be closed because of the presence of $w_1 : \varphi$ and $w_1 : \psi$ in B'.

Example 3.1.1 illustrates how unit propagation works. From the example we can see that the application of unit propagation reduced the number of branches. For a real world problem that contains a large number of formulae, considerable branching can be saved because of unit propagation, which can improve the performance of a theorem prover significantly.

3.2 Simplification

Simplification is a technique widely used in theorem provers. It aims to reduce the size of the input problem formulae without changing their logical properties (satisfiability or validity). When the input formulae become smaller, better performance of the theorem prover can be expected.

Simplification is usually used as a pre-processing technique performed before proof search. It utilizes a set of rules or equations to transform a formula into a smaller new formula with the new one logically equivalent to the old one. During this transformation formulae which are obviously satisfiable or unsatisfiable are replaced. The complexity of this transformation should not be more than that of solving the problem itself. A variety of simplification techniques that can be used on modal formulae were investigated in [HS98].

$$\phi \wedge \varphi \Rightarrow \bot, \quad \text{if } \varphi = \bot \text{ or } \varphi = \neg\phi \tag{3.2.1}$$
$$\phi \wedge \varphi \Rightarrow \phi, \quad \text{if } \varphi = \phi \text{ or } \varphi = \top \tag{3.2.2}$$
$$\phi \vee \varphi \Rightarrow \top, \quad \text{if } \varphi = \top \text{ or } \varphi = \neg\varphi \tag{3.2.3}$$
$$\phi \vee \varphi \Rightarrow \phi, \quad \text{if } \varphi = \bot \text{ or } \varphi = \phi \tag{3.2.4}$$
$$\Box\phi \Rightarrow \top, \quad \text{if } \phi = \top \tag{3.2.5}$$
$$\Diamond\phi \Rightarrow \bot, \quad \text{if } \phi = \bot \tag{3.2.6}$$
$$\neg\phi \Rightarrow \bot, \quad \text{if } \phi = \top \tag{3.2.7}$$
$$\neg\phi \Rightarrow \top, \quad \text{if } \phi = \bot \tag{3.2.8}$$

Table 3.1: Rules of simplification

A set of rules is employed to perform simplification. The rules are shown in Table 3.1. The simplification rules are not difficult to understand. Most of them are tautologies of propositional logic. For example the rule (3.2.1) in Table 3.1 means that the formulae $\phi \wedge \bot$ and $\phi \wedge \neg\phi$ can be replaced by $\bot$. Rules (3.2.5) and (3.2.6) represent the equivalence $\Box\top \leftrightarrow \top$ and $\Diamond\bot \leftrightarrow \bot$.

Simplification can be implemented recursively. The input formula is checked for any applicable simplification rule. Every application of rule leads to a substitution which reduces the complexity of the input formula. After a substitution, the resulting formula is subjected to simplification again. This process stops when it is found that no more rules can be applied.

Example 3.2.1. *Given a formula* $\phi = \Box((p \rightarrow q) \vee \neg(p \rightarrow q))$*, perform simplification.*
Let $\theta = (\phi \rightarrow \varphi)$*, then* $\phi = \Box(\theta \vee \neg\theta)$*.*
We can apply the simplification rule 3.2.3 to ϕ *to obtain* $\Box(\top)$*.*
We can then apply the simplification rule 3.2.5, to $\Box(\top)$ *and obtain* $\top$*.*

Example 3.2.1 shows how simplification works. We can see from this example that a modal formula is directly reduced to a logic constant.

3.3 Lexical normalization

Lexical normalization is also a technique performed in pre-processing. It chooses a subset of the logical operators as the seminal operators and replaces the other operators with the seminal operators. After unifying the presentation of the formulae, it is possible to detect complementary complex formulae earlier. For example, the seminal operators could be $\neg$, $\vee$ and $\Diamond$, and the process of lexical normalization could be illustrated in following example.

$$\phi \wedge \psi \Rightarrow \neg(\neg\phi \vee \neg\psi) \tag{3.3.9}$$
$$\Box\phi \Rightarrow \neg(\Diamond\neg\phi) \tag{3.3.10}$$

Table 3.2: Rules of lexical normalization

Table 3.2 shows a set of transformation rules that can be used in lexical normalization. The operators of $\neg$, $\wedge$ and $\vee$ have been selected as the seminal operators. The rules are used to change a formula that has non-seminal operators like $\wedge$ and $\Box$ into a formula only has seminal operators.

Example 3.3.1. *Lexical normalization*

$$(p \vee \Diamond q) \wedge \phi \wedge \neg p \wedge \Box\neg q$$
$$\Rightarrow \neg(\neg\underline{(p \vee \Diamond q)} \vee \neg\phi \vee \underline{(p \vee \Diamond q)})$$

After applying the transformation rules, it can easily be seen that the two underlined subformulae are complementary and therefore the formula is trivially unsatisfiable.

3.4 Complement splitting

Complement splitting, also called *semantic branching*, is a technique to reduce the redundancy caused by branching. In [GS96], Giunchiglia and Sebastiani spotted a problem of disjunction expansion that has affected the efficiency of tableau-based theorems. They proposed complement splitting as a solution to alleviate the problem.

In tableau-based theorem proving, β-rules are used to expand disjunctions. By applying a β-rule, two branches are created. These two branches are not guaranteed to be disjoint with each other. To elaborate this problem, we use the formulae shown in Table 3.3 as an example. In the example formula set, ϕ can be an arbitrary formula.

$$p \vee q \tag{3.4.11}$$
$$\neg p \vee q \tag{3.4.12}$$
$$p \vee \neg q \tag{3.4.13}$$
$$\phi \tag{3.4.14}$$

Table 3.3: Example of complement splitting

Figure 3.1 is the tableau proof for the example problem. From the proof we can see that the two branches that consist of emphasized edges contain the same set of formulae. These two branches have been constructed twice. Considering ϕ can be a formula of any size, the cost of this redundancy may be significant.

To eliminate this kind of redundancy, complement splitting uses a new rule for disjunction expansion. The rule is defined in Table 3.4. We can see that when the new rule expands a disjunction, it creates two branches, and it not only puts the disjuncts into the branches separately but also puts the negation of one of the disjuncts into a branch. By doing this, complement splitting makes the two branches disjoint with each other.

$$\frac{\phi \vee \psi}{\begin{array}{c|c} \phi & \neg\phi \\ & \psi \end{array}}$$

Table 3.4: Rule of complement splitting

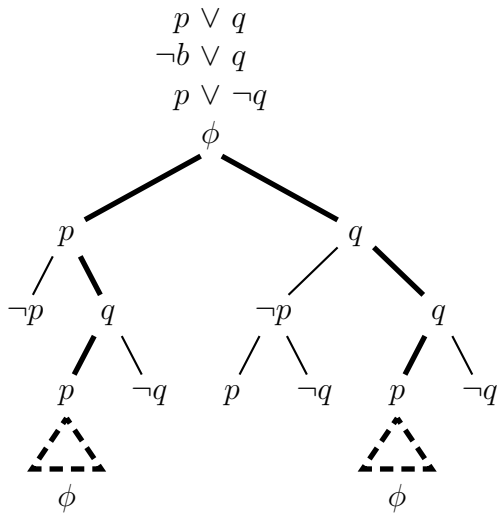


Figure 3.1: Example of derivation tree

Though complement splitting can remove some redundancy, it is not always the optimal choice. Because complement splitting adds an extra formula to one of the branches, if this formula is a complex one, the overhead of calculation may compromise the benefit of using complement splitting. In [HPS98], Horrocks and Patel-Schneider made the observation that while complement splitting is very effective with randomly generated problems, it does not work very well with realistic description logic knowledge bases. In some cases complement splitting even degrades the performance of the prover.

3.5 Backjumping

Backjumping is a kind of non-chronological backtracking method [Bak95, Dec89, SS77]. It aims to avoid unproductive backtracking by using dependency information.

In a tableau derivation, when a clash is found, the current branch needs to be closed and backtracking has to be performed. A straightforward way to do backtracking is chronological backtracking, which selects the most recent previously expanded disjunction as the point to backtrack to and removes the formulae after that point. The problem is that the formulae removed during chronological backtracking may have nothing to do with the clash. Therefore the proof search may be hindered by the same clash many times until the formula that actually caused the clash is removed by backtracking. This scenario is called *thrashing*[Mac87].

Thrashing can be alleviated by using backjumping. The tableau derivation for the formulae in Table 3.5 is shown to illustrate how backjumping works. The tableau proof is given in Figure 3.2.

$$p \vee q \tag{3.5.15}$$
$$s \vee t \tag{3.5.16}$$
$$\phi_1 \vee \psi_1 \tag{3.5.17}$$
$$\vdots$$
$$\phi_n \vee \psi_n \tag{3.5.18}$$
$$\neg p \vee \neg s \tag{3.5.19}$$

Table 3.5: Example formula set for backjumping

In Figure 3.2, each node represents a formula. The number at the beginning of a formula is the id of this formula. The numbers inside the square brackets at the end of a formula is called the *dependency set*, which contains the ids of the formulae that this formula depends on.

We can see from the tableau proof that formulae 6, 7, 8 and 9 come from formulae 1, 2, 3 and 4 respectively via disjunction expansion. When formula 10 is derived from formula 5, a clash is found between formulae 10 ($\neg p$) and 6 (p). This branch is closed. Then we try to find the most recent disjunction whose disjunct has caused this closure. And this disjunction must have an unexplored disjunct. In this case formula 5 meets the requirements. Backtracking is performed on formula 5. Another branch is created with formula 11. The dependency set of formula 11 includes not only its parent formula but also the formulae that caused the closure the first branch. Suppose d is the parent disjunction, f is the previously failed disjunct and D is the dependency set of the closure caused by f. The dependency set of the new disjunct is $(D \cup \{d\})\backslash\{f\}$. When formula 11 is derived, a clash is found between formula 11 ($\neg s$) and formula 7 (s). This time using the dependency sets and backjumping, choose formula 2 as the backtracking point. Formula 3 to 4 are bypassed. This is where chronological backtracking and backjumping depart. In this situation, chronological backtracking would select the immediate previous disjunction to backtrack, which is formula 4. Because formula 4 has nothing to do with the clashes, the branch will be closed by the same clashes we had before. If there are n disjunctions from formula 3 to formula

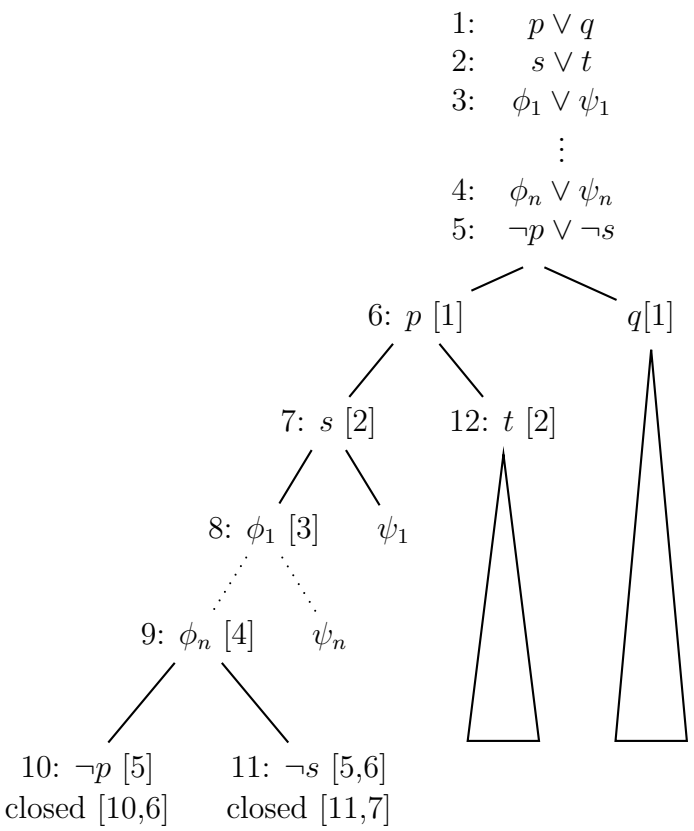


Figure 3.2: Tableau derivation for formulae in Table 3.5

4, it will take 2^n backtrackings for chronological backtracking to reach formula 2.

Backjumping has been implemented in FaCT and DLP [Hor97, PSH99]. Empirical tests have shown that backjumping is one of the most effective optimization techniques. It can dramatically improve the performance of the inference procedure for both random problems and realistic knowledge bases.

3.6 Caching

Caching was discussed in [Hor97] as a technique to improve the efficiency of description logic reasoning. Because of the similarity between description logic and modal logic, caching can also be used in modal logics. During tableau derivation for description logics (modal logics) the same sets of formulae are often propagated to different worlds. The satisfiability of a world is only decided by the set of formulae in that world. Therefore, the intuition of caching is that if a set of formulae appears in more than one world, we can keep the satisfiability result of this set of formulae after the first time they are solved and reuse the result later.

We use two sets S and U to record satisfiability and unsatisfiability results respectively. S and U contain sets of formulae. Given a world w, let Ω_w be a set that contains all the formulae in the current branch that have the label w. Suppose w has a predecessor w' and the ν rule has been applied to all the possible formulae on w'.

- If there is a $\Theta \in S$ such that $\Omega_w \subseteq \Theta$, then Ω_w is satisfiable.
- If there is a $\Theta \in U$ that $\Theta \subseteq \Omega_w$, then Ω_w is unsatisfiable.

Example 3.6.1.

$$\begin{aligned}
&\Diamond(p \wedge r) && (f_1)\\
&\Diamond q && (f_2)\\
&\Box\psi_1 && (f_3^1)\\
&\Box\psi_2 && (f_3^2)\\
&\vdots\\
&\Box\psi_n && (f_3^n)\\
&\Box(p \wedge q) && (f_4)
\end{aligned}$$

First, the π rule is applied to f_1. A world w_1 is created. Then the ν rule is applied to formulae f_3^1 to f_3^n and f_4. We have $\Omega_{w_1} = \{p, r, \psi_1, \ldots, \psi_n, q\}$. If Ω_{w_1} is satisfiable, add Ω_{w_1} to S. Suppose $S = \{\{p, r, \psi_1, \ldots, \psi_n, q\}\}$. Now apply the π rule to f_2, create another world w_2 and apply the π rule to formulae f_3^1 to f_3^n and f_4. We have $\Omega_{w_2} = \{q, \psi_1, \ldots, \psi_n, p\}$. Since there is a set in S that is a superset of Ω_{w_2}, we know that Ω_{w_2} is satisfiable without any further derivation. When $\psi_1, \ldots, \psi_n$ are very complicated the performance gain of caching in this example is significant.

Caching has been incorporated in FaCT and DLP [PSH99]. Empirical results have shown that caching can successfully alleviate the computational difficulty of description (modal) logic reasoning. Caching is especially effective for the problems with heavily nested modalities like the TANCS problems [MD00, PS00, Hor00].

3.7 Selection heuristics

Heuristics are widely used in theorem provers to improve their performance. Heuristics adapt certain strategies to guide proof search when there is non-determinism. In particular, in a tableau-based algorithm, heuristics are used to select a formula to be processed next when there is more than one formula available.

In tableau-based theorem proving, there are two kinds of non-determinism:

- Select a formula from a list of unexpanded formulae (e.g. those on the current branch). Because the set of available formulae can be interpreted as the conjunction of these formulae, this form of selection is called *conjunct selection*.

- Select a formula from the disjuncts of a disjunctive formula. We call this form of selection *disjunct selection* .

Different strategies should be used in conjunct selection and disjunct selection to get better performance. For conjunct selection, a formula that is likely to close more branches should be selected first, because when the selected formula is found to lead to a contradiction, the other candidates in the formula list no longer need to be explored. For disjunct selection, the formula that is least likely to lead to a contradiction should be selected first, because when the selected formula leads to a model, there is no need to explore the other alternative disjuncts.

There is a variety of heuristic methods. Some of them intuitively give priority to the formulae that are easier to handle. There are two kinds of formulae which are more expensive to process than others. Firstly, disjunctions require branching and backtracking, which can include intensive calculation and a great number of memory operations. Secondly, diamond formulae generate new worlds which may cause lots of expansions of box formulae. And if caching is implemented, the newly generated world has to be compared with the existing worlds. Therefore some heuristic methods process disjunctions and diamond formulae as late as possible.

The heuristic Maximum Occurrence in disjunctions of Minimum Size, also known as MOMS [Fre95] is a technique for disjunct selection. It considers all disjunctions of minimum size and counts the positive and negative occurrences of disjuncts. Working with complement splitting and unit propagation, MOMS intends to maximize the possibility of deterministic expansion.

$$\phi \vee \psi_1 \tag{3.7.20}$$
$$\phi \vee \psi_2 \tag{3.7.21}$$
$$\vdots$$
$$\phi \vee \psi_n \tag{3.7.22}$$

Table 3.6: Example formula set for MOMS

A set of formulae is given in Table 3.6. Because in these formulae, ϕ has the highest number of occurrences, MOMS first selects ϕ to do complement splitting. If Formula 3.7.20 is the one chosen by conjunct selection, one of the two branches created by MOMS contains ϕ and the other branch contains $\neg\phi$ and ψ_1. Since there are more occurences of ϕ than occurences of $\neg\phi$ in the formula set, the branch that has $\neg\phi$ is tested first. By using unit propagation, deterministic expansion is performed on the rest of the formulae and yields ψ_2, ..., ψ_n.

MOMS is incorporated in FaCT and DLP [PSH99]. Empirical tests have been performed. The effectiveness of MOMS is discussed in [HPS02]. The test results show that MOMS interacts adversely with backjumping.

It is worth noting that the strategies we recommended for disjunct selection do not totally agree with MOMS. In the example shown above, after two branches are created, one has ϕ, another has $\neg\phi$ and ψ_1, MOMS selected the second branch to test first which resulted in adding $\neg\phi$, ψ_2, ..., ψ_n to the branch. Another strategy is to select the first branch, in this way only ϕ will be added to the branch. Compared with MOMS, the new strategy puts fewer formulae into the branch, which will more likely lead to satisfiability. This phenomena was also discovered by Hladik [Hla00]. He named the new strategy inverse of MOMS (iMOMS). On the test results presented in his paper iMOMS had a better performance than MOMS.

Chapter 4

Dynamic backtracking

Previous efforts in this area have shown that intelligent backtracking algorithms play a significant (if not the most important) role among all the optimization techniques for tableau theorem provers. Intelligent backtracking algorithms are improvements over traditional chronological backtracking. Examples of intelligent backtracking algorithms include backjumping, conflict-directed backjumping and dynamic backtracking. In this chapter, the focus is on dynamic backtracking. First we introduce dynamic backtracking for propositional clauses, then we extend that algorithm to modal clausal formulae and finally we present a dynamic backtracking algorithm for general modal formulae.

The rest of this chapter is organized as follows: Section 4.1 introduces intelligent backtracking techniques. Section 4.2 describes a dynamic backtracking algorithm for propositional clauses. Section 4.3 presents a dynamic backtracking algorithm for general propositional formulae. Section 4.4 describes a dynamic backtracking algorithm for modal logic.

4.1 Intelligent backtracking

Backtracking is widely used to solve problems that involve non-determinism. Solving these problems needs exponential space but the memory consumption can be dramatically reduced with clever backtracking techniques.

In recent years, developments in artificial intelligence led to thorough reinvestigations of backtracking algorithms. Chronological backtracking is not efficient because it does not remember the reason for its earlier failed attempts. As a consequence it may waste too much effort on the same failure. In other words,

if the reasons for earlier failure can be used to guide the ensuing proof search, the performance can probably be significantly improved. Researchers have discovered this and proposed new backtracking algorithms. Examples include backjumping [Gas79] and conflict-directed backjumping [Pro93]. Conflict-directed backjumping employs conflict sets to keep the information collected from previous failures. Each conflict set corresponds to a disjunctive formula. The set contains the formulae that were in contradiction with the disjuncts of this disjunctive formula that have been tried earlier. With the help of conflict sets conflict-directed backjumping can find the decision point which is the true source of the current failure. Though conflict-directed backjumping saves time by performing less backtracking, it undoes too much work unnecessarily during backtracking. In [GM94], Ginsberg therefore proposed a new form of backtracking, called *dynamic backtracking*, for solving constraint satisfaction problems (CSPs). Dynamic backtracking can be viewed as an improvement on conflict-directed backjumping. It has mechanisms not only to reduce the amount of backtracking performed but also to undo as little work as possible during backtracking. Constraint satisfaction problems can be translated into propositional clauses. Therefore we can use dynamic backtracking to solve propositional formulae in clausal form.

Now we use abstract derivation trees to explain how dynamic backtracking algorithm works and to discuss the differences between chronological backtracking, conflict-directed backjumping and dynamic backtracking. In these derivation trees, nodes connected by wider edges are on the current branch. The symbol '×' means that the branch is closed because of a clash.

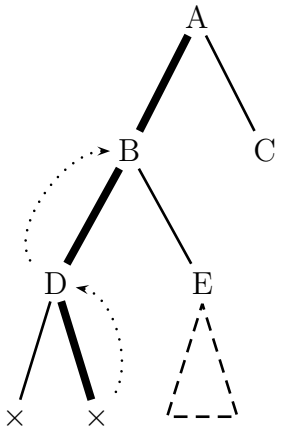


Figure 4.1: Derivation tree of chronological backtracking

Figure 4.1 shows the derivation tree of chronological backtracking. Nodes

A, B and D are on the current branch. Branches are closed below D. When backtracking, chronological backtracking always looks up the nearest upper node to check if there is any alternative available. In this case, D does not have any alternative, then chronological backtracking goes to B and derives another branch including E.

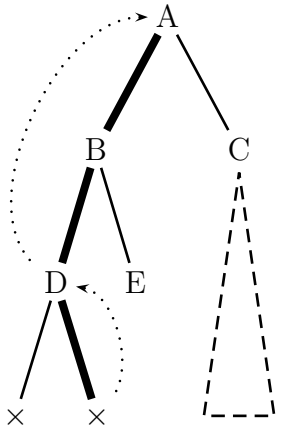


Figure 4.2: Derivation tree of backjumping and conflict-directed backjumping

Figure 4.2 shows the derivation tree of backjumping or conflict-directed backjumping. Like Figure 4.1, branches are closed below D due to clashes. Suppose the clashes involve A and do not involve B. Chronological backtracking will still backtrack on B no matter what happens and the new branches will be closed by the same clashes again. While backjumping and conflict-directed backjumping can recognize the situation and perform backtracking on A instead of B.

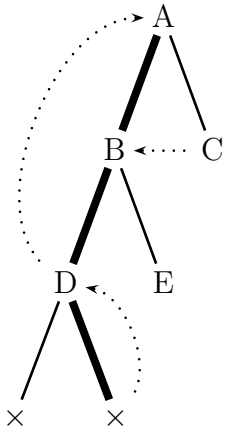


Figure 4.3: Derivation tree of dynamic backtracking

Figure 4.3 shows the derivation tree for dynamic backtracking. Under the same circumstances as we described in Figure 4.2, dynamic backtracking can also

find A as the backtracking point. When backtracking, chronological backtracking, backjumping and conflict-directed backjumping all abandon the branch already derived below the backtracking point and develop another new branch which inevitably repeat some work that has been done before. In contrast, instead of removing the whole branch below the backtracking point, dynamic backtracking only replaces the nodes derived form the backtracking point with the new nodes and therefore avoids redoing the branch again.

A concrete example is given below to further explain dynamic backtracking. The derivation tree for this problem is given in Figure 4.4.

Example 4.1.1.

$$a \vee b \qquad (f_1)$$
$$\phi \vee \psi \qquad (f_2)$$
$$\neg a \vee c \qquad (f_3)$$
$$\neg a \vee \neg c \qquad (f_4)$$

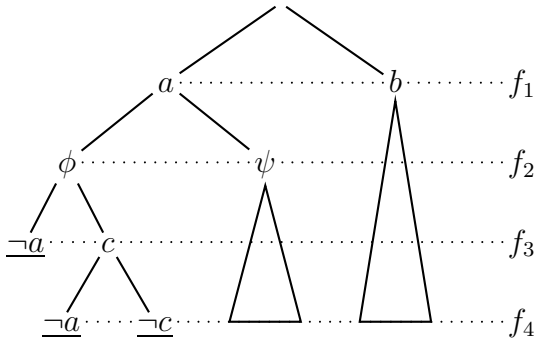


Figure 4.4: Derivation tree for Example 4.1.1

Each time a clash is found and backtracking is needed, conflict-directed backjumping records the set of disjunctions, whose subformulae are involved in the clash. These sets are called *conflict sets*. Let f_i be a disjunctive formula. Let $C(f_i)$ be the conflict set for f_i. Initially all the conflict sets are set to $\emptyset$. We perform depth-first search on the tree shown in Figure 4.4 from left to right. We find the first clash at level 3, where both a and $\neg a$ are present in the current branch. This means there is a clash. Conflict-directed backjumping records the conflict information by setting $C(f_3) := C(f_3) \cup \{f_1\} = \emptyset \cup \{f_1\} = \{f_1\}$, because

a is from f_1 and $\neg a$ is from f_3. Then backtracking is performed on f_3 and another branch is selected for f_3. The next clash happens when we expand f_4. If we select $\neg a$, it is in conflict with f_1, therefore we set $C(f_4) := \{f_1\}$. Now we backtrack and select $\neg c$, it is in conflict with f_3. So we set $C(f_4) := C(f_4) \cup \{f_3\} = \{f_1, f_3\}$. Now we can use the conflict sets to find out where to backtrack to. First, we consider the conflict set for f_4, and consider the formula that is deepest in the derivation tree. This is f_3. Due to both of f_3's branches having been explored, we perform the same operation to f_3's conflict set as we did to f_4's conflict set, namely, we look for the deepest formula in $C(f_3)$. Now f_1 is the deepest formula (actually $C(f_3)$ only has one item). Hence we backtrack to f_1, formulae f_2, f_3 and f_4 are eliminated from the current branch, and their conflict sets are cleared.

If chronological backtracking was used instead, we would backtrack to f_2 instead of f_1. f_2 can however be extremely complicated. Since conflict-directed backjumping avoids looking through the sub tree of f_2, the search space of conflict-directed backtracking is considerably smaller than that of chronological backtracking.

Dynamic backtracking records the same kind of information as conflict-directed backjumping, but at a finer level. Dynamic backtracking employs sets, called *eliminating explanations* to keep the information collected from previous failures. Given a disjunctive formula, for each of its disjunct there is a eliminating explanation which records the formulae that were in contradiction with this disjunct in earlier derivation. Let f_i be a disjunction and τ a disjunct of f_i. $E(f_i, \tau)$ denotes the elimination explanation for f_i with respect to τ. The difference between eliminating explanations and conflict sets is that eliminating explanation not only keep the information for each disjunction but also for every disjunct individually. Conflict sets do not distinguish information for different disjuncts.

Given the formula from Example 4.1.1, dynamic backtracking works in the same way as conflict-directed backjumping until it finds a clash. After the clash between a, the left child of f_1 and $\neg a$, the left child of f_3 occurs, an eliminating explanation is recorded as $E(f_3, \neg a) = \{f_1\}$. It means that $\neg a$, a disjunct of f_3 is blocked because of the currently selected disjunct of f_1. After the clash between $\neg a$ the left child of f_4 and a the left child of f_1 occurs, we set $E(f_4, \neg a) = \{f_1\}$. And after the clash between $\neg c$, the right child of f_4 and c, the right child of f_3, we set $E(f_4, \neg c) = \{f_3\}$. Now we use the eliminating explanations to find where to backtrack to. For this purpose, we need a temporary set T, which

serves as a buffer. Let T be the union of all the eliminating explanations of f_4. So, $T = \{f_1, f_3\}$ and the deepest formula in T is f_3. Since f_3 has no unexplored branch, we replace f_3 in T by the union of all the eliminating explanations of f_3. The new T is $\{f_1\}$. Again it chooses f_1 to backtrack to. Unlike conflict-directed backtracking, dynamic backtracking does not remove the formulae f_2, f_3 and f_4 from the current branch. Only f_1's left child a is removed and replaced by its right child b. Since the selected branch of f_1 has changed, we need to update the eliminating explanations of all the disjuncts that previously were contradicting with f_1. These disjuncts can be found by looking for the eliminating explanations that involve f_1. The status of these disjuncts are changed from blocked to unexplored and their eliminating explanations are cleared. Compared with conflict-directed backtracking, dynamic backtracking avoids re-expanding one of the branches of f_2, f_3 and f_4. Therefore the search space saved is subject to the size of these formulae.

4.2 Dynamic backtracking for propositional clauses

The previous section explains how dynamic backtracking works on propositional clauses. Now we present an algorithm implementing dynamic backtracking.

Before the description of the algorithm, we mention the data structures, functions and flags used in the algorithm.

I:	The initial problem as a set of clauses.
B:	A sequence of formulae. It contains all the formulae in the current branch.
D:	A sequence of non-unit disjunctive formulae. It contains all the disjunctions on the current branch, which have been expanded.
T:	A temporary set of non-unit disjunctive formulae. subformulae of the formula ϕ.
$ParentDisjs(\phi)$:	A function that returns disjunctive super-formulae of the formula ϕ.

$Clashes(B)$:	A function that returns a set of ordered pairs (ϕ, ψ), where ϕ and ψ are complementary formulae on the branch B, and ϕ was put into B earlier than ψ.
$E(f, \tau)$:	A set of non-unit disjunctive formulae. It is the eliminating explanation for τ. τ is a disjunct of the disjunctive formula f.
unknown/*active*/*blocked*:	Flags to indicate a formula's state.
expanded:	A flag to indicate if any expansion rule has been applied to a formula.
$Unexpanded(B)$:	A function that returns a formula f, $f \in B$ and f is not marked as *expanded*.

The algorithm of dynamic backtracking for clausal propositional formulae is given in Algorithm 1. Since this algorithm is dedicated to formulae in clausal form, $ParentDisjs(\phi)$ only returns a single disjunctive formula. A formula's state can be *unknown*, *active* or *blocked*. At the beginning, all of the formulae are initialized to be *unknown*. A formula's status is switched from *unknown* to *active* after that formula is expanded and put into the current branch. If that formula is involved in a clash and is backtracked, its status will be changed to *blocked*. Flag *expanded* is used to distinguish the unexpanded formulae and the formulae that have been expanded.

Algorithm 1 implements the tableau calculi we introduced in Chapter 2. When a clash is found, Algorithm 2 and 3 are invoked to find the backtracking point and perform backtracking. In the step 6 of Algorithm 2, the reason that we look through D in reverse order is that we want to find the most recent disjunction that is relevant to the clash. If we select an arbitrary disjunction from T, it is possible that some consistent fully expanded branch is missed and consequently the completeness of the algorithm is compromised. The speciality of dynamic backtracking lies in the step 3 to 7 of Algorithm 3, where it only abandons the subformulae of disjunction at the backtracking point and keeps the other formulae. When a formula f is removed from the current branch due to backtracking, we need to find all the formulae that were previously blocked by f and initialize their status and elimination explanations. When f is changed, we are no longer

Algorithm 1 proof_search

Input: I: a set of input formulae

Output: Returns "satisfiable" if the conjecture is satisfiable, otherwise returns "unsatisfiable".

```
1: B := I, D := ∅
2: while Clashes(B) ≠ ∅ or Unexpanded(B) ≠ null do
3:   if Clashes(B) = ∅ then
4:     φ = Unexpanded(B)
5:     if φ is a conjunction then
6:       for all ψ such that ψ is a conjunct of φ do
7:         set B := B ∪ {ψ}
8:         set ψ to be active
9:       end for
10:    else if φ is a disjunction then
11:      set ψ to be one of φ's disjuncts
12:      set B := B ∪ {ψ}
13:      set D := D ∪ {ψ}
14:      set ψ to be active
15:    end if
16:    set φ to be expanded
17:  else
18:    f = find_dynamic_backtracking_point(Clashes(B))
19:    if f = "null" then
20:      return "unsatisfiable"
21:    else
22:      dynamic_backtrack(f)
23:    end if
24:  end if
25: end while
26: return "satisfiable"
```

Algorithm 2 *find_dynamic_backtracking_point*

Input: S: a set of complementary formulae

Output: Returns the formula where backtracking is to be performed. If such a formula is not found, returns "null"

```
1: T := ∅
2: for all f ∈ S do
3:    set c to be the clause the f belongs
4:    T := T ∪ {c}
5: end for
6: for all ψ ∈ D in reverse order do
7:    if ψ ∈ T then
8:       for all π such that π is a disjunct of ψ do
9:          if π is neither active nor blocked then
10:            set ρ to be ψ's current active disjunct
11:            set E(ψ, ρ) = T − {ψ}
12:            return ψ
13:         end if
14:      end for
15:      for all μ such that μ is a disjunct of ψ do
16:         T = T ∪ E(ψ, μ)
17:      end for
18:      T = T − {ψ}
19:   end if
20: end for
21: return "null"
```

Algorithm 3 *dynamic_backtrack*

Input: ψ: the disjunction where backtracking is to be performed

Output: none

```
1: set ρ to be ψ's current active disjunct
2: set ρ to be blocked
3: for all γ such that γ's elimination explanation includes ρ do
4:    set γ to be unknown
5:    clear γ's elimination explanation
6: end for
7: remove ρ from B
8: set π to be a disjunct of ψ that is not blocked
9: set π to be active
10: set B := B ∪ {π}
```

sure whether those formulae are blocked or not, so we have to initialize them and perform a search on them later if necessary.

Example 4.2.1.

$$a \vee b \qquad (f_1)$$
$$\phi \vee \psi \qquad (f_2)$$
$$\neg a \vee c \qquad (f_3)$$
$$\neg a \vee \neg c \qquad (f_4)$$

We use the formulae shown in Example 4.2.1 to illustrate how dynamic backtracking works on general formulae. Actually, a tree-based structure is not appropriate to describe the derivation procedure using dynamic backtracking. Therefore we will utilize a table to explain how dynamic backtracking works. The table has three columns entitled 'disjunction', 'disjunct' and 'E'. The contents of the column 'disjunction' denote the disjunctions in the current branch. The contents of the column 'disjunct' denote the current selected branch and the contents of the column 'E' are the eliminating explanations.

Suppose our search strategy is depth-first, left to right. First we expand f_1, its left branch a is selected. When we expand f_2, we select ϕ. Then f_3 is expanded, its left disjunct $\neg a$ is put into the current branch. The derivation status becomes:

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	$\neg a$	

There is a clash on the current branch between formulae a and $\neg a$. They belong to clauses f_1 and f_3 respectively. Therefore the temporary set $T = \{f_1, f_3\}$. The more recent one, f_3 is selected to backtrack and its another disjunct is selected. f_1 is added into the elimination explanation for the disjunct $\neg a$ in f_3.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	c	$\neg a : f_1$

We continue by expanding f_4 and put its left disjunct into the current branch.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	c	$\neg a : f_1$
f_4	$\neg a$	

There is a clash on the current branch between formulae a and $\neg a$. They belong to clauses f_1 and f_4 respectively. Therefore the temporary set $T = \{f_1, f_4\}$. The more recent one, f_4 is selected to backtrack and its another disjunct is selected. f_1 is added into the elimination explanation for the disjunct $\neg a$ in f_4.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	c	$\neg a : f_1$
f_4	$\neg c$	$\neg a : f_1$

There is another clash on the current branch between formulae c and $\neg c$. They belong to clauses f_3 and f_4 respectively. Therefore the temporary set $T = \{f_3, f_4\}$. The more recent one, f_4 is selected to backtrack, but another disjunct of f_4 has been blocked by f_1. Hence f_1 is added into T, f_4 is taken from T. The temporary set T becomes $\{f_1, f_3\}$ and the next most recent disjunction f_3 is selected to backtrack. Again the alternative disjunct of f_3 is also blocked by f_1. Since f_1 is already in T, f_3 is taken from T, T becomes $\{f_1\}$. Now the last formula in T, f_1 is selected to backtrack. Its current selected disjunct a is removed from the current branch, all the elimination explanations related to a are updated. Another disjunct of f_1, b is selected.

Disjunction	*Disjunct*	*E*
f_1	b	$a : \emptyset$
f_2	ϕ	
f_3	c	
f_4	$\neg c$	

The clash between c and $\neg c$ still exists on the current branch. Again we need to apply dynamic backtracking. These two formulae belong to f_3 and f_4 respectively. The temporary set T becomes $\{f_3, f_4\}$. The more recent one, f_4 is selected to backtrack. This time, the other disjunct of f_4, $\neg a$, has been released due to the backtracking performed on f_1. We can backtrack on f_4 and select $\neg a$.

Disjunction	*Disjunct*	*E*
f_1	b	$a : \emptyset$
f_2	ϕ	
f_3	c	
f_4	$\neg a$	$\neg c : f_3$

So far every formula has been expanded and there is no clash on the current branch. The derivation terminates and the input formulae are satisfiable.

4.3 Dynamic backtracking for general propositional formulae

The dynamic backtracking algorithm we presented above is designed for formulae in clausal form. Given an arbitrary formula we have to transform it into clauses before our algorithm can be applied. This transformation can cause significant growth in terms of the size of a formula and the number of the variables. To avoid this we present a new algorithm which works on any form of formula.

The data structures, functions and flags are defined as follows:

I: A set of formulae as the initial problem.

B: A sequence of formulae. It contains all the formulae in the current branch.

D: A sequence non-unit disjunctive formulae. It contains all the disjunctions on the current branch, which have been expanded.

$ParentDisjs(\phi)$: A function that returns disjunctive super-formulae of the formula ϕ.

$Clashes(B)$:	A function that returns a set of ordered pairs (ϕ, ψ), where ϕ and ψ are complementary formulae on the branch B, and ϕ was put into B earlier than ψ.
$E(f, \tau)$:	A set of non-unit disjunctive formulae. It is the eliminating explanation for τ. τ is a disjunct of the disjunctive formula f.
unknown/*active*/*blocked*:	Flags to indicate a formula's state.
expanded:	A flag to indicate if any expansion rule has been applied on a formula.
$Unexpanded(B)$:	A function that returns a formula f, $f \in B$ and f is not marked as *expanded*.

Most of the definitions are same as the dynamic backtracking clausal form version, except that a new function $ParentDisjs(\phi)$ is introduced which returns all of the disjunctions that are ancestors of ϕ.

The algorithm of *proof_search* is the same as the clausal form version. Hence we won't give it again. The other algorithms, *find_dynamic_backtracking_point* and *dynamic_backtracking* are described in Algorithm 4 and 5 respectively.

The difference between the algorithm for clausal formulae and the algorithm for non-clausal formulae is that while traditional dynamic backtracking only looks for a disjunction which has an unblocked disjunct from the elimination explanations, non clausal dynamic backtracking has to find such a disjunction that either itself or one of its subformulae is contained in the elimination explanations.

Example 4.3.1.

$$a \vee^1 b \qquad (f_1)$$

$$\phi \vee^2 \psi \qquad (f_2)$$

$$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d)) \vee^3 \gamma \qquad (f_3)$$

$$c \vee^6 d \qquad (f_4)$$

Algorithm 4 *find_dynamic_backtracking_point*

Input: S: a set of complementary formulae

Output: Returns the formula where backtracking is to be performed. If such a formula is not found, returns "null".

```
1: T := ∅
2: for all f ∈ S do
3:    T := T ∪ ParentDisjs(f)
4: end for
5: for all ψ ∈ D in reverse order do
6:    if ψ ∈ T then
7:       for all π such that π is a disjunct of ψ do
8:          if π is neither active nor blocked then
9:             set ρ to be ψ's current active disjunct
10:            set E(ψ, ρ) = T − {ψ}
11:            return ψ
12:         end if
13:      end for
14:      for all μ such that μ is a disjunct of ψ do
15:         T = T ∪ E(ψ, μ)
16:      end for
17:      T = T ∪ ParentDisjs(ψ)
18:      T = T − {ψ}
19:   end if
20: end for
21: return "null"
```

Algorithm 5 *dynamic_backtrack*

Input: ψ: the disjunction where backtracking is to be performed

Output: none

```
1: set ρ to be ψ's current active disjunct
2: set ρ to be blocked
3: for all γ such that γ's elimination explanation includes ρ do
4:    set γ to be unknown
5:    clear γ's elimination explanation
6: end for
7: remove ρ and its subformulae from B and D
8: set π to be a disjunct of ψ that is not blocked
9: set π to be active
10: set B := B ∪ {π}
```

We use the formulae shown in Example 4.3.1 to illustrate how dynamic backtracking works on general formulae. Like the previous example, our search strategy is still depth-first, left to right. When we expand f_1, we select its left branch a. When we expand f_2, we select ϕ. The derivation status becomes:

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	

Now we expand f_3 and its left branch is selected, which is a conjunction. Then we apply expansion rule for conjunctions and put both subformulae into the current branch.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d))$	

Next we expand f_4 by selecting $\neg a$. It is in conflict with a in the current branch. Then we need to perform backtracking on f_4 and select another branch $\neg c$. The elimination explanation is recorded.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d))$	
f_4	$\neg c$	$\neg a : f_1$

Like the last step, we expand f_5 by selecting $\neg a$. It is in conflict with a in the current branch. Then we need to perform backtracking on f_5 and select another branch $\neg d$. The elimination explanation is recorded.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d))$	
f_4	$\neg c$	$\neg a : f_1$
f_5	$\neg d$	$\neg a : f_1$

When we try to expand f_6, its left branch is in conflict with f_4 and its right branch is in conflict with f_5.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d))$	
f_4	$\neg c$	$\neg a : f_1$
f_5	$\neg d$	$\neg a : f_1$
f_6	d	$c : f_4; d : f_5$

Now we need to decide upon which formula we should backtrack. For this purpose, we introduce a temporary set T, which is initialized with the elimination explanation of f_6, $T = \{f_4, f_5\}$. Since neither f_4 nor f_5 has a branch available, we make $T := (T - \{f_4\} - \{f_5\}) \cup E(f_4) \cup E(f_5) = \{f_1\}$. Now it is time to be careful. If we used the algorithm for formulae in clausal form, we would perform backtracking on f_1, which is wrong. By applying the algorithm for general formulae, $T := T \cup \{f_3\} = \{f_1, f_3\}$. Because f_3 is more recent than f_1, we should backtrack on f_3. In consequence f_3's subformulae f_4 and f_5 are removed from the current branch and the elimination explanations related to those formula need to be cleared.

Disjunction	*Disjunct*	*E*
f_1	a	
f_2	ϕ	
f_3	γ	$((\neg a \vee^4 \neg c) \wedge (\neg a \vee^5 \neg d)) : f_1$
f_6	d	

4.4 Dynamic backtracking for general modal formulae

Not all of the modal logic formulae can be transformed into strict clausal form. But the original dynamic backtracking (DB) proposed by Ginsberg can only deal with formulae in clausal forms. If his algorithm is used on non-clausal formulae, it

could miss some branches of the derivation tree and therefore becomes incomplete. Here we extend the traditional dynamic backtracking algorithm to handle modal formulae in any form (clausal and non-clausal form).

The data structures, functions and flags used in our algorithms are defined as follows:

I:	A set of initial formulae.
B:	A sequence of labelled formulae. It contains all the labelled formulae on the current branch.
D:	A sequence of non-unit disjunctive formulae. It contains all the disjunctions on the current branch, which have been expanded.
W:	A set of worlds.
R:	A set of pairs (w_1, w_2), with $w_1, w_2 \in W$.
$Succ_R(w)$:	A function that returns a set of worlds. For every w' such that $(w, w') \in R$, $w' \in Succ_R(w)$.
$BoxWorld_w$:	A set of formulae, $w \in W$. For every ϕ such that $w : \Box\phi) \in B$, $\phi \in BoxWorld_w$.
$DiaWorld_w$:	A set of labelled formula, $w \in W$. World w is generated because of the expansion of the formulae in $DiaWorld_w$.
$ParentDisjs(\phi)$:	The disjunctive super-formulae of the formula ϕ.
$Clashes(B)$:	A function that returns a set of ordered pairs (ϕ, ψ), where ϕ and ψ are complementary formulae on the branch B, and ϕ was put into B earlier than ψ.
$E(f, \tau)$:	A set of non-unit disjunctive formulae. It is the eliminating explanation for τ. τ is a disjunct of the disjunctive formula f.

unknown/active/blocked: Each formula has a flag indicating its state. A formula's state can be *unknown*, *active* or *blocked*. *unknown* is a formula's initial state. A formula is set to be *active* if it is in the current branch. A formula is set to be *blocked* if it caused a contradiction.

expanded: A flag that indicates if any expansion rule has been applied to that formula.

$Unexpanded(B)$: A function that returns a formula f, $f \in B$ and f is not marked as *expanded*.

When a formula is removed from the branch, its flags and data structures will be cleared.

The dynamic backtracking algorithms for modal formulae are shown in Algorithms 6, 7 and 8.

The formulae shown in Example 4.4.1 are used as the input formulae to explain how dynamic backtracking works for modal formulae.

Example 4.4.1.

$$\Diamond a \vee c \qquad (f_1)$$
$$\phi \vee \psi \qquad (f_2)$$
$$\Box(\neg a \vee \neg b) \vee c \qquad (f_3)$$
$$\Box b \vee \Box\neg a \qquad (f_5)$$

We still stick to depth-first left-to-right search strategy. f_1 has a disjunct which is a diamond formula. We create a new world named w_1. After expanding f_2 and f_3, the state is shown by the table below.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	$\Box(\neg a \vee \neg b)$	

Algorithm 6 proof_search

Input: I: a set of input formulae

Output: Returns "satisfiable" if the conjecture is satisfiable, otherwise returns "unsatisfiable".

set $D := \emptyset$, create world w and $W := \{w\}$
for all f such that $f \in I$ **do**
  $B := B \cup \{w : f\}$
end for
while $Clashes(B) \neq \emptyset$ or $Unexpanded(B) \neq null$ **do**
  if $Clashes(B) = \emptyset$ **then**
    $w : \phi = Unexpanded(B)$
    if ϕ is a conjunction **then**
      for all ψ such that ψ is a conjunct of ϕ **do**
        set $B := B \cup \{w : \psi\}$ and set $w : \psi$ to be *active*
      end for
    else if ϕ is a disjunction **then**
      set ψ to be one of ϕ's disjuncts
      set $B := B \cup \{w : \psi\}$, $D := D \cup \{w : \psi\}$ and set $w : \psi$ to be *active*
    else if ϕ is a diamond formula **then**
      set ψ to be ϕ's immediate subformula
      create a new world w', set $W := W \cup \{w'\}$ and $R := R \cup \{(w, w')\}$
      set $DiaWorld_{w'} := DiaWorld_{w'} \cup \{w : \phi\}$
      set $B := B \cup \{w' : \psi\}$ and set $w' : \psi$ to be *active*
      for all π such that $\pi \in BoxWolrd_w$ **do**
        set $B := B \cup \{w' : \pi\}$
      end for
    else if ϕ is a box formula **then**
      set ψ to be ϕ's immediate subformula
      for all w' such that $w' \in Succ_R(w)$ **do**
        set $B := B \cup \{w' : \psi\}$ and set $w' : \psi$ to be *active*
      end for
      set $BoxWorld_w = BoxWorld_w \cup \{\psi\}$
    end if
    set $w : \phi$ to be *expanded*
  else
    $f = find\_dynamic\_backtracking\_point(Clashes(B))$
    if $f =$ "null" **then**
      return "unsatisfiable"
    else
      $dynamic\_backtrack(f)$
    end if
  end if
end while
return "satisfiable"

Algorithm 7 *find_dynamic_backtracking_point*

Input: S: a set of complementary labelled formulae

Output: Returns the labelled formula where backtracking is to be performed. If such a formula is not found, returns "null".

1: $T := \emptyset$
2: **for all** $f \in S$ **do**
3: $\quad T := T \cup ParentDisjs(f)$
4: **end for**
5: **for all** $\psi \in D$ in reverse order **do**
6: $\quad$ **if** $\psi \in T$ **then**
7: $\quad\quad$ **for all** π such that π is a disjunct of ψ **do**
8: $\quad\quad\quad$ **if** π is neither *active* nor *blocked* **then**
9: $\quad\quad\quad\quad$ set ρ to be ψ's current active disjunct
10: $\quad\quad\quad\quad$ set $E(\psi, \rho) = T - \{\psi\}$
11: $\quad\quad\quad\quad$ **return** ψ
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad\quad$ **for all** μ such that μ is a disjunct of ψ **do**
15: $\quad\quad\quad$ $T := T \cup E(\psi, \mu)$
16: $\quad\quad$ **end for**
17: $\quad\quad$ $T := T \cup ParentDisjs(\psi)$
18: $\quad\quad$ set w to be the world that ψ is labelled
19: $\quad\quad$ set δ to be the labelled formula in $DiaWorld_w$
20: $\quad\quad$ $T := T \cup ParentDisjs(\delta)$
21: $\quad\quad$ $T = T - \{\psi\}$
22: $\quad$ **end if**
23: **end for**
24: **return** "null"

Algorithm 8 *dynamic_backtrack*

Input: ψ: the disjunction where backtracking is to be performed
Output: none

1: set ρ to be ψ's current *active* disjunct
2: set ρ to be *blocked*
3: **for all** γ such that γ's elimination explanation includes ρ **do**
4: set γ to be *unknown*
5: clear γ's elimination explanation
6: **end for**
7: all the labelled formulae resulting from ρ are removed from B, D, $DiaWorld_w$ and $BoxWorld_w$ ($w \in W$).
8: all the worlds and relations resulting from ρ are removed from W and R.
9: set π to be a disjunct of ψ that is not *blocked*
10: set π to be *active*
11: set $B := B \cup \{\pi\}$

In the next step, we expand the selected disjunct of f_3. It is a box formula and we propagate the formula to world w_1. The left branch of the resulting disjunction is selected. Now we get a clash between f_4 and f_1.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	$\Box(\neg a \vee \neg b)$	
f_4	$w_1 : \neg a$	

We perform backtracking on f_4 and select the right branch. The eliminating explanation is recorded. Now we expand f_5, and the left branch is selected. It is a box formula and is propagated to world w_1.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	$\Box(\neg a \vee \neg b)$	
f_4	$w_1 : \neg b$	$\neg a : f_1$
f_5	$w_1 : b$	

We have a clash between f_4 and f_5. We backtrack on f_5, the right branch is selected, and the eliminating explanation is recorded.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	$\Box(\neg a \vee \neg b)$	
f_4	$w_1 : \neg b$	$\neg a : f_1$
f_5	$w_1 : \neg a$	$\Box b : f_4$

A clash occurs between f_5 and f_1 and there is no unexplored branch in f_5. The eliminating explanation for f_5 is $E(f_5) = \{f_4, f_1\}$. The backtracking temporary set becomes $T = E(f_5)$. The deepest formula in the set is f_4. But f_4 has no unexplored branch either. Then replace f_4 in T by f_4's parent disjunctive formula f_3 and f_4's eliminating explanation, which is $\{f_1\}$. T becomes $\{f_3, f_1\}$. Now the deepest formula in T is f_3 and f_3 has a unexplored branch. Therefore we backtrack on f_3, the right branch of f_3 is selected and the eliminating explanations for f_3 and f_5 are updated.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	c	$\Box(\neg a \vee \neg b) : f_1$
f_5	$w_1 : \neg a$	

However, backtracking on f_3 does not immediately solve the clash between $w_1 : a$ and $w_1 : \neg a$. To make the current branch clash free, we need to run dynamic backtracking again. Since f_5's alternative disjunct ($\Box b$) has been released in last step, we can now perform backtracking on f_5. After $\Box a$ is removed, the current branch does not have any clash. And after $\Box b$ is derived and propagated to world w_1, a model is found for the input formulae and the derivation terminates.

Disjunction	Disjunct	E
f_1	$w_1 : a$	
f_2	ϕ	
f_3	c	$\Box(\neg a \vee \neg b) : f_1$
f_5	$w_1 : b$	$\Box\neg a : f_1$

Chapter 5

Forward reasoning

Forward reasoning is a set of techniques that aim to reduce redundancies by looking into the search space that has not yet been explored. Examples of forward reasoning techniques are the looking ahead mechanism in SAT solvers and forward checking methods in Constraint Satisfaction Problems systems. Two forward reasoning techniques are used in our system. They are forward checking and forward implication.

Forward checking is well-known as one of the most promising prior pruning techniques for Constraint Satisfaction Problems [HE80]. In [Pro93], Prosser pointed to a hybrid algorithm which incorporates forward checking and conflict-directed backtracking as the most efficient algorithm among the nine algorithms he has tested. This result was confirmed later in [SG95].

To the best of our knowledge, no effort has been made to use forward checking in modal logic theorem provers. It would be interesting to see if forward checking works for modal logic. It would be also interesting to see how a hybrid algorithm, combining forward checking and dynamic backtracking, competes with other algorithms in modal logic reasoning.

Another forward reasoning technique is forward implication. In contrast to forward checking, forward implication makes use of implication instead of complementarity to prune the search space.

The purpose of this chapter is to describe an extended forward checking algorithm which is designed to handle modal formulae and to work with dynamic backtracking. In this chapter, we will also introduce forward implication for both propositional logic and modal logic.

The rest of this chapter is organized as follows: Section 5.1 introduces the

motivation for forward checking and briefly explains how forward checking works on propositional formulae. Section 5.2 describes the forward checking algorithm for propositional logic. Section 5.3 explains the precomputed connections used to determine complementary formulae in modal logic. Section 5.4 introduces the forward checking algorithm for modal logic. Section 5.5 shows how forward checking works with dynamic backtracking. Section 5.6 explains how to release a blocked formula. Section 5.7 introduces the forward implication for propositional logic. Section 5.8 describes the forward implication algorithm for modal logic.

5.1 Introduction

Tableau-based algorithms traditionally perform consistency checking in a backward fashion. When a new formula is added to the current branch, the algorithm looks through the formulae that are already on the branch. If a formula is found contradicting the newly added formula then a clash has been found and backtracking will be performed. We say this procedure uses backward reasoning because all of the formulae being checked already exists on the branch. In contrast, forward checking techniques look through the formulae that have not been added to the branch yet.

The reason why we are investigating a new technique instead of backward checking is that all of the tableau-based procedures based on backward checking suffer from an important problem. The decisions made at earlier branch points have greater impact on the system's performance. In other words, earlier decisions are more costly than later ones. It often happens that after a lot of proof steps have been performed a clash is found and it is caused by a few steps made at the earlier stages. If a clash can be spotted earlier, certain unnecessary inference steps might be saved. Therefore efficiency is likely to be improved. Since we are working on non-clausal formulae, unit propagation is in general not very helpful to identify clashes earlier.

Figure 5.1 shows why finding clashes earlier can improve the performance of proof search. In the derivation tree shown in Figure 5.1, formulae f_1 to f_4 are the input formulae. Suppose our default strategy is to always consider the left-most case of a branching point first. Expanding formula f_1, we select a and add it to the branch. Then f_2 is expanded similarly and b is added to the branch. So far nothing unusual has happened. There is no clue if we have made a wrong

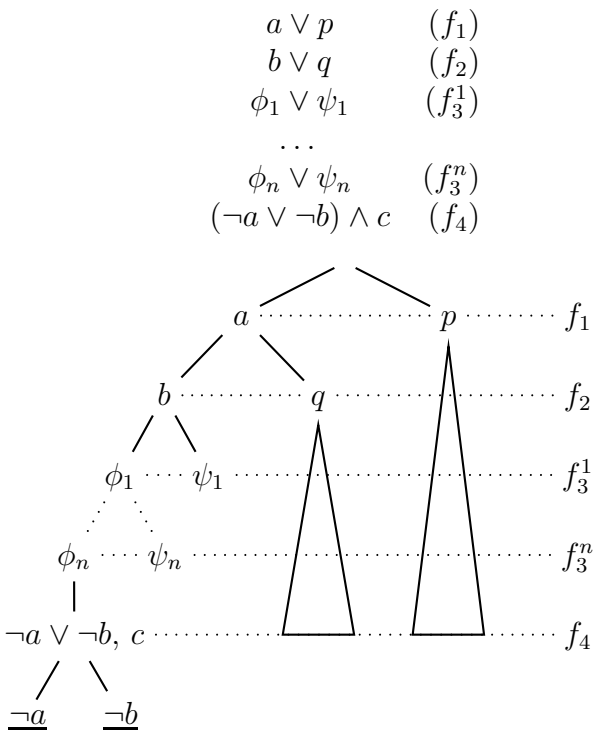


Figure 5.1: Example of derivation tree

decision. In the following steps, we carry on expanding formulae f_3^1 to f_3^n. These formulae can be very complicated and can have a large search space. Finally, we reach f_4 and expanding it we get two new formulae $\neg a \vee \neg b$ and c. Immediately we find that both of the two disjuncts of the formula $\neg a \vee \neg b$ are blocked. To solve these clashes, we have to change the decision we made when we expanding f_1 and f_2. If we could have spotted these clashes right after f_2 is expanded, we would have been able to save a lot of time spent on expanding the formulae f_3^1 to f_3^n. This can be achieved by using forward checking.

The way that forward checking works can be better explained by showing the inference process using formula tree graphs. Figure 5.2 shows a formula tree for the input formulae in Figure 5.1.

In a formula tree, every node of the tree is either an operator or a propositional literal. A formula is represented by a tree, whose root node is the formula's top operator. A formula needs to be transformed into negation normal form before it is presented with a formula tree. Hence negation occurs only immediately in front of atomic formulae. A literal is presented in one node no matter it is positive or negative.

The expansion in a tableau can be compactly captured by using markings of

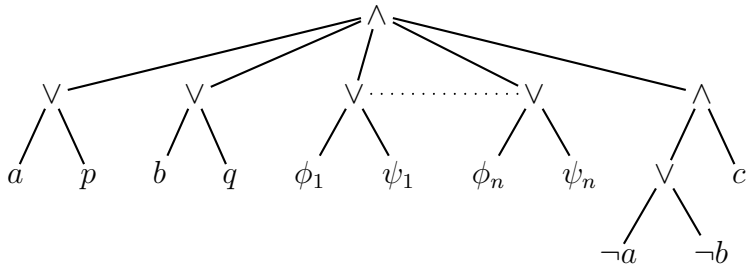


Figure 5.2: Example of forward checking in propositional logic: step 1

the formula tree that represents the input problem. We use two types of markings. A formula is underlined if it has been added to the branch. If it is believed that a formula should not be added to the branch, we say that it has been *rejected*. We indicate that a formula has been rejected by crossing it off using a back slash.

We can think of the underlined formulae as being true in the current branch and consider the formulae crossed off as being false in the current branch.

If we reconsider the previous example, the application of the α rule means that the formulae f_1 to f_4 in Figure 5.1 are added to the branch. This is indicated by underlining these formulae in the formula tree in Figure 5.3.

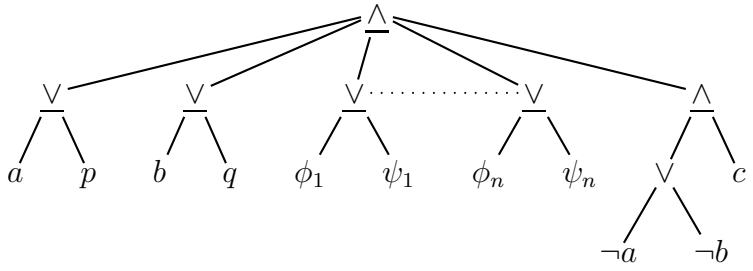


Figure 5.3: Example of forward checking in propositional logic: step 2

As shown in Figure 5.4, we now select the left-most disjunction (f_1 in Figure 5.1) to expand first and put a into the branch. Now we do forward checking. It is found that there is formula $\neg a$ that contradicts a. Therefore we block $\neg a$ which is indicated by crossing $\neg a$ off.

In the next step we expand the second disjunction (f_2 in Figure 5.1) from the left and put b into the branch. As in the last step, forward checking blocks b's complementary formula $\neg b$. In order to deal with non-clausal formulae, our forward checking algorithm propagates the blocking from the bottom of the formula tree to upper levels. Now we find that for the formula $\neg a \vee \neg b$, both of its two disjuncts are blocked. Hence $\neg a \vee \neg b$ is also blocked. This formula's immediate parent formula is a conjunction. The blockage can be propagated upward again.

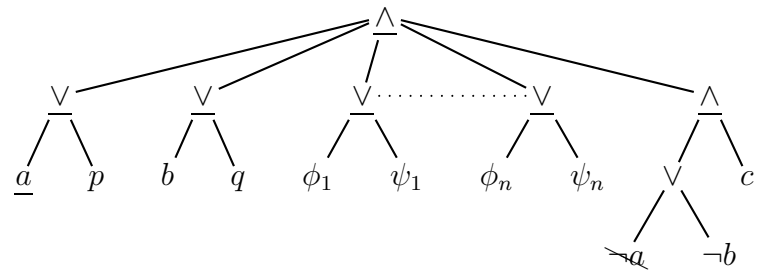


Figure 5.4: Example of forward checking in propositional logic: step 3

Therefore formula $(\neg a \vee \neg b) \wedge c$ is blocked. But this formula is actually already underlined which means that it is true. At this point, we realize that there is a contradiction. This triggers backtracking to be performed.

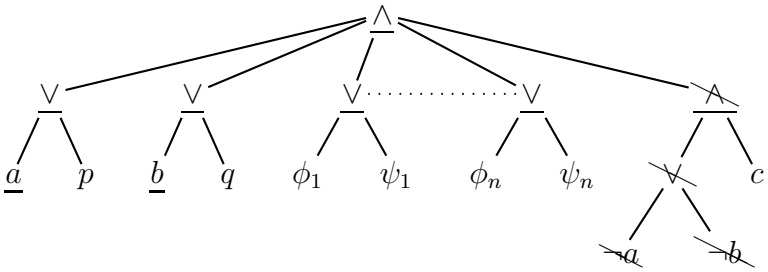


Figure 5.5: Example of forward checking in propositional logic: step 4

This example illustrates that compared with traditional tableau inference steps, forward checking can help tableau-based procedures reduce the search space. Forward checking actively looks for the formulae that are complementary to a newly added formula. By propagating the blockage upward, we are able to find situations where backtracking is needed earlier. As a consequence many unnecessary inference steps can be avoided. A formal description of forward checking algorithm will be given in the next section.

In the examples above, we have given an informal description of how forward checking works. It often happens that forward checking needs to go through the formula tree looking for complementary formulae whenever a new formula is added to the branch. The cost of this procedure is linear in the size of the input formula. For real world problems, this overhead may degrade the performance substantially. There is a way to optimise forward checking using the fact that all of the input formulae are already known before proof search starts. We can look through the input formulae, find the complementary formulae and build connections among them. Thus later, when we do forward checking, we only need to follow these precomputed connections instead of the whole formula tree.

5.2 Forward checking for propositional logic

In propositional logic, precomputed connections are built between complementary formulae. Here we give the definition of two formulae being complementary:

Definition 5.2.1. *ϕ and ψ are* complementary *iff*

i. $\phi = \neg\psi$, if ψ is not a negated formula, or

ii. $\phi = \theta$, if $\psi = \neg\theta$.

In Definition 5.2.1, '=' is basically syntactic equality. Some preprocessing techniques can be used to help syntactic matching. These techniques include lexical normalization. As an inverse process of negation normalization, lexical normalization brings the negations outside and put them on parent formulae. For example, given two formulae $f_1 = a \wedge b \wedge c$ and $f_2 = \neg a \vee \neg b$. Lexical normalization transforms f_2 to $\neg(a \wedge b)$. Then it is recognized that the first subformula of f_1 $(a \wedge b)$ is complementary to f_2. Hence connection is built between the subformula of f_1 $(a \wedge b)$ and f_2.

The forward checking algorithm for propositional logic is described in Algorithm 9. The input to the algorithm, ν, is the formula newly added to the branch. $comp_\nu$ denotes the precomputed list of the formulae that are complementary to ν. In the algorithm, for each formula in $comp_\nu$ the procedure performing forward checking is called. If no contradictions are found during forward checking, the procedure returns true. Otherwise, the procedure returns false and backtracking will be performed afterwards in this case.

Algorithm 9 forward_checking_loop(ν)

Input: ν: the newly added formula
Output: Returns false if a contradiction is found otherwise returns true.

```
1: for all φ such that φ ∈ comp_ν do
2:   if forward_checking(φ) = false then
3:     return false
4:   end if
5: end for
6: return true
```

The forward checking algorithm is shown in Algorithm 10. The algorithm can be written in a recursive way, but we designed it as a non-recursive algorithm

Algorithm 10 forward_checking(f)

Input: f: the formula to be checked.
Output: Returns false if a contradiction is found otherwise returns true

```
1: loop
2:   if f has been added to the branch then
3:     return false
4:   end if
5:   block(f)
6:   pf ← parent(f)
7:   if pf is an 'and' formula then
8:     f ← pf
9:   else if pf is an 'or' formula then
10:    if all of pf's disjuncts have been blocked then
11:      f ← pf
12:    else
13:      return true
14:    end if
15:  end if
16: end loop
```

for the sake of performance. Most of the algorithm is inside an infinite loop. At the beginning of the loop, we check if the formula f which is to be blocked is on the branch. If the formula is blocked, the algorithm returns and tells the caller that a contradiction has been found. If f is not in the branch yet, we set it to be blocked. We get the parent formula of f using the function $parent(f)$. There are two cases in which we make f be its parent formula and perform another iteration. One is when this parent formula is a conjunction. Another is when this parent formula is a disjunction and all of its disjuncts have been blocked. In other cases the algorithm exits from the loop and returns.

5.3 Precomputed connections in modal logic

The precomputed connections used in forward checking for propositional logic need to be enhanced to handle modal logic. In modal logic, two subformulae ψ and $\neg\psi$ occurring in a formula are not necessarily complementary. It depends on their modality, or in other words, on the worlds in which these formulae hold. If ψ and $\neg\psi$ are true in different worlds, i.e. we have $w_1 : \psi$ and $w_2 : \neg\psi$ in the current branch where $w_1 \neq w_2$, the formulae do not conflict with each other.

This means we need to develop an algorithm that is able to find complementary formulae that can possibly share the same world and build connections between them.

To do forward checking in modal logic, a new notion of *complement-related* is defined by Definition 5.3.1. The idea of this notion is to define the relation between two modal formulae that can cause a clash in a tableau derivation. The definition is supported by Lemma 5.3.1 and 5.3.2, which can be easily proven in modal logic.

Definition 5.3.1. *Formulae ϕ and ψ are* complement-related *iff*

i. *ϕ and ψ are complementary according to Definition 5.2.1,*

ii. *if $\phi = \Diamond\theta_1$, $\psi = \Box\theta_2$, and θ_1 and θ_2 are complement-related, or*

iii. *if $\phi = \Box\theta_1$, $\psi = \Box\theta_2$, and θ_1 and θ_2 are complement-related.*

Lemma 5.3.1. *If ϕ and ψ are complementary according to Definition 5.2.1 then*

$$\Diamond\phi \wedge \Box\psi \vDash \bot.$$

Lemma 5.3.2. *If ϕ and ψ are complementary according to Definition 5.2.1 then*

$$\Diamond\top \wedge \Box\phi \wedge \Box\psi \vDash \bot.$$

In Lemma 5.3.2, $\Diamond\top$ must be true to make the whole statement correct. However, we do not need to confirm the truth of $\Diamond\top$ while building precomputed connections. It is guaranteed by the expansion rule for box operators. Box formulae won't be expanded unless there already exists an accessible world.

In order to find the complement-related modal formulae, we use the notion of *unifiable* modal formulae. To decide if two formulae are unifiable, we compute a formula's *prefix*. Informally we say two modal formulae are unifiable if their prefixes can be unified.

The prefix of a formula f consists of all of the modal operators on the path from f to the top-most operator (the root of the formula tree). The formal

description of how a prefix is generated is given in Algorithm 11. The algorithm assumes the input formula is in negation normal form. In Algorithm 11, P denotes a stack. Hence whenever a new item pushed in, it is located at the beginning of P. When the algorithm terminates, P contains the prefix of formula f.

Algorithm 11 $prefix(f)$

Input: f:a formula
Output: P: a list containing the prefix

```
1: loop
2:    pf ← parent(f)
3:    if pf does not exist then
4:       return P
5:    end if
6:    if pf is a 'box' formula then
7:       push pf into P
8:    else if pf is a 'diamond' formula then
9:       push pf into P
10:   end if
11:   f ← pf
12: end loop
```

For example, consider the formula

$$f = p \wedge (\Box\Box q \vee \Box p \vee \Diamond\neg r) \wedge \neg(\Box p \wedge \Box\neg r \wedge \Box\Diamond q) \tag{5.3.1}$$

Its negation normal form is

$$f' = p \wedge (\Box\Box q \vee \Box p \vee \Diamond\neg r) \wedge (\Diamond\neg p \vee \Diamond r \vee \Diamond\Box\neg q) \tag{5.3.2}$$

Figure 5.6 shows the formula tree for the negation normal form of formula 5.3.1. We encode f by giving every subformula of f a unique number. B is used to represent a box operator and D represents a diamond operator. The prefixes for the atomic formulae are given as follows:

- $P_2 = \emptyset$
- $P_6 = B_4 B_5$
- $P_8 = B_7$
- $P_{10} = D_9$

- $P_{13} = D_{12}$
- $P_{15} = D_{14}$
- $P_{18} = D_{16}B_{17}$

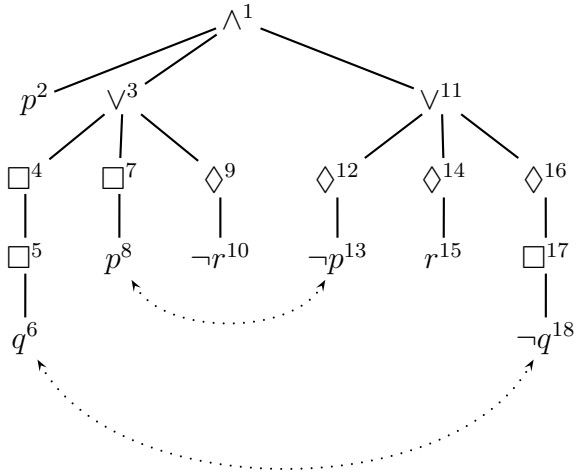


Figure 5.6: Formula tree with connections for (5.3.1)

Given two formulae ϕ and ψ, and their prefixes P_ϕ and P_ψ, we can use Algorithm 12 to determine if ϕ and ψ are complementary in modal logic K. If the algorithm returns true, that means ϕ and ψ are complementary in modal logic K. Otherwise, they are not. At the beginning of the algorithm we make sure that the two formulae ϕ and ψ are negations of each other. Then we look through each item in P_ϕ, named a and its corresponding item in P_ψ, named b. If it is not the case that a and b are both diamond operators and they are not the same formula occurence, a and b are unifiable. In other words, under following circumstances a and b are deemed to be unifiable.

- In a and b, one is a diamond operator another is a box operator.
- a and b are both box operators.
- a and b are the same operator occurence.

Now we can use the Algorithm 12 to find the complementary connections in the modal formula shown in Figure 5.6. Formula 13 ($\neg p$) is the negation of formula 8 (p). The prefix of formula 8 is

$$P_8 = B_7$$

Algorithm 12 Procedure *complementary_K*$(\phi, P_\phi, \psi, P_\psi)$

Input: Formula ϕ and its prefix P_ϕ
Formula ψ and its prefix P_ψ
Output: Returns true if ϕ and ψ are complementary to each other otherwise returns false.

```
1: if φ is not a negation of ψ then
2:   return false
3: end if
4: if the length of P_φ and P_ψ are not the same then
5:   return false
6: end if
7: for n ← 1 to length(P_φ) do
8:   a ← P_φ[n]
9:   b ← P_ψ[n]
10:  if a is a diamond formula then
11:    if b is a diamond formula then
12:      if a ≠ b then
13:        return false
14:      end if
15:    end if
16:  end if
17: end for
18: return true
```

and the prefix of formula 13 is

$$P_{13} = D_{12}.$$

Since the first (the only) item in their prefix are box and diamond formulae respectively, formula 8 and 13 are complementary in modal logic K. Formulae 6 (q) and 18 ($\neg q$) are also complementary. The prefix of formula 6 is

$$P_6 = B_4 B_5,$$

the prefix of formula 18 is

$$P_{18} = D_{16} B_{17}.$$

The first two items in these two prefixes are box and diamond operators respectively and the second ones are both box operators. Therefore according to our algorithm formulae 6 and 18 are also complementary in modal logic K. Let's consider another pair of conflicting formulae 10 ($\neg r$) and 15 (r). The prefix of formula 10 is

$$P_{10} = D_9$$

and the prefix of formula 15 is

$$P_{15} = D_{14}.$$

The items in these two prefixes are both diamond operators but D_9 and D_{14} are different operator occurences. So formulae 10 and 15 are not complementary in modal logic K.

We have introduced complementarity in modal logic K. It is possible to define complementarity in other modal logics. In [OK96], Otten and Kreitz proposed a unification procedure to deal with prefixes in modal logics such as D, T, 4 and S5. The unification procedure includes a set of unification rules. The rules are different for each modal logic. Suppose in a certain kind of modal logic a pair of prefixes are given. If one of them can be transformed into another after applying the unification rules designed for this modal logic, we say that these pairs of prefixes are unifiable in this modal logic. Any pair of conflicting formulae with unifiable prefixes are complementary in this particular modal logic. In [Sch97, Sch98], the prefixes are presented in *path logic*. Unification rules for

expressive modal logics are defined based on this path logic. Other related work was proposed in [Gov95]. In his work, Governatori used a labelled tableau system to handle variants of modal logics.

5.4 Forward checking for modal logic

Forward checking for modal logic is more complicated than for propositional logic. This is reflected in two aspects. First, determining complementarity in modal logics is less straight forward. Computation is needed to test if two prefixes are unifiable. Second, blockage in modal logics not only involves formulae but also involves worlds.

To handle modal logic, we have enhanced the forward checking algorithm described in Algorithm 10. The modal logic version of forward checking is shown in Algorithm 13.

Algorithm 13 $forward\_checking(f, w)$

Input: f: a formula
w: a world
Output: Returns false if a contradiction is found, otherwise returns true.

```
1: loop
2:   if w:f has been added to the branch then
3:     return false
4:   end if
5:   block(f, w)
6:   pf ← parent(f)
7:   if pf is an 'and' formula then
8:     f ← pf
9:   else if pf is an 'or' formula then
10:    if all of pf's disjuncts have been blocked in w then
11:      f ← pf
12:    else
13:      return true
14:    end if
15:  else if pf is a 'box' or 'diamond' formula then
16:    f ← pf
17:    w ← parent_world(w)
18:  end if
19: end loop
```

Comparing Algorithm 13 to the forward checking algorithm for propositional

logic, Algorithm 10, we see the difference is in lines 15 and 16 in Algorithm 13. As we have mentioned, the blocking in modal logic depends on a certain world. When we block a formula f associated with a world w, f may not be true in w but f still can hold in other worlds. Another difference related to the case that when the parent formula is a modal formula, we need to find the predecessor of the current world w in the Kripke model that we have built so far in the derivation. To give an explanation, we suppose f's parent formula is f'. Since f' is either a box formula or a diamond formula, $w : f$ must be the result of applying the ν-rule or the π-rule to $w' : f'$, where w' is a predecessor of w.

We use an example to explain how forward checking works in modal logic. Formula (5.4.3) is used as the input formula. Its formula tree is shown in Figure 5.7. The modally complementary formulae are linked by dotted lines.

$$p \wedge (\Diamond p \vee \Box r) \wedge (\Diamond q \vee \Diamond \neg r) \wedge (\Box \neg p \vee \Box \neg q) \tag{5.4.3}$$

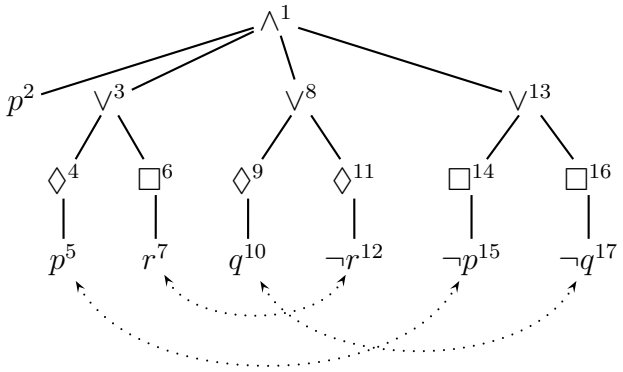


Figure 5.7: Example of forward checking in modal logic: step 1

Figure 5.8 is a compact representation of a branch after a few tableau inference steps. Labels are attached in front of the formulae. These labels consist of sets of worlds. Given a formula f and its label L, f is true in each underlined world in L. f cannot be true (blocked) in each crossed out world in L. At the beginning, an initial world w_0 is created. The top conjunction formula 1 is expanded and its subformulae 2, 3, 8 and 13 are set to be true in w_0. Then we expand the disjunction 3 and select 4. Formula 4 is a diamond formula. The π-rule is applied to formula 4 and a new world w_1 is created and becomes the successor world of w_0. Formula 5 is set to hold in w_1. Since formula 5 is connected with 15, formula 5 and formula 15 cannot both be true in a world. Hence formula 15 is blocked

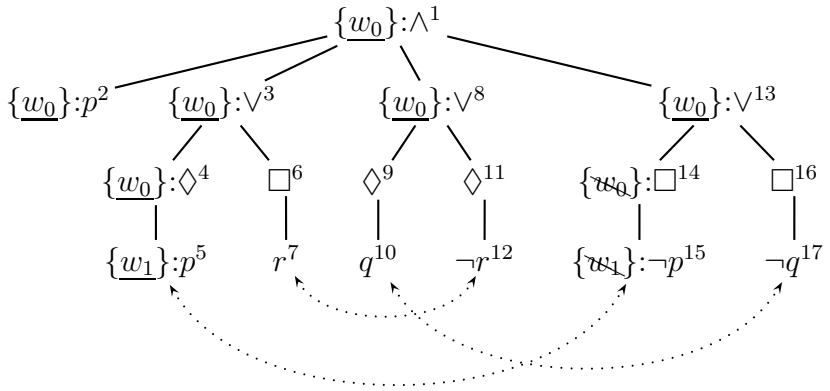


Figure 5.8: Example of forward checking in modal logic: step 2

in world w_1. Now we can propagate the blocking at 15 upwards. The parent formula of formula 15, formula 14, is a box formula. A ν-rule needs to be applied to formula 14 to make formula 15 true in world w_1. According to the ν-rule, in order to block formula 15 in world w_1, formula 14 needs to be blocked in the predecessor world of w_1, which is w_0.

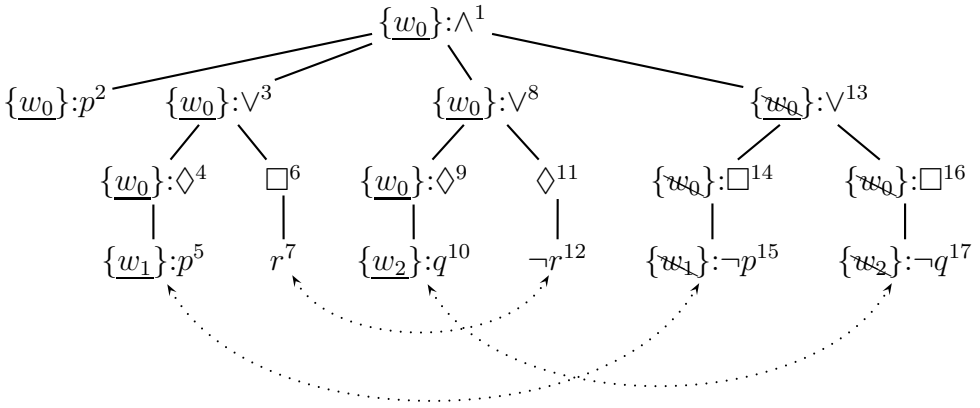


Figure 5.9: Example of forward checking in modal logic: step 3

Figure 5.9 represents the continuation of the derivation. Formula 8 has been expanded. We set the disjunct 9 to true. Again, since 9 is a diamond formula, we create another successor world of w_0, named w_2. Formula 10 is set to be true in world w_2. There is a connection between formula 10 and 17. Therefore forward blocking blocks the formula 17 in w_2. This blocking is to be propagated upwards. As formula 17's parent formula is the box formula 16, only the application of a ν-rule can make formula 17 true in world w_2. And the premise of the ν-rule requires that formula 16 is true in world w_2's predecessor world, which is w_0.

Hence to block formula 17 in w_2, forward checking blocks formula 16 in w_0. Both of the two disjuncts of formula 13 are blocked in world w_0. As a consequence, formula 13 is blocked in w_0. But formula 13 is already true in world w_0 (which is indicated by underlining). Thus a contradiction is found and backtracking is needed.

5.5 Forward checking and dynamic backtracking

Dynamic backtracking is a very important optimisation technique. In Chapter 4 we have shown that it can significantly improve the performance of a theorem prover. So a question comes up: Can forward checking work with dynamic backtracking so that we can combine the strength of these techniques. We will show the answer is yes.

Dynamic backtracking needs to have facilities to record the reasons of earlier failures. That information contains two parts: one is the formula ϕ that has been proved cannot be true, another is the formula instances that caused the rejection of ϕ. Each formula instance also has two parts: one is the formula ψ, another is the world in which ψ holds.

In the forward checking algorithm we introduced above, the blocked formula is not removed from its parent formula, instead, we add some extra information which indicates that this formula has been blocked. For propositional logic, the information is simply a switch showing if the formula is blocked. For modal logic, the information needs to contain the worlds in which the formula is blocked. To make forward checking work with dynamic backtracking, we need to extend the information recorded further. It needs to include not only a world but also the formula that causes this formula to be blocked.

A new structure called a blocking explanation is used to make forward checking work with dynamic backtracking. Suppose the blocking explanation of a formula f is $(w: f_1, f_2, ..., f_n)$, where w is the world in which the formula is blocked and f_1 to f_n are the formulae that block f.

The dynamic backtracking algorithm we introduced in Chapter 4 is for traditional backward consistency checking. Most parts of the algorithm do not need to be changed to work with forward checking. Only the function to find the complementary pairs needs to be modified. Suppose a contradiction is found in

formula f. In order to get the explanations for this contradiction, we need to collect all of the blocking explanations for every subformula of f.

Algorithm 14 *blocking_explanation*(f, w)

Input: f: formula that has been blocked.
w: the world in which f is blocked.
Output: E: a set of formulae that contains the blocking explanations *of* f.

```
1: E ← ∅
2: if f is an 'or' formula then
3:    for each subformula s of f do
4:       E ← E ∪ blocking_explanation(s, w)
5:    end for
6: else if f is a 'and' formula then
7:    for each subformula s of f do
8:       if s is blocked then
9:          E ← E ∪ blocking_explanation(s, w)
10:      end if
11:   end for
12: else if f is a 'diamond' or 'box formula then
13:   s ← f's subformula
14:   w' ← w's child world that has been blocked
15:   E ← blocking_explanation(s, w')
16: else if f is a atomic formula then
17:   E ← EXP(w)
18: end if
19: return E
```

Figure 14 shows a recursive algorithm that goes through the subformulae to collect the blocking explanations. If formula f is a disjunctive formula, all of the blocking explanations of its subformulae need to be collected. If f is a conjunctive formula, only the explanations for one of its blocked subformulae are needed. For diamond and box formulae, we need to change the world w to its blocked child world and collect the explanations for that world.

Again, we use an example to demonstrate how forward checking works with dynamic backtracking. Given a modal formula f, its formula tree with connections is shown in Figure 5.10. It is worth noting that ϕ and ψ can represent formulae of arbitrary complexity.

After a few inference steps, the result can be seen in Figure 5.11. w_0 is the initial world. Formula 1 and its subformulae are accepted and set to be true in w_0. Formula 3 is expanded first. A new world w_1 is created. Formula 4 is set to

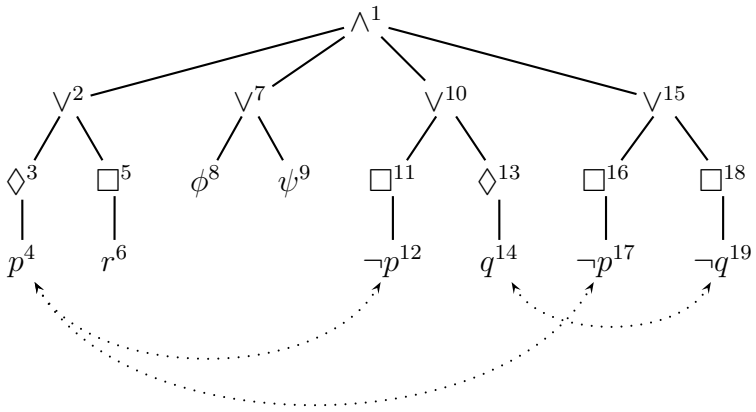


Figure 5.10: Example of forward checking with dynamic backtracking: step 1

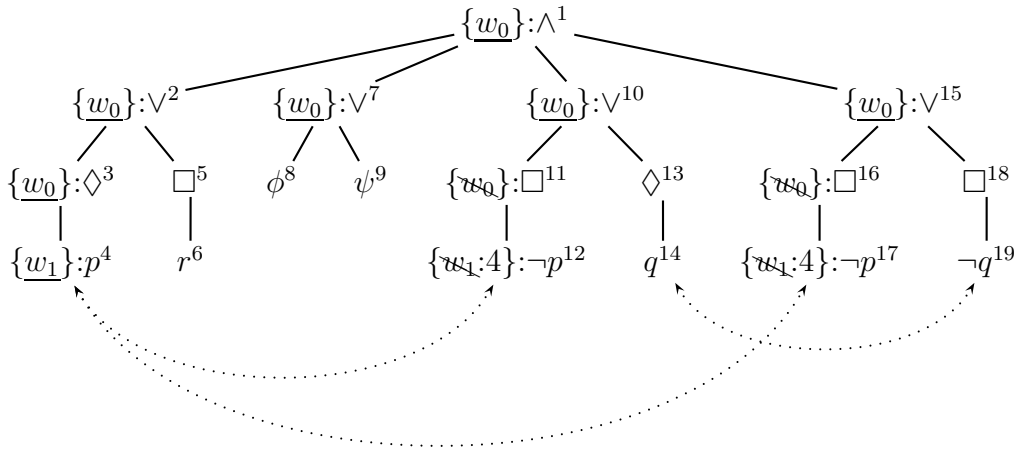


Figure 5.11: Example of forward checking with dynamic backtracking: step 2

be true in w_1. Since formulae 12 and 17 are complementary to formula 4, we set 12 and 17 to be blocked in w_1 and put 4 into the explanation.

As shown in Figure 5.12, formula 8 was accepted in world w_0. Due to the blocking at formula 11, formula 13 is the only available disjunct for the disjunction 10. Since 13 is a diamond formula, world w_2 is generated and formula 14 is set to be true in w_2. Formula 19 is complementary to 14. Therefore formula 19 is blocked in w_2 and formula 18 is blocked in w_2's parent world w_0. Because both formula 16 and 18 are blocked in w_0, formula 15 is blocked in w_0. But 15 has been accepted in w_0. Hence a contradiction is found. Now we can use Algorithm 14 to find out where to backtrack to. Because formula 15 is a disjunction, we need to look through both of its subformulae 16 and 18. They are box formulae. Formula 16's subformula 17 is blocked in world w_1 by formula 4. Formula 18's

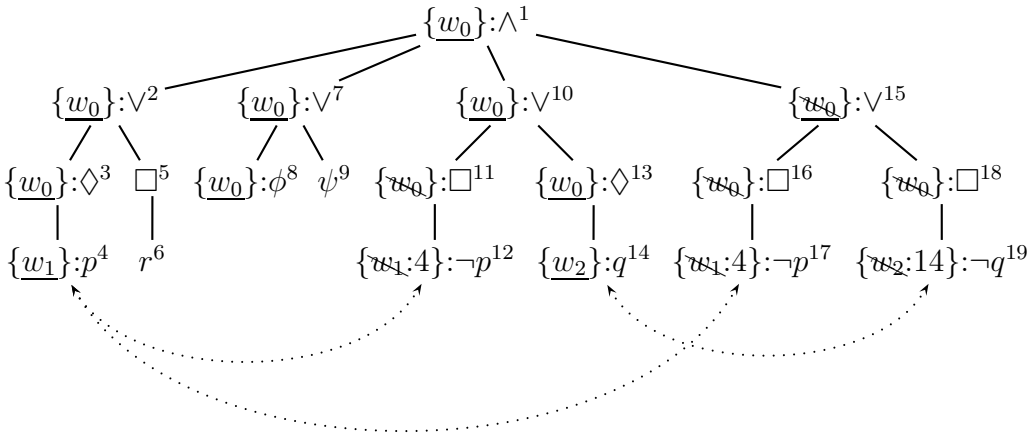


Figure 5.12: Example of forward checking with dynamic backtracking: step 3

subformula 19 is blocked in world w_2 by formula 14. So far the explanation set contains formula 4 and 14. Their parent disjunctions are formulae 2 and 10 respectively. According to our algorithm of dynamic backtracking, we look into formula 10, which is the more recent one of these two formulae. Formula 10 has two subformulae, 11 and 13. Currently formula 13 is accepted. The other one, formula 11 has been blocked. Again, we visit formulae 11 and its subformula 12 and find out the blocking explanation is formula 4. Then we use formula 4 to replace formula 14 in the explanation set. So far the only item left in the explanation set is formula 4 and it has an unexplored disjunct. Therefore we need to do backtracking on formula 4.

5.6 Releasing blocked formulae

In the previous section, we described a way of combining dynamic backtracking with forward checking. We focused on the new structure, the blocking explanation, and how a blocking explanation is used in the hybrid algorithm. Now we introduce another important procedure, releasing blocked formulae. When a formula is backtracked, it is to be removed from the current branch and the formulae that it has blocked need to be released.

Algorithm 15 describes how a blocked formula is released when the formula that caused this blocking is removed due to backtracking. Suppose $w : \phi$ is the formula we want to backtrack on. For every formula that has been blocked by $w : \phi$, we invoke the procedure *release_blockage*. By releasing the formula f in

Algorithm 15 *release_blockage*(f, w)

Input: f: formula to be released. w: the world in which f is blocked.
Output:

```
1: loop
2:   if f is blocked in w then
3:     release(f, w)
4:   else
5:     return
6:   end if
7:   pf ← parent(f)
8:   if pf is an 'or' formula then
9:     f ← pf
10:  else if pf is an 'and' formula then
11:    if none of pf's conjuncts are blocked in w then
12:      f ← pf
13:    else
14:      return
15:    end if
16:  else if pf is a 'box' or 'diamond' formula then
17:    f ← pf
18:    w ← parent_world(w)
19:  end if
20: end loop
```

w, we remove w and ϕ from f's blockage status. Then we look into f's parent formula pf. If pf is a disjunction, when one of its disjuncts is released, pf needs to be released as well. If pf is a conjunction, when f is released and no other conjunct of pf is blocked, pf has to be released. If pf is a box or diamond formula, w needs to be replaced by its parent world. We traverse the formula tree bottom up repeating the above procedure until we find a formula that is not blocked.

To demonstrate how this algorithm works, we continue with the example used in the last section. In the last section, we have reached the stage where backtracking is needed on formula 4.

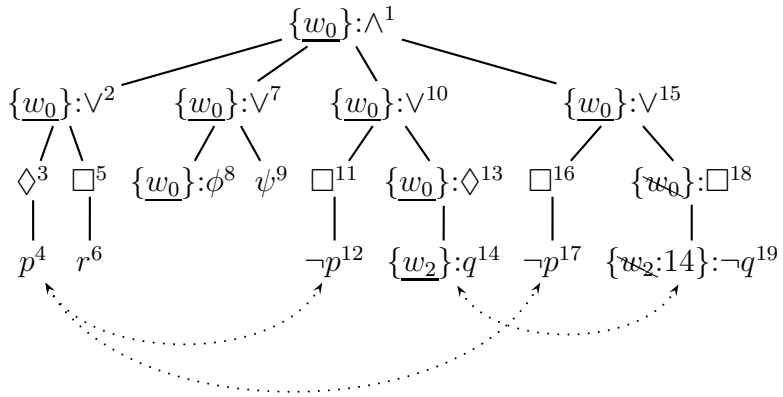


Figure 5.13: Example of releasing blocked formula

The result after formula 4 being backtracked is shown in Figure 5.13. Formula 4 is no longer true in world w_1. In Figure 5.12, two formulae are blocked by formula 4 in w_1. They are formulae 12 and 17. First we release formula 12 in w_1. By releasing it, we remove the label attached to formula 12. According to Algorithm 15, formula 12's parent formula 11 is visited. Since it is a box formula and w_1's parent world is w_0, formula 11 is released in w_0. Then we move to formula 17. We release formula 17 in w_1. Box formula 16 is the parent formula of formula 17. Hence formula 16 is released in world w_0. Formula 16's parent formula is disjunctive formula 15. Formula 15 is also released in w_0.

5.7 Forward implication

So far we have discussed forward checking algorithms. The idea of forward checking is to use the complementary formulae to identify clashes earlier and avoid some unnecessary work. Apart from forward checking, another mechanism for forward reasoning is forward implication. The idea of forward implication is similar to subsumption deletion including backward subsumption and forward subsumption in resolution algorithms for clausal formulae [GL85, Vor95]. Subsumption deletion detects subsumption relation between the newly derived clause and the existing clauses and removes the subsumed clause to avoid redundancies. Forward implication is based on implication relationships. Every time a new formula is derived, forward implication checks if any formula that is waiting to be expanded is implied by the derived formulae on the current branch. If such a formula exists, it will be spared from future expansion.

It is important to detect the implication relations efficiently, because this operation can be expensive and can become a bottleneck of the system.

For this purpose, precomputed connections are also used in forward implication as they are in forward checking. But they are not intended to connect complementary formulae but to reflect implication relations. To build these connections, we first generate the prefixes using the same algorithm as in forward checking. Once the prefixes are built, we can check if there is any implication relation between two specific formulae.

In our system, a bottom-up approach is used for implication detection. To find which formulae are implied by a set of derived formulae, we start from formulae syntactically identical to the derived formulae. These formulae are located at the leaves of the formula tree. Travelling upwards along the tree, forward implication checks if their parent formulae are in an implication relationship as well. Suppose ϕ and ψ are two different occurrences of syntactically identical formulae. Then clearly $\phi \rightarrow \psi$ and $\psi \rightarrow \phi$ and we add so-called *implication connections* between ϕ and ψ, in both directions. In the next step, the parent formulae of the identical formulae are visited to see if they are in implication relationships as well. This procedure is based on the properties that:

$$\phi \rightarrow \psi \vDash \phi \rightarrow (\psi \vee \delta)$$

and

$$(\phi_1 \rightarrow \psi_1) \wedge (\phi_2 \rightarrow \psi_2) \vDash \phi_1 \wedge \phi_2 \rightarrow (\psi_1 \wedge \psi_2)$$

We demonstrate forward implication by proving the satisfiability of a formula given in Figure 5.14.

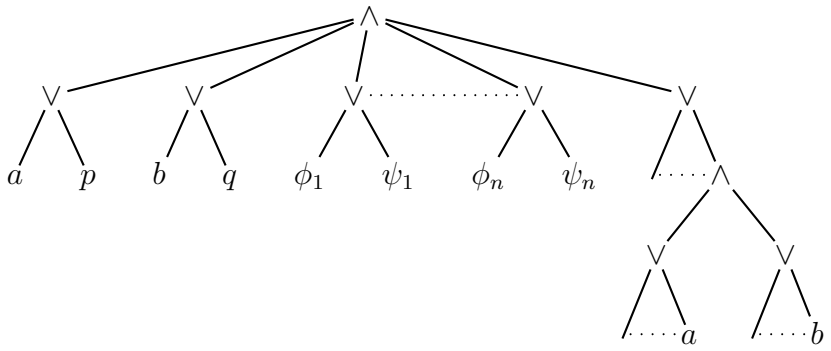


Figure 5.14: Example of forward implication: step 1

The first few steps are shown in Figure 5.15. As before, the top conjunction is expanded first. Using the default strategy, the β rule is applied to the left-most disjunction and a is selected and added to the branch. There is another positive occurrence of a on the formula tree. This situation is recognized by forward implication through the implication connections and the second occurrence of a is set to be implied. Its parent formula is a disjunctive formula. Therefore the disjunction is also implied by the first occurrence of a. Wave markers are used to indicate the formulae having been implied in Figure 5.15. Implied formulae are ignored in further expansions.

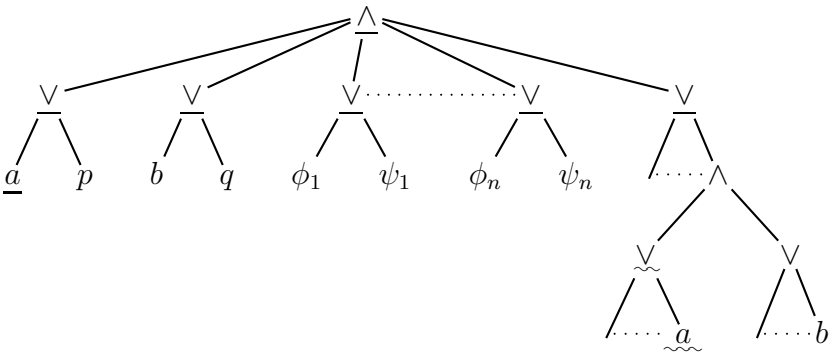


Figure 5.15: Example of forward implication: step 2

As shown in Figure 5.16, next we expand the second left-most disjunction and put b into the branch. Again, there is another positive occurrence of b in the

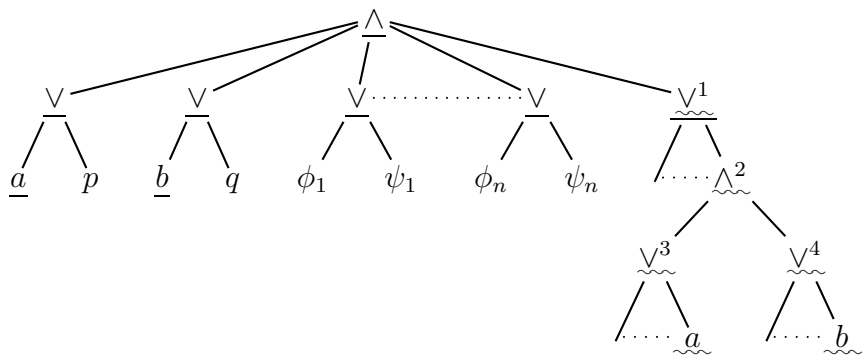


Figure 5.16: Example of forward implication: step 3

formula tree. The second occurrence of b and its parent disjunction (formula 4) are implied. Then because both formulae 3 and 4 are implied, the conjunction of the two, i.e. formula 2, is also implied. This means formula 2 is redundant and so is formula 1 because it is a disjunction. Formula 1 is now marked with an underline and a wave which indicates that it is both accepted and implied at the same time.

When a formula is set to be implied, the tableau-based procedure can ignore it even though it has not been expanded. As we can see from the example in Figure 5.16, formula 1 has been added to the branch. Since it is also implied, we do not need to expand it anymore. In that example, formula 1 can be very expensive to handle. Hence forward implication can save considerable search space.

The algorithm for forward implication is shown in Algorithm 16. The input of this algorithm is the formula to be implied. There is no output for this algorithm. It uses certain rules to check if the input formula and its parent formulae should be implied. If so, the algorithm sets the formulae to be implied and these formulae will be ignored in future formula expansion. Like forward checking, forward implication has been realised in iterative form instead of as a recursive solution. At the beginning of the loop, formula f is set to be implied. If f is in the branch already, the algorithm returns. Otherwise, we get f's parent formula. In two cases, the algorithm makes f be its parent formula and starts another iteration. One case is when the parent formula is a disjunction. The other case is when the parent formula is a conjunction and all of its conjuncts have been implied.

Algorithm 16 $forward\_implication(f)$

Input: f: the formula to be implied.
Output:

```
1: loop
2:    set f to be implied
3:    if f has been added to the branch then
4:       return
5:    end if
6:    pf ← parent(f)
7:    if pf is an 'or' formula then
8:       f ← pf
9:    else if pf is an 'and' formula then
10:      if all of pf's conjuncts have been implied then
11:         f ← pf
12:      else
13:         return
14:      end if
15:   end if
16: end loop
```

5.8 Forward implication for modal logic

Like forward checking, forward implication can be used in modal logic as well. The adaptation of forward implication from propositional logic version to modal logic version is similar to the forward checking case.

Definition 5.8.1 below defines the notion of *implication-related*. If two formulae are implication-related, there is some level of implication relationship between these two formulae. The property of being implication-related is supported by Lemma 5.8.1.

Definition 5.8.1. *Formula ϕ and ψ are implication-related iff*

i. $\phi \rightarrow \psi$, *or*

ii. *if* $\phi = P_1\theta_1$, $\psi = P_2\theta_2$, *and* θ_1 *and* θ_2 *are implication-related where* P_1 *and* $P_2 \in \{\Box, \Diamond\}$.

Lemma 5.8.1 below can be easily proven in modal logic.

Lemma 5.8.1. *If* $\vDash \phi \rightarrow \psi$ *then*

i. $\vDash \Diamond\phi \rightarrow \Diamond\psi$

ii. $\vDash \Box\phi \rightarrow \Box\psi$

iii. $\vDash \Diamond\top \wedge \Box\phi \rightarrow \Diamond\psi$

This theorem describes how implication relation works with modal operators. It is worth noting that the relation between $\Diamond\phi$ and $\Box\psi$ is not mentioned in the lemma. It is because when $\phi \rightarrow \psi$, it is not true that $\Diamond\phi \rightarrow \Box\psi$. Somehow, there is still some level of implication relationship between $\Diamond\phi$ and $\Box\psi$ because these two formulae are unifiable. But this relationship is only restricted to the world generated while expanding $\Diamond\phi$.

Precomputed connections are built between implication-related formulae. Forward implication for modal logic accesses the formulae that have precomputed connections with the newly derived formula and goes upwards towards the root of the formula tree to check if the parent formulae are implied. The modal logic forward implication algorithm needs to deal with not only the propositional logic operators but also the modal logic operators.

Algorithm 17 *implication_K*$(\phi, P_\phi, \psi, P_\psi)$

Input: Formula ϕ and its prefix P_ϕ
Formula ψ and its prefix P_ψ
Output: Returns true if $P_\phi\phi$ and $P_\psi\psi$ are implication-related otherwise returns false.

```
1: if φ is not implication-related to ψ then
2:    return false
3: end if
4: if the length of P_φ and P_ψ are not the same then
5:    return false
6: end if
7: return true
```

Given two formulae ϕ and ψ, and their prefixes P_ϕ and P_ψ, we can use Algorithm 17 to determine if $P_\phi\phi$ and $P_\psi\psi$ are implication-related in modal logic K. If the algorithm returns true, that means $P_\phi\phi$ and $P_\psi\psi$ are implication-related in modal logic K. Otherwise, they are not implication-related. At the beginning of the algorithm we test whether the two formulae ϕ and ψ are implication-related. Then we test whether the two prefixes have the same number of modalities. If that is true, then according to Definition 5.8.1 $P_\phi\phi$ and $P_\psi\psi$ are implication-related.

Now we can use the algorithm described above to build the implication connections in the modal formula shown in Figure 5.17. Formula 6 ($\neg q$) is the same

as formula 18 ($\neg q$). The prefix of formula 6 is

$$P_6 = B_4 B_5$$

and the prefix of formula 18 is

$$P_{18} = D_{16} B_{17}.$$

Since they have the same number of modal operators, a implication connection is built between these two formulae. For the same reasons, formulae 8 and 13, formulae 10 and 15 are also connected.

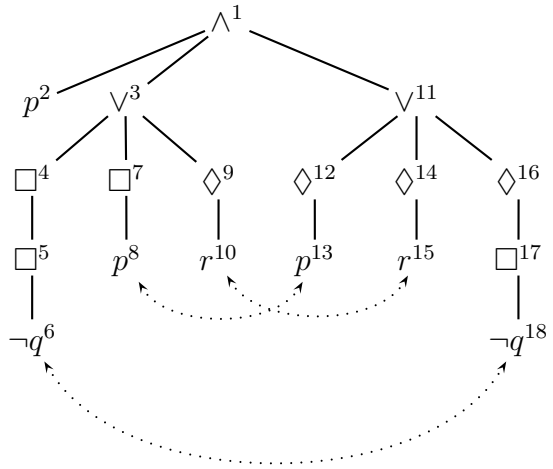


Figure 5.17: Formula tree with implication connections

The implication connections give us a clue about where we should look for implication relations. When we followed the connection and reached a formula, an algorithm will be used to search towards the top of the formula tree to find the parent formulae that are also implied.

The modal logic forward implication algorithm is shown in Algorithm 18. Suppose ϕ is a newly derived formula and it is set to be true in world w. Formula f is connected with ϕ by an implication connection. Taking f, w and the prefix of ϕ as the parameters, the algorithm checks which ones of f's parent formulae are implied by ϕ.

Like forward checking, forward implication is also associated with a certain world. When a formula f, associated with a world w, is implied by ϕ, f is ignored in w but f still needs to be derivable in other worlds.

The algorithm is based on the logical properties we mentioned earlier. When

the parent formula of f is a conjunction or disjunction, the algorithm works in the same way as the propositional logic version of forward implication. When the parent formula of f is a diamond formula, we can conclude that the parent formula pf is implied by ϕ. To find out in which world pf is implied, we need to find the predecessor of the current world w in the Kripke model that we have built so far in the derivation. The existence of $\Diamond\top$ is guaranteed by the expansion rule for box formulae. If the parent formula of f is a box formula, we need to check the corresponding modality in the prefix of ϕ. If it is a box operator, pf is implied. But if it is a diamond operator, pf is not implied by ϕ and the implication search ends.

We use a example to explain how forward implication works in modal logic. The formula tree of the example formula is shown in Figure 5.18. The implication connection has been built between formula 12 and 17.

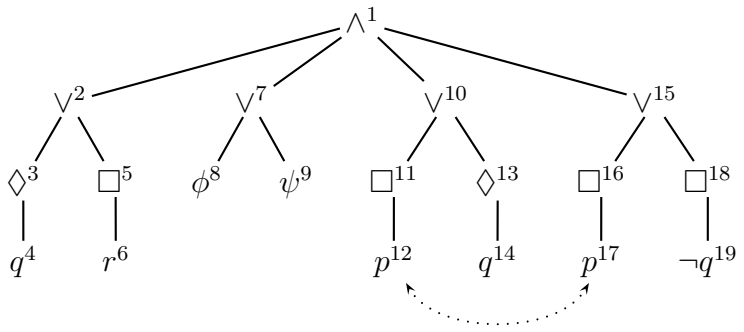


Figure 5.18: Example of forward implication in modal logic: step 1

The top formula is expanded first. A initial world w_0 is created. Suppose the search strategy is from left to right. Formula 2 is expanded next and then formula 3. A successor of world w_0 is created named w_1.

When formula 12 is expanded, formula implication follows the implication connection and implies formula 17 in world w_1. Since 17's parent formula is a box formula and formula 12's parent formula is also a box formula, formula 16 is set to be implied in world w_0. Furthermore, formula 16's parent formula 15 is a disjunction. Therefore formula 15 is also implied. Because formula 15 is already accepted, implication search ends. Formula 15 will be ignored in the future derivation.

Algorithm 18 $forward\_implication(f, w, P_\phi)$

Input: f: the formula to be implied.
w: a world.
P_ϕ: the prefix of ϕ.
Output:

```
1: loop
2:   set n to the index of the last modality in P_φ
3:   set f to be implied in w
4:   if w:f has been added to the branch then
5:     return
6:   end if
7:   pf ← parent(f)
8:   if pf is an 'or' formula then
9:     f ← pf
10:  else if pf is an 'and' formula then
11:    if all of pf's conjuncts have been implied in w then
12:      f ← pf
13:    else
14:      return
15:    end if
16:  else if pf is a 'diamond' formula then
17:    f ← pf
18:    w ← parent_world(w)
19:    n = n − 1
20:  else if pf is a 'box' formula then
21:    if P_φ[n] is a 'box' operator then
22:      f ← pf
23:      w ← parent_world(w)
24:      n = n − 1
25:    else
26:      return
27:    end if
28:  end if
29: end loop
```

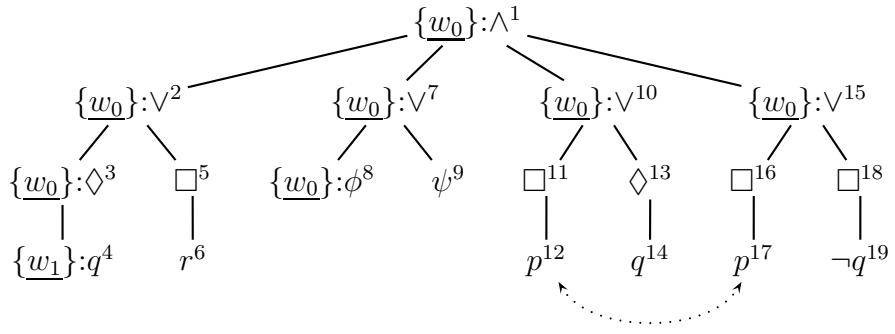


Figure 5.19: Example of forward implication in modal logic: step 2

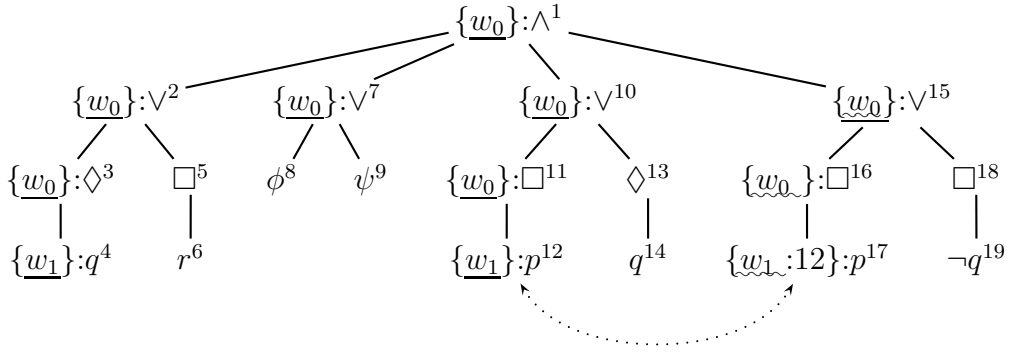


Figure 5.20: Example of forward implication in modal logic: step 3

Chapter 6

The MLTP system

This chapter describes the implementation details of our Modal Logic Theorem Prover, called MLTP. It was developed as a platform to investigate the tableau-based algorithms and optimization techniques for modal logics. Optimization techniques including dynamic backtracking and forward checking have been incorporated into MLTP. MLTP also features generic frameworks and data structures.

The rest of this chapter is organized as follows: Section 6.1 describes the MLTP system. Section 6.2 explains how syntax parsing is performed in MLTP. Section 6.3 introduces a generic framework used in MLTP. Section 6.4 describes the inference loop of MLTP. Section 6.5 presents a higher level language that can be used by users to specify MLTP's behavior. Section 6.6 introduces two kinds of output that MLTP can generate.

6.1 System description

MLTP is implemented in the C++ programming language. Since we care about the efficiency of our prover, it is better to choose a language which can fully exploit the ability of the prover instead of restraining it. Concerning efficiency, C++ is one of the fastest languages available. C++ supports lower level memory operation such as pointers. With this support, we can develop sophisticated data structures, which will improve the performance of the system. At design level, C++ supports object oriented design. This is particularly helpful to the development of a complicated system like a theorem prover. C++ also features a technique called STL (Standard Template Library). It helps in developing

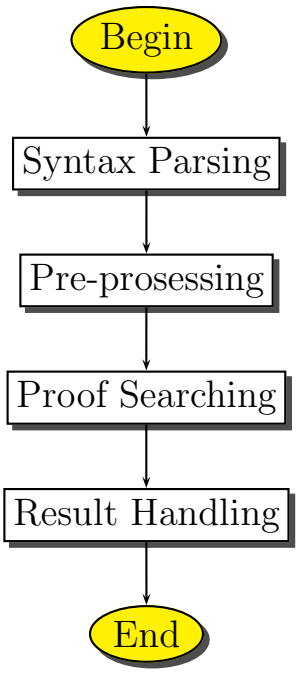


Figure 6.1: System structure of MLTP

generic algorithms for different types of input data. We found it very useful and using it saved considerable work. As far as we know only a few modal logic theorem provers are implemented in C++ or C. FaCT [Hor97] was originally implemented in LISP, recently reimplemented in C++ and renamed as FaCT++ [TH06]. Regarding other well-known provers: DLP [PS98] is implemented in ML, Lotrec [CFG+01] is in Java and Racer [HM01] is in LISP. As a highly profiled language, C++ has been widely used in industry to develop commercial systems. It is therefore interesting to see how suitable C++ is for modal logic system programming.

However, there is a downside to using C++ as the implementation language. The developer needs to take care of lower level operations, such as memory allocation and releasing. This requires the developer to have a good knowledge of how the computer system works. For the same reason, the system is prone to memory access errors. Therefore good programming and debugging skills are essential.

The top-level structure of MLTP is shown in Figure 6.1. The system runs from the top, down through all of the modules. The first module, *syntax parsing*,

takes an input file from the user. The input file contains important information such as the description of the problem, the conjecture formula, the axioms and some other supplementary data. Syntax parsing extracts useful information from the input file and transforms it into a data structure that can be accessed more efficiently by the system. Next, the *pre-processing* module performs operations on the data structure that contains the conjecture formula created by syntax parsing. The operations performed as part of pre-processing usually rewrite the conjecture formula to make the derivation easier and faster without changing the logical conclusion. Then, the resulting formula is passed to the *proof searching* module. That is the actual part that performs the derivations. It uses the labelled tableau algorithm to check if the conjecture formula is satisfiable or not. An *event-handler framework* is adopted to enable the users to specify the behavior of the system. The *result handling* module generates outputs according to the user's demands. These outputs not only include the result of the inference but can also include the time spent on solving this problem and the number of inference steps and backtrackings that have been performed during the derivation. These modules will be explained in more detail in the following sections.

6.2 Syntax parsing

To use the system in order to solve a problem, the user has to provide an input file recognizable by MLTP. The syntax parser of MLTP was developed with the help of Bison. Bison is a general-purpose parser generator that converts an annotated context-free grammar into a parser for that grammar. The syntax of the input file is compatible with that of MSPASS [HS00]. The file contains the various forms of detail about a problem. In the file users can give descriptions of themselves or the problem in natural language. Users can also specify what logic they are working with and the axioms. The constants, predicates and variables need to be defined in that file. The formulae must be given in a prefixed form.

The purpose of formula parsing is to extract the information from the input file and put that information into data structures conveniently accessible by the system. For most of the information, no complicated data structures are needed. Simple strings or integers would be enough. But special attention is needed for the formulae in the input file, given either as axioms or as problem formulae. These formulae and their subformulae will be intensively explored during the

derivation. To make the formula accessing more efficient, we need specialized data structures.

Like most of logic applications, MLTP uses tree-like data structures to store formulae. Every node of the tree contains a symbol and pointers to its subformulae. This tree-like data structure can handle logical operators of arbitrary, finite arity. The nodes of the tree do not have to be allocated next to each other in the memory. They are connected by pointers. In this way, when a formula is to be modified, only a few pointers need changing instead of moving blocks of memory.

6.3 Framework

In recent years, there have been many successful implementations of tableau-based algorithms for modal logics or description logics. At the end of Chapter 2, we have mentioned well-known systems like RACER, FACT and DLP. These systems are heavily optimised and provide very powerful provers for the logics the systems have been designed to deal with. However, too much emphasis on efficiency may cause some other problems to be ignored. Most of the systems mentioned above are restricted to a few fixed logics, and their inference strategies and optimised algorithm are hard-coded in the implementation. This sacrifice of flexibility for efficiency is not optimal for all kinds of problems and all kinds of logics. There are variants of modal and description logics, which differ in their expressive power. Choice of one over another is driven by modelling needs and computational constraints of the application. For example, the logic of actions and plans, which is usually used to formalize agents, is likely to have different semantic and computational properties from the logic of electric circuits. Even with the same logics, different applications may need different inference strategies. We have therefore developed a general framework to facilitate the development of a generic theorem prover.

Nowadays most of the heavily-optimised modal logic theorem prover have certain strategies hard-coded in their system. For example, DLP and FaCT use a strategy that performs modal processing only after all propositional processing is completed for a world [PS00, Hor97]. This strategy is the result of applying conjunction and disjunction expansion rules before possibility and necessity expansion rules. Though in general this strategy leads to a good performance, it is not always the optimal solution for any given problem.

For example, given a formula, $\Box p \wedge \Diamond \neg p \wedge (\phi_1 \vee \varphi_1) \wedge \ldots \wedge (\phi_n \vee \varphi_n)$, expanding the conjunctions and disjunctions first would lead to fruitless expansions of the n disjunctions, $\phi_i \vee \varphi_i$ $(i = 1, \ldots, n)$, before coming to $\Box p \wedge \Diamond \neg p$ and finding a clash. This conservative search strategy has to explore 2^n paths before a conclusion is drown. But if an alternative strategy is taken to apply possibility and necessity rules first, then the result will come out just after exploration of *one* path of 2 steps. Though an optimisation technique called backjumping has been adopted by some of the heavily-optimised systems like FaCT and DLP to improve this inefficient situation [Hor97], at least one path of n steps has to be explored and the extra effort of maintaining a dependency record is needed to utilize this technique.

The example above shows that appropriate rule order can improve the decision procedure to a substantial extent. Unfortunately, there is no existing theory which can guarantee to find the best rule order to use for an arbitrary formula. But this would surely be of great value and deserves more research. Therefore MLTP is implemented in a generic way to serve this purpose.

To achieve generality, our system uses an *event-handler framework*. An event is defined to identify a situation in which something just happened or something is about to happen requiring the system's attention. For example a clash-found event comes up when the system finds a clash. A backtracking event is raised when the system needs to backtrack. A handler is a procedure that is designed to handle a certain event. An event can have an arbitrary number of handlers. A handler has to register before it can be used by the system. The purpose of the registration is to establish a connection between the handler and the event. Registration can be done at anytime, even while the system is running. This is very useful when interaction between the user and the system is needed during the derivation.

A handler can be seen as an independent module that focuses on one task. In this way, the system becomes an integration of different modules. This flexible infrastructure has three advantages. First, it is easy to maintain. Optimized theorem provers are usually very complicated. It is not only because their data structures are sophisticated but also because their process control is not straight forward. Different operations have to be performed in different situations. The addition of optimization methods may increase the complexity to a large extent. It is often a disaster for the developers when something fundamental needs to be changed. From the software engineering point of view, it is always recommended

to divide a system into independent modules. Second, the event-handler framework makes the system highly specifiable. The advantage of our event-handler framework is that for each event additional features and techniques can be added easily. Users can write their own handler. These customized handlers can then be compiled separately and integrated seamlessly into the system. Third, using independent modules can considerably improve the readability of the program. It becomes also easier when modifications need to be done to the system.

The events defined in our system are listed below. The handlers that have been implemented in MLTP are given in brackets.

- Pre-processing (Normalization, Simplification, Pre-computed connections)
- Conjunct selecting (Heuristics)
- Formula expanding (Inference rules)
  - Disjunct selecting (Heuristics)
- Consistency checking (Backward checking, Forward checking)
- Backtracking point searching (Chronological backtracking, Dynamic backtracking)
- Proof Backtracking (Chronological backtracking, Dynamic backtracking)

Pre-processing is a event that is triggered outside the proof search procedure. It works on the input formulae given by the user. The formulae are transformed from one form to another. Some of these transformations are necessary to the inference procedure. For instance, negation normal form is required by some expansion rules. Pre-computed connections are needed by forward checking. Transforming into clausal form is important if the inference procedure only works on clauses. Other transformations like simplification are not crucial to the inference procedure. But they are very helpful to improve the performance of the prover. The handlers that have been implemented in MLTP for pre-processing include negation normal form transformation, simplification and pre-computed connection generation.

Conjunct selecting is to select the next formula to process from all of the non-atomic formulae on the current branch. The event is named conjunct selecting because the logical relation among the formulae on the current branch is

conjunction. To make fetching the unexpanded formulae more efficient, there is a waiting list dedicated to the non-atomic formulae on the current branch. In every iteration of the proof search, a new formula is to be selected from that waiting list according to some strategies. Many techniques can be used here via the handler of this event. For example, various heuristics can be implemented to obtain shorter proofs. The application order of the expansion rules can also be specified by selecting the formulae that are to be handled first. The handlers implemented in MLTP for conjunct selecting include several selection heuristics such as processing minimum sized formulae first, processing deterministic expansion first and processing propositional formulae before modal formulae.

Formula expanding is to expand the formula that has been selected by the conjunct selecting handler. There are four types of formulae in our system for modal logic. They are conjunctions, disjunctions, box formulae and diamond formulae. These formulae are treated differently according to the expansion rules of the tableau algorithm. The expansion of a disjunction is a non-deterministic procedure. A disjunct selecting event is defined for this procedure. The handler of this event can employ heuristic methods to select a disjunct that is likely to lead to a more compact proof. In MLTP the handlers for formula expanding implement the expansion rules of the labelled tableau calculus introduced in Section 2.5.

Disjunct selecting selects the next formula to process from a set of subformulae of a disjunctive formula. Like conjunct selecting, various heuristics can be applied here to minimize the length of the proof. While conjunct selecting attempts to select the formula that has least chance to lead to a satisfiable model, disjunct selecting tries to find the formula that is most likely to lead to a satisfiable model. So far in MLTP, disjunct selecting shares the same handlers as conjunct selecting.

Consistency checking is to make sure that no clashes exist in the current branch. It is performed every time a new formula is added into the current branch. The handler of this event can do consistency checking in a conventional way by looking through the current branch to check if there is any clash caused by the newly added formula. As opposed to forward checking we call it backward checking. Or it can do forward checking by looking through the unexplored search space and rule out formulae that can potentially cause a clash. Backward checking and forward checking have been implemented in MLTP as the handlers for consistency checking.

Backtracking point searching is triggered when a clash is found and backtracking is needed. As we mentioned earlier, backtracking can be separated into two steps. This event is for the first step, which is to find the right point to backtrack to. The handler of this event can apply chronological backtracking by using the immediate previous disjunction as the backtracking point. On the other hand it can apply a dependency directed backtracking algorithm by choosing the disjunction that has caused that clash as the backtracking point. Chronological backtracking and dynamic backtracking have been implemented in MLTP as the handlers for backtracking point searching.

Proof backtracking is the next step following backtracking point searching. When the backtracking point is located by the backtracking point searching handler, this event is triggered to perform backtracking. The handlers implemented in MLTP for this event can be categorized into two types. One is to undo all of the proof steps between the latest step and the backtracking point. We call this chronological proof backtracking. Another is to only undo the formula at the backtracking point. This formula's dependent formulae are also backtracked. We call it dependency directed proof backtracking. Different combinations of handlers for backtracking point searching and proof backtracking can result in three kinds of backtracking technique. They are chronological backtracking, dependency directed backtracking and dynamic backtracking. Chronological backtracking is the result of using chronological backtracking point searching with chronological proof backtracking. Dependency directed backtracking is obtained by using dependency directed backtracking together with chronological proof backtracking. And dynamic backtracking is dependency directed backtracking plus dependency directed proof backtracking.

6.4 The inference loop

Having introduced all of the events, we now introduce the inference loop of the tableau inference procedure. The inference loop is a critical part of our theorem proving system. It takes the pre-processed input formula and performs proof search until the search algorithm terminates.

The infrastructure of the inference loop is shown in Figure 6.2. After being pre-processed, the input formula enters the inference loop and is observed to be unexpanded. Conjunct selecting selects an unexpanded formula for processing in

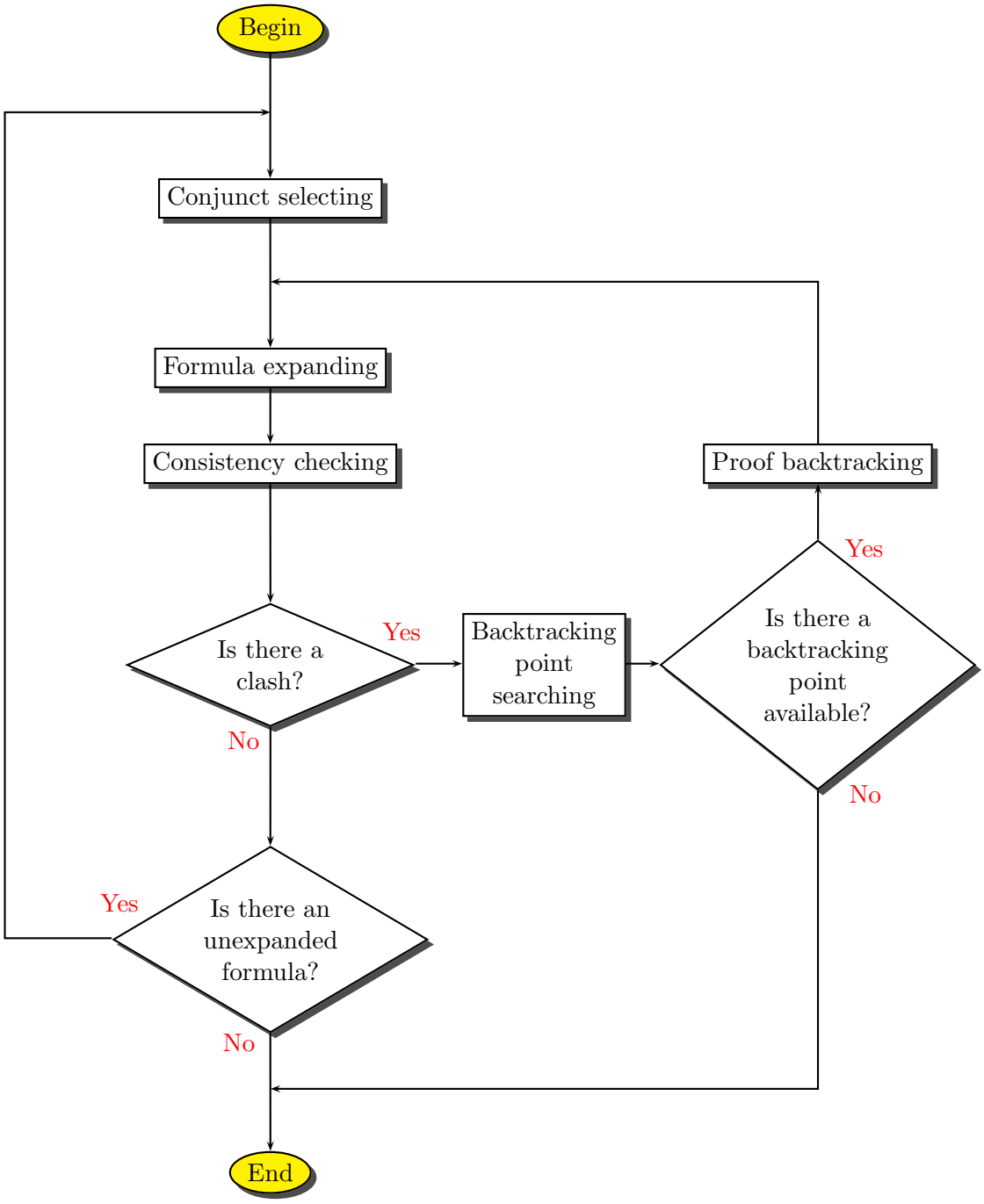


Figure 6.2: Infrastructure of the inference loop

this iteration. The selected formula is put into the formula expanding handler. An expansion rule is applied to this formula according to its top operator. The resulting formulae are put into the current branch. Now, consistency checking undertakes some procedure to make sure that there is no clash in the current branch. If a clash is found, the backtracking point searching handler is called to find out where to backtrack to. If no backtracking point is available, the algorithm terminates with the conclusion that the input formula is not satisfiable. Otherwise if a backtracking point is found, backtracking is performed to that point. Then the disjunction at the backtracking point needs to be re-expanded and an alternative disjunct is selected. Consistency checking is applied again and looks through the current branch for clashes. If no clash is found, and there are still formulae waiting to be expanded, the algorithm goes back to the beginning of the loop to start another iteration. If there is no unexpanded formula, this inference loop terminates with a conclusion that the input formula is satisfiable.

Every module in the inference loop can comprise other modules. For instance, the formula expanding handler can execute different modules according to the type of formula being processed. Figure 6.3 shows the modules involved in formula expanding. As can be seen, there are four modules directly accessible from formula expanding. If the formula to be expanded is a conjunction, the handler of and formula expanding is invoked. For the other kinds of formulae, or formula expanding, box formula expanding and diamond formula expanding are called respectively. Among those formula expanding handlers, disjunction expanding is the only one that is non-deterministic. That means any disjunct can be selected. The handler of disjunction expanding employs another module called disjunct selecting to choose a disjunct. Heuristic algorithms can be used in disjunct selecting to pick the formula that is more likely to lead to a shorter derivation.

One of the advantages of this event-handler framework is that it is capable of meeting the unpredictable needs of user defined event handlers. A reference of the inference data is passed to every handler as the parameter. Through the inference data a handler can access all of the context information about this inference. The inference data also has stacks dedicated to the arguments where a handler can retrieve its arguments sent by its caller. Every handler becomes an independent module. Event handlers share the same interface and can be invoked in a uniform way. They can be developed and compiled separately. Sometimes

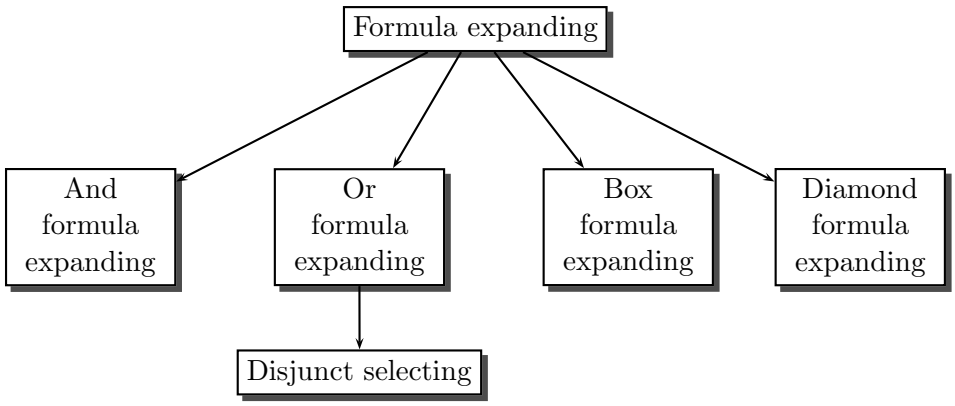


Figure 6.3: Structure of Formula expanding

it is difficult to obtain the source code of the system. This framework enables the users to develop their own event handlers and compile them even without the source code of a system. If the system is deployed as an inference server running in the background, it would be desirable that the system can add new function modules dynamically. With this event-handler framework, the handlers created by the user can be compiled as libraries and loaded by the system without having to stop the inference service.

6.5 Specification language

The framework we introduced above provides the users with flexible facilities. Through the framework users not only can specify the system using existing modules but also can write their own modules. Some users may want to implement some new techniques that are not currently supported by our system. We have designed a higher level language so that users can change existing modules or develop their own modules. Implementing a new technique usually means defining new data structures. To make developing new data structures and functions in our system easier, we invented a high level specification language. This language is syntactically similar to C++. But it offers a simpler and uniform way to build and access data structures. Our language has two kinds of statements for data structure definition. One is to use the keyword "struct" to define a data structure

like we can do in C or C++. Another one is to use the keyword "connection" to define relations between data structures.

Algorithm 19 Example of structure definition

```
1: struct Inference {};
2: struct Formula {};
3: struct World {};
4: struct Formula_instance {};
5: @connection Formula_world_fi{
6:         Formula;
7:         World;
8:         Formula_instance;
9: }
10: @connection Inf_fi {
11:        Inference(
12:                inf_fi_accepted,
13:                inf_fi_unexpanded,
14:        );
15:        Formula_instance;
16: }
```

Algorithm 19 is an example to show how data structures are defined with our language. In this example, four data structures are defined using the keyword "struct". They are Inference, Formula, World and Formula_instance. The Inference data structure represents a session of the inference procedure. It contains the context information to support all of the modules. The other structures represent their corresponding concepts in the labelled tableau algorithm for modal logics. For instance, a formula_instance represents a labelled formula.

The relations between the data structures are defined using the keyword "@connection". In Algorithm 19, two connections are defined. The first connection reflects the relation between a formula instance, a formula and a world. As a labelled formula, a formula instance is a combination of a formula and a certain world. The second connection defines the relations between an inference and a formula instance. There are three connection points between a inference and the formula instance. "inf_fi_accepted" keeps a record of the formula instances that have been accepted and put into the current branch. "inf_fi_unexpanded" contains the formula instances that have not been expanded yet. These connection points are called *ports*. There should be at least one port for every data structure in a connection. If the ports are not specified explicitly like we did for

"inf_fi_accepted" and "inf_fi_unexpanded", a default port will be created using the name of the connection but changing the first letter to lower case.

The connections in our system give great flexibility. They can model different kinds of relationships such as one-to-one, one-to-many and many-to-many relationships. A one-to-one relationship is the simplest relationship. A formula and its id have a one-to-one relationship because a formula can only have one id and a particular id can only be given to one formula. A one-to-many relationship exists between a formula instance and a world since a formula instance can only be true in one world but a world can hold multiple formula instances. An example of a many-to-many relationship is the dependency relation among different formula instances. It is because a formula instance can be the logical premise of multiple other formula instances and a formula instance can also be derived from more than one other formula instances.

Based on the data structure introduced above, we have developed a high level language to handle these data structures in a simple and uniform way. Algorithm 20 shows how to use our language to implement the expansion rule for and formulae.

Algorithm 20 Example of expand_and_formula (Inference* inf)

Input: inf: a reference to the inference data structure
Output: none

```
 1: Formula_instance* fi = inf->arg(FI);
 2: Formula* f = fi->formula_world_fi[1];
 3: World* w = fi->formula_world_fi[2];
 4: Formula* subf;
 5: foreach (subf, f->formula_formula_child[2]) {
 6:       Formula_instance* newfi = new Formula_instance();
 7:       addconn(subf->formula_world_fi, w->formula_world_fi,
 8:             newfi->formula_world_fi);
 9:       if (getconn(inf->inf_fi_accepted,newfi->inf_fi) == NULL) {
10:             addconn(inf->inf_fi_accepted, newfi->inf_fi);
11:       }
12:       if (getconn(inf->inf_fi_unexpanded,newfi->inf_fi) == NULL) {
13:             addconn(inf->inf_fi_unexpanded, newfi->inf_fi);
14:       }
15: }
16: return
```

The algorithm takes a reference of the inference as the input. In the first line, the argument is extracted. fi is the formula instance to be expanded. The operator

[n] ($n > 0$) is designed to access a data structure that is involved in a connection. n is the index of the data structure in the name of the connection. For example, in the second and the third line of the algorithm, the formula and the world are obtained from the connection formula_world_fi. They are respectively the first and the second part in the name of the connection separated by "_". In the 5th line, the keyword "foreach" works on a list and for every item in the list the following operations are undertaken. The function "addconn" is to establish a connection among the objects in its argument list. The order of the parameters should be in accordance with the name of the connection. The function "getconn" is to test if there is a connection (created by "addconn") among the given parameters.

The first line of Algorithm 20 defines fi as the formula instance to be expanded, which has been passed to this handler as an argument. Lines 2 and 3 get the formula and the world that fi is associated with. Line 4 defines a temporary formula subf. Line 5 starts a loop which goes through every child formula of f and assigns subf to be this child formula. Line 6 creates a new formula instance named newfi. Line 7 establishes a bond between the child formula f, the world w and the new formula instance newfi. Line 9 checks if this new formula instance newfi is in the current branch. If the answer is no, then newfi is put into the branch. Line 12 checks if newfi is in the waiting list of unexpanded formulae. If newfi is not found in the list, then it is added into the list. The statements numbered from 6 to 14 are performed for every subformula of fi.

6.6 Proof generation

Being able to output the proof steps is very important for an automated reasoning system. The proof output shows how a derivation is performed and what happened inside the theorem prover. It helps the users to have a better understanding of the system and the inference procedure. Proof output is also very helpful when the system is used in a logic class as a teaching assistant tool.

In our system, proof generation is handled as a special event that can be fired at many places, even inside another event. The handler of this event generates proof steps in a customized format and outputs them to a specified device. There are two different styles of output that our system can generate. They are the *memory snapshot style* and the *indented style*.

The indented style proof used in MLTP was inspired by pdl-tableau, a prototype solver for propositional dynamic logic [Sch03]. An example proof is shown in Listing 6.1. The proof consists of a list of all of the formula instances generated during the derivation in chronological order. Indentations are used to set off the proof steps after branching. For a particular formula instance, the formula instances that have been derived prior to it and share the same branch are directly above it either in the same column or in a column to the left.

A line in indented style proof uses two kinds of notations. One is for formula instances, another is for clashes. A formula instance is presented using a notation in the form of:

$$n(fn, wn, pfi[n])[rule, info].$$

Let this formula instance be I. In the notion, the first n is the id of this formula instance I. fn is the id of the formula that I represents. wn is the world that the formula instance I holds. $pfi[n]$ is the parent formula instance from which I was derived. $rule$ is the expansion rule used to generate the formula instance I. $info$ is supplementary information about the expansion rule used. For the conjunction and disjunction rules, $info$ contains the total number of conjuncts or disjuncts and the index of this formula instance I. For the box and diamond rules, $info$ contains the worlds involved in this derivation.

A clash is presented with a notation like:

$$clash : [fi1 : fi2].$$

$fi1$ and $fi2$ are the two formulae that caused this clash.

When a disjunction is expanded, the formula instance containing the selected disjunct is output in an indented way. The width of the indentation for this formula instance is calculated in the following way. Suppose the width of each indentation is s, the total number of the disjuncts is n and the width of the indentation for the newly derived disjunct is w. Then we have

$$w = s \times (n - 1).$$

Every time backtracking happens, the amount s of indentation is undone by moving the proofs to the left.

Listing 6.1: Indented proofs

```
Formula (normalized & simplified) :
1:and(2:or(3:dia(4:r1,
                 5:c1),
           6:dia(7:r1,
                 8:c2)),
      9:box(10:r1,
            11:and(13:-c1,
                   15:-c3)))
Proofs:

1(f1,w1,pfi[])
2(f2,w1,pfi[1])  [and,1/2]
3(f9,w1,pfi[1])  [and,2/2]
    4(f3,w1,pfi[2])  [or,1/2]
    5(f5,w2,pfi[4])  [dia,w1->w2]
    6(f11,w2,pfi[3])  [box,w1->w2]
    7(f13,w2,pfi[6])  [and,1/2]
    8(f15,w2,pfi[6])  [and,2/2]
    clash:[fi7:fi5]
9(f6,w1,pfi[2])  [or,2/2]
10(f8,w3,pfi[9])  [dia,w1->w3]
11(f11,w3,pfi[3])  [box,w1->w3]
12(f13,w3,pfi[11])  [and,1/2]
13(f15,w3,pfi[11])  [and,2/2]

==> Problem Satisfiable
```

Listing 6.1 shows an example of an indented style proof for the given formula:

$$(\Diamond c1 \vee \Diamond c2) \wedge \Box(\neg c1 \wedge \neg c3).$$

Being normalized and simplified the conjecture formula is presented in a tree-like structure at the beginning of the proof. In the first step, the formula instance 1 is derived and it holds in the initial world $w1$. It is the initial formula, and therefore has no predecessor. In the second step, a formula instance 2 is derived from the formula instance 1 by applying the conjunction rule. The supplementary information "1/2" means that there are two conjuncts and this is the first one. The next step is similar to step 2, only the second conjunct is derived. Step 4 is indented because a disjunction, formula instance 2, has been expanded. The disjunct, formula 3, is selected first to form formula instance 4. In the next step, a diamond formula is expanded. The supplementary information "w1−>w2" means that a new world w2 has been created which is a successor of world w1. Formula instance 6 is derived from a box formula expansion. The supplementary information "w1−>w2" in the context of a box formula expansion means that the box formula propagated from world w1 to w2. Further down a few steps, a clash is found between the formula instance 7 and 5. The disjunction, formula 2, needs to be re-expanded. And this time another disjunct, formula 6, is selected. Because backtracking has been performed, the proofs are moved to the left to undo the indentation. After the formula instance 13, all of the formulae have been expanded and no clash has been found, therefore the derivation ends with the conclusion that this problem is satisfiable.

A memory snapshot proof reveals more information than an indented style proof. Memory snapshot style output contains not only details about logic derivations but also status of the automated reasoning system. After each inference step, the content of the important data in the memory is listed. These lists include the current branch, the unexpanded formulae, the existing worlds, and the clashes.

Listing 6.2 is an example of a memory snapshot proof for the formula

$$\Diamond c1 \wedge (\Box\neg c1 \vee c2).$$

Like the indented proofs, the normalized and simplified conjecture formula is

Listing 6.2: Memory snapshot proofs

```
Formula (normalized & simplified) :
1:and(2:dia(3:r1,
            4:c1),
      5:or(6:box(7:r1,
                 9:-c1),
           10:c2))
Proof:
#[1]INITIALIZE
coll_accepted: 1(f1,w1)
coll_unexpanded: 1
coll_world: 1()

#[2]EXPAND_FORMULA: 1(f1,w1)
coll_accepted: 1(f1,w1) 2(f2,w1) 3(f5,w1)
coll_unexpanded: 2 3
coll_world: 1()

#[3]EXPAND_FORMULA: 2(f2,w1)
coll_accepted: 1(f1,w1) 2(f2,w1) 3(f5,w1) 4(f4,w2)
coll_unexpanded: 3
coll_world: 1(2) 2()
......

#[7]EXPAND_FORMULA: 5(f6,w1)
coll_accepted: 1(f1,w1) 2(f2,w1) 3(f5,w1) 4(f4,w2)
               5(f6,w1) 6(f9,w2)
coll_unexpanded:
coll_world: 1(2) 2()

#[8]CHECK_CONSISTENCY
coll_clash: 1{[6][4]}

#[9]BACKTRACK: 3(f5,w1)
coll_accepted: 1(f1,w1) 2(f2,w1) 3(f5,w1) 4(f4,w2)
coll_unexpanded:
coll_world: 1(2) 2()

#[10]RECHOOSE_BRANCH
coll_accepted: 1(f1,w1) 2(f2,w1) 3(f5,w1) 4(f4,w2) 7(f10,w1)
coll_unexpanded:
coll_world: 1(2) 2()

==> Problem Satisfiable
```

listed at the beginning of the proof. The proofs are organized into sections separated by empty lines. Each section represents an inference step. The first line of a section of proofs starts with "#", followed by the number of the inference step and the name of the operation performed in the inference step. The remaining lines of a section of proofs are lists of data. These lists are important data structures used by the inference procedure. For example, "coll_accepted" contains the accepted formula instances on the current branch. These instances are presented in the form of s(fm,wn), where s is the id of the formula instance, m is the id of a formula which can be found in the formula tree at the beginning of this proof and n is the id of a world. "coll_unexpanded" contains the ids of the unexpanded formulae. "coll_world" consists of the worlds in the Kripke frame. Each world is given in the form of n(m), where n is the id of this world and m is the id of the successor worlds. "coll_clash" gives the clashes found in the system. Clash information has the form t{[m],[n]}, where t is the id of this clash, m and n are the two formulae causing this clash.

In Listing 6.2, the first step creates an initial world, world 1. This world and the conjecture formula form a formula instance and put this instance into the current branch. In the second step, formula instance 1 is expanded. Its two conjuncts are put into the current branch. In the third step, formula instance 2 is expanded. Because it is associated to a diamond formula, world 2 is created which is a successor of world 1. Steps 4 to 6 are not listed because they are similar to step 2 and 3. In step 7, formula instance 5 is expanded which is associated to formula 6, a box formula. This expansion adds formula instance 9 into the branch. Step 8 performs consistency checking. A clash is found between formula instances 6 and 4. In step 9, backtracking is performed to release the clash. Formula instance 3 is selected as the backtracking point. Formula instances 5 and 6 are removed from the current branch. Step 10 re-expands the disjunctive formula instance 3. Another disjunct, formula instance 7, is added into the current branch. So far all of the formula instances have been expanded and no clashes has been found. Hence the prover makes the conclusion that the conjecture formula is satisfiable.

The memory snapshot proof is useful for the developer of a theorem prover to debug the system. In the development of an automated reasoning system, it is inevitable that some errors happen in the system. These errors can be classified into two categories. They are program errors and algorithm errors. Program errors are caused by mistakes in the code. Typical program errors include misuse

of pointers and forgetting to initialize or release memory. Algorithm errors happen when the algorithm does not work in the way that it is supposed to. Both program errors and algorithm errors can lead to incorrect operations and make the system to generate wrong results. Some of these errors can be detected and reported by the operating system. Hence it is easier for the developers to fix these kinds of errors. However, many of the errors can not be detected by the operating system. To identify these errors, a developer can look into the proof outputs to find out what goes wrong. The memory snapshot proof can be used here to tell the developer what is going on the system.

Unfortunately, sometimes the problem becomes more complicated. Some errors only happen when the prover is trying to solve a complex formula. The output of the proofs contains thousands of lines. It is too big to be checked by hand. Our experience shows that this kind of error is the most difficult to fix. To solve this kind of problem, we have developed a proof verifier. It automatically checks the memory snapshot proofs generated by the prover. The properties to be checked are encoded as rules in the proof verifier. If the verifier finds anything in the proof outputs that is not working in accordance with the rules, it will report this as an exception. We have found that the proof verifier is very helpful to identify errors in the prover.

Chapter 7

Evaluation

The purpose of this chapter is to discuss the evaluation results of our modal logic theorem prover MLTP. Based on empirical tests, we evaluate techniques that we introduced earlier including dynamic backtracking and forward checking. We also compare our system with other well-known systems like MSPASS, FACT++ and RACER.

The rest of this chapter is organized as follows: Section 7.1 introduces the methodology of benchmarking. Section 7.2 describes the problem suites used in the empirical tests. Section 7.3 presents the results of simplification. Section 7.4 gives the results of different backtracking techniques. Section 7.5 gives a comparison of forward checking and backward checking. Section 7.6 compares the efficiency and generality of different automated reasoning systems. Section 7.7 discusses the fundamental low level data structures and their evaluation. Section 7.8 is about the efficient use of caches.

7.1 Benchmarking methodology

Empirical tests usually collect performance data via tests on suites of formulae. Then comparison is made on the performance data from different systems or from runs of one system using different optimisation techniques [Hor97, HS97]. In the tests described in this chapter, two kinds of data are used as a measure of performance. They are CPU time and search space.

Earlier research based on empirical experiments usually used CPU time as the most important measure of the performance [Hor97, HS97]. From an application's

point of view, the usefulness of a proof system is significantly affected by the computational intractability of its algorithm. It can be seen from the earlier analysis that for tableaux algorithms, computational intractability is more correlated with CPU time than with other possible measures like storage requirement. As our tests show (see below), the performance is mainly affected by non-deterministic expansions while the storage requirements are only affected by deterministic expansions. The other reason which motivates the choice of CPU time is that most of the programming languages have incorporated functions to obtain system time easily.

However, there are some drawbacks to using CPU as a measure, because not only can the theoretical algorithms affect the CPU time, but the implementation style, programming language and test platform also significantly influence the execution time.

The search space measure indicates that how many backtracking steps have to be performed during satisfiability tests. This factor measures the improvements caused by taking optimisations, which find the unsatisfiability inherited in a modal formula before search it.

7.2 Problem suite

To reach a reliable conclusion, empirical tests need to be performed on a great number of example problems. It is not trivial to select good test instances when evaluating the performance of a algorithm. Good instances give good measures of system's performance. Inappropriate test instances could lead to wrong conclusions.

Earlier empirical studies employed random formula generators to supply the problems used in tests [Hor97, HS97]. Randomly generated modal formulae were first used by Giunchiglia and Sebastiani [GS96] to test the provers KSAT_0 and KSAT. They designed a random generator for formulae in the propositional modal logic K and tested their systems on formulae produced by the generator. As pointed out in [HS97], some versions of their generator generate trivially satisfiable problems. For 3CNF clauses of degree 0, it can generate formulae like $\neg\Box(p \vee \neg p \vee p)$. Thus, random modal KCNF formulae contain tautological and contradictory subformulae. It is easy to remove these subformulae without affecting satisfiability.

Giunchiglia-Sebastiani problems are used in the empirical experiments described in this chapter. The parameters used to generate the problem set are as following:

- the number of clauses in the main formula, L,
- the number of propositional variables, N,
- the number of modalities, M,
- the number of subformulae per disjunction, K,
- the probability that an atom is propositional, P,
- the maximum modal depth, D.

We utilize the random formulae generator by fixing N, M, P, and generate random formulae for various values of L, and used the resulting formulae to test MLTP.

The parameters of the Giunchiglia-Sebastiani generator are set as follows: N=5, M=1, K=3, D=2, P=0.5. Different values of L are used to generate different problem sets. For each L, 100 random problems are produced. In this chapter, tests are performed on the problem sets with L from 5 to 40. Since the number of variables is 5, the number of clauses ranges from 5 to 200.

Besides randomly generated problems, we also used the TANCS-2000 problems [MD00] in some of the empirical experiments. The benchmark problems are constructed by translating QBF formulae into modal logic K.

Three translation schemes are used to generate TANCS-2000 problems. The formulae relatively easy to handle are created using Schmidt-Schauss-Smolka translation [SSS91]. The formulae of medium difficulty are generated by Ladner translation [Lad77]. The hard formulae are constructed with a Halpern translation.

The QBF formulae are generated using the following parameters:

- the number of clauses, C,
- the depth of the alternation, D,
- the number of variables, V,

- the number of variables in each clause, K.

For each variable in a clause, there is a possibility of 50% that the variable is negated.

The test results of the Giunchiglia-Sebastiani problems are processed and displayed in plots. In these plots, the number of clauses is used as the x-axis. For the tests about runtime, the CPU time in seconds is used as the y-axis. For the tests concerning backtracking, the number of backtracking steps is used as the y-axis.

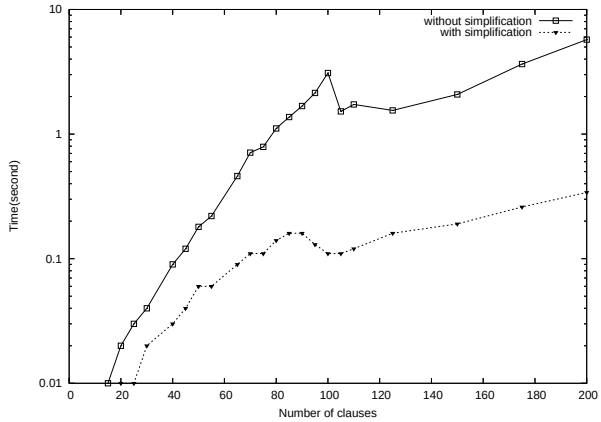
Two kinds of plots are used to get a better understanding of the results. They are the 50% percentile plot and the 100% percentile plot. A 50% percentile plot contains the data of the median case. Whereas a 100% percentile plot consists of the data of the worst case. To produce these plots, the results from each group of problems are sorted in ascending order. In the sorted list of results, the data in the middle is picked to create the 50% percentile plot, the data at the end is used for the 100% percentile plot.

The machine we used to perform the tests is a Dell workstation with Linux operating system. It has a 1.2GHz Intel Pentium 4 CPU and 512MB main memory. While doing the tests, a limitation of 1000 seconds CPU time is imposed on the runtime of the prover. That means when the derivation time exceeds 1000 seconds, the reasoning process is immediately stopped no matter the result has been worked out or not.

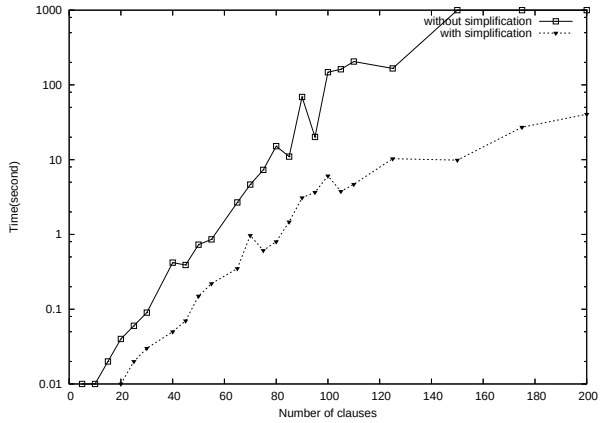
7.3 Simplification

The simplification techniques discussed in Section 3.2 have been implemented in MLTP. Usually they are performed as part of pre-processing prior to the start of the proof search. Propositional and modal logic tautologies are used in simplification to reduce the complexity of conjecture formulae.

In Figure 7.1, the graphs of tests with and without simplification are plotted. Dynamic backtracking is performed in those tests. In the plot of 50% percentile, simplification improves the prover's performance on the problems that have more than 15 clauses. Especially, for the problems with more than 80 clauses, the tests with simplification is at least 10 times faster than tests without simplification. In the plot of 100% percentile, the tests with simplification also shows a better performance on most of the problems. Without simplification, the prover failed



a) 50% percentile CPU time



b) 100% percentile CPU time

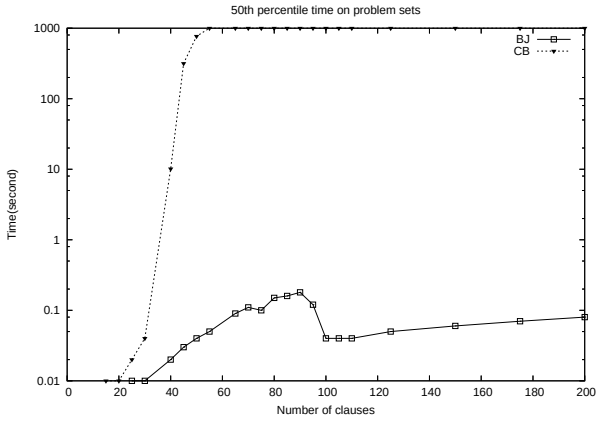
Figure 7.1: Test results: Simplification

to terminate on some of the problems with more than 150 clauses, whereas all of tests with simplification can terminate within 100 seconds.

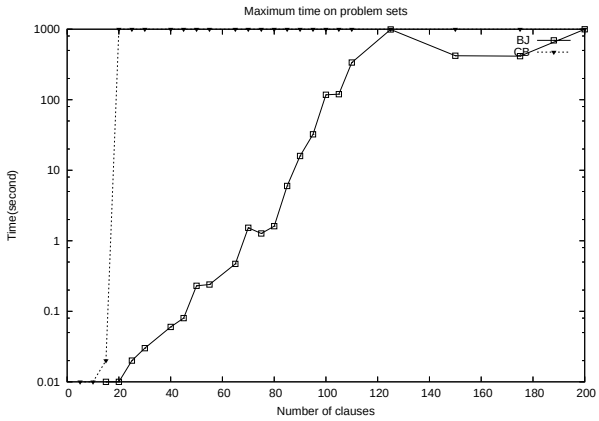
We can see from the results that simplification significantly improves the prover's performance on the Giunchiglia-Sebastiani benchmark problems. As we have mentioned earlier, the Giunchiglia-Sebastiani problems contain tautological and contradictory subformulae. Therefore by using simplification, the complexity of the problems can be reduced considerably.

7.4 Backtracking

Three kinds of backtracking techniques are implemented in MLTP. They are chronological backtracking, backjumping and dynamic backtracking. Chronological backtracking is the basic technique, which only chooses the backtracking point and does backtracking in a chronological way. Backjumping extends chronological backtracking with the ability to choose the backtracking point in a non-chronological way. Dynamic backtracking further improves chronological backtracking so that it not only chooses the backtracking point but also does backtracking in a non-chronological way.



a) 50% percentile

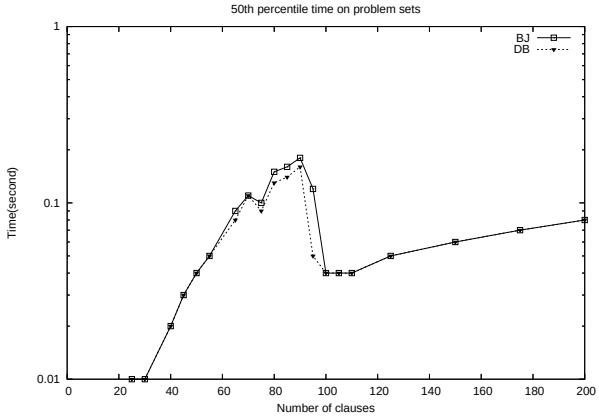


b) 100% percentile

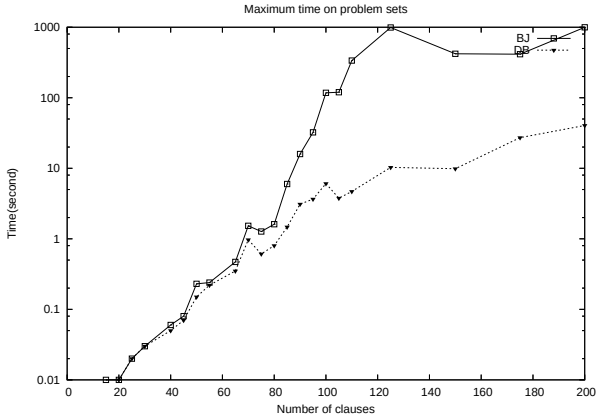
Figure 7.2: Test results: Chronological backtracking vs. Backjumping

The results of tests using chronological backtracking are plotted against results of tests using backjumping in Figure 7.2. The input formulae used in these tests are all simplified. We can see that in most of the cases backjumping outperforms

chronological backtracking. In the 50% percentile plot, chronological backtracking could not solve most of the problems that have more than 50 clauses within the given time limit. In contrast, the average time backjumping need to solve all of the problems is less than 1 second. In the 100% percentile plot, chronological backtracking failed to solve the problems that have more than 20 clauses. On the other hand, backjumping successfully solved most of the problems. Only on the most difficult problems from the two classes with 120 and 200 clause backjumping failed to terminate within the given time limit.



a) 50% percentile CPU time



b) 100% percentile CPU time

Figure 7.3: Test results: Backjumping vs. Dynamic backtracking

The results of tests using backjumping are plotted against results of tests using dynamic backtracking in Figure 7.3. In the plot of 50% percentile CPU time, there is no significant difference between backjumping and dynamic backtracking. Only on the problems of the size between 60 and 100 clauses, dynamic backtracking has

a slightly better performance than backjumping. In the plot of 100% percentile cpu time, we can see that dynamic backtracking performs much better than backjumping especially on the problems of greater size. For problems that have more than 100 clauses dynamic backtracking improves backjumping by at least one order. While backjumping failed to solve some of the problems, dynamic backtracking managed to terminate on all of the problems.

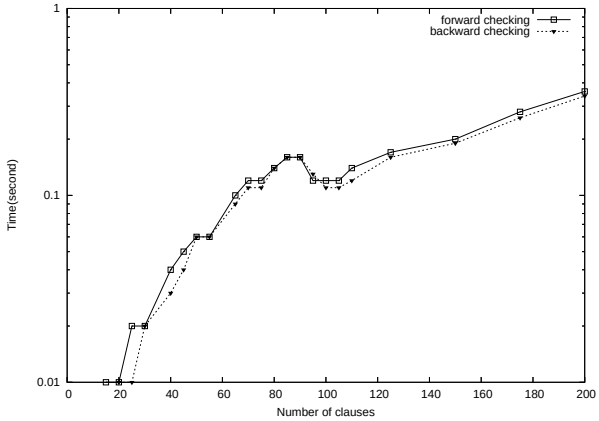
From the test results, we can see that backjumping and dynamic backtracking outperform chronological backtracking by far in both average cases and worst cases. Dynamic backtracking only performs better than backjumping noticeably on the hard problems. In general, dynamic backtracking and backjumping give similar performance. When implementing a theorem prover, if efficiency is the most important factor, the results obtained suggest that dynamic backtracking would be the best choice. But considering the additional complexity of dynamic backtracking, backjumping is also recommended.

7.5 Forward checking

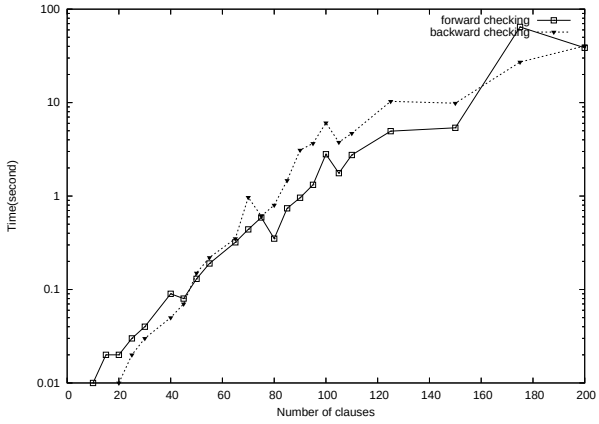
In Chapter 5, we introduced forward checking as a kind of forward reasoning mechanism. Forward checking is an alternative method for consistency checking to backward checking. As a traditional method, backward checking looks through the formulae already derived to see if there is any formula complementary to the newly derived formula. Using pre-computed connections, forward checking aims to find the clashes earlier so that unnecessary work can be saved. In particular, when a new formula is derived, unlike backward checking, forward checking does not search the current branch for clashes but looks through the formulae that have not been expanded yet and rules out the ones that can cause clashes.

In the empirical tests discussed in this section, dynamic backtracking is used to work with forward checking or backward checking.

Figure 7.4 shows the plots comparing forward checking and backward checking. In the plot of 50% percentile, forward checking and backward checking have similar performance on all of the problems. Backward checking works slightly better than forward checking. In the plot of 100% percentile, forward checking is slower than backward checking on small problems (less than 40 clauses) probably because of the overload introduced by creating pre-computed connections. For problems with 80 to 150 clauses, forward checking performs better than backward



a) 50% CPU time

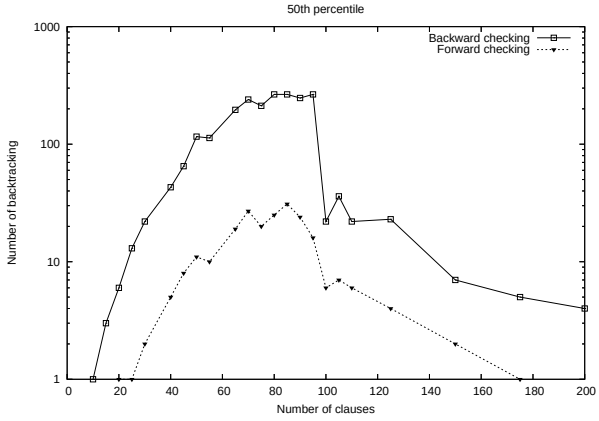


b) 100% CPU time

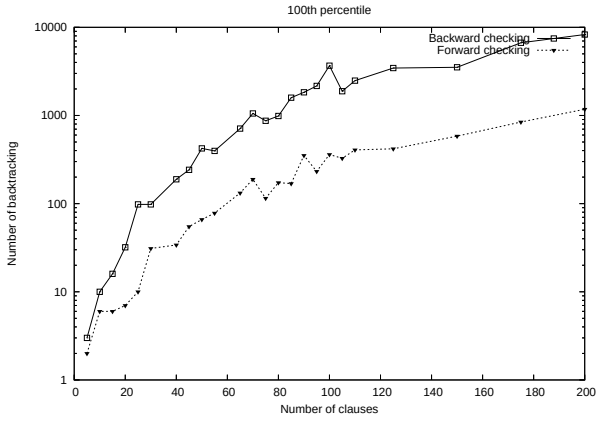
Figure 7.4: Test results: Forward checking vs. Backward checking

checking.

Figure 7.5 compares the number of backtracking steps performed by MLTP using forward checking and backward checking respectively. From the plots we can see that forward checking reduced the number of backtracking steps very successfully. Referring to the 50% percentile plot in Figure 7.4 we can see from the 50% percentile plot in Figure 7.5 that the groups of problems that take more time to solve are also the groups of problems on which more backtracking steps are performed. Forward checking involves fewer backtracking steps than backward checking on all of the problems. For most of the problems that have more than 175 clauses, forward checking can complete the proof search straightforwardly without a single backtracking step. In the plot of 100% percentile, forward checking also performs less backtracking steps than backward checking.



a) 50% percentile



b) 100% percentile

Figure 7.5: Backtracking steps: Forward checking vs. Backward checking

Besides randomly generated problems, we also used the TANCS-2000 problems [MD00] to compare the performance of forward checking and backward checking. The problems we tested are the modal QBF problems in the modal PSPACE division with the translation described in [SSS91]. The specification of the problems is p-qbf-cnfSSS-K4 with V=4, C={10,20,30,40,50} and D={4,6}. For each specification there are eight problems. Table 7.1 present the test results in 10ms for backward checking and forward checking respectively. The content of the tables are the geometric mean of the running time of the eight problems in each class.

We can see from the results in Table 7.1 that backward checking performs better than forward checking on the problems we tested. This is because, compared with the randomly generated problems the TANCS-2000 problems involve

Modal QBF	C10	C20	C30	C40	C50
cnfSSS-V4-D4	12	18	20	20	24
cnfSSS-V4-D6	196	2926	3502	168	77

a. Results of backward checking

Modal QBF	C10	C20	C30	C40	C50
cnfSSS-V4-D4	17	27	25	24	25
cnfSSS-V4-D6	461	7984	17560	474	172

b. Results of forward checking

Table 7.1: TANCS-2000 results: forward checking & backward checking

heavily nested modal formulae. This feature increases the costs of generating the precomputed connections which compromises the benefits of forward checking. It is also worth noting that our test results are not as good as results of the other highly optimized tableau-based theorem provers, DLP and FACT [PS00, Hor00]. This is due to the caching technique implemented in DLP and FACT which is not incorporated as yet in MLTP. Caching improves a prover's performance by keeping and reusing the satisfiability status of a formula. Caching works especially well on problems that have heavily nested modalities [PS00, Hor00].

From the test results we can see that when working with dynamic backtracking, forward checking does not have an advantage over backward checking. Though there is evidence that forward checking significantly reduces the number of backtracking steps, the additional computational difficulties of forward checking have an impact on its performance. For the problems with many nested modalities, the overload of forward checking considerably slows it down.

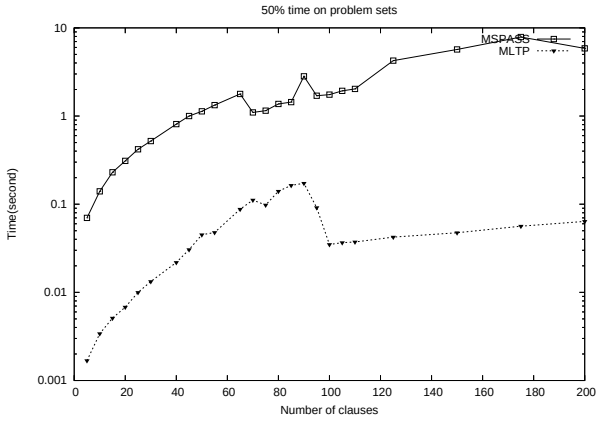
7.6 System comparison

To evaluate our system, we compare MLTP with state-of-the-art theorem provers, MSPASS [HS00] and RACER [HM01]. Randomly generated problems are used in the empirical tests shown below.

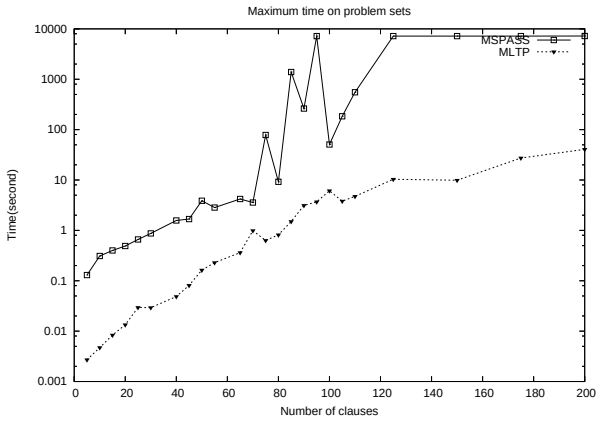
As a resolution based theorem prover, MSPASS handles modal logic formulae by transforming them into first order logic. Therefore MSPASS can handle all modal logics that can be translated into first order logic. In the empirical experiments, MSPASS (v.2.0g1.4) is used with the following settings.

-DocProof=0	-PGiven=0	-PProblem=0	-PKept=0
-EMLTranslation=2	-EMLFuncNary=1	-Sorts=0	-Select=2
-CNFStrSkolem=0	-CNFOptSkolem=0		

RACER is dedicated to description logics. It is a highly optimized tableau prover that can handle many expressive modal logics. Capable of both TBox reasoning and ABox reasoning, RACER is widely used as an efficient solver in applications of description logics. In the empirical experiments, RACER (v.1.7) is used as a black box without any special settings.



a) 50% percentile CPU time

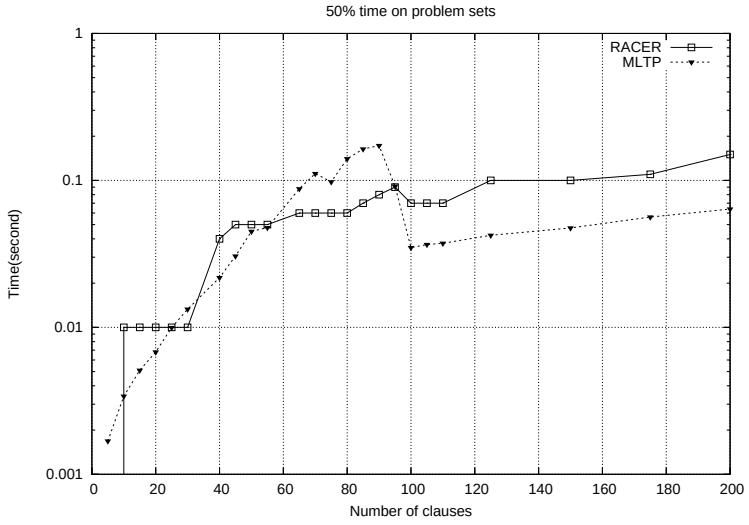


b) 100% percentile CPU time

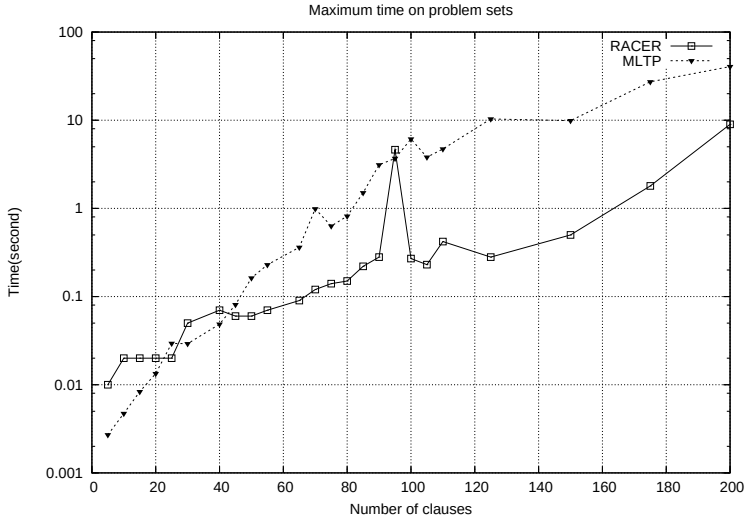
Figure 7.6: Test results: MLTP vs. MSPASS

Figure 7.6 shows the plot comparing the performance of MLTP and MSPASS on the Giunchiglia-Sebastiani problems. In the tests, MLTP used dynamic backtracking with backward checking. From the plot, we can see that MLTP outperforms MSPASS on all of the problems. In the 100% percentile plot, MSPASS

failed to solve some of the problems within the given time limit while MLTP successfully solved all of the problems.



a) 50% CPU time



b) 100% CPU time

Figure 7.7: Test results: MLTP vs. RACER

Figure 7.7 shows the plots comparing the performance of MLTP and RACER. In the plot of 50% percentile, we can see that MLTP and RACER have similar performance on all of the problems. There is hump in the middle of the MLTP plot. That means the problem with 60 to 100 clauses takes more time for MLTP to solve than the others. On the other hand, the plot of RACER is smoother. The problems difficult for MLTP do not cause much more trouble than the others

for RACER. In the plot of 100% percentile, RACER performs better than MLTP on the problems that have more than 60 clauses by around 1 order.

On the problems we have tested, MLTP has comparable performance with the state-of-the-art automated reasoning systems. Considering that efficiency is not the only purpose of MLTP and some of the useful optimization techniques (e.g. complement splitting and caching) have not been implemented in MLTP, the results of MLTP are quite acceptable.

The purpose of MLTP is to offer an efficient and generic theorem prover. We have shown that in terms of efficiency MLTP is comparable with the well-known highly optimized systems like MSPASS and RACER. Next we are going to compare MLTP against other provers not only on the performance but also on the extent of generality. The systems included in our comparison are RACER, LWB, Lotrec, TWB and MLTP.

Features	RACER	LWB	Lotrec	TWB	MLTP
logic scope	DLs	many logics	MLs...	MLs...	ML(K)
Optimization	Yes	Yes	No	some	Yes
syntax grammar	fixed	fixed	specifiable	specifiable	specifiable
deduction rules	fixed	fixed	specifiable	specifiable	specifiable
strategies	fixed	fixed	specifiable	specifiable	specifiable
backtracking method	fixed	fixed	fixed	fixed	specifiable

Table 7.2: Comparison of theorem provers

Table 7.2 compares the features of different systems. RACER can handle a wide range of description logics. LWB can deal with many kinds of logics including modal logics (KT, S4, S5), tense logics, linear logic and intuitionistic logic. Lotrec can handle modal logics or logics that have relational semantics. TWB can work on modal logics and temporal logics. MLTP so far can only handle modal logic K, but it is not difficult to extend it to handle other logics.

RACER is a highly optimized prover. LWB also incorporates many well known optimization techniques. As far as we know, Lotrec has no optimization techniques implemented. TWB has optimizations like simplification and back-jumping available, but not as a standard. MLTP has a variety of optimizations incorporated.

Regarding to the specifiability of logics, inference rules and heuristics, RACER and LWB have most of the details hard coded in their system. There are few

things customizable by the users in these two systems. By contrast, in Lotrec, TWB and MLTP users can specify their own logic, deduction rules and strategies. Only in MLTP can different backtracking methods be specified.

7.7 Optimizing data structures

An automated theorem prover is different to most of the user interactive applications that we use everyday not only because provers are more computationally complicated and they involve more correlative data structures but also because they are more dynamic. Normally a prover takes a logic formula as its input. The size of the formula can be up to hundreds of clauses. The formula is parsed and stored in some data structure accessible to the prover. Then a logical inference procedure is performed on the formula. The time consumption of this procedure ranges from microseconds to hours (days) depending on the scale of the formula and the inference algorithm. Lots of temporary data is generated during the inference procedure. This data is heavily interrelated. Considering these relations as bidirectional links we can reach a certain data item through several alternative routes of access. Which route is the most efficient depends on the current system memory state (system state). The system state is different for problems and it changes while the system is running. For those reasons, the optimal access varies according to the formula and the stage of the inference procedure.

A generic automated theorem prover should allow the users to specify the strategies and modify the algorithm to some extent. Data structures of hard-coded systems are normally designed for one particular kind of operation and can become clumsy for the others. High level generality inevitably needs flexible data structures that are able to fit into different operations. It appears that efficiency and generality have similar requirements on the data structures.

Collections are widely used in computer programs. Data of the same type are stored in a collection so that procedures can be performed on not just one but a set of data by iterating through the collection. Inserting, removing and searching are operations that can be performed on a collection. Techniques commonly used to implement a collection are arrays, lists, associative arrays and hash tables. These techniques have different time complexities with respect to the operations of the collection.

To increase the speed of the collection operations, we keep an item's position

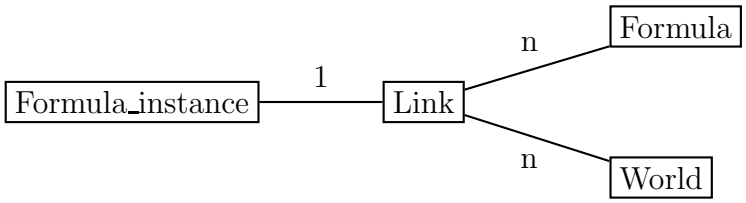


Figure 7.8: Example of Entity-Relationship Diagram 1

in the memory where it is stored when we put the item into a collection. The memory position will be used later by the other operations performed on that item. It is required that a collection does not change an existing item's memory position until this item is removed. Arrays do not meet that need because in arrays an item's position changes when deletion or insertion happens on any item before that item. Associative arrays need a value as the key to access an item. The time cost of searching, deleting and inserting in an associative array is $O(log(n))$ only if you use the information which has been set as the key to access the array. That obviously is not compatible with the requirements of a generic system. The same applies to hash tables. In particular, is very difficult to find an appropriate hash function for changeable data structures. This leaves lists as our only choice for implementing collections. Inserting and deleting items in a list only takes constant time. The drawback of lists is that searching is slow, and has $O(n)$ complexity. But there are mechanisms, which we will introduce later, that can improve the performance.

Figure 7.8 is an Entity-Relationship Diagram that presents the idea of the data structure as used in our system. We call it a relation-based structure. In the figure, "Formula_instance", "Formula" and "World" are three classes of data. They are related to each other. A formula instance is an instance of a certain formula at a certain world. "Link" represents their relationship and connects them together. The label "n" of the connection between Formula and Link means that a formula can have multiple such relations. That is, a formula can have different formula instances with respect to different worlds. Similarly a world can contain multiple formula instances, each represents a particular formula. The

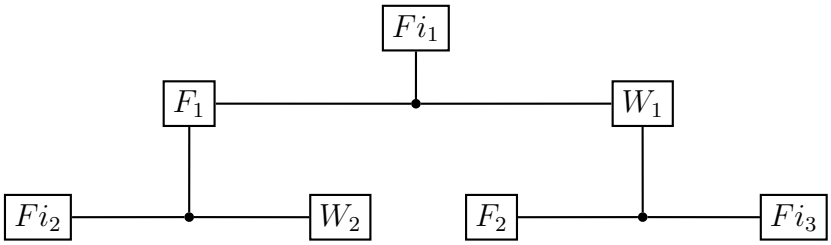


Figure 7.9: Example Entity-Relationship Diagram 2

label "1" of the connection between the formula instance and link means that a formula instance can only represent one formula and exists in one world.

With the relation-based data structure, we can perform various operations conveniently. Suppose we have a formula f and a world w, and we want to find the formula instance connected with f and w. Now we have two options. We can either look through the collection of f to see if there is an item connected to w or we can look through the collection of w to find an item connected to f. The time consumption of these two ways depends on the size of the collections of f and w. As we mentioned before, the system state is dynamic. It is impossible to predict which of the two ways of look-up is better beforehand. Because of the generality of relation-based data structures, we are free to choose either way. So we can simply compare the size of the two collections and use the smaller one to undertake the operation.

Figure 7.9 gives a concrete example of the model described in 7.8. In this figure, links are simplified to dots. F_1, F_2 are formulae, W_1, W_2 are worlds and Fi_1, Fi_2 and Fi_3 are formula instances. Fi_1 is an instance of the formula F_1 at the world W_1. Fi_2 is another instance of F_1 at the world W_2. And Fi_3 is an instance of F_2 at W_1. Thus the collection of F_1 contains two links and the collection of W_1 has two links, too. If the formula F_1 and the world W_1 are given, we need to find the formula instance that both F_1 and W_1 are connected to. Since the collection of F_1 and the collection of W_1 both contain two items, either we can look through F_1's collection and find the link connected to W_1, or we can look through W_1's collection and find the link connected to F_1. In another case, if F_2 and W_1 are given, the formula instance Fi_3 is to be found. Since F_2's collection

has one item while W_1's has two, it is better to look through F_2's collection than to look through W_1's collection.

Consider the situation in which we utilize associative arrays or hash tables. We may use the formula as the key and the formula instance as the value. If in the system the number of formulae is n and each of them has just one instance in the world w, the time needed for searching for a formula instance is $O(log(n))$ for associative arrays and optimally constant for hash tables. But with our method, the time cost is constant. On an alternative we can use worlds as the key. If in the system the number of worlds is n and f has one instance in each of these worlds, the relation-based structure also outperforms associative array and hash table.

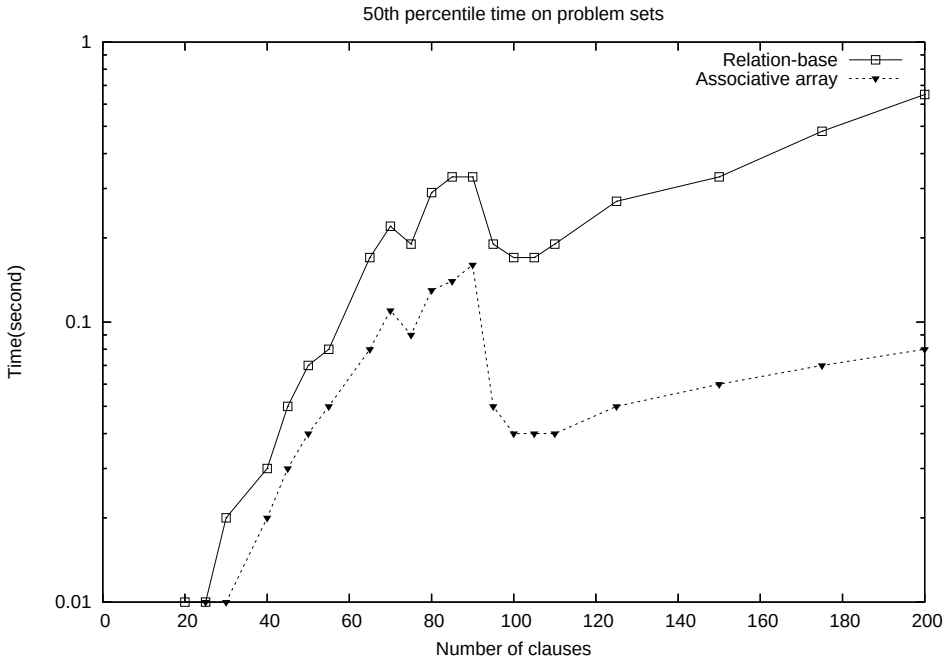


Figure 7.10: Test results (50%): Relation-based structure vs. Associative array

Figures 7.10 and 7.11 show the evaluation results of relation-based structures against the results of associative arrays. The Giunchiglia-Sebastiani problems are used in these tests. The CPU times in Figure 7.10 are the 50% percentile time for each set of the data tested. The CPU times in Figure 7.11 are the maximum time for each set of the data. From those figures, we can see that systems with relation-based structures are slower than systems with associative arrays. But the differences are not significant. The reason for this is mainly because looking through a list is more expensive than looking through an associative array. With

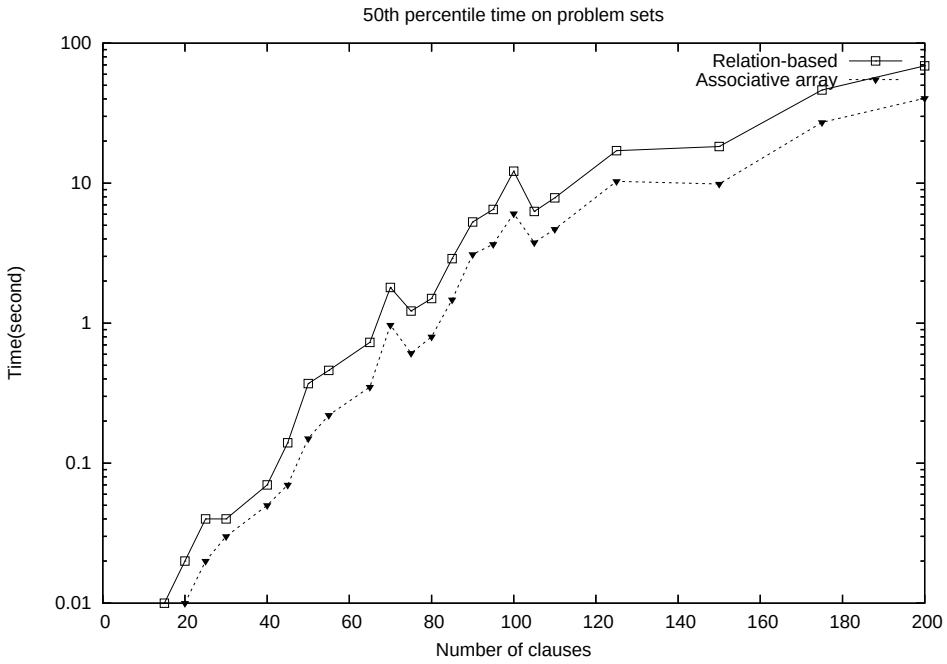


Figure 7.11: Test results (100%): Relation-based structure vs. Associative array

this acceptable efficiency sacrifice, our system gains great generality and uniformity. Performance of relation-based structures can be improved by optimisation techniques. Some of these techniques will be discussed in the following section.

7.8 Cache efficient implementation

Our MLTP prover is implemented in C++. Like most sophisticated C++ programs, the system uses a lot of pointer references. We expected pointer redirecting to be very cheap, only taking a fraction of the system's runtime. However, according to the profile we made for our system based on a run solving a formula of considerable difficulty, some operations that only involve a few pointer redirections are surprisingly expensive. These operations are fundamental and frequently executed. They are performed thousands of times during a run on some complicated formula. Therefore a small fraction may add up to something significant. To understand this problem, some facts about the memory architecture of modern CPUs needs to be introduced.

Modern machines employ a hierarchical memory design. Most of the x86-based machines including Intel and AMD CPUs utilise a memory hierarchy which

consists of main memory, L1 cache and L2 cache. A cache is a special type of memory. Compared to main memory, a cache has a smaller size and a much higher read and write speeds.

When certain data in memory is required by an application, the address of the data is given. The machine first looks into the L1 cache. If the data is found in L1 cache, it is called a *cache hit*. Otherwise a *cache miss* or more specifically a *L1 miss* happens. When there is a L1 miss, the machine checks the L2 cache. If there is a *L2 miss* as well, the main memory will be searched. If the data is found in the main memory, it will be copied into the caches. The machine will apply some prefetching strategy to anticipate the memory that will possibly be required in the future. This memory will be copied into the caches as well. In case the caches are full, some data needs to be removed to make room for new data. The data to be removed from the caches is determined by a replacement policy. A widely used replacement policy is to select the least recently used data (LRU). Because caches can be more rapidly accessed than main memory, the time loss caused by a cache miss is called *miss penalty*. On a modern machine, the miss penalty of L1 can be around 10 cycles and the miss penalty of L2 is up to 200 cycles.

To improve the performance of a program we need to reduce the cache misses as much as possible. Theorem provers usually handle a lot of data even though the size of the cache is small. Therefore it is impossible to avoid all cache misses. But we can design our program in a way compatible with the prefetching strategy to increase the possibility that the data to be accessed has been put into the cache during the previous steps. A prefetching strategy widely used in modern machines is to select a continuous block of memory including the data being required. This is based on the assumption that most of the time memory accesses are local. To work with this strategy, we need to optimise the locality of our program's memory accesses.

Pointer redirecting usually accesses a data item stored remotely. Therefore, to maximize cache hits pointer references should be minimized. As an example, in Table 7.3, the design of $Formula\_instance_1$ is better than the design of $Formula\_instance_2$ from the cache efficiency point of view, because the sub-structures of $Formula\_instance_1$ are contiguous while the sub-structures of $Formula\_instance_2$ are probably not.

For the same reason, at the level of data structures, arrays are more favourable

```
struct Formula_instance_1
{
  struct Formula formula;
  struct World world;
}
```

```
struct Formula_instance_2
{
  struct Formula *formula;
  struct World *world;
}
```

Table 7.3: Two designs of different cache efficiency

than lists, because arrays consist of contiguous memory locations while lists usually contain memory locations far away from each other. But lists have their own advantages and at the level of program design, in some scenarios lists are better choices. There are adjustments that can be made to improve cache efficiency. The following is an example. In some implementations of lists, there is no counter indicating how many elements have been stored. To check if a list is empty, one usually uses two pointers, one pointing to the beginning of the list the other pointing to the end of the list. If they are pointing to the same address, the list is empty otherwise it is not empty. During processing two pointer redirections are involved. We can improve this empty check operation by using a counter, which has an initial value 0. Whenever a deletion or insertion happens, the counter is updated. Then an empty check can be reduced to checking if the counter equals 0.

Another method we applied to improve the cache performance is using a buffer in the lists. This mechanism is based on a phenomenon that some data stored in a list is more frequently accessed than the others. Data accessed recently has a copy in a buffer which is allocated locally. When data is to be accessed, before looking through the list we first try to find it in the buffer. When the size of the buffer is more than 1, a least recently used policy (LRU) is applied to replace the data in the buffer.

To see how much improvement our cache optimization can make, we ran a few tests on the problem sets we used previously. In those tests we not only measured CPU time but also collected L1 and L2 cache misses using *Valgrind*. By doing that we get a clearer view of how cache misses and CPU time performance are connected.

In Figure 7.12*a* and 7.12*b* we give the 50% percentile and 100% percentile CPU time measured before and after cache optimization. From that figure we can see that there are improvements in the middle part of the diagram for the problems starting from 50 clauses to 100 clauses. There are no significant differences on

the remaining part of the diagram except that for the problems with 30 and 105 clauses the performance is slower than before the optimization in the order of 10^{-3} and 10^{-2} seconds respectively.

From Figure 7.12*a* and 7.12*b* we can conclude that the cache optimization we applied improved the average performance of some of the medium sized problems while cache optimization improved the worst performance of most of the problem sets.

Figure 7.12*a* is used as a reference in Figure 7.12*c* and 7.12*e*. For each group of formulae with certain clauses, we measured the L1 misses and L2 misses of the particular formula that had an everage CPU time in the group according to Figure 7.12*a*. Using Figure 7.12*b* as a reference, we get Figure 7.12*d* and 7.12*f*. That is for each group of formulae with certain clauses, we measured the L1 misses and L2 misses of the particular formula that had the worst CPU time in the group in Figure 7.12*b*.

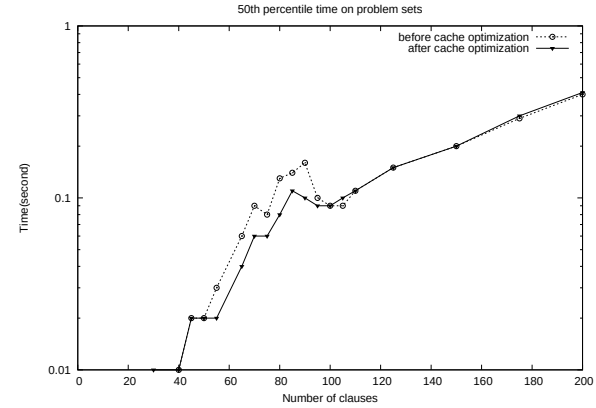
From those figures we can see that the change of the CPU time is in accordance with the change of cache misses. Wherever there is a improvement of runtime, there is a decrease of L1 and L2 cache misses and vice versa. Because of the different implementation of L1 and L2 cache, the extra time required by an L1 miss is much less than for an L2 miss. Therefore reducing the L2 misses is more important for improving the performance of our system.

Figure 7.13 is generated in a similar way as Figure 7.12. In Figure 7.13 *a* and 7.13*b* we give the 50% and 100% percentile L2 misses measured before and after cache optimization. Figure 7.13*a* is used as a reference in Figure 7.13*c* and 7.13*e*. For each group of formulae with certain clauses, we measured the L1 misses and CPU time of the particular formula that had an everage L2 misses in the group according to Figure 7.13*a*. Using Figure 7.13*b* as a reference, we get Figure 7.13*d* and 7.13*f*. That is for each group of formulae with certain clauses, we measured the L1 misses and CPU time of the particular formula that had the worst L2 misses in the group in Figure 7.13*b*. The results in Figure 7.13 agree with the conclusion we made from Figure 7.12.

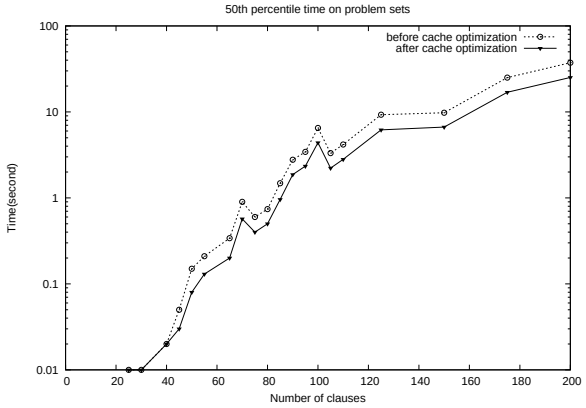
Figure 7.12 and 7.13 together support the conclusion that our cache optimization methods reduced cache misses, and hence improved system performance. It seems that the optimization works especially well on the hard problems.

This is the start rather than the end of the story. We applied a single tier buffer on the fundamental data structure level. There are other techniques available

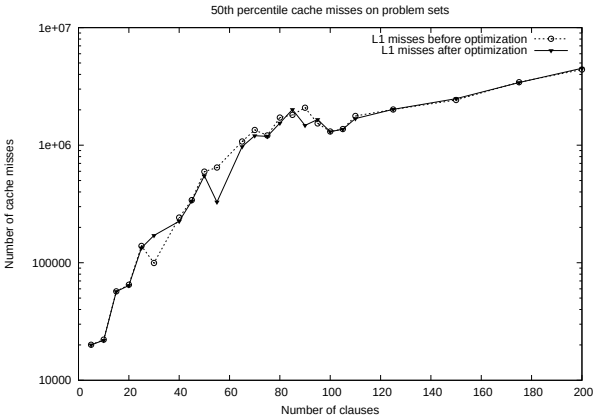
to improve cache performance. Some of them are variants of the technique we used. Firstly, buffers of different sizes may have different effect on the system performance. Secondly, the buffers can have a hierarchy. That is, having more than one tier, different tiers of buffer use different techniques and have different access time and miss penalties like the L1 and L2 caches in the CPU. Thirdly, cache optimization can be applied to different levels of a system. The effects of these techniques are still to be investigated.



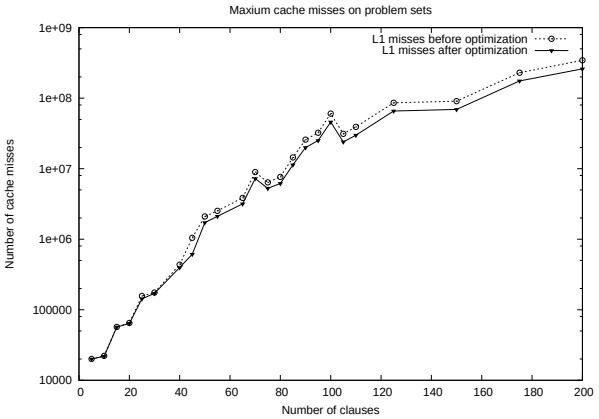
a) 50% percentile CPU time



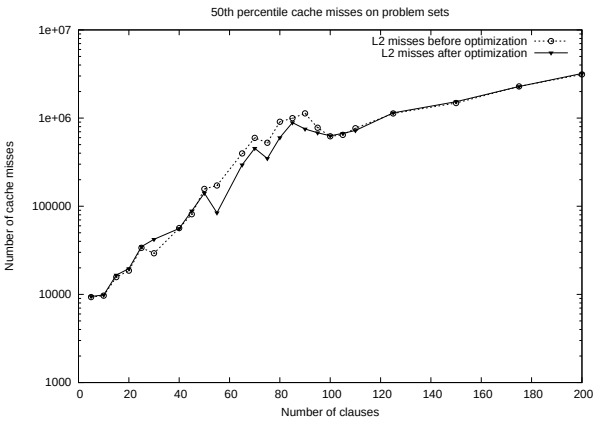
b) 100% percentile CPU time



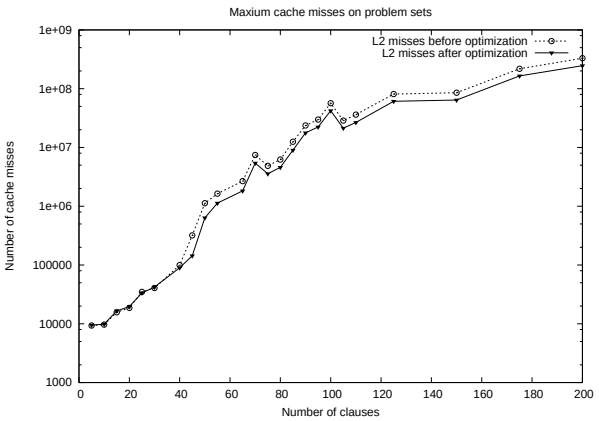
c) L1 misses



d) L1 misses

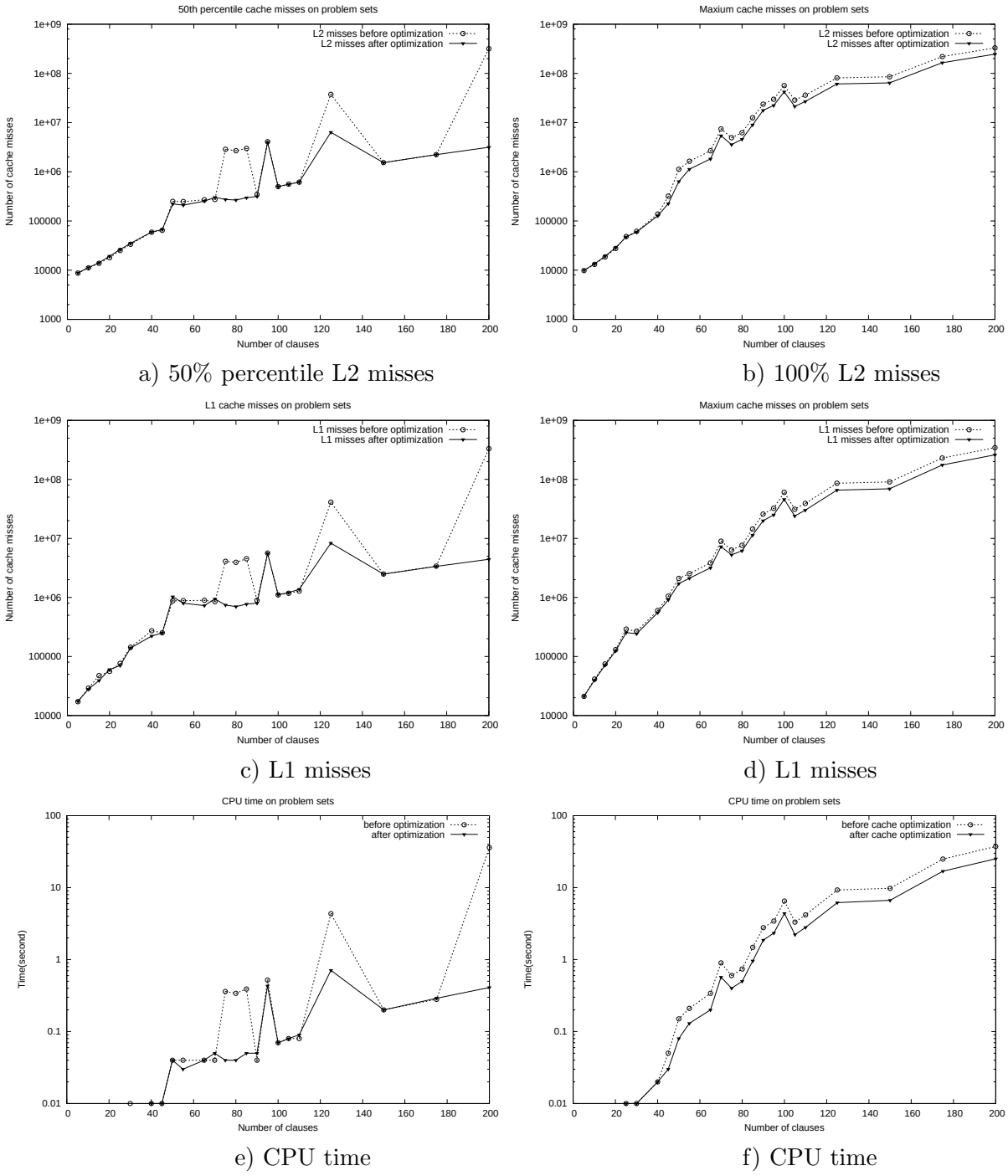


e) L2 misses



f) L2 misses

Figure 7.12: CPU time: before optimization vs. after optimization



a) 50% percentile L2 misses

b) 100% L2 misses

c) L1 misses

d) L1 misses

e) CPU time

f) CPU time

Figure 7.13: Cache misses: before optimization vs. after optimization

Chapter 8

Conclusion

The objective of this thesis was to investigate and develop both efficient and generic reasoning in modal logic. For this purpose, we developed and implemented MLTP. which is a generic platform for automated reasoning system in modal logic K. MLTP uses a labelled tableau approach. Several optimization techniques for reasoning are incorporated in MLTP. As a generic theorem prover, MLTP incorporates an event-handler framework and flexible data structures, which enable users to customize the system using a specification language.

The optimization techniques implemented in MLTP include simplification, backjumping, dynamic backtracking and forward checking. Empirical experiments have been performed to evaluate these techniques. According to the results, we can draw the following conclusions.

- Simplification can successfully improve the efficiency of solving problems that have redundancies.
- Backjumping significantly improves chronological backtracking.
- Dynamic backtracking is better than backjumping on harder problems.
- Forward checking did not show any obvious improvement over backward checking as expected. On the TANCS-2000 problems, it even performed worse. This is probably because the advantage of forward checking is compromised by its extra computational complexity.
- The overall performance of MLTP is comparable to the state-of-art theorem provers on the Giunchiglia-Sebastiani random problems. But MLTP does not perform as well on the problems that have heavily nested modalities like

the TANCS-2000 problems. It should however be kept in mind that MLTP still lacks some standard optimization mechanisms, for instance, caching of previous subderivations.

MLTP was developed not only to be an efficient theorem prover but also to provide greater flexibility. To achieve this goal, MLTP has incorporated an event-handler framework, a generic data structure and a macro language. The event-handler framework divides the system into loosely related modules so that the behavior of the system can be easily changed by combining different modules. The generic data structure makes it possible to perform different operations in a uniform way. Thus, it is possible for users to define a macro language in order to specify some highly operational techniques like advanced backtracking. With these facilities, MLTP can be easily extended to handle more expressive modal logics.

In summary, the main contributions of this thesis are:

- A dynamic backtracking algorithm for modal logics.
- A forward reasoning mechanism for modal logic, which incorporates forward checking and forward implication.
- The discussion and evaluation of the optimization techniques including simplification, backjumping, dynamic backtracking and forward checking.
- Facilities to support generic reasoning including the event-handler framework, generic data structures and the macro language.
- An investigation of memory cache efficiency.

The work presented in this thesis has suggested several promising directions for further research.

- Incorporate more optimization techniques like caching and complement splitting in MLTP and study their effectiveness.
- Implement forward implication in MLTP and test its performance.
- Improve forward checking and investigate techniques that can help to improve its efficiency.

- Extend MLTP to handle more logics. This would require the addition of a blocking mechanism.
- Develop user-friendly interfaces to make it easier for users to work with MLTP.

Bibliography

[AG03] P. Abate and R. Goré. The tableaux work bench. In *Proc. TABLEAUX 2003*, volume 2796 of *LNAI*, pages 230–236. Springer, 2003.

[Bak95] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.

[BDRV01] P. Blackburn, M. De Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[BHN+92] F. Baader, B. Hollunder, B. Nebel, H. Profitlich, and E. Franconi. An empirical analysis of optimization techniques for terminological representation systems, or making KRIS get a move on. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 270–281, Morgan-Kaufmann, 1992.

[Cat91] L. Catach. TABLEAUX: A general theorem prover for modal logics. *Journal of Automated Reasoning*, 7:489–510, 1991.

[CFG+01] L. F. Cerro, D. Fauthoux, O. Gasquet, F. Massacci, A. Herzig, and D. Longin. Lotrec: Generic tableau prover for modal and description logic. In *Proc. IJCAR 2001*, volume 2083 of *LNAI*, pages 453–458. Springer, 2001.

[Dec89] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1989.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Fit83] M. C. Fitting. *Proof Methods for Modal and Intuitionistic Logics*. Library of Congress Cataloging in Publication Data, 1983.

[Fit90] M. C. Fitting. *First order logic and automated theorem proving*. Berlin: Springer-Verlag, Berlin, 1990.

[Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[Gas79] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[GG02] D. Gabbay and F. Guenthner. *Handbook of philosophical logic*. Kluwer Academic Publishers, 2002.

[GL85] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *J. ACM*, 32(2):280–295, 1985.

[GM94] M. L. Ginsberg and D. A. McAllester. Gsat and dynamic backtracking. In P. Torasso, J. Doyle, and E. Sandewall, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237. Morgan Kaufmann, 1994.

[Gor99] R. Goré. Tableau methods for modal and temporal logics. In M. D'Agostino, D. Gabbay, R. Haehnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Kluwer, 1999.

[Gov95] G. Governatori. Labelled tableaux for multi-modal logics. In P. Baumgartner, R. Hhnle, and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *LNAI*, pages 79–94, Berlin, 1995. Springer-Verlag.

[GS96] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 304–314. Morgan Kaufmann, San Francisco, California, 1996.

[HC96] E. G. Hughes and J. M. Cresswell. *A new introduction to modal logic*. Routledge, 1996.

[HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HJSS96] A. Heuerding, G. Jäger, S. Schwendimann, and M. Seyfried. The Logics Workbench LWB: A snapshot. *Euronmath Bulletin*, 2(1):177–186, 1996.

[Hla00] J. Hladik. Implementing the n-ary description logic GF1-. In *Proceedings of the International Workshop in Description Logics 2000 (DL2000)*, Aachen, Germany, 2000.

[HM01] V. Haarslev and R. Möller. RACER system description. In *Proc. IJCAR 2001*, volume 2083 of *LNAI*, pages 701–706. Springer, 2001.

[Hor97] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, The University of Manchester, 1997.

[Hor00] I. Horrocks. Benchmark analysis with fact. In *TABLEAUX*, pages 62–66, 2000.

[HPS98] I. Horrocks and P. F. Patel-Schneider. Comparing subsumption optimizations. In *Proc. of the 1998 Description Logic Workshop (DL'98)*, volume 11 of *CEUR (`http://ceur-ws.org/`)*, pages 90–94, 1998.

[HPS02] I. Horrocks and P. F. Patel-Schneider. Evaluating optimised decision procedures for propositional modal $\mathbf{K}_{(\mathbf{m})}$ satisfiability. *J. of Automated Reasoning*, 28:173–204, 2002.

[HS97] U. Hustadt and R. A. Schmidt. On evaluating decision procedures for modal logics. In M. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, volume 1, pages 202–207. Morgan Kaufmann, August 1997.

[HS98] U. Hustadt and R. A. Schmidt. Simplification and backjumping in modal tableau. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference,*

TABLEAUX'98, Oisterwijk, The Netherlands, Proceedings, volume 1397 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1998.

[HS00] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Artificial Intelligence*, pages 67–71. Springer, 2000.

[Kri63] S. Kripke. Semantical considerations on modal logic. *Acta philosphica fennica*, 16:83–94, 1963.

[Lad77] R. E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

[Mac87] A. K. Mackworth. Constraint satisfaction. *Encyclopedia of Artificial Intelligence*, pages 205–211, 1987.

[MB04] W. Minker and S. Bennacef. *Speech and Human-Machine Dialog.* Springer, 2004.

[MD00] F. Massacci and F. M. Donini. Design and results of tancs-2000 non-classical (modal) systems comparison. In *TABLEAUX*, pages 52–56, 2000.

[MDAP99] R. Hahnle M. D. Agostino, D. M. Gabbay and J. Posegga. *Handbook of tableau methods.* Kluwer Academic, 1999.

[Mey03] J. J. C. Meyer. Modal epistemic and doxastic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic (2nd edition)*, volume 10, pages 1–38. Kluwer, 2003.

[MvdH95] J. J. C. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science.* Cambridge University, 1995.

[OK96] J. Otten and C. Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. In *Proceedings of TABLEAUX-96*, Lecture Notes in Artificial Intelligence. Springer, 1996.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[PS98] P. F. Patel-Schneider. DLP system description. In *Proc. Description Logics*, volume 98 of *CEUR Workshop Proceedings*, pages 87–89, 1998.

[PS00] P. F. Patel-Schneider. Tancs-2000 results for dlp. In *TABLEAUX*, pages 72–76, 2000.

[PSH99] P. F. Patel-Schneider and I. Horrocks. DLP and fact. In *Analytic Tableaux and Related Methods*, pages 19–23, 1999.

[RU71] N. C. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, New York, 1971.

[Sch97] R. A. Schmidt. *Optimised Modal Translation and Resolution*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[Sch98] R. A. Schmidt. *E*-unification for subsystems of S4. In T. Nipkow, editor, *Rewriting Techniques and Applications: 9th International Conference, RTA'98, Proceedings*, volume 1379 of *Lecture Notes in Computer Science*, pages 106–120, Tsukuba, Japan, 1998.

[Sch03] R. A. Schmidt. *pdl-tableau*. http://www.cs.man.ac.uk/ schmidt/pdl-tableau/, 2003.

[SG95] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. *Proceedings of IJCAI-95*, pages 646–651, 1995.

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[TH06] D. Tsarkov and I. 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.

[Vor95] A. Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

Index