MODAL SATISIFIABILITY IN A CONSTRAINT LOGIC ENVIRONMENT

by

LYNETTE STEVENSON

Submitted in fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR : PROF K. BRITZ
JOINT SUPERVISOR : Mrs T. HöRNE

November 2007

SUMMARY

The modal satisfiability problem has to date been solved using either a specifically designed algorithm, or by translating the modal logic formula into a different class of problem, such as a first-order logic, a propositional satisfiability problem or a constraint satisfaction problem. These approaches and the solvers developed to support them are surveyed and a synthesis thereof is presented.

The translation of a modal K formula into a constraint satisfaction problem, as developed by Brand et al. [18], is further enhanced. The modal formula, which must be in conjunctive normal form, is translated into layered propositional formulae. Each of these layers is translated into a constraint satisfaction problem and solved using the constraint solver ECL^iPS^e. I extend this translation to deal with reflexive and transitive accessibility relations, thereby providing for the modal logics KT and $S4$. Two of the difficulties that arise when these accessibility relations are added are that the resultant formula increases considerably in complexity, and that it is no longer in conjunctive normal form (CNF). I eliminate the need for the conversion of the formula to CNF and deal instead with formulae that are in negation normal form (NNF). I apply a number of enhancements to the formula at each modal layer before it is translated into a constraint satisfaction problem. These include extensive simplification, the assignment of a single value to propositional variables that occur only positively or only negatively, and caching the status of the formula at each node of the search tree. All of these significantly prune the search space. The final results I achieve compare favorably with those obtained by other solvers.

Key terms

Modal satisfiability, modal validity, constraint satisfaction problem, constraint solver, tableau system, first-order translation, ECL^iPS^e, modal logics K, KT, $S4$, constraint logic programming

Table of Contents

List of Tables

Acknowledgements

I would like to express my sincere thanks and appreciation to the following people:

- My husband, for all his encouragement and the many sacrifices he has had to make while I was studying.
- My supervisor, Professor Arina Britz, for her ideas, expertise and guidance.
- My co-supervisor, Tertia Hörne, for her suggestions, recommendations and her attention to detail.

  Thank you both for the hours you spent assisting me and for your encouragement and support.
- Sebastian Brand, for kindly letting me have the code to his KCSP solver.

Chapter 1

Introduction

Modal logic was originally conceived as the logic of *necessary* and *possible* truths and hence can be used to qualify the truth condition of statements [48]. From these philosophical beginnings, modal logic has expanded into many different application areas which include mathematics, linguistics, computer science and artificial intelligence. This makes it a very diverse field and it appears as:

- a logic of necessity and possibility,
- a language for studying provability and expressibility in various formal theories,
- a language for talking about relational and topological structures and their uses in computer science,
- a knowledge representation formalism,
- a language for talking about the behavior of programs,
- a formalism for representing linguistic meaning,

among others [134]. Because of its ability to deal with relational structures, it has been applied in areas such as program verification, hardware verification, database theory and distributed computing. It has lately found application on the Semantic Web.

In this dissertation, I look specifically at the modal satisfiability problem, which is the problem of determining whether or not a modal formula is satisfiable. This class of problem is decidable, which means that there exists an algorithm which will always determine whether or not a formula is satisfiable. However, the algorithm is **PSPACE**-complete, and so it can, in the worst case, take exponential time to produce a solution and can require polynomial space to do so. Hence, this is by no means a simple problem to solve.

I begin by looking at the automated approaches that have been developed to date to solve the modal satisfiability problem. These approaches fall into two distinct categories. Either a special, purpose-build algorithm has been developed, or the modal formula has been translated into a *different* class of problem and the highly optimized solvers of that class have been applied to solve it. The most widely used purpose-built algorithms are the tableau approach and the Gentzen sequent calculus. Modal formulae have been translated into a first-order logic, a propositional satisfiability problem or a constraint satisfaction problem, among others.

The translation into a *constraint satisfaction problem* is limited to the modal logic K and only deals with formulae that are in conjunctive normal form (CNF). Brand et. al [18] proposed an approach that translates a modal formula into a layered set of constraint satisfaction problems. Each layer is solved using the state-of-the-art constraint solver, ECL^iPS^e [130]. This approach returns good results and hence is a good candidate for further development.

I extend this solver to deal with reflexive and transitive accessibility relations, on which the modal logics KT and $S4$ are based. When these properties are added to an existing tableau or sequent solver for the first time, the unmodified algorithm loops within worlds in the case of reflexivity, and produces infinite loops in the case of a transitive accessibility relation. The various solutions developed to overcome these problems are identified and discussed.

In order to adequately test the performance of my solver, suitable test data sets are required. I identify the Heuerding / Schwendimann benchmark data sets as the most appropriate. They consist of 9 different classes of valid and non-valid problems of increasing complexity.

The first solver I develop deals with reflexivity. When reflexivity is applied to a modal formula, the complexity of the clauses increases and conjunctive normal form is lost. If the resultant formula is converted back into conjunctive normal form, its complexity increases further, as the number of additional clauses generated is an exponential factor of the number of original positive modal literals. I propose and implement two prototypes – one in which the formulae are maintained in conjunctive normal form and one in which they are not.

The initial prototypes are essentially the original solver for the modal logic K, with the addition of reflexivity. The initial results are not good and so I propose a series of enhancements. These enhancements focus on the simplification of the propositional formula at each modal layer *before* the constraint satisfaction problem is generated. Brand et al. [18] assign a value from the domain $\{0,\ 1,\ u\}$ to each propositional literal, where u is taken to mean that an actual value has not been assigned to the literal. I took this concept further as follows. If a propositional literal occurs only positively or only negatively in the propositional formula of a particular modal layer, it can be assigned a value of 1 or 0 respectively. This means that the clauses in which the literal occurs are $True$, and so do not need to be included in the translation into a constraint satisfaction problem. This significantly prunes the search space. Another major enhancement involves caching the status of the formula at each node, thereby avoiding unnecessary reprocessing – it was found that in some cases, the same formula occurs on many nodes of the tree. The application of these and other enhancements returns favorable results. Of the two prototypes, the one in which the formula is *not* maintained in conjunctive normal form returns better results.

The second solver I develop deals with reflexive and transitive accessibility relations. The application of the rules of reflexivity and transitivity result in even more complex formulae, which are once more not in conjunctive normal form. In this case, the conversion back to CNF is prohibitive, as far too many additional clauses are generated. The initial prototype does not return good results – in fact, for one of the classes of data sets, it is able to solve only one non-valid and two valid problems. Further simplification returns better results. Firstly, simplification is applied *before* the application of reflexive and transitive accessibility rules. Secondly, simplification is applied more extensively. Modal clauses are no longer in conjunctive normal form, but are in negation normal form. This means that, within a clause, one can have a disjunction which includes a modal formula. Simplification is now applied to these modal formulae, which was not the case previously. This prototype returns good results which compare favorably with those of existing solvers for which benchmark data is available, although it is not the best of these.

The data structures used prohibit the implementation of any further optimizations. This is however what one would expect from a prototype which has been extended and extended again, and which was not originally designed for the functionality which has been added. To improve these results, the prototypes need to be either rewritten or converted into an alternate constraint-based package, with the focus being on the optimization of the data structures.

I conclude with a discussion of potential application areas to which the prototypes can be applied.

The remainder of this dissertation is organized as follows. Chapter 2 provides a formal overview of modal logic and its computational complexity. It provides an overview of three application areas of modal logics, these being the modal logics of knowledge and belief, description logic and modal temporal logic. Chapter 3 provides a detailed overview of automated approaches and solvers developed to solve the modal satisfiability problem. Chapter 4 discusses the reflexive and transitive prototypes I have developed and their enhancements and implementation. Chapter 5 concludes the dissertation with a discussion of future research areas.

Chapter 2

An overview of modal logic

This chapter aims at providing an understanding of what modal logic is. It describes the type of problems that can be solved using modal logic and discusses the complexity of providing suitable algorithms to solve them.

After a brief introduction, we formally define modal logic from a syntactic and semantic point of view. There are many different types of modal logic. We provide details of the basic normal modal logics and their axioms, as these form the building blocks from which more complex logics can be built. We look at an example of a simple modal formula and a syntactic and semantic proof of its validity, to gain an insight into the difference between these two approaches. We will focus on two of these normal logics, namely the modal logics *KT* and *S4*, in the chapters which follow.

We then look at some of the problems that can be solved using modal logic. Of these, we select the *modal satisfiability problem*, which is the problem of determining whether or not a modal formula is satisfiable, for further study. We need a means of measuring the complexity of this problem and so look into *computational complexity theory*. Computational complexity theory defines how to categorize problems in terms of the CPU time and memory requirements of an algorithm – it measures whether a problem *can be solved*, regardless of the resources it will require to do so.

The modal satisfiability problem is *decidable*, meaning that an effective algorithm exists that is able to determine whether any given formula is satisfiable or unsatisfiable. However, its complexity class tells us that it will take exponential time in the worst case to provide a solution. Hence, this is a complex problem to solve and we cannot in general expect quick answers.

In the chapters which follow, we will refer to certain characteristics of modal logic, in particular its relation to first-order logic, its decidability and its tree model property. These are briefly discussed.

We conclude this chapter by looking at three application areas of modal logic. We begin with a brief overview of the modal logics of knowledge and belief. We look in particular at description logics, a family of knowledge representation languages that are closely allied to modal logic. We make this choice for two reasons. Firstly, the type of problem that can be expressed in description logic is intuitive and easy to understand. Secondly, the next chapter looks at solvers that have been developed to solve the modal satisfiability problem. Much of the development work that has taken place in the description logic arena is applicable to modal logic because of a strong correspondence between the two – and visa versa. The final application area we consider is modal temporal logic. We look at the temporal constraint satisfaction problem in some depth and its solution using Allen's interval algebra [3]. A translation of the interval algebra into a modal temporal logic has been defined [41]. It is of interest to us as this modal temporal logic requires a transitive accessibility relation. The prototype we develop in chapter four implements transitivity and so could potentially be applied to solve this class of problem.

By the end of this chapter, we will have gained a good insight into modal logic, its complexity and the type of problems it can solve. This leads us to the next chapter, where we will study the solvers that have been developed to date to solve the modal satisfiability problem.

The reader is referred to comprehensive texts such as Chellas [21], Goldblatt [47] and more recently Blackburn et al. [16] for in-depth details of modal logic. A good overview of the history of modal logic is provided in [48].

2.1 What is modal logic?

Modal logic was originally developed in philosophy to distinguish between two different modes of truth – it was conceived as the logic of *necessary* and *possible* truths [48]. Nowadays computer scientists use modal logic to reason about subjects such as knowledge, belief and time. Modal logic can be viewed as an extension of propositional logic to which *modalities* have been added – instead of a proposition being just true or false, it may in addition be *necessarily true* or *possible true.*

Propositional logic allows reasoning about statements. The smallest units of reasoning are *atomic propositions* – denoted p, p_1, …, q, q_1, …. Atomic propositions can be combined to form *compound propositions* using *connectives* such as *not*, *or* and *and*, which are represented symbolically as $\neg$, $\vee$ and $\wedge$ respectively. Examples of compound propositions are $\neg p$, $(p \vee q)$ and $(p \wedge q)$, also referred to as *well-formed formulae* (wffs).

To this framework, the two modalities *necessarily* ($\Box$) and *possibly* ($\Diamond$) are added. They are interpreted differently in different modal logics. For example, in modal temporal logic, they are used to express statements such as 'φ will *always* be true in the future' or 'φ will be true *sometime* in the future'. In the logic of knowledge and belief, they are used to reason about what is *known* and what is *believed*.

Let us consider the modality 'always true' further. Depending on the context, 'always true' could mean 'true in the future' or 'true at all worlds' or 'true at all states'. Similarly, 'possibly true' could mean 'true at some non-specific moment(s) in time' or 'true at some world(s)' or 'true at some state(s)'. In the case of time, the relationship between moments in time is the natural chronological order. If we have worlds or states u, w and v, their relationship is *accessibility*, stating which world or state can be accessed from another. We could, for example, have that u can access v and w, whereas v and w cannot be accessed from each other. Various modal logics have been developed that allow reasoning about truth from the perspective of different moments in time, worlds or states, using an appropriately defined accessibility relation.

These concepts were formalized in the 1950s by Kripke [78], although the theory was developed by several authors including Hintikka [62]. Kripke gave a precise mathematical meaning to the notion of modalities in terms of possible world models and provided a precise syntax and semantics for modal logics, as we shall see in the next section.

2.2 The formal definition of modal logic

We now provide a formal overview of the syntax and semantics of modal logic.

Definition 2.2.1. *The basic modal language is defined using a set Φ of atomic propositions whose elements are denoted p, p_1, ..., q, q_1, ..., the propositional connectives $\neg$ and $\wedge$, and the unary modality $\Box$. The set of well-formed formulae generated from Φ, denoted Fma(Φ), is generated by the rule*

$$\varphi ::= p \mid \bot \mid \neg\varphi \mid \varphi \wedge \psi \mid \Box\varphi$$

where p ranges over the elements of Φ, $\perp$ is the falsum and φ, $\psi, \ldots$ are modal formulae.

This definition means that a formula is either a propositional letter, the propositional constant falsum, a negated formula, a conjunction of formulae, or a formula prefixed by the box modality.

Additional connectives that are commonly used are $\vee$ (or), $\rightarrow$ (implies) and $\leftrightarrow$ (if and only if). These connectives are defined in terms of $\neg$ and $\wedge$ – that is, $\neg(p \vee q) \equiv \neg p \wedge \neg q$; $p \rightarrow q \equiv \neg p \vee q$ and $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$. An additional unary modal connective $\Diamond$ is defined as $\Diamond \equiv \neg\Box\neg$. Whenever a collection of formulae has a common syntactic form, this form can be represented as an *axiom schema* or *axiom*. For example, the axiom schema $\Box\varphi \rightarrow \varphi$ denotes the collection of formulae $\{\Box\psi \rightarrow \psi : \psi \in Fma(\Phi)\}$ and has the meaning that 'if φ is necessarily true, then φ is true'.

The next step is to formally define the semantics of modal logic. Kripke semantics (also known as *relational semantics* or *frame semantics*) defines the semantics of modal logic in terms of *relational models* and *frames* or *structures* and is defined as follows:

Definition 2.2.2. *A Kripke frame is defined as a tuple $\mathcal{F} = (W, R)$. A Kripke model* (*or a modal model*) *is defined as a tuple $\mathcal{M} = (W, R, V)$, where W is a nonempty set of worlds or states denoted u, u_1, ..., v, v_1, ..., w, w_1 ...; R is a binary accessibility relation over W, and V is a function or interpretation that associates with each propositional letter in Φ a set of worlds in W.*

The interpretation $V(p)$ tells us at which worlds a propositional letter p is true, and the binary accessibility relation uRv tells us that world v is accessible from world u.

We next define what it means for a formula φ to be true at a given world u in a model $\mathcal{M}$.

Definition 2.2.3. *The statement 'φ is true at world u in model $\mathcal{M}$', denoted $\mathcal{M} \models_u \varphi$, is defined inductively as follows:*

$\mathcal{M} \models_u p$	*iff* $u \in V(p)$ *and* $p \in \Phi$
$\mathcal{M} \models_u \neg\varphi$	*iff not* $\mathcal{M} \models_u \varphi$
$\mathcal{M} \models_u (\varphi \wedge \psi)$	*iff* $\mathcal{M} \models_u \varphi$ *and* $\mathcal{M} \models_u \psi$
$\mathcal{M} \models_u \Box\varphi$	*iff for all* $v \in W$, uRv *implies* $\mathcal{M} \models_v \varphi$ *– that is,* φ *holds at all worlds* v *accessible from* u
$\mathcal{M} \models_u \Diamond\varphi$	*iff* $\mathcal{M} \models_v \varphi$ *for some* $v \in W$ *such that* uRv

This definition gives us the truth of a formula at a particular world. The next step is to define the *satisfiability* and *validity* of a formula in a model $\mathcal{M}$ and a frame $\mathcal{F}$.

Definition 2.2.4. *A formula* φ *is satisfiable in a frame* $\mathcal{F} = (W, R)$ *if there exists some Kripke model* $\mathcal{M} = (W, R, V)$ *based on* $\mathcal{F}$ *such that for some* $u \in W$, $\mathcal{M} \models_u \varphi$.

Definition 2.2.5. *A formula* φ *is valid in a model* $\mathcal{M} = (W, R, V)$*, denoted* $\mathcal{M} \models \varphi$*, iff* $\mathcal{M} \models_u \varphi$ *for all worlds* u *in* W*. A formula is valid in a frame* $\mathcal{F} = (W, R)$ *iff it is valid in all models* $\mathcal{M}$ *based on* $\mathcal{F}$*. It can also be valid in a class of frames or in all frames.*

The above concepts are illustrated in the following example.

Example 2.2.6. Consider the set $\Phi = \{p,\, q,\, r\}$ of propositions. Let $W = \{w_1,\, w_2,\, w_3,\, w_4\}$ and $R = \{w_1Rw_2,\, w_1Rw_3,\, w_3Rw_4\}$. Let $V_1(p) = \{w_1,\, w_2,\, w_3\}$, $V_1(q) = \{w_3\}$, $V_1(r) = \{w_2,\, w_4\}$ and $\mathcal{M}_1 = (W, R, V_1)$. Let $V_2(p) = \{w_1,\, w_2,\, w_3,\, w_4\}$, $V_2(q) = \{w_3\}$, $V_2(r) = \{w_2,\, w_4\}$ and $\mathcal{M}_2 = (W, R, V_2)$.

$\mathcal{M}_1$ and $\mathcal{M}_2$ are represented graphically in Figure 2.1.

We can state the following in terms of Definition 2.2.3:

1. For $\mathcal{M}_i$, $i = 1,\, 2$, the following hold: $\mathcal{M}_i \models_{w_1} \Box p$, $\mathcal{M}_i \models_{w_3} \Box r$ and $\mathcal{M}_i \models_{w_1} \Diamond q$. In terms of Definition 2.2.4, the modal formulae $\Box p$, $\Box r$ and $\Diamond q$ are thus *satisfiable.*

2. For $\mathcal{M}_i$, $i = 1,\, 2$, the following does not hold : $\mathcal{M}_i \models_{w_1} \Box q$.

3. For $\mathcal{M}_2$, we have $\mathcal{M}_2 \models_{w_3} \Box p$ which is not the case in $\mathcal{M}_1$.

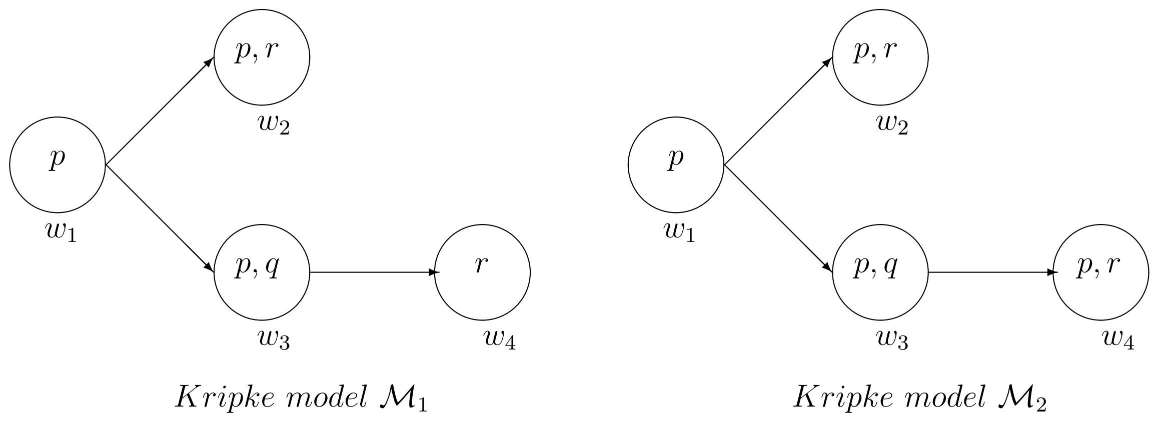


Figure 2.1: A simple example of Kripke models

4. In terms of Definition 2.2.5, the formula p is *valid* in $\mathcal{M}_2$, as $\mathcal{M}_2 \models_u p$ for all worlds u in W. This formula is however not valid in $\mathcal{M}_1$.
5. In terms of Definition 2.2.5, the formula p is *not valid* in frame $\mathcal{F} = (W, R)$, as it is not valid in the model $\mathcal{M}_1$. $\dashv$

2.2.1 Normal forms

To simplify the processing of a modal formula, a standard practice is to first reduce it to a standard representation, called *normal form* or *canonical form*, in which the number of connectives in the formula is reduced to a minimal subset.

The following definitions are applicable and will be used throughout the work that follows.

Definition 2.2.7. *A propositional atom is any propositional formula that cannot be decomposed propositionally.*

Definition 2.2.8. *A modal atom is any modal formula that cannot be decomposed propositionally – that is, any formula whose main connective is not propositional.*

Definition 2.2.9. *A propositional literal is either a propositional atom or its negation. A positive propositional literal is a propositional atom; a negative propositional literal is a propositional atom with a single negation.*

Definition 2.2.10. *A modal literal is either a modal atom or its negation. A positive modal literal is a modal atom; a negative modal literal is a modal atom with a single negation.*

Definition 2.2.11. *A unit modal literal is either of the form $\Box l$ or $\neg\Box l$, where l is a propositional literal.*

A unit modal literal is a modal literal; whereas a modal literal is not necessarily a unit modal literal.

Definition 2.2.12. *A propositional clause is composed of the disjunction of propositional literals (the literals are logically OR-ed).*

Definition 2.2.13. *A modal clause is composed of the disjunction of propositional and / or modal literals.*

Definition 2.2.14. *A propositional unit clause is a clause that is composed of a propositional literal.*

Definition 2.2.15. *A modal unit clause is a clause that is composed of a single modal literal.*

Definition 2.2.16. *The depth of a modal formula φ, written depth(φ), is defined as the maximum number of nested modal operators in φ.*

Example 2.2.17. The definitions above are clarified in the following:

- p is a propositional atom; whereas $\neg p$ is not.
- p and $\neg p$ are propositional literals; p is a positive propositional literal or propositional atom; $\neg p$ is a negative propositional literal.
- $\Box\Box p$ and $\Box(p \vee q)$ are modal atoms; whereas $\neg\Box p$ is not.
- $\Box(p \vee q)$ and $\neg\Box(p \vee q)$ are modal literals; $\Box(p \vee q)$ is a positive modal literal or modal atom; $\neg\Box(p \vee q)$ is a negative modal literal.
- $\Box p$ is a unit modal literal; whereas $\Box(p \vee q)$ is not – it is a modal unit clause.
- In the modal formula

$$\Box\Box p \wedge (p_1 \vee \neg q) \wedge p_2 \wedge \Box(p_3 \vee p_4)$$

$(p_1 \vee \neg q)$ is a propositional or modal clause with no modal literals and hence depth 0; p_2 is a propositional unit clause; $\Box(p_3 \vee p_4)$ is a modal unit clause; $\Box\Box p$ and $\Box(p_3 \vee p_4)$ have modal depths 2 and 1 respectively. The depth of this modal formula is thus 2. ⊣

We have stated that, prior to processing a modal formula, it needs to be reduced to a normal form.

Definition 2.2.18. *A modal formula φ is in negation normal form (NNF) if negation occurs only immediately before propositional and modal atoms and the only Boolean connectives it contains are $\{\neg, \wedge, \vee\}$.*

This reduction is achieved by applying the following rule:

Rule 2.2.19. *A modal formula is reduced to negation normal form (NNF) as follows:*

- *Double-negations are eliminated and negations occur only immediately before atoms by the application of:*

$$\begin{array}{lll} (i) & \neg\neg\varphi_1 \equiv \varphi_1 & (\textit{double negation}) \\ (ii) & \neg(\varphi_1 \vee \varphi_2) \equiv (\neg\varphi_1 \wedge \neg\varphi_2) & (\textit{De Morgan's law}) \\ (iii) & \neg(\varphi_1 \wedge \varphi_2) \equiv (\neg\varphi_1 \vee \neg\varphi_2) & (\textit{De Morgan's law}) \end{array}$$

- *The Boolean connectives equivalence ($\leftrightarrow$) and implication ($\rightarrow$) are replaced with their definitions:*

$$\begin{array}{ll} (iv) & (\varphi_1 \leftrightarrow \varphi_2) \equiv (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2) \\ (v) & (\varphi_1 \rightarrow \varphi_2) \equiv (\neg\varphi_1 \vee \varphi_2) \end{array}$$

Note that the clauses in an NNF modal formula can consists of the disjunction of propositional literals, modal literals and / or modal formulae.

Example 2.2.20. An example of a formula in negation normal form (NNF) is

$$p_1 \wedge (p_2 \vee (\neg p_3 \wedge p_4 \wedge (\Box(p_5 \vee p_6))))$$

whereas the formula

$$p_1 \leftrightarrow (\neg\neg p_2 \rightarrow (\neg p_3 \wedge p_4))$$

is not. Clauses p_1 and $(p_2 \vee (\neg p_3 \wedge p_4 \wedge (\Box(p_5 \vee p_6))))$ of the first formula are NNF clauses. $\dashv$

Definition 2.2.21. *A modal formula is in conjunctive normal form (CNF) if it is a conjunction of modal clauses (Definition 2.2.13). Conjunction normal form is achieved by applying the following law of distribution:*

$$\varphi \vee (\varphi_1 \wedge \ldots \wedge \varphi_n) \equiv (\varphi \vee \varphi_1) \wedge \ldots \wedge (\varphi \vee \varphi_n)$$

Example 2.2.22. An example of a formula in conjunctive normal form (CNF) is

$$(\varphi_1 \vee \varphi_2) \wedge (\varphi_3 \vee \varphi_4) \wedge \varphi_5$$

whereas the formula

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3 \vee \varphi_4)$$

is not.

The formula $(\varphi_1 \vee (\varphi_2 \wedge \varphi_3))$, when converted to conjunctive normal form, becomes $(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$. $\dashv$

Note that a modal formula which is in conjunctive normal form is also in negation normal form. A modal formula which is in negation normal form is not however in conjunctive normal form.

2.3 Basic normal modal logics

The definition of modal logic can be extended in various ways, such as by adding further axioms or by placing restrictions on the accessibility relation R. In this way, deductively stronger modal logics can be defined. For example, we may want R to represent the flow of time, in which case it needs to be transitive. If we have moments in time t_1, t_2 and t_3 and we have t_1Rt_2 (t_2 is in the future of t_1) and t_2Rt_3 (t_3 is in the future of t_2), then we must be able to deduce t_1Rt_3 (t_3 is in the future of t_1). A mechanism is required to formalize such inferences. We next look at how this can be achieved from an axiomatic and semantic perspective.

We begin by providing an axiomatic definition of a propositional logic L.

Definition 2.3.1. *Given a language based on a countable set Φ of atomic formulae, a logic L is defined as any set $\Lambda \subseteq Fma(\Phi)$ that includes all tautologies and is closed under the rule of modus ponens:*

$$\textit{if } \varphi,\ \varphi \rightarrow \psi \in L, \textit{ then } \psi \in L.$$

This definition can be applied to different logics, including modal logics. We now look at the definition of a normal modal logic that will by default have the properties defined above.

Definition 2.3.2. *A modal logic is called normal if it includes the axiom*

$$K : \Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$$

and is closed under the rule of necessitation: from φ infer $\Box\varphi$.

The smallest normal logic is called K and is defined as $K = \bigcap\{\Lambda : \Lambda$ is a normal logic$\}$. K has been named after Kripke. In the next chapter we will see that the modal logic K plays a prominent role in the solvers we will be discussing and some solvers provide for only K.

K can be extended to create the basic normal modal logics, which are obtained from various combinations of the following five basic axioms:

Definition 2.3.3. *The axioms of modal logic include:*

$$\begin{array}{ll}
\mathrm{D} : & \Box\varphi \rightarrow \Diamond\varphi \\
\mathrm{T} : & \Box\varphi \rightarrow \varphi \\
\mathrm{B} : & \varphi \rightarrow \Box\Diamond\varphi \\
4 : & \Box\varphi \rightarrow \Box\Box\varphi \\
5 : & \Diamond\varphi \rightarrow \Box\Diamond\varphi
\end{array}$$

From these five axioms, 15 basic normal logics have been derived, some of the more well-known of which are *KT*, *K4*, *S4* and *S5*. They are obtained as follows:

- *T* or *KT* is obtained by adding the axiom *T* to the logic *K*.

- *K4* is obtained by adding the axiom *4* to the logic *K*.

- *KT4* is obtained by adding the axioms *T* and *4* to the logic *K*. It is also referred to as the logic *S4*.

- *KTB4* is obtained by adding the axioms *T*, *4* and *B* to the logic *K*. It is also referred to as logic *S5* – the modal logic of knowledge.

These axioms can be used to prove the validity of a formula in a given logic. We look at the syntactic proof of the following modal formula, using the axioms of the logic *KT*.

Example 2.3.4. The proof of $\varphi \rightarrow \Diamond\varphi$ in modal logic *KT* is derived as follows:

1.	$\Box\neg\varphi \rightarrow \neg\varphi$	*T*
2.	$\varphi \rightarrow \neg\Box\neg\varphi$	*1, propositional logic (PL)*
3.	$\Diamond\varphi \leftrightarrow \neg\Box\neg\varphi$	*definition of* $\Diamond$
4.	$\varphi \rightarrow \Diamond\varphi$	*2, 3, PL*

⊣

When we look at this proof, we see that it is not intuitive – if we were assigned the task of proving the formula from scratch, it would not be obvious where to begin or which axioms to use. If we could not derive a given formula because it was not a theorem, there would be no guarantee that we would realize there is no proof. Axiomatic systems are notoriously bad for proof search – they give no guidance on either how to look for a proof or how to establish the lack of a proof. Furthermore, their axioms have little intuitive content.

Frames on the other hand are a more natural way of representing a logic. We now look at the semantic representation of the above axioms.

Definition 2.3.5. *The axioms of Definition 2.3.3 can be represented semantically in a frame* $\mathcal{F} = (W, R)$ *as follows:*

- *D is true in* $\mathcal{F}$ *iff R is* serial *– that is, iff for every* $u \in W$ *there is some* $v \in W$ *such that uRv.*

- *T is true in $\mathcal{F}$ iff R is* reflexive *– that is, iff for every $u \in W$, uRu.*

- *4 is true in $\mathcal{F}$ iff R is* transitive *– that is, iff for every u, v, $w \in W$, if uRv and vRw then uRw.*

- *5 is true in $\mathcal{F}$ iff R is* euclidean *– that is, iff for every u, v, $w \in W$, if uRv and uRw then vRw.*

- *B is true in $\mathcal{F}$ iff R is* symmetric *– that is, iff for every $u, v \in W$, if uRv then vRu.*

Hence, the Kripke semantics of K is defined with respect to the set of all frames, KT with respect to the set of all reflexive frames, *S4* with respect to the set of transitive and reflexive frames and *S5* with respect to the set of transitive, symmetric and reflexive frames.

It must be noted that all axioms do not have an equivalent semantic characterization in terms of the accessibility relation, and vice versa. For example, consider the accessibility relation R which satisfies the condition that, for every $u, v \in W$, if uRv, then it is not the case that vRu. There is no modal axiom which corresponds to this R. Similarly, the McKinsey schema, $\Box\Diamond\varphi \rightarrow \Diamond\Box\varphi$, cannot be defined in terms of an accessibility relation [47].

We now look at how to provide a semantic proof for the formula $\varphi \rightarrow \Diamond\varphi$ of Example 2.3.4. First, we need to understand some of the background of semantic proofs. Recall that for a formula to be valid, it must be true in *all* models (Definition 2.2.5). The approach taken to produce such a proof is to show that the *negation* of the formula is false in *all* models.

Example 2.3.6. We need to prove the validity of $\varphi \rightarrow \Diamond\varphi$ in the class of reflexive frames. Suppose we have an *arbitrary model* $\mathcal{M} = (W, R, V)$ based on $\mathcal{F} = (W, R)$. Suppose W contains any number of worlds, including an *arbitrary world* u, and suppose R is reflexive.

We consider the truth of the formula $\neg(\varphi \rightarrow \Diamond\varphi)$ or $\varphi \wedge \Box\neg\varphi$ at the arbitrary world u.

Suppose φ and $\Box\neg\varphi$ are $True$ at u. We have by Definition 2.2.3 that

$$\mathcal{M} \models_u \Box\varphi \text{ iff for all } v \in W,\ uRv \text{ implies } \mathcal{M} \models_v \varphi$$

Since R is reflexive, uRu, and so $\neg\varphi$ is $True$ at u – a contradiction.

Since u and $\mathcal{M}$ were arbitrarily chosen, we have shown that $\varphi \wedge \Box\neg\varphi$ is $False$ at all worlds and all models.

Hence, $\varphi \rightarrow \Diamond\varphi$ is valid. $\dashv$

This proof is more intuitive than the corresponding syntactic proof. We will see in the next chapter that a formula such as this is easily proved using proof methods such as a tableau proof.

The theory presented so far adapts easily to logics with more than one modality, and we extend Definition 2.2.1 accordingly [47].

Definition 2.3.7. *The basic multi-modal language is defined using a set Φ of atomic propositions whose elements are usually denoted p, p_1, ..., q, q_1, ..., the propositional connectives $\neg$ and $\wedge$, and a collection of modalities $\{\Box_i : i \in I\}$. The set of well-formed formulae generated from Φ, denoted $Fma_I(\Phi)$, is generated by the rule*

$$\varphi ::= p \mid \bot \mid \neg\varphi \mid \varphi \wedge \psi \mid \Box_i\varphi$$

where p ranges over the elements of Φ, $\bot$ is the falsum and φ, ψ ... are modal formulae.

We now have formulae $\Box_i\varphi$ for each $\varphi \in \text{Fma}(\Phi)$ and each $i \in I$. The modalities $\Box_i$ are treated in the same way we treated $\Box$ previously, with the dual modalities $\Diamond_i$, defined by $\Diamond_i \equiv \neg\Box_i\neg$, corresponding to $\Diamond$. $K(m)$, also referred to as K_m, is now defined as a *multi-modal logic* with a set of m modal operators $B = \{\Box_1, \Box_2, ..., \Box_m\}$. The Kripke structure for $K(m)$ is a tuple $\mathcal{M} = (W, R_1, R_2, ..., R_m, V)$ where each R_i is a binary relation on the worlds of W. Definition 2.2.3 needs to be extended to include

$$\begin{array}{ll} \mathcal{M} \models_u \Box_i\varphi & \textit{iff for all } v \in W,\ uR_iv \textit{ implies } \mathcal{M} \models_v \varphi \\ \mathcal{M} \models_u \Diamond_i\varphi & \textit{iff } \mathcal{M} \models_v \varphi \textit{ for some } v \in W \textit{ such that } uR_iv \end{array}$$

Although we will not focus on multi-modal logics, the above has been provided for completeness.

2.4 Modal logic reasoning tasks

We now look at the sort of reasoning tasks that are typically required in a modal logic. We begin with the definitions of *decidability* and the *modal satisfiability* and *modal validity* reasoning tasks, and then provide an overview of the *model checking task*.

Definition 2.4.1. *A problem is decidable (computable) if and only if there exists an algorithm that is capable of providing a correct answer of 'yes' or 'no' for all valid inputs to the algorithm, in a finite number of algorithmic steps.*

Definition 2.4.2. *The modal satisfiability problem is the problem of determining whether or not a modal formula φ is satisfiable in some model.*

Definition 2.4.3. *The modal validity problem is the problem of determining whether or not a modal formula φ is valid in a model, in a frame or in a class of frames* (*such as the class of reflexive frames*).

It is easy to see that φ is valid if and only if $\neg\varphi$ is unsatisfiable – that is, the modal validity problem is the dual of the modal satisfiability problem. If we have a method of solving the one, we will be able to solve the other. Both of these problems are decidable. We will focus on the *modal satisfiability problem* in the chapters which follow.

The *model checking task* is more complex and can be one of three tasks. It can verify whether a formula φ is satisfiable at some world w in a model $\mathcal{M}$ or it can verify whether the formula is satisfiable at all worlds in the model. A more complex form of model checking involves returning the set of worlds in a finite model $\mathcal{M}$ at which the modal formula φ is satisfiable (which is not a decision problem).

Model checking is a very important reasoning task that is used, for example, to verify formal systems. In the case of a hardware or software design, it verifies whether or not a formal specification (formula) is satisfiable in a model representing the design. Such a specification will typically be expressed as a modal temporal logic formula. It can be used in the design of a concurrent system to verify that deadlock cannot occur. An example of temporal logic model checking of programs is given in [24], and an example of the verification of a concurrent system using model checking is to be found in [23].

The satisfiability and validity problems are both decision problems. As a next step, we need to define a means of measuring their *complexity*.

2.5 The computational complexity of automated reasoning

For any algorithm, there are two kinds of complexity measures to consider – the time it takes to find a solution (the number of computational steps required) and the space requirements of the solution (the amount of memory required). Both the time and space requirements are typically measured as functions of the size of the input. The resources an algorithm uses can be measured by looking at its *worst-case complexity* – we measure the performance of the most difficult problem instance the algorithm can be given to solve.

To measure computational complexity, a robust mathematical model of computability is required. One of the most widely used models is the Turing machine, the details of which are provided in texts such as [35]. Briefly, Turing machines were developed by Alan Turing [121] and are based on the concept that an algorithm is computable if one can specify a finite sequence of instructions which, when followed, results in the completion of the algorithm.

Because of its simplicity, the Turing machine model is widely used in theoretical computer science, particularly in complexity theory. It is remarkably robust and general.

The following definitions are standard definitions that form part of any discussion on complexity theory [9, 16].

Definition 2.5.1. *An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size – that is, if there is a polynomial p such that for each input of size n, the algorithm terminates after at most $p(n)$ steps.*

Definition 2.5.2. ***P** is the class of decision problems that are polynomially bounded.*

Problems in **P** can be solved in a deterministic Turing machine in deterministic polynomial time. This class of problems is referred to as *tractable* or 'not-so-hard'. However, a problem that is *not* in **P** is extremely expensive and probably impossible to solve in practice. Such problems are termed *intractable* and are defined in terms of the class **NP**.

Definition 2.5.3. ***NP*** *is the class of decision problems for which a given proposed solution for a given input can be checked in polynomial time to see if it is a solution.*

NP is the class of problems that are *decidable* in non-deterministic polynomial time – a non-deterministic Turing machine may take exponential time to solve a problem but will take polynomial time to verify a potential solution.

The type of problem that belongs in **NP** is solved using a non-deterministic algorithm in which the problem is decomposed into two separate steps – a non-deterministic search for a solution, followed by the deterministic verification of the solution. The non-deterministic search is essentially a guided guess of a possible solution, which is followed by a verification phase which is *tractable* (*solvable* in polynomial time). If the guess fails, the search continues and the next possible solution is verified. This process is repeated until either a solution is found or the problem is found to be unsolvable.

NP-hard problems are at least as hard as any problem in **NP**, while **NP**-complete problems are in **NP** and are **NP**-hard.

Definition 2.5.4. ***PSPACE*** *is the class of problems solvable by a deterministic Turing machine using only polynomial space.*

A **PSPACE** algorithm may generate many different possible solutions, each of which will require polynomial space to solve. Because there could be many such solutions, it could take exponential time to generate them. Since each solution is discarded once it has been generated and tested, simplistically such an algorithm runs in polynomial space.

A decision problem is **PSPACE**-complete if it is in **PSPACE** and every problem in **PSPACE** can be reduced to it in polynomial time. **PSPACE**-complete problems can be thought of as the hardest problems in **PSPACE**.

Definition 2.5.5. ***EXPTIME*** *is the class of problems deterministically solvable in exponential time – a problem is solvable in exponential time if there is an exponentially time bounded Turing machine that solves it.*

Many modal logics fall within this important class.

Definition 2.5.6. ***NEXPTIME*** *is the class of problems solvable using an exponentially bounded non-deterministic Turing machine.*

Like **NP** algorithms, **NEXPTIME** algorithms have a 'guess and check' profile with the crucial difference being that the guessed solution may be exponentially large in the size of the input and so the deterministic checking that follows may take exponentially many steps in the size of the input.

These classes are related as follows:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$$

with **NEXPTIME** being the most complex class. In an ideal world, we would want all our algorithms to fall into the **P** class.

An unanswered question in computational complexity theory is whether or not **P** = **NP**. Recall that a problem in **P** can be *solved* in polynomial time, whereas a problem in **NP** can only be *verified* in polynomial time. If **P** = **NP**, it would mean that it is as easy to verify a solution as it is to compute the solution.

The reader is referred to [9] and [16] for further details and examples of these classifications.

2.6 The computational complexity of modal satisfiability

We will not be looking at the specific details of a modal satisfiability algorithm here – that is the task of the next chapter.

Cook [26] showed that the problem of deciding whether a formula of propositional logic is satisfiable is **NP**-complete. Since propositional logic formulae are a subset of modal formulae, this means that the modal satisfiability problem is at least **NP**-complete.

Ladner [80] further quantified the computational complexity of the modal satisfiability problem for various systems of modal logic. He showed that there are polynomial space bounded algorithms for deciding whether or not a formula is provable in any one of K, KT and $S4$. They are in **PSPACE** and are **PSPACE**-complete using deterministic space $O(n^2)$, $O(n^3)$ and $O(n^4)$ respectively. He also showed that *S5*-satisfiability is in **NP** and is therefore **NP**-complete.

Halpern [54] subsequently showed that the **PSPACE**-completeness results of Ladner also hold for the modal logics $K(m)$, $KT(m)$ and $S4(m)$, for m ≥ 1. He also showed that, if there is a bound on the number of primitive propositions in an *S5*-satisfiability problem, it can be solved in linear time.

Hudelmaier [68] showed that satisfiability in *S4* is $O(n^2 \; log \; n)$-space computable and in K and KT it is $O(n \; log \; n)$-space computable. Nguyen [94] reduced the bound for *S4* to $O(n \; log \; n)$ and Hemaspaandra [57] reduced the bound for K to $O(n)$.

These results tell us that the modal satisfiability problem for the modal logics we have discussed could take exponential time to solve, but that it takes polynomial time to verify a solution and requires polynomial space. Note that some of the more complex modal logics have complexity results in **EXPTIME** and **NEXPTIME**. We will see in the next chapter that the solvers we look at have been limited to the simpler modal logics.

2.7 Some interesting properties of modal logics

Before we conclude this section, we need to discuss some properties of modal logics that we will be referring to in the next chapters. We will also be looking at the translation of modal logic into first-order logic and since there is an interesting difference in the decidability of these logics, a brief introduction is appropriate.

There is a standard translation of a modal logic into a first-order logic – both talk about relational structures, with first-order logic making use of the 'for all' ($\forall$) relation as opposed to the 'always' ($\Box$) relation of modal logic. Every normal modal formula can be translated into a first-order formula, as we will see in the next chapter. The first-order formulae that correspond to the modal logic formulae under this standard translation form the *modal fragment of first-order logic.*

Modal logic is decidable as we have seen, whereas first-order logic is not. This means that we can translate a decidable formula into an undecidable one. The reason for the decidability property of modal logic has caused much interest – especially when one considers the apparent similarity between modal logic and first-order logic. Vardi [127] attributes this to the *tree model property* of modal logic, which tells us that every satisfiable formula has a model that is a tree, with the formula being satisfiable at the root of the tree. The tree model property provides a powerful tool for proving decidability results and for constructing efficient decision procedures.

Some modal logics also have the *finite model property* which means that, for a set of formulae φ, every satisfiable formula in φ is satisfiable in a finite model. Decidability of certain modal logics can be shown by establishing this property.

Now that we have defined modal logic and have looked at the sort of problems that need to be solved, we look at some of its application areas. There are many areas to which modal logic has been applied – we look at a very small subset.

2.8 An overview of application areas of modal logic

We briefly discuss three families of modal logics that are central to computing. These are the logics of knowledge and belief, description logics and modal temporal logics.

2.8.1 Modal logics of knowledge and belief

We touch briefly on modal logics of knowledge and belief in which the properties of knowledge need to be defined. The field grew out of the work of Hintikka [62] whose idea was that, in each state of the world, an agent has other states or worlds he considers possible. Some of these possible worlds may be indistinguishable from the true world to an agent. An agent is then said to know a fact φ if φ is true in all the worlds he thinks possible. Areas of interest are then to consider what an agent knows about the world, and also what he knows about what other agents know and do not know. This kind of reasoning is crucial in bargaining and economic decision making. It is also of interest to be able to differentiate between common knowledge and the implicit knowledge of a group of agents [55]. Modal logic lends itself well to expressing these concepts.

By imposing various conditions on the accessibility relation, we can capture a number of interesting axioms. For example, if we require that the world that an agent finds himself in is always one of the worlds he considers possible (which amounts to saying that the possibility relation is reflexive), then it follows that the agent does not know false facts. Similarly, we can show that if the relation is transitive, then an agent, who knows a given fact, knows that he knows it. If we impose no restrictions, then the resulting logic is the modal logic K [56].

These logics also need to be able to deal with multi-agent systems which are important in many areas in computer science. Multi-agents are applicable in multi-threading systems and multi-processor computers and can be used to model the behavior of robots and to define web-services, which are agent-like. This is another vast area of research – the reader is referred to [36] and [53] for good overviews of the logics of knowledge and belief and to [133] for an insight into its application areas. Details of some of the limitations of agent-based logics are discussed in [112].

2.8.2 Description logic

Description logic is used to represent and reason about knowledge. It is used to represent the knowledge of an application domain (the world) by defining the *concepts* of the domain (sets of objects) and the *roles* of the domain (binary relationships on objects). Complex concepts, denoted C, C_1, ..., D, D_1, ..., and rules, denoted R, R_1, ..., are built from atomic ones using the set of *constructors* of the description logic, which are typically conjunction ($\sqcap$), disjunction ($\sqcup$), negation ($\neg$), value restriction ($\forall R.C$) and exist restriction ($\exists R.C$). For example, $\forall R.C$ restricts the set of objects to those that are in the relationship R only with objects belonging to C (i.e. they are not related by R to any objects outside of C).

The concepts and roles defined create the *world description* of the application domain. We can infer implicitly represented knowledge from explicitly represented knowledge using various inference capabilities provided in the logic. For instance, the *subsumption* algorithm is used to determine subconcept-superconcept relations. C is subsumed by D ($C \sqsubseteq D$) if and only if all instances of C are also instances of D. Determining subsumption is thus the problem of checking whether one concept is considered more general than another.

From a semantic point of view, concepts are given a set-theoretic interpretation. A concept is interpreted as a set of individuals. Roles are interpreted as sets of pairs of individuals. The domain of interpretation is arbitrary and can be infinite.

We look at a simple example which represents knowledge concerning people – the domain. The relevant *concepts* we define in this domain are Person, Female,

Mother, Woman, Child and Man. Note that these concepts are variable-free – they apply to all individuals in the domain who belong to the concept. *Roles* express additional information about the individual objects. We define the role $\mathsf{hasChild}$ and can associate the concept Mother with it.

We can now express 'persons that are not female' and 'individuals, all of whose children are female' as

$$\mathsf{Person} \sqcap \neg\mathsf{Female} \qquad \text{and} \qquad \forall\mathsf{hasChild}.\mathsf{Female}$$

In this simple example, we have defined a knowledge base of concepts, we have defined a role and we have built the new concept, 'persons that are not female', which we can identify with the concept Man. An inference task in this domain might be to determine whether Mother is subsumed by Woman.

Another important kind of role restriction is given by *number restrictions* that restrict the cardinality of the sets of fillers of roles. For instance, the concept

$$(\geq 3\ \mathsf{hasChild}) \sqcap (\leq 2\ \mathsf{hasFemaleRelative})$$

represents the concept of 'individuals having at least three children and at most 2 female relatives'. Number restrictions are a distinguishing feature of description logics and an equivalent is not found in modal logic.

Description logic initially developed independently of modal logic. Schild [110] identified the relationship between the two which brought these formerly unrelated fields together. This allowed some of the theories and implementations already in place for modal logics to be applied to description logics. Conversely, the highly optimized implementations developed for description logics can be applied in a modal context.

The correspondence between the two logics was defined in terms of the description logic $\mathcal{ALC}$ developed by Schmidt-Schauß and Smolka [113].

Definition 2.8.1. *The description logic $\mathcal{ALC}$ is defined in terms of the following formation rules, where c denotes a concept symbol and r a role symbol:*

$$\begin{array}{lcl} C, D & \rightarrow & c \mid \top \mid C \sqcap D \mid \neg C \mid \forall R.C \\ R & \rightarrow & r \end{array}$$

The formal semantics of $\mathcal{ALC}$ is defined as follows:

Definition 2.8.2. *Let $\mathcal{D}$ be a set called the domain. An extension function ε over $\mathcal{D}$ is a function mapping concepts C to subsets of $\mathcal{D}$ and roles R to subsets of $\mathcal{D} \times \mathcal{D}$ such that*

$$\begin{array}{lcl} \varepsilon[\top] & = & \mathcal{D} \\ \varepsilon[C \sqcap D] & = & \varepsilon[C] \cap \varepsilon[D] \\ \varepsilon[\neg C] & = & \mathcal{D} \setminus \varepsilon[C] \\ \varepsilon[\forall R.C] & = & \{d \in \mathcal{D} : \forall <d,e> \in \varepsilon[R].e \in \varepsilon[C]\} \end{array}$$

Viewing $\mathcal{ALC}$ from the modal logic perspective, concepts can be viewed as modal formulae in a multi-modal language $K(m)$ and can be interpreted as the set of worlds in which the modal formula holds. In this case, $\forall$ becomes a modal operator since it is applied to formulae. Thus, $\neg C_1 \sqcap \forall R.(C_2 \sqcap C_3)$ can be expressed by the propositional modal formula $\neg C_1 \wedge \Box_R(C_2 \wedge C_3)$. $\Box_R(C_2 \wedge C_3)$ can be read as 'agent R *knows* proposition $C_2 \wedge C_3$', meaning that, in every world accessible from R, both C_2 and C_3 hold. Thus, the domain of the extension function can be read as the set of worlds, concepts can be interpreted as the set of worlds in which they hold and roles can be interpreted as accessibility relations.

Further, $\forall R.C$ can be expounded as 'all worlds in which agent R knows proposition C' instead of 'all objects for which all Rs are in C'.

The correspondence is established formally by defining a function f that maps $\mathcal{ALC}$-concepts to $K(m)$ formulae as follows:

$$\begin{array}{lcl} f(C) & = & p_C \\ f(\top) & = & True \\ f(C \sqcap D) & = & f(C) \wedge f(D) \\ f(\neg C) & = & \neg f(C) \\ f(\forall R.C) & = & \Box_R f(C) \end{array}$$

This correspondence led to the proof that $\mathcal{ALC}$ is a notational variant of the propositional modal logic $K(m)$ and that satisfiability of $K(m)$ has the same computational complexity as $\mathcal{ALC}$.

We will see in the next chapter that some of the solvers have been developed specifically to address description logics but can be applied to modal logic problems as well.

Description logic has been used in different application domains such as natural language processing, configuration of technical systems and database schemata and queries. It has been used of late to develop applications for the Semantic Web, which is an active research area. Good references on current developments are to be found in the papers of annual conferences such as ‘International Workshop on Description Logics’ and ‘Principles and Practice of Semantic Web Reasoning’.

A detailed overview of description logic can be found in the Description Logic Handbook by Baader et al. [7].

2.8.3 Modal temporal logic

Modal temporal logic provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. It is a vast and varied research area and many different approaches have been taken to solve it. It emerged as an area of research in the 1950s, due largely to Arthur Prior [105, 106, 107]. Born some decades later than modal logic, it turned out technically much like it. It was based on philosophical arguments concerning the structure of time, on linguistics where tenses and other temporal expressions are expressed in natural languages, and to some extent on mathematics.

In a modal temporal logic, various temporal operators or modalities are introduced to describe and reason about how the truth values of assertions vary with time, in accordance with how time has been defined. Time can be viewed as *discrete* as represented using natural numbers, *continuous* as represented by reals or rationals, *linear* or *branching* where branching usually occurs in the future, *complete* or *incomplete*. Temporal entities can then be defined as *points*, *intervals* or *durations*.

Prior’s temporal logic, called *tense logic*, is a two-directional modal logic with modalities F – *at least once in the future* and P – *at least once in the past*. Subsequently, stronger languages were defined that contained additional modalities – Kamp, for example, added the modalities *Since* and *Until* [77]. Temporal logic evolved in complexity, resulting in different temporal logics. *Linear temporal logic* has time represented as discrete points, ordered like natural numbers with only one possible future for each time entity. *Branching temporal logic* is based on a branching notion of time where a time entity may have several possible successor time entities. There are furthermore various branching temporal logics such as *computation tree logic* that defines a time tree.

An early application of temporal logic was identified by Pnueli [104]. He was one of the first to recognize that it could be used to specify and verify the correctness of computer programs, especially those that are non-terminating or that are continuously operating concurrent programs such as operating systems. In an ordinary sequential program, formal verification of program correctness is reasonably straightforward, as programs move from an initial state through to a final state. However, in the case of an operating system that is non-terminating and that interacts with its environment, these methods are not applicable. In this environment, the operators of temporal logic can be utilized to deal with their time-varying behavior. These ideas were subsequently explored and extended by a number of researchers. Temporal logic has now been used or proposed for use in virtually all aspects of concurrent program design, including specification, verification, development and mechanical program synthesis.

Good overviews of temporal logics are available in papers such as [22, 34, 118, 124].

We now look specifically at the temporal constraint satisfaction problem and its relation to modal temporal logic, as this is an application area which relates well to the work which follows.

Temporal Constraint Satisfaction

Temporal constraint satisfaction is a search technique used to represent and answer queries about the timing of events and the temporal relations that hold between them and is particularly useful in, for example, scheduling problems. In temporal constraint satisfaction problems (TCSPs), variables can be time points, time intervals or durations, and constraints represent sets of allowed temporal relations between them. The domain of time point and duration variables can be either the set of integers or rational numbers, whereas the domain of time interval variables is the set of ordered pairs of either set. There are several classes of temporal constraints, these being *qualitative*, *quantitative* (metric) or a combination of the two. The reader is referred to [88] and [115] for further details of these classes.

An approach that has been successfully deployed to solve temporal constraint satisfaction problems is Allen's interval algebra.

Allen's interval algebra

Allen [3] defined a temporal representation based on time intervals and the binary relations between them. It is used to express qualitative relations between intervals and enables temporal knowledge that is possibly indefinite and incomplete to be represented. His interval algebra consists of the thirteen atomic interval relations shown in Figure 2.2. These relations consist of 6 basic relations and their inverses, as well as the equality relation.

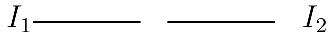
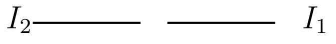
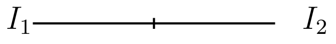
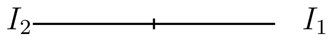
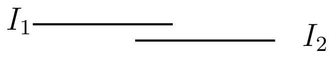
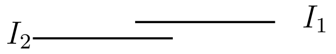
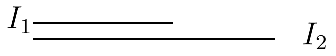
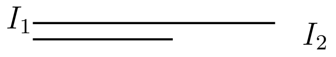
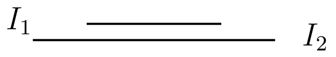
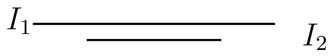
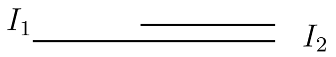
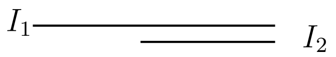
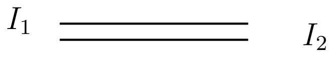
Relation	Abbreviation	Representation
I_1 *before* I_2	*b*	
I_1 *after* I_2	*bi*	
I_1 *meets* I_2	*m*	
I_1 *met by* I_2	*mi*	
I_1 *overlaps* I_2	*o*	
I_1 *overlapped by* I_2	*oi*	
I_1 *starts* I_2	*s*	
I_1 *started by* I_2	*si*	
I_1 *during* I_2	*d*	
I_1 *contains* I_2	*di*	
I_1 *finishes* I_2	*f*	
I_1 *finished by* I_2	*fi*	
I_1 *equals* I_2	*eq*	

Figure 2.2: Allen's 13 basic relations between intervals

Given two time intervals I_1 and I_2, their relative position can be described in terms of these thirteen relations. An unknown relation can be represented as the disjunction of any of the 13 interval relations in the set $I = \{b, bi, m, mi, o, oi, s, si, d, di, f, fi, eq\}$. For example, I_1 $\{m, b, o\}$ I_2 is interpreted as either 'I_1 *meets* I_2', or 'I_1 *before* I_2', or 'I_1 *overlaps* I_2'. There are 2^{13} possible sets of such relations (8192).

Three standard operations are applied to these sets of relations, these being *converse*, *intersection* and *composition*. As an example, the converse of the relation *before* is *after*, while the intersection of the relations $\{b, m, o\}$ and $\{m, eq\}$ is $\{m\}$. Allen

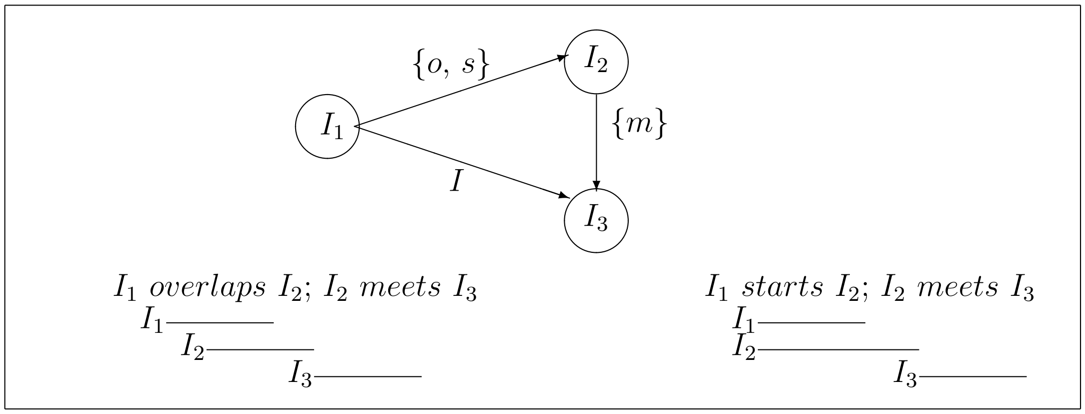
[3] defined a lookup table which is used to determine the composition of two sets of relations. In terms of this table, the composition of the relations *before* and *finishes* is {*before, during, overlaps, meets, starts*}.

Temporal information defined in terms of the interval algebra can be represented using *temporal constraint graphs* or *interval algebra networks*. Nodes are used to represent intervals and arcs or edges are labeled with the set of possible relations between the two intervals. Such a network is thus a network of binary constraints.

Using the interval algebra representation, the fundamental reasoning task is to instantiate the variables with values from their domains in such a way that all constraints are satisfied.

We illustrate these concepts with the following example.

Example 2.8.3. Consider the following temporal constraint graph in which I_1, I_2 and I_3 are intervals:


$\{o, s\}$
I_1
I_2
I_3
$\{m\}$
I
I_1 *overlaps* I_2; I_2 *meets* I_3
I_1
I_2
I_3
I_1 *starts* I_2; I_2 *meets* I_3
I_1
I_2
I_3

The label I between nodes I_1 and I_3 denotes that this relation is unknown.

In the two diagrams beneath the temporal constraint graph, the two scenarios that arise from the relations *overlaps* and *starts* that are possible between intervals I_1 and I_2 are presented. It is clear that the relation between I_1 and I_3 must be $\{b\}$ or $before$. ⊣

According to van Beek [122], finding a consistent scenario and finding the feasible relations between nodes in an interval algebra network are both **NP**-complete and are thus intractable in the worst case.

Allen [3] defined the path consistency algorithm to compute the strongest implied relation between pairs of intervals. This algorithm has been used extensively and, according to [115], it is still used as the major constraint propagation algorithm for interval TCSPs. A significant body of research exists which looks at ways in which temporal constraint satisfaction algorithms for the interval algebra can be improved. The reader is referred to papers such as [79, 92, 123, 91, 114, 120] for further details.

The translation of the interval algebra into a modal temporal logic

We now look at how a temporal constraint satisfaction problem which is expressed using Allen's interval algebra can be translated into a modal temporal logic. We begin by defining a modal temporal logic $\mathcal{ML}_2$, with two modalities [47].

Definition 2.8.4. *Consider a propositional language with two modalities, $\Box_F$ and $\Box_P$, meaning respectively henceforth (at all future times) and hitherto (at all past times). In a Kripke frame, $\mathcal{F} = (W, R_F, R_P)$,*

$$\begin{aligned} \mathcal{M} \models_s \Box_F \varphi \quad & \textit{iff } sR_Ft \textit{ implies } \mathcal{M} \models_t \varphi \\ \mathcal{M} \models_s \Box_P \varphi \quad & \textit{iff } sR_Pt \textit{ implies } \mathcal{M} \models_t \varphi \end{aligned}$$

where the connective sR_Ft is read as 't is in the future of s' and sR_Pt as 't is in the past of s'.

Because a temporal ordering is by its nature *transitive*, we can define a *time frame* to be any structure $\mathcal{F} = (W, R)$ in which R is a *transitive* relation on W, with the modeling above.

We need to define the concept of 'always' in this structure and so the following definition is appropriate.

Definition 2.8.5. *In a temporal logic, $\Box\varphi$ meaning 'always φ' and $\Diamond\varphi$ meaning 'at some time φ' are defined as follows:*

$$\Box\varphi = \Box_P\varphi \wedge \varphi \wedge \Box_F\varphi$$

$$\Diamond\varphi = \Diamond_P\varphi \vee \varphi \vee \Diamond_F\varphi$$

We can now express Allen's interval algebra in this basic temporal logic $\mathcal{ML}_2$. This translation is taken from [41].

Algorithm 1. *Intervals I_i and I_j in the interval algebra are expressed as propositional atoms p_i and p_j, each of which denotes an interval. Therefore we use these terms interchangeably and can refer to p_i as either a propositional atom or an interval. For atomic formulae we apply the translation:*

equals(p_i, p_j)	$\Box(p_i \leftrightarrow p_j)$
before(p_i, p_j)	$\Box(p_i \rightarrow \neg p_j \wedge \Diamond_F p_j) \wedge \Diamond(\neg p_i \wedge \neg p_j \wedge \Diamond_P p_i \wedge \Diamond_F p_j)$
meets(p_i, p_j)	$\Box(p_i \rightarrow \neg p_j \wedge \Diamond_F p_j) \wedge \neg\Diamond(\neg p_i \wedge \neg p_j \wedge \Diamond_P p_i \wedge \Diamond_F p_j)$
overlaps(p_i, p_j)	$\Diamond(p_i \wedge p_j) \wedge \Diamond(p_i \wedge \neg p_j \wedge \Diamond_F p_j) \wedge \Diamond(p_j \wedge \neg p_i \wedge \Diamond_P p_i)$
starts(p_i, p_j)	$\Box(p_i \rightarrow p_j) \wedge \Diamond(p_j \wedge \neg p_i) \wedge \Box(p_i \wedge p_j \rightarrow \Box_P(p_j \rightarrow p_i))$
during(p_i, p_j)	$\Box(p_i \rightarrow p_j) \wedge \Diamond(p_j \wedge \neg p_i \wedge \Diamond_P p_i) \wedge \Diamond(p_j \wedge \neg p_i \wedge \Diamond_F p_i)$
finishes(p_i, p_j)	$\Box(p_i \rightarrow p_j) \wedge \Diamond(p_j \wedge \neg p_i) \wedge \Box(p_i \wedge p_j \rightarrow \Box_F(p_j \rightarrow p_i))$

with the added constraint that for each propositional atom p_i,

$$\Diamond p_i \wedge \Box(\Diamond_P p_i \wedge \Diamond_F p_i \rightarrow p_i) \tag{2.1}$$

Equation (2.1) ensures that the propositional atom p_i associated with the interval variable i is interpreted by a non-empty convex set.

The interval algebra problem of Example 2.8.3 is translated into this modal temporal logic as follows:

$$\begin{aligned}
&((\Diamond(p_1 \wedge p_2) \wedge \Diamond(p_1 \wedge \neg p_2 \wedge \Diamond_F p_2) \wedge \Diamond(p_2 \wedge \neg p_1 \wedge \Diamond_P p_1)) \vee \\
&(\Box(\neg p_1 \vee p_2) \wedge \Diamond(p_2 \wedge \neg p_1) \wedge \Box(\neg p_1 \vee \neg p_2 \vee \Box_P(\neg p_2 \vee p_1)))) \wedge \\
&\Box((\neg p_2 \vee \neg p_3) \wedge (\neg p_2 \vee \Diamond_F p_3)) \wedge \neg\Diamond(\neg p_2 \wedge \neg p_3 \wedge \Diamond_P p_2 \wedge \Diamond_F p_3)
\end{aligned}$$

We can see that the above formula does not have the intuitive content of the interval algebra representation. However, the benefit of this translation lies in the fact that a modal temporal logic is able to deal with more powerful temporal problems than the interval algebra.

2.9 Final remarks

In this chapter, we have provided a high level overview of modal logic. In [16], the authors tell us that there are many different definitions and views of modal logic – a thorough understanding of it therefore requires intensive study.

We have identified one of the reasoning tasks in modal logic – modal satisfiability – to investigate further and in the next chapter will be looking at various solvers that have been developed to solve it. We have seen that, although a well-designed algorithm will be able to solve any problem in this class, the time taken to do so will be exponential in the worst case. This leads us to expect that these algorithms need to focus on reducing or pruning the search space and traversing it as efficiently as possible. They need to be highly optimized if they are going to be able to produce realistic results – results that run in an acceptable time. We will also notice that the solvers we look at have been developed for a limited subset of normal modal logics, another indication of the complexity of this class of problem.

We have looked at various application areas of modal logic and have looked in some detail at description logic, on which many of the solvers we will be looking at have been based. We have looked in detail at the temporal constraint satisfaction problem. It can be represented in a temporal modal logic that requires a transitive accessibility relation. This is of particular interest to us in the work which follows.

In the next chapter, we look at solvers which have been developed to solve the modal satisfiability problem and focus on the optimizations which have been applied.

Chapter 3

Modal satisfiability solvers

In the previous chapter, we identified the modal satisfiability problem as an area to investigate further. Many different approaches have been followed to develop algorithms that can adequately address the complexity of modal satisfiability. Some of these resulted in the development of highly optimized solvers, while others are of academic interest. These solvers fall into two major categories – either special purpose-built algorithms have been developed, or the modal logic has been translated into a different class of problem. We look in detail at the tableau approach (a special purpose-built algorithm), and the translation of modal logic into first-order logic, a propositional satisfiability problem (SAT) and a constraint satisfaction problem (CSP). The translation approach has the distinct advantage that, once the formula has been translated into the new class, it can be further processed by the sophisticated existing tools and provers of that class. First-order theorem proving is one of the most mature subfields in automated theorem proving, while the SAT-based theorem prover developed by Davis, Putnam, Logemann and Loveland in 1962 [27, 28] still forms the basis of many efficient complete SAT solvers. The constraint logic programming language ECL^iPS^e [2] is a sophisticated tool which has been developed to deal with constraint satisfaction problems.

The first approach we consider is the tableau system, which is widely used. It has been applied in description logics where a lot of research effort has gone into its enhancement. We look at two implementations of tableau systems – one with an implicit accessibility relation in which worlds are not specified, and the other with an explicit accessibility relation in which the tableau rules include named worlds. Some of the more mature solvers are based on the second approach. We look at the complexity of solving problems in tableau systems by considering a specific example and find that the order in which clauses and variables are processed greatly affects the time taken to solve a problem. There is also non-determinism in the order of execution of tableau rules. The most efficient solvers developed to date have addressed these problems by implementing techniques that include good search heuristics, the optimization of data structures and the simplification of formulae.

The second approach is the translation of modal formulae into first-order logic. The translation can be relational, functional or semi-functional, which is a combination of the two. In the case of the relational translation, formulae are directly translated into first-order logic with the possible world structure of the modal logic being explicitly represented using a predicate symbol. In the case of the functional translation, the accessibility relation is decomposed into a set of accessibility functions that map worlds to accessible worlds. The relational translation has the problem of non-termination when resolution is applied to formulae, while the functional translation results in long and complex formulae. Ohlbach et al. [97] concluded that this approach is not entirely satisfactory – it has primarily been of interest in understanding why modal logic is decidable, whereas first-order logic is not.

The third approach is the translation of modal formulae into propositional logic formulae. The satisfiability (SAT) problem of propositional formulae is a widely researched field. A modal formula can be viewed as a layered set of propositional formulae, with each layer being solved by a SAT solver. The most commonly used SAT algorithm is the DPLL algorithm that forms the basis of the KSAT solver [46] we will be looking at. This solver produces good results for the modal logics K and KT.

Finally, we look at the translation into constraint satisfaction problems. When a modal formula is translated into layered propositional formulae, each layer can in turn be translated into a constraint satisfaction problem. We look in depth at this approach and its solution in a constraint logic programming environment. The K_KCSP solver [18] produces good results for the modal logic K, which motivates investigating it further in the next chapter.

Most of these solvers have been benchmarked using commonly available data sets and so we are able to compare their performance. We have limited our investigation to those solvers that have demonstrated the best results to date. We base this choice on the results obtained at the TANCS conferences that are a major forum for the presentation of new research on all aspects of automated reasoning. In particular, the

TANCS 'Non Classical Systems Comparison' of 1998, 1999 and 2000 [10, 85, 87] were conferences that were held to conduct an experimental analysis of theorem provers and satisfiability testers for expressive modal and description logics. Their aim was to compare the performance of such solvers in an experimental setting by providing meaningful test data sets. The K_KCSP solver was however not benchmarked at these conferences as it was only developed in 2004.

The Gentzen sequent calculus, which is a special purpose-build algorithm, is very similar to the tableau approach and so, even though an impressive solver has been developed using this approach, we only look briefly at their commonality. We outline some other approaches that are also translations but leave it to the reader to investigate them further.

3.1 Tableau systems

We begin this section by looking at the theory of tableau systems. Tableau systems have some distinct advantages over other approaches – in particular, they are easy to understand and implement. They are also easy to extend. For example, the tableau system for propositional logic has been extended to the basic modal logic K by simply adding an additional rule, as we will see further on.

We begin with the following notational conventions:

- φ, ψ, φ_i and ψ_i denote well-formed formulae.
- X, Y, Z denote finite (possibly empty) sets of well-formed formulae.
- $(X; Y)$ denotes $X \cup Y$ and $(X; \varphi)$ denotes $X \cup \{\varphi\}$.
- $\neg X$ denotes $\{\neg\varphi \mid \varphi \in X\}$, $\Box X$ denotes $\{\Box\varphi \mid \varphi \in X\}$ and $\neg\Box X$ denotes $\{\neg\Box\varphi \mid \varphi \in X\}$.

Tableau systems for modal logics consist of a set of tableau rules that can be applied to modal formulae to produce *refutation proofs* – that is, the validity of a set of formulae X is proved by showing that its negation is unsatisfiable. A tableau proof is a binary tree of nodes where the *root node* contains the set of formulae $\neg X$ and the *child nodes* are obtained by the application of the tableau rules to the parent node. The set of formulae X is decomposed by eliminating one connective at each new node. The rules are applied systematically, subject to a partially defined order on their application. If no more tableau rules can be applied to a node, it is termed an *end node* or a *leaf node*.

In the case of modal logics, two approaches can be followed when dealing with the accessibility relation – it can be dealt with either implicitly or explicitly. In implicit systems, the accessibility relation is built into the rules that are directly applied to the formulae. In explicit systems, the accessibility relation is incorporated into labels attached to each node.

An excellent account of proof methods for modal logics is provided in the book by Fitting [37]. The paper by Góre [49] is also a good reference as it provides a comprehensive overview of tableau methods for many different modal logics.

3.1.1 Modal tableau systems with implicit accessibility

We now look at how a tableau system $\mathcal{C}L$ with implicit accessibility for a modal logic L (Definition 2.3.1) is defined.

Definition 3.1.1. *A tableau system $\mathcal{C}L$ for a modal logic L is a finite collection of tableau rules $\rho_1, \rho_2, \ldots, \rho_n$, each of which is identified by its rule name. A tableau rule ρ consists of a numerator $\mathcal{N}$ above the line and a finite list of denominators $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$ below the line, separated by either vertical bars or by semicolons:*

$$(\rho_1) \qquad \frac{\mathcal{N}}{\mathcal{D}_1 | \ldots | \mathcal{D}_k} \qquad \textit{or} \qquad (\rho_2) \qquad \frac{\mathcal{N}}{\mathcal{D}_1 ; \ldots ; \mathcal{D}_k}$$

where $\mathcal{N}$, $\mathcal{D}_1$, ... are finite sets of formulae.

Denominators separated by vertical bars are placed on different branches of the parent node; denominators separated by semicolons are placed together on a single branch of the parent node.

A tableau rule with vertical bars, (ρ_1), is read in a downward direction as follows: 'if the numerator is L-satisfiable, then so is at least one of the denominators'. These are branching rules. A tableau rule with semicolons, (ρ_2), is read in a downward direction as follows: 'if the numerator is L-satisfiable, then so are all of its denominators'. The numerator of each tableau rule contains one or more distinguished formulae called *principal formulae*.

We present the tableau algorithm, as per [49]:

Algorithm 2. *A $\mathcal{C}L$-tableau for a set of formulae X is a finite tree with root $\neg X$ whose nodes carry finite formula sets. A tableau rule with numerator $\mathcal{N}$ is applicable to a node carrying set Y if Y is an instance of $\mathcal{N}$. Steps for extending the tableau are:*

- *choose a leaf node n carrying Y, where n is not an end node and choose a rule ρ that is applicable to Y;*
- *if ρ has k denominators and there are k successor nodes to n, each node i will be labeled with an appropriate instantiation of denominator $\mathcal{D}_i$; in the case of a single successor node, it will be labeled with an appropriate instantiation of the denominators $\mathcal{D}_1; ...; \mathcal{D}_k$*

all with the proviso that, if a successor node s carries a set Z and Z has already appeared on the branch from the root to s, then s is an end node.

A branch in the tableau is closed if its end node contains $\bot$; otherwise it is open. The tableau is closed if all its branches are closed. If the tableau is closed, X is valid.

We now define soundness and completeness of a tableau system $\mathcal{C}L$ for a modal logic L with respect to L-frames of the logic.

Theorem 3.1.2. *(Soundness)*

If there is a closed $\mathcal{C}L$-tableau for $Y \cup \{\neg\varphi\}$, then any L-model that makes Y true at world w must make φ true at world w.

Theorem 3.1.3. *(Completeness)*

If every L-model that makes Y true at world w also makes φ true at world w, then some $\mathcal{C}L$-tableau for $Y \cup \{\neg\varphi\}$ must close.

These soundness and completeness results are proved in [49].

Having defined the tableau algorithm, the next step is to define the tableau rules for the basic modal logic K [49]:

Definition 3.1.4. *The tableau rules $\mathcal{C}K = \{(\bot), (\wedge), (\vee), (\neg), (K)\}$ for the basic modal logic K are defined as follows.*

$$(\bot)\quad \frac{X;\varphi;\neg\varphi}{\bot} \qquad (\wedge)\quad \frac{X;\varphi\wedge\psi}{X;\varphi;\psi} \qquad (\vee)\quad \frac{X;\varphi\vee\psi}{X;\varphi \quad | \quad X;\psi}$$

$$(\neg)\quad \frac{X;\neg\neg\varphi}{X;\varphi} \qquad (K)\quad \frac{Z;\Box X;\Diamond\varphi}{X;\varphi}$$

In the case of a set of modal formulae $X;\ \neg(\varphi \wedge \psi)$, the formula $\neg(\varphi \wedge \psi)$ is replaced with $\neg\varphi \vee \neg\psi$ using standard propositional rules before the tableau rules are applied. In the case of a set of modal formulae $\Box X;\ \neg\Box Y$, we need to first convert $\neg\Box Y$ to $\Diamond\neg Y$ before the (K) rule can be applied.

These tableau rules can be categorized as *static* and *transitional* rules. The intuition behind this distinction is that in the static rules, the numerator and denominator represent the same world, whereas in the transitional rules, the numerator and denominator represent different worlds. In the above, the static rules are $(\bot)$, $(\wedge)$, $(\vee)$ and $(\neg)$, while the transitional rule is (K).

The order of application of these rules is non-deterministic and the order of formulae in a set is immaterial, that is, $\{X;\varphi;\psi\} \equiv \{X;\psi;\varphi\}$. Since the $(\vee)$ rule is the only rule for which more than one branch is generated, this rule is generally only applied if no other rule can be applied. These branching points give rise to the search component in tableau proofs and the failure of a branch causes backtracking to a previous branch point, whereupon the remaining branches are searched.

A very important rule is the *cut* rule that encodes the 'law of the excluded middle' and is non-deterministic [49].

Definition 3.1.5. *The cut rule is defined as follows:*

$$(cut)\quad \frac{X}{X;\varphi \quad | \quad X;\neg\varphi}$$

where the formula φ is arbitrary.

If φ is a subformula of X, the above rule is known as the *analytical cut*. The application of this rule leads to disjoint branches, which results in shorter proofs [49].

Example 3.1.6. We now demonstrate these concepts using the formula

$$\Box(p \to q) \to (\Box p \to \Box \mathrm{q})$$

Its negation and simplification give $\Box(\neg p \vee q) \wedge (\Box p \wedge \neg\Box q)$. We write the formula with the rule applied to it on the branch below, and proceed to apply the tableau rules until either the tableau is closed or no more rules can be applied. The tableau for this formula is presented in Figure 3.1.

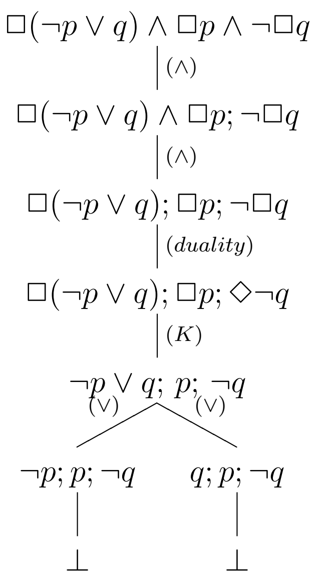

$\Box(\neg p \vee q) \wedge \Box p \wedge \neg\Box q$
$(\wedge)$
$\Box(\neg p \vee q) \wedge \Box p; \neg\Box q$
$(\wedge)$
$\Box(\neg p \vee q); \Box p; \neg\Box q$
$(duality)$
$\Box(\neg p \vee q); \Box p; \Diamond\neg q$
(K)
$\neg p \vee q;\ p;\ \neg q$
$(\vee)$
$(\vee)$
$\neg p; p; \neg q$
$q; p; \neg q$
$\bot$
$\bot$

Figure 3.1: Implicit tableau proof for $\Box(p \to q) \to (\Box p \to \Box q)$

Since both branches are closed, the tableau is closed, which means that $\Box(p \to q) \to (\Box p \to \Box q)$ is a theorem in the modal logic K. $\dashv$

The soundness and completeness of the $\mathcal{C}K$ tableau system are proved by Góre [49]. The tableau system $\mathcal{C}K$ is easy to extend to include additional properties of accessibility relations. The tableau systems for the logics KT and $S4$, already defined in Section 2.3, are as follows:

Definition 3.1.7. *The tableau rules $\mathcal{C}T$ for the modal logic KT consist of the rules of $\mathcal{C}K$ (Definition 3.1.4) and the static tableau rule (T):*

$$(T) \quad \frac{X; \Box\varphi}{X; \Box\varphi; \varphi}$$

Definition 3.1.8. *The tableau rules $\mathcal{C}S4$ for the modal logic S4 consist of the rules of $\mathcal{C}K$ (Definition 3.1.4), the static tableau rule (T) (Definition 3.1.7) and the transitional rule (S4):*

$$(S4) \quad \frac{\Box X; \Diamond\neg\varphi}{\Box X; \neg\varphi}$$

The additional rules required for the various modal logics are defined in the paper by Góre [49], together with the soundness and completeness proofs of each.

3.1.2 Modal tableau systems with explicit accessibility

Various modal tableau systems with explicit representation of accessibility relations have been developed, examples of which are the single step tableau (SST) [84] and the explicit single step tableau [49], which was later extended to become the free-variable tableau [12]. Note that we only give an outline of these proof systems. For more details, the reader is referred to the papers already mentioned.

Single step tableau

In tableau systems where the accessibility relation is represented explicitly, each node of the tableau contains a structured label, together with the formulae applicable at the node. We now define how these labels are structured.

Definition 3.1.9. *Let Γ denote a set of labels. A label σ is defined as a nonempty sequence of positive integers separated by dots. A label τ is said to be an extension of a label σ if $\tau = \sigma.n_1.n_2 \ldots n_k$ for some $k \geq 1$ with each $n_i \geq 1$.*

Definition 3.1.10. *A set of labels Γ is strongly generated if:*

- *there is some root labeled $\rho \in \Gamma$ such that every other label in Γ is an extension of ρ; and*

- *$\sigma.n \in \Gamma$ implies $\sigma \in \Gamma$.*

A labeled formula is a structure of the form $\sigma :: \varphi$ where σ is a label and φ is a formula. A labeled tableau for a finite set of formulae $X = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ is a tree where each node contains a single labeled formula, constructed in a systematic fashion, meaning that the rules are applied according to a specified ordering. A tableau branch is any path from the root downwards in such a tree. A branch is closed if it contains the labeled formulae $\sigma :: \varphi$ and $\sigma :: \neg\varphi$. A tableau is closed if every branch is closed.

Definition 3.1.11. *The rules that contain $\Diamond$ are referred to as π-rules, while the rules that contain $\Box$ are referred to as υ-rules.*

There are three types of tableau rules for the single step tableau – the rules which apply to classical propositional logic, the υ-rules applicable to formulae of the form $\sigma :: \Box\varphi$, and the π-rule applicable to formulae of the form $\sigma :: \Diamond\varphi$.

Definition 3.1.12. *The single step tableau rules for the modal logic K are:*

$$(l\bot)\quad \frac{\sigma :: \varphi \wedge \neg\varphi}{\sigma :: \bot} \qquad (l\neg)\quad \frac{\sigma :: \neg\neg\varphi}{\sigma :: \varphi} \qquad (l\wedge)\quad \frac{\sigma :: \varphi \wedge \psi}{\sigma :: \varphi; \sigma :: \psi}$$

$$(l\vee)\quad \frac{\sigma :: \varphi \vee \psi}{\sigma :: \varphi \quad | \quad \sigma :: \psi} \qquad (lK)\quad \frac{\sigma :: \Box\varphi}{\sigma.n :: \varphi} \qquad (l\pi)\quad \frac{\sigma :: \Diamond\varphi}{\sigma.n :: \varphi}$$

Note that in the case of the (lK) rule, n denotes an existing world (one already named); in the case of the $(l\pi)$ rule, n denotes a new world which is directly accessible from the world already named. As before, the $(l\vee)$ rule is the only rule that causes the tree to branch.

Further υ-rules are added for the modal logics KT, $S4$ and so on.

When generating a tableau proof, a tableau rule is applied to a formula and the next node, which is either in the current or the next world, is generated. No further action is required unless the rule is a υ-rule: a formula of the form $\sigma :: \Box\varphi$ requires φ to be added to *each* world $\sigma.n$ that is generated, as φ is true at each of these. Recall that in the case of implicit tableau systems, only a single node was generated by the (K) rule, and so this was not an issue.

These concepts are demonstrated in the example below, which is a closed tableau.

Example 3.1.13. The single-step tableau proof for the modal K formula

$$\Box(p \to q) \to (\Box p \to \Box q)$$

is given in Figure 3.2. It is first negated to give $\Box(\neg p \vee q) \wedge \Box p \wedge \neg\Box q$. Note that this is the formula that was used in Example 3.1.6.

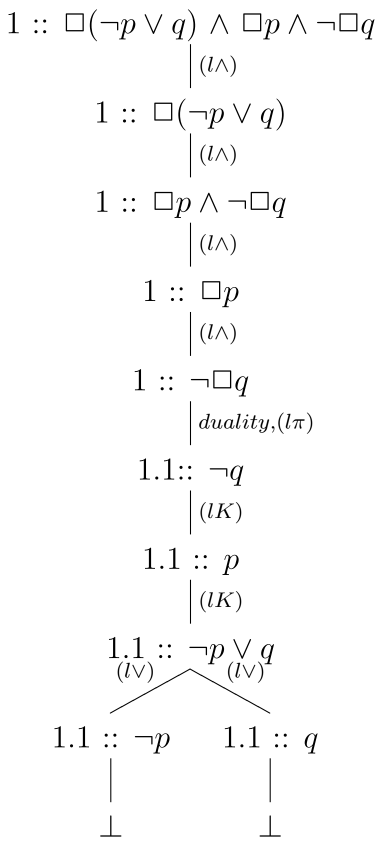


Figure 3.2: Single step tableau proof for $\Box(p \to q) \to (\Box p \to \Box q)$

Note that, to improve clarity, we have placed worlds which are on the same branch below each other. ⊣

Góre [49] and Massacci [86] proved that the single step tableau (SST) rules are sound and complete. The single step tableau rules for the modal logic K (Definition

3.1.12) are easily extended to other modal logics by adding additional rules, as was the case with the modal tableau systems with implicit accessibility (Definitions 3.1.7 and 3.1.8).

Free-variable tableau

Beckert and Góre [12] extended the single step tableau system to the *free-variable tableau system.* This system makes use of free-variable semantic tableau – labels now become $\sigma.x : X$ where the variable x can be either a *free variable* or a *universal variable.*

As we have seen, the interpretation of $\sigma : \Box\varphi$ is that the possible world σ satisfies the formula $\Box\varphi$. In the free-variable tableau approach, the box rule (lK) reduces the formula $\sigma : \Box\varphi$ to the labeled formula $\sigma.x : \varphi$ that contains the *free variable* x in its label and has the intuitive reading, 'the possible world $\sigma.x$ satisfies the formula φ'. This means that the actual value of x does not have to be guessed at the point where $\Box\varphi$ is reduced. Instead, we defer the choice of x until enough information is available to make a choice that immediately closes a branch of the tableau.

Since x represents the next accessible world, in theory it must be instantiated to the same value on all branches. However, instantiating a free variable to close one branch may make it impossible to close other branches. Beckert and Hähnle [13] showed that under certain conditions, $\sigma : \Box\varphi$ has a universal nature and the free variable x can be instantiated in one way to close one branch and in a different way to close another branch – the original binding can be undone without losing soundness. In this case, x is said to be *universal* and $\sigma.x : \varphi$ effectively says that 'all successors of the possible world σ satisfy φ', thereby capturing the Kripke semantics for $\Box\varphi$.

By allowing the choice of variables to be deferred until more information becomes available, the search space is reduced and the non-determinism inherent in automated proof search is also reduced.

Example 3.1.14. The difference between using free variables and universal variables is illustrated in Figure 3.3 in which the negated labeled tableau formulae considered are $1 : \Diamond\neg p \vee \Diamond\neg q$ and $1 : \Box(p \wedge q)$. The tree on the left-hand side uses free variables; the tree on the right-hand side uses universal variables.

In the proof on the left-hand side, we obtain $1.x_1 : p \wedge q$, creating the free variable

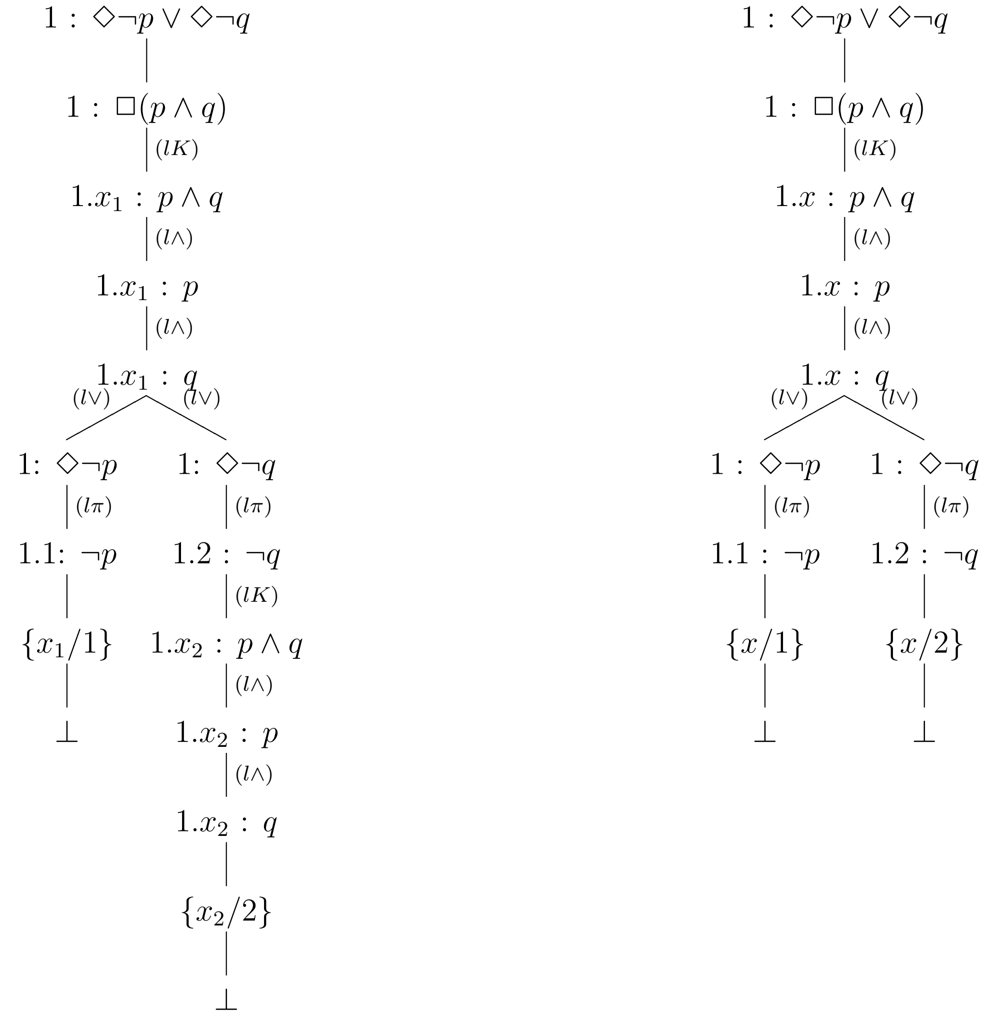


Figure 3.3: Comparison of free variable and universal variable tableau proofs for $(\Diamond\neg p \vee \Diamond\neg q) \wedge \Box(p \wedge q)$

x_1. Applying the conjunctive rule $(l\wedge)$ to this formula gives $1.x_1 : p$ and $1.x_1 : q$. Next we apply the disjunctive rule $(l\vee)$ to the root and split the tableau into two branches containing $1 : \Diamond\neg p$ and $1 : \Diamond\neg q$. Applying the diamond rule $(l\pi)$ gives 1.1 : $\neg p$ on the first branch where 1.1 is a label new to the tableau. We can now close the left branch by setting $x_1 := 1$. All occurrences of x_1 are now bound to 1. To close the right branch, we now need to apply the box-rule again to 1: $\Box(p \wedge q)$ to obtain $1.x_2 : p \wedge q$, thereby creating a new free variable x_2. Applying the conjunctive rule again, we close the second branch by setting $x_2 := 2$.

In the proof on the right-hand side, we create a universal variable x instead of the free variables x_1 and x_2. In this case, before proceeding to the second branch, we undo the binding of x and so can close the second branch by setting $x := 2$. Thus the variable x is instantiated in multiple ways. $\dashv$

The reader is referred to the paper by Beckert and Góre [12] for further details of this approach.

3.1.3 An analysis of the tableau approach

We now examine the efficiency of tableau systems. We have seen that the tableau system $\mathcal{C}K$ has well-defined rules that are applied one at a time until the set of formulae has been proved satisfiable or unsatisfiable. Each time a rule is applied, the set of formulae is simplified and so we have a decision procedure.

However, when we look further, we find there is non-determinism in the choice of formulae, nodes and rules – we need to decide which formula to process next, on which node and which rule to apply. Tableau systems generate a search space and so the overall efficiency of a proof procedure depends on how this search space is explored. Search strategies need to prune the search space as much as possible and need to be able to recognize and eliminate redundancies in the formulae. Inference steps that cannot contribute to the final solution should be avoided as far as possible.

In order to understand the nature of this problem, let us consider the following formula to which the implicit $\mathcal{C}K$ tableau rules are applied. We will not negate the formula to generate a refutation proof – we will only look at its *satisfiability.*

$$\varphi = (\varphi_1 \vee \psi_1) \wedge (\varphi_2 \vee \psi_2) \wedge \ldots \wedge (\varphi_n \vee \psi_n) \wedge \psi \tag{3.1}$$

φ_i, ψ_i and ψ are used to denote formulae of any complexity. The more complex these formulae are, the more complex the tableau expansion would be. We present a simple scenario below.

Suppose that ψ is unsatisfiable. When the implicit $\mathcal{C}K$ tableau rules are applied with $n = 2$ in the formula φ, the tableau in Figure 3.4 is generated.

We have applied the analytical tableau rules from left to right and note the following:

- We process each of φ_1, φ_2, ψ_1 and ψ_2 twice and ψ four times. This problem arises because the alternative branches of the search tree are not disjoint, which causes

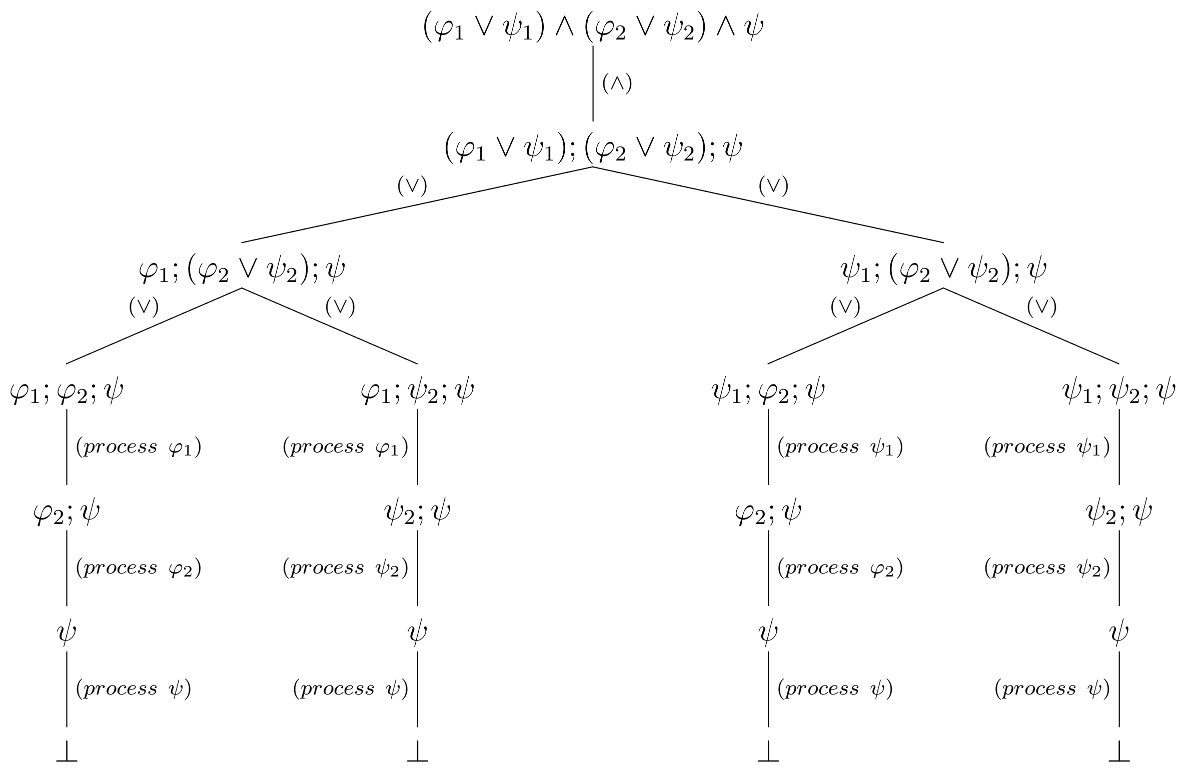


Figure 3.4: Tableau proof for $(\varphi_1 \vee \psi_1) \wedge (\varphi_2 \vee \psi_2) \wedge \psi$

unnecessary and costly expansion of formulae. This is a major inefficiency of the syntactic tableau algorithm and is known as *thrashing*.

- If the order of the formulae had been $\psi \wedge (\varphi_1 \vee \psi_1) \wedge (\varphi_2 \vee \psi_2)$, ψ would have been processed once and found to be unsatisfiable. Hence, a good heuristic is required to determine the order of processing, particularly in the case where φ is complex.

- Suppose $(\varphi_2 \vee \psi_2)$ in Equation (3.1) was in fact $(\varphi_2 \vee True) = True$. Equation (3.1) could then have been simplified prior to the generation of the tableau proof, which would have resulted in a reduced search tree. This process is known as *simplification*.

- Suppose that φ_1 and φ_2 represent a formula ψ' that is unsatisfiable. We would process the clause $(\psi' \vee \psi_1)$, find that ψ' is unsatisfiable, and then process ψ_1. If, when we came to process $(\psi' \vee \psi_2)$, we had recorded that ψ' was unsatisfiable, we would simply select ψ_2 in this branch, thereby traversing the search space more efficiently. This process is known as *caching*.

This example does not serve to present the full complexity of tableau systems – it does however give some idea of the magnitude of the problem.

Most successful implementations of proof procedures do not rely only on the strength of the calculus. They apply various strategies, heuristics, and redundancy elimination to the system as well as sophisticated optimizations that include good data structures and strong indexing techniques.

3.1.4 Tableau-based solvers

A number of tableau-based solvers have been developed to date. For example, Beckert and Góre [12] developed a tableau-based solver that applies the free-variable tableau rules discussed above. This solver was initially called ModLeanTAP, and was later revised to become leanK 2.0.

As far as tableau provers are concerned, at the TANCS-2000 conference we have already mentioned, the tableau solvers FaCT [64], DLP [103] and RACE [51] were submitted and each returned good results. It must be noted that the primary focus of each of these solvers was description logics, although they are also able to deal with the modal logics $K(m)$ and $K4(m)$. The optimizations applied in FaCT and DLP are particularly well described in [65].

FaCT and DLP are heavily optimized decision procedures for description and modal logics. They use a method that is now standard for subsumption testing, namely translating subsumption tests into satisfiability tests and checking for satisfiability using an optimized tableau method. Many of the DLP optimizations originated from the work done by Horrocks using FaCT [63].

The key optimizations are as follows:

- *Lexical Normalization*: Incoming formulae are converted into normal form and common sub-formulae are uniquely stored. This allows the early detection of clashes – formulae $\neg\varphi$ and φ will be easily identified. Lexical simplification also takes place – formulae such as $\Box True$ and $\varphi \vee \neg\varphi$ are eliminated. The detection and handling of contradictory conjuncts can make a dramatic difference in solution time.

- *Semantic branching*: When DLP and FaCT decide to branch on a formula, they pick an element in the formula and assign *True* and *False* in turn to that element. Semantic branching on a formula $\varphi_1 \vee \varphi_2$ becomes $\varphi_1; \varphi_1 \vee \varphi_2$ and $\neg\varphi_1; \varphi_1 \vee \varphi_2$ – this explores each section of the search space only once. However, if φ_1 is a large formula, it can result in a significantly larger search space. This does not however seem to be a major concern.

- *Simplification*: Simplification is used to reduce the amount of non-determinism in the expansion of node labels. It is also called *boolean constraint propagation* – it looks for disjuncts in unexpanded disjunctions whose value is constrained due to values in the other disjuncts. For example, $(\varphi_1 \vee \varphi_2) \wedge \neg\varphi_1$ will become $\varphi_2 \wedge \neg\varphi_1$, thereby removing a choice point; if we have $\varphi_1 \wedge (\varphi_1 \vee \varphi_2)$, the formula $(\varphi_1 \vee \varphi_2)$ will be discarded.

- *Heuristic guided search*: A heuristic needs to be established to determine which sub-formula to branch on first. DLP uses simple heuristics to guide search.

- *Caching*: During satisfiability checking, the status of each node is cached. This reduces search time when a similar node is encountered and its satisfiability status is already known. Although it increases the space requirement, caching has produced dramatic gains in speed.

- *Dependency directed backtracking (backjumping)*: For each subformula, information on the choice points that led to its deduction are kept. When backtracking to a previous choice point, the algorithm checks to see if the clash depends on the choices made at the current node. If not, the branch at this node need not be considered and the previous node is then explored. Backjumping significantly reduces the search space.

RACE [50] and its successor, RACER [52], are based on the tableau calculus. They use optimized search techniques to guarantee good average-case performance. The techniques used are the standard optimization techniques; these being dependency directed backtracking, caching and semantic branching.

A comparison of RACE, DLP and FaCT is to be found in the paper detailing the results of the TANCS-2000 conference [87] Of these, DLP was the fastest.

We can see from the above discussion that good performance of tableau algorithms requires heavy optimization. This result is not surprising, considering the PSPACE-complete nature of the modal satisfiability problem.

3.1.5 Generic tableau solvers

For completeness, we note that generic tableau solvers have been developed to enable experimentation with many different modal and description logics. A disadvantage of strongly optimized solvers is that they are limited to a few logics.

TABLEAUX [87] is an example of a theorem proving system that is based on an adapted version of the traditional semantic tableau method. It explicitly builds a model but, as we would expect, faces problems with complexity. It has however proved to be a valuable tool for studying multimodal logics. Lotrec [31] is another automated theorem prover that was developed to evaluate different modal and description logics. It is also based on tableau methods.

3.2 Gentzen sequent calculus

We include a brief overview of the Gentzen sequent calculus, as it is also a special purpose-built algorithm. A good reference hereof is to be found in the paper [89].

The Gentzen sequent calculus uses a backward proof search approach that takes a modal formula and reduces it to the axioms of the system. A formula φ is unsatisfiable if and only if $\neg\varphi$ is provable.

Definition 3.2.1. *The sequent calculus rules for the modal logic K are:*

$$(\wedge)\quad \frac{\varphi, X \qquad \psi, X}{\varphi \wedge \psi, X} \qquad\qquad (\vee)\quad \frac{\varphi, \psi, X}{\varphi \vee \psi, X} \qquad\qquad (\Box)\quad \frac{\varphi, X}{\Box\varphi; \Diamond X}$$

where φ and ψ are formulae and X represents sets of formulae.

Note that these rules are applied using a bottom-up approach. One begins with the modal formula in the denominator and it is then reduced using the rules. One can see the similarity between these rules and those of the tableau approach, which decomposes the formula into a tree using a top-down approach. However, in the sequent calculus, the $(\wedge)$ rule generates branches, whereas in the tableau approach, the $(\vee)$ rule generates branches. If one of the branches fails, the whole proof attempt fails, whereas with the tableau approach, backtracking will occur.

Góre [49] discusses the connection between modal tableau systems and modal sequent systems and shows that a sequent is provable if and only if there is a closed tableau for the formula. We do not provide further details.

3.2.1 Sequent solvers

A sequent solver of note is the Logics Workbench LWB [59, 58], an interactive system for the modal logics K, KT, *S4* and *S5*, the multimodal logics $K(m)$, $KT(m)$ and *S4*(m) and various other logics. All decision procedures are based on sequent calculi and a separate specifically optimized procedure has been developed for each logic. Simplification of formulae is used, as was the case with tableau solvers, although this was not the case in its earlier versions. Structure sharing has also been implemented to increase efficiency.

Use-checking is applied to the backward application of the $(\wedge)$ rule to prune branches. Recall that this rule takes $A \wedge B, \Gamma$ and creates two branches A, Γ and B, Γ respectively. If Γ holds on the first branch, irrespective of A, it is not necessary to process the second branch.

The LWB software is accessible on the World Wide Web [75] and can be used to check the satisfiability or validity of a formula in any of the above-mentioned modal logics. The speed of processing complex formulae is impressive. This solver continues to be enhanced and improved.

3.3 The translation of modal logic into first-order logic

Theorem provers for first-order logics have well developed and mature proof procedures. First-order logic is semi-decidable and so various strategies have been put in place to prune the search space and recognize and eliminate redundancies. Because the development of proof procedures is complex, there are considerable benefits in translating modal logic into first-order logic – we need to define appropriate translation rules and test them to establish how well a particular proof procedure works. It is important to remember that we may be translating a decidable modal logic into a semi-decidable first-order representation.

Before we look at various translations and their effectiveness, we first need to understand the rules of first-order theorem proving which we will look at only briefly. For a more comprehensive overview, the reader is referred to textbooks such as [93].

3.3.1 First-order resolution theorem proving

We first define the language of first-order logic. We then define *prenex normal form* and *Skolem normal form*, which render first-order formulae into a form that can be used in resolution theorem proving.

Definition 3.3.1. *The language L of first-order logic consists of the following sets of primitive symbols:*

- *Variables: x, y, z, v, x_0, x_1, ...*
- *Constants: c, d, c_0, d_0, ...*
- *Connectives: $\vee, \neg, \wedge$*
- *Quantifiers: $\forall, \exists$*
- *Predicate symbols: P, Q, R, P_1, P_2, ..., which can have arity n = 1, 2, ...*
- *Function symbols: $f, g, h, f_0, f_1, ...$, which can have arity n = 1, 2, ...*
- *Punctuation: the comma and left and right parenthesis.*

Every variable, constant symbol and n-ary function symbol is called a term, and terms with no variables are called variable-free terms or ground terms. An atomic formula is an expression of the form $R(t_1, ..., t_n)$ where R is an n-ary predicate symbol and $t_1, ..., t_n$ are terms. Every atomic formula is a formula. If φ and ψ are formulae, so are $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$ and $(\neg\varphi)$. If v is a variable, then $((\exists v)\varphi)$ and $((\forall v)\varphi)$ are also formulae. An occurrence of a variable v in a formula φ is bound if there is a subformula ψ of φ containing the occurrence of v such that ψ begins with $((\forall v)$ or $((\exists v)$. An occurrence of v in φ is free if it is not bound.

First-order theorem provers use *refutation proofs* with the goal of obtaining a contradiction and their most widely used inference rule is *resolution.* Before attempting a resolution proof of a first-order formula, it is necessary to transform the formula, which must be in conjunctive normal form (Definition 2.2.21). The first transformation is into *prenex normal form* in which all quantifiers are moved to the beginning of the formula. This is then followed by *Skolemization*, which returns a *universal formula* φ which has only universal quantifiers ($\forall$).

Definition 3.3.2. *A first-order formula φ is in prenex normal form (PNF) if it is of the form*

$$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n \psi$$

where each $\mathcal{Q}_i \in \{\exists, \forall\}$ and ψ is quantifier-free.

Transformation of a first-order formula into prenex normal form is achieved by applying the following transformations, the correctness of which is proved in texts such as [93].

Lemma 3.3.3. *For any string of quantifiers $\mathcal{Q}x = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n$ where each $\mathcal{Q}$ is $\forall$ or $\exists$, and any formulae φ and ψ, we have the following provable equivalences:*

1. $\mathcal{Q}x\neg(\forall y)\varphi \equiv \mathcal{Q}x(\exists y)\neg\varphi$
2. $\mathcal{Q}x\neg(\exists y)\varphi \equiv \mathcal{Q}x(\forall y)\neg\varphi$
3. $\mathcal{Q}x((\forall y)\varphi \vee \psi) \equiv \mathcal{Q}x(\forall z)(\varphi(y/z) \vee \psi)$
4. $\mathcal{Q}x((\exists y)\varphi \vee \psi) \equiv \mathcal{Q}x(\exists z)(\varphi(y/z) \vee \psi)$

where z is a variable not occurring in φ or ψ or among the x_i and where (y/z) means that variable y is replaced with variable z.

Note that all connectives need to be transformed into $\neg$ and $\vee$ before the above Lemma can be applied.

Example 3.3.4. Let $\varphi = \forall x(\neg\forall y\, R(x, y) \wedge \forall z P(z))$. The following sequence transforms this formula into PNF by applying the above rules:

$$\begin{aligned}
\varphi &= \forall x(\neg\forall y\, R(x, y) \wedge \forall z P(z)) \\
&\equiv \forall x(\exists y \neg R(x, y) \wedge \forall z P(z)) \\
&\equiv \forall x \exists z_1(\neg R(x, z_1) \wedge \forall z P(z)) \\
&\equiv \forall x \exists z_1 \forall z_2(\neg R(x, z_1) \wedge P(z_2))
\end{aligned}$$

$\dashv$

Definition 3.3.5. *A formula φ is in Skolem normal form iff it is of the form*

$$\varphi = (\forall x_1)(\forall x_2) \dots (\forall x_n)\varphi'$$

where φ' is a quantifier-free formula, which is in conjunctive normal form (Definition 2.2.21).

Theorem 3.3.6. *For every first-order formula φ, there is a formula φ' in Skolem normal form such that φ is satisfiable iff φ' is satisfiable. φ' can be obtained from φ through the process of Skolemization.*

The proof of the Skolemization theorem is omitted – the reader is referred to texts such as [93] and [14]. Skolemization is achieved by applying the following Lemma:

Lemma 3.3.7. *For any sentence*

$$\varphi = \forall x_1 \dots \forall x_n \exists y \psi$$

of a language L, φ and

$$\varphi' = \forall x_1 \dots \forall x_n \psi(y/f(x_1, \dots, x_n))$$

are equisatisfiable when f is a function symbol not in L.

The intuition is that one can choose a function f such that, for any given $x_1, ..., x_n$, there is some y that makes the formula true if and only if $f(x_1, \dots, x_n)$ makes the formula true.

Example 3.3.8. Applying Skolemization, a formula such as

$$\varphi = \forall x_1 ... \forall x_n \exists y_1 ... \exists y_m R(x_1, ..., x_n, y_1, ... y_m)$$

will be replaced by

$$\varphi = \forall x_1 ... \forall x_n R(x_1, ..., x_n, f_1(x_1, ..., x_n), \dots, f_m(x_1, ..., x_n))$$

where each f_i is a new function symbol. $\dashv$

The above is a universal formula that can be used to test satisfiability – we would test the satisfiability of $R(x_1, \ldots, x_n, f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$.

We now look at *unification* and *resolution*. Resolution has only one rule, the aim of which is to reduce the non-determinism in a proof by eliminating clauses. This elimination process is facilitated by first applying unification to the clauses. We do not however give a comprehensive theoretic exposition hereof and attempt to explain both concepts via an example. See [93] and [14] for further details.

Example 3.3.9. Suppose we have a formula which consists of the conjunction of the two clauses

$$C_1 = P(f(x), y) \vee \neg Q(a, b, x)$$

and

$$C_2 = \neg P(f(g(c)), g(d)) \vee R(y)$$

We first apply *unification* in which we substitute literal $g(c)$ for x and $g(d)$ for y in C_1 to get

$$C_1' = P(f(g(c)), g(d)) \vee \neg Q(a, b, g(c))$$

We next apply *resolution* in which, if we have clause $C_1 = l \vee C_1'$ and $C_2 = \neg l \vee C_2'$ where l is any literal, we return $C = C_1' \vee C_2'$. C is referred to as the *resolvent*.

In the above, $l = P(f(g(c)), g(d))$, giving resolvent $C = \neg Q(a, b, g(c)) \vee R(y)$. $\dashv$

Resolution is combined with unification to give a proof procedure for the full predicate logic.

3.3.2 Translation of modal logic into first-order logic

Translation methods can be based on either the syntax or the semantics of a modal logic. The direct syntactic encoding of a Hilbert-style axiomatization of the modal logic into a first-order logic is not the optimal way of proving theorems in these logics [97]. We therefore do not consider syntactic translations further.

We look at translation methods based on the possible worlds semantics of modal logic. In cases where the accessibility relations are specified only implicitly as Hilbert axioms, a translation method is required to provide an explicit axiomatization of the properties of the accessibility relation. The paper by Ohlbach [96] provides the details of such translation methods.

There are two commonly followed approaches to this translation. The first involves a *relational translation* in which the possible world structure of the modal logic is explicitly represented using a predicate symbol. The second approach is a *functional translation* method that decomposes the accessibility relation into a set of accessibility functions that map worlds to accessible worlds. However, each of these approaches has certain drawbacks, as a result of which a *semi-functional translation* has been considered that combines the strengths of both.

3.3.3 The relational translation

This approach encodes the semantics of the source logic by simply transcribing it into the target logic. We look at the modal logic K and its translation $ST(w,\ \varphi)$ in which w is a world and φ is a modal formula. The translation consists of a binary predicate symbol R representing the accessibility relation and unary predicate symbols representing propositional letters. Modal formulae are represented as φ, ψ, $\ldots$; predicate symbols as P, Q,… and worlds as u, v, w, … with c representing a constant world.

Definition 3.3.10. *The recursive definition for the standard relational translation is:*

$$\begin{aligned}
&ST(w,p) = P(w)\\
&ST(w,\neg\varphi) = \neg ST(w,\varphi)\\
&ST(w,\varphi\wedge\psi) = ST(w,\varphi)\wedge ST(w,\psi)\\
&ST(w,\varphi\vee\psi) = ST(w,\varphi)\vee ST(w,\psi)\\
&ST(w,\Box\varphi) = \forall v(wRv\rightarrow ST(v,\varphi))\\
&ST(w,\Diamond\varphi) = \exists v(wRv\wedge ST(v,\varphi))
\end{aligned}$$

Note that the propositional connectives $\rightarrow$ and $\leftrightarrow$ need to be eliminated before the relational translation is applied.

Example 3.3.11. Using the relational translation, the modal K formula

$$\Box\Diamond p \rightarrow \Diamond\Box p$$

is translated as follows:

$$\begin{aligned}
&ST(w,\, \neg\Box\Diamond p \vee \Diamond\Box p) \\
&\quad = \quad ST(w,\, \neg\Box\Diamond p) \vee ST(w, \Diamond\Box p) \\
&\quad = \quad \neg ST(w,\, \Box\Diamond p) \vee \exists v(wRv \wedge ST(v,\, \Box p)) \\
&\quad = \quad \neg\forall v(wRv \rightarrow ST(v,\, \Diamond p)) \vee \exists v(wRv \wedge \forall u(vRu \rightarrow ST(u,\, p))) \\
&\quad = \quad \neg\forall v(wRv \rightarrow \exists u(uRv \wedge ST(u,\, p))) \vee \exists v(wRv \wedge \forall u(vRu \rightarrow P(u))) \\
&\quad = \quad \neg\forall v(wRv \rightarrow \exists u(uRv \wedge P(u))) \vee \exists v(wRv \wedge \forall u(vRu \rightarrow P(u))) \\
&\quad = \quad \forall v(wRv \rightarrow \exists u(uRv \wedge P(u))) \rightarrow \exists v(wRv \wedge \forall u(vRu \rightarrow P(u)))
\end{aligned}$$

Since this holds for all worlds w, it can be written as

$$\forall w(\forall v(wRv \rightarrow \exists u((vRu \wedge P(u))) \rightarrow \exists v(wRv \wedge \forall u((vRu \rightarrow P(u)))))) \tag{3.2}$$

⊣

The standard relational translation considers modal languages as fragments of first-order or other predicate logics. In Ohlbach [97], this first-order fragment is analyzed and proved to have the finite model property as well as the tree model property (Section 2.7) – hence it is robustly decidable and has good logical and computational behavior, as was shown by Vardi [127].

We can easily extend the standard relational translation described above to cater for a different logic. For example, the multi-modal logic $K(m)$ can be obtained by replacing the translation for $ST(w,\, \Box\varphi)$ with

$$ST(w, [a]A) = \forall v(wR_a v \rightarrow ST(v, A))$$

in Definition 3.3.10.

Similar extensions can be applied to define the translation for many other modal logics.

We demonstrate this approach in the following example where we apply the standard relational translation to a modal formula and then Skolemize it to generate a universal formula. We then apply unification and resolution.

Example 3.3.12. Consider the modal formula $\Box(p \rightarrow \Diamond p)$, which is satisfiable. We apply the relational translation, omitting the details of some of the steps relating to propositional logic.

$$\begin{aligned}
ST(w, \Box(p \rightarrow \Diamond p)) &= \forall v(wRv \rightarrow ST(v, (p \rightarrow \Diamond p))) \\
&= \forall v(wRv \rightarrow (ST(v, \neg p) \vee ST(v, \Diamond p)) \\
&= \forall v(wRv \rightarrow (\neg P(v) \vee \exists u(vRu \wedge ST(u, p)))) \\
&= \forall v(\neg wRv \vee \neg P(v) \vee \exists u(vRu \wedge P(u))) \\
&\simeq \forall v(\neg wRv \vee \neg P(v) \vee (vRf(v) \wedge P(f(v)))) \\
&\qquad \text{(Skolemization on } \exists) \\
&\simeq \forall v((\neg wRv \vee \neg P(v) \vee vRf(v)) \wedge (\neg wRv \\
&\qquad \vee \neg P(v) \vee P(f(v)))) \\
&\qquad \text{(application of CNF (Definition 2.2.21))} \\
&\simeq (\neg wRv \vee \neg P(v) \vee vRf(v)) \wedge (\neg wRv \\
&\qquad \vee \neg P(v) \vee P(f(v))) \text{ (universal formula)}
\end{aligned}$$

Resolution is now applied to the two disjunctive clauses:

1. $\neg wRv \vee \neg P(v) \vee vRf(v)$
2. $\neg wRv \vee \neg P(v) \vee P(f(v))$

These clauses have two resolvents giving:

3. $\neg wRv \vee \neg P(v) \vee \neg P(f(v)) \vee P(f^2(v))$

   (resolve on $vRf(v)$ in 1. and $\neg wRv$ in 2.)
4. $\neg wRf(v) \vee f(v)Rf^2(v) \vee \neg wRv \vee \neg P(v)$

   (resolve on $\neg P(v)$ in 1. and $P(f(v))$ in 2.)

Clauses 2 and 4 resolve to produce

5. $\neg wRv \vee \neg P(v) \vee \neg wRf^2(v) \vee f^2(v)Rf^3(v)) \vee \neg wRf(v)$

   (resolve on $P(f(v))$ in 2. and $\neg P(v)$ in 4.)

Clauses 2 and 5 resolve again and produce a clause with even higher term complexity. ⊣

In the example above, the size of the clauses becomes more and more complex. This set has infinitely many resolvents, which shows that a standard resolution procedure does not terminate for the relational translation of satisfiable modal formulae.

This translation into first-order logic unfortunately has an inherent non-determinism when standard resolution techniques are applied to it. For unsatisfiable formulae, any complete resolution procedure generates a contradiction, whereas for satisfiable formulae, resolution may not terminate. Plain resolution unfortunately does not take into consideration the characteristics of the original modal logic, which is decidable.

Various approaches have been developed to address these shortcomings. One approach is to develop special resolution refinements that can then to be applied to the translated formulae – ordering and selection-based refinements are presented in a paper by de Nivelle et al. [30]. Another approach makes use of the observation that quantifiers in the relational translation are guarded by the accessibility relation – this fragment of first-order logic is called the *guarded fragment*. A resolution decision procedure has been developed for this guarded fragment [29]. None of these approaches however generates particularly good results.

An alternative approach was developed by Areces et al. [6] that exploits the tree model property of modal logic by encoding layering present in tree models into the syntax of modal formulae. Formulae are first translated into an extended multi-modal language where each modal depth has its own modal operators and its own propositional letters. This avoids the situation where clauses in different levels in the tree are resolved.

Example 3.3.13. If one applies this approach to the formula $\Box(p \to \Diamond p)$ used in Example 3.3.12, the translated formula is reduced instead to the clauses:

1. $\neg wR_1y \vee \neg P_1(v) \vee vR_2f(v)$
2. $\neg wR_1y \vee \neg P_1(v) \vee P_2(f(v))$

Note that the details of this derivation are available in [6]. The literals with subscript i correspond to a modal operator or propositional letter occurring at modal depth i. We can no longer resolve these two clauses, which gives a better result. $\dashv$

The above approach was benchmarked against other provers and returned some good results.

In general, the relational translation without extra strategies does not return good results, particularly as resolution is possible between two literals that do not occur at the same world. This is clearly seen in the difference between the resolution proofs of Examples 3.3.12 and 3.3.13. A further observation is that it is very difficult to read resolution proofs, as can be seen in Example 3.3.12.

3.3.4 The functional translation

Because of the limitations discussed above, the functional translation was developed, where the standard Kripke semantics are reformulated by decomposing the accessibility relation into a set AF of accessibility functions which map worlds to accessible worlds.

Definition 3.3.14. *For any binary relation R on a non-empty set W of worlds, we can define a set AF_R of accessibility functions – that is, a set of partial functions $\gamma : W \to W$, such that*

$$\forall u, v(uRv \leftrightarrow (\exists \gamma(\gamma \in AF_R \wedge \gamma(u) = v)))$$

Thus, for any worlds u and v, uRv if and only if there is some partial function γ mapping u to v.

For simplicity, $\gamma(u)$ is represented as $[u\gamma]$, $\delta(\gamma(u))$ as $[u\gamma\delta]$ and so on. Note that this notation reflects paths in the underlying frame. For example, $[u\gamma\delta]$ denotes the world reached from the world u via the functions γ and δ – it reflects a single path.

It is important to note that there are many ways in which a relation R can be decomposed. Consider the relation $R_1 = \{w_1Rw_2, w_1Rw_3, w_2Rw_4, w_2Rw_5\}$. It can be decomposed into two functions $\{\gamma_1, \gamma_2\}$ either as

$$\begin{aligned} &\gamma_1(w_1) = w_2; \gamma_1(w_2) = w_4 \\ &\gamma_2(w_1) = w_3; \gamma_2(w_2) = w_5 \end{aligned}$$

or

$$\begin{aligned} &\gamma_1(w_1) = w_2; \gamma_1(w_2) = w_5 \\ &\gamma_2(w_1) = w_3; \gamma_2(w_2) = w_4 \end{aligned}$$

Hence, there are many possible sets of accessibility functions that can represent the same binary relations. An accessibility function γ is termed *maximally defined* whenever, for each u and v with uRv, $\gamma(u)$ is defined.

When the accessibility relation R is *serial* (Definition 2.3.5), it can be decomposed into a set of total functions – recall that, for a serial accessibility relation, it holds that for every $u \in W$, there is some $v \in W$ such that uRv. However, when R is *not* serial, we need to define a special predicate de_R called the *dead-end* predicate, to ensure a set of total functions [97]. This is necessary as, for every $u \in W$, there is not necessarily some $v \in W$ such that uRv. Hence, we do not always have a set of total functions.

Definition 3.3.15. *The dead-end predicate de_R is defined as follows:*

$$\forall u(de_R(u) \leftrightarrow \forall \gamma(\gamma \in AF_R \rightarrow [u\gamma] = \perp))$$

Definition 3.3.16. *A non-serial relation R can now be defined in terms of a set AF_R of total functions $\gamma : W \rightarrow W$ as follows:*

$$\forall u, v(uRv \leftrightarrow (\neg de_R(u) \wedge \exists \gamma(\gamma \in AF_R \wedge [u\gamma] = v)))$$

We have expressed the accessibility relation as a set of accessibility functions and can now define the functional translation. The target logic will be a multi-sorted logic with sorts W and AF – that is, it has two distinct sets of variables W and AF. The variables $u, v, w, ...$ are of sort W; the functional variables are denoted by $\gamma, \gamma_1, \gamma_2, ...$ and are of sort AF.

Definition 3.3.17. *The functional translation $FT(w, A)$ is defined as follows:*

$$\begin{array}{lcl}
FT(w, p) & = & P(w) \\
FT(w, \neg\varphi) & = & \neg FT(w, \varphi) \\
FT(w, \varphi \wedge \psi) & = & FT(w, \varphi) \wedge FT(w, \psi) \\
FT(w, \varphi \vee \psi) & = & FT(w, \varphi) \vee FT(w, \psi)
\end{array}$$

with the translation of the modal atoms being defined according to the accessibility relation:

$$FT(w, \Box\varphi) = \begin{cases} \forall \gamma : AF \; FT([w\gamma], \varphi), & \textit{serial accessibility relation} \\ \neg de(w) \rightarrow \forall \gamma : AF \; FT([w\gamma], \varphi) & \textit{otherwise} \end{cases}$$

$$FT(w, \Diamond\varphi) = \begin{cases} \exists\gamma : AF\ FT([w\gamma], \varphi), & serial\ accessibility\ relation \\ \neg de(w) \wedge \exists\gamma : AF\ FT([w\gamma], \varphi) & otherwise \end{cases}$$

In order to add additional properties of accessibility relations, the associated axioms must be translated into the functional language. This is usually straightforward, but the translated axioms are in general equations. For example, the functional counterpart of reflexivity is

$$\forall w \exists\gamma : AF[w\gamma] = w$$

and of transitivity is

$$\forall w \forall\gamma_1, \gamma_2 \exists\gamma((\neg de(w) \wedge \neg de([w\gamma_1])) \to [w\gamma_1\gamma_2] = [w\gamma]).$$

This adds a complexity to the translation that we did not have with the relational translation. The reader is referred to the paper [97] for further details.

We provide the functional translation of the modal K formula $\Box\Diamond p \to \Diamond\Box p$, but without the details of the derivation. Its translation is

$$\forall w(\neg de(w) \to \forall\gamma(\neg de([w\gamma]) \wedge \exists\delta P([w\gamma\delta])) \to (\neg de(w) \wedge \exists\gamma(\neg de[w\gamma] \to \forall\delta P([w\gamma\delta])))) \tag{3.3}$$

Note the difference between Equation (3.3) and the standard relational translation of this formula, given in Equation (3.2) (Example 3.3.11). In Equation (3.2), the accessibility relations between the worlds w, u and v are specified, while Equation (3.3) refers only to the world w and specifies the paths from w.

Further simplification of Equation (3.3) is required before being able to apply resolution, as it contains existential quantifiers. Recall that existential quantifiers need to be eliminated in order to obtain a universal formula. However, whenever we replace existentially quantified variables with Skolem functions, it can result in complex terms being built up during resolution – as was the case with the relational translation. Skolem functions could be avoided if it was possible to pull existential quantifiers originating from $\Diamond$-operators over universal quantifiers origination from $\Box$-operators. That is, if we could change an expression such as $\forall x \exists y P[xy]$ into $\exists y \forall x P[xy]$, we could replace the existential quantifier y with a constant to give $\forall x P[xc]$.

Ohlbach and Schmidt [98] show that this is possible for *functional frames* that are maximal, these being those frames in which the set of accessibility functions contains all possible accessibility functions. They call this the *quantifier exchange rule* – the quantifier exchange operator Υ swops quantifiers according to the principle:

$$\exists\gamma\forall\delta A \leftrightarrow \forall\delta\exists\gamma A$$

This rule was incorporated in the *optimized functional translation* that was defined by Schmidt [111] and that consists of a sequence of transformations. The first transformation, which is the functional translation, gives the *basic path logic* in which the AF-terms are called *basic paths.* The next transformation applies the operator Υ to yield the optimized functional translation. The final step is the formulation of a clausal form that contains only Skolem constants. Ordinary resolution can now be applied to the resultant formula.

The optimized functional translation returns good results.

3.3.5 The semi-functional translation

The semi-functional approach was first proposed by Nonnengart [95]. It combines the advantages of the relational and functional translation while trying to avoid their respective disadvantages. One of the advantages of the relational translation is that the translation result mirrors the Kripke semantics of the modal logic, while its major disadvantage is its exponential growth. The functional translation provides a very compact translation result in the case of serial modal logics but has the disadvantage that other properties of accessibility relations such as reflexivity, transitivity, symmetry and so on have to be encoded specifically and thus require a strong equality handling mechanism.

In the semi-functional translation, the $\Diamond$-operator is translated functionally and the $\Box$-operator is translated relationally. Since a different translation method is applied to the $\Diamond$ and $\Box$ operators, we must ensure that $\Box A \leftrightarrow \neg\Diamond\neg A$. Nonnengart [95] proved this result for serial modal logics by showing that every world that is accessible via the accessibility functions is also accessible via the accessibility relation. The disadvantage of this translation is that it has been applied primarily to serial modal logics.

For more details of this approach, refer to Ohlbach et al. [97].

3.3.6 First-order translation solvers

We have outlined the relational and functional translations, as well as the semi-functional adaptation proposed to ensure adequate decision procedures. We now look at various translation-based solvers.

Weidenbach et al. [132] developed a translation-based solver that makes use of the *optimized functional translation*. It uses a module FLOTTER as a translator of modal logic into first-order logic which is in clausal normal form. It then uses SPASS, an existing fast and sophisticated state-of-the-art theorem prover, as the theorem prover for the resulting first-order logic. We have seen that the optimized functional translation results in a translation to which ordinary resolution without refinement strategies can be applied. Hence such translated formulae can be fed into any state-of-the-art theorem prover.

Another optimized functional translation called TA was developed that also uses SPASS as the first-order theorem prover [69, 72]. However, TA was found to perform poorly against various other solvers [97].

MSPASS [71] was then developed as an enhancement of SPASS. It supports all the translations we have discussed and includes optimizations that are specific to first-order logic. It was benchmarked at the TANCS-2000 conference and its performance was very good for some of the test sets, but poor on others [71].

We have already discussed the approach taken by Areces et al. [6] on page 59. When this approach was tested using an existing first-order solver, Bliksem, it produced reasonable results for some problem sets but was unable to solve others. The authors conclude that the problem of finding efficient proof procedures for the translated formulae has not yet been satisfactorily solved.

Ohlbach et al. [97] concluded that the problem of finding efficient proof procedures for translated modal logic formulae has not yet been satisfactorily solved. Special purpose-built refinements to complement the translations are required if an acceptable performance is to be achieved. Hence, the translation approach has certain limitations. A major benefit has been to provide an understanding of how the good computational and logical behavior of propositional modal logics is related to first-order logics, which are undecidable.

3.4 The SAT-based approach

Giunchiglia and Sebastiani [46, 45] developed a modal logic solver called KSAT that makes use of a propositional decision procedure (or SAT solver) to handle the propositional logic that is embedded in each modal layer of a set of modal formulae. As with the first-order translation, the benefit of this approach is that SAT solvers are mature, efficient and highly optimized. KSAT indirectly benefits from any improvements made to its SAT solver component. Of the available SAT solvers, the most efficient are based on the Davis-Putnam-Longemann-Loveland (DPLL) procedure [28, 27] that we now discuss.

3.4.1 The DPLL SAT algorithm

The propositional satisfiability problem (SAT) is applicable to classical propositional logic formulae. It is a decision procedure that determines if there is some assignment of true and false values to the variables in the Boolean formula such that the formula evaluates to true. A formula of propositional logic is said to be satisfiable if truth values can be assigned to its variables in such a way that the formula is true. In other words, it is a sub-problem of the modal satisfiability problem, but without any modalities.

The SAT problem has typically been solved using the DPLL SAT procedure which was developed by Davis, Putnam, Longemann and Loveland in the early 1960s. It is applicable only to formulae in conjunctive normal form (CNF) (Definition 2.2.21) and makes use of a two-phase approach – the first phase simplifies the set of formulae as much as possible, while the second phase applies a heuristic search.

The first phase of this procedure makes use of *unit subsumption* and *unit resolution* to eliminate clauses and simplify the set of formulae. This combination is referred to as *unit propagation.*

Rule 3.4.1. *A formula $\varphi = l \wedge (\varphi_1 \vee l) \wedge \varphi_2$ in CNF is logically equivalent to the formula $\varphi' = l \wedge \varphi_2$. Removing the clause $(\varphi_1 \vee l)$ from φ to produce φ' is called unit subsumption.*

The unit clause l is subsumed by the clause $(\varphi_1 \vee l)$, which may therefore be removed from φ without affecting the satisfiability of the resulting formula.

Rule 3.4.2. *A formula $\varphi = l \wedge (\varphi_1 \vee \neg l) \wedge \varphi_2$ in CNF is logically equivalent to the formula $\varphi' = l \wedge \varphi_1 \wedge \varphi_2$. Replacing the clause $(\varphi_1 \vee \neg l)$ with φ_1 to produce φ' is called unit resolution.*

Unit resolution is applied to φ and $\neg l$ can be removed from each clause in which it occurs without affecting the satisfiability of the resulting formula.

Unit subsumption and unit resolution are applied to all propositional unit clauses in the formula until either a contradiction is achieved or there are no more propositional unit clauses in the formula to which it can be applied.

Example 3.4.3. Suppose we have the propositional formula

$$\varphi = l \wedge (\psi_1 \vee l) \wedge (\psi_2 \vee \neg l) \wedge \varphi_1$$

where ψ_1 and ψ_2 are disjunctions of any number of literals and φ_1 is a conjunction of any number of clauses. For φ to be satisfiable, l must be assigned the value *True*. Then $(\psi_1 \vee l)$ is *True* and it can be removed by unit subsumption. In the clause $(\psi_2 \vee \neg l)$, $\neg l$ must be false. Hence, it is removed by unit resolution and the clause is replaced with ψ_2. This formula is simplified to become $\varphi' = l \wedge \psi_2 \wedge \varphi_1$. $\dashv$

The second phase of the DPLL SAT procedure applies a heuristic to select a suitable literal l to process – this is termed *splitting*. The procedure then evaluates the satisfiability of $\varphi \cup \{l\}$. If it is not satisfiable, it verifies the satisfiability of $\varphi \cup \{\neg l\}$. Since only one of these two formulae can be satisfiable, this is a proof procedure. It continues in this manner until the satisfiability or otherwise of the formula has been determined. Note that in some cases, a partial assignment may suffice (when we remove a clause such as $(\psi \vee l)$, we have not assigned values to the variables in ψ).

Algorithm 3. *The DPLL algorithm can be summarized in the following pseudo-code in which φ is a propositional formula:*

```
function DPLL($\varphi$)
    $\varphi_1$ := unit_propagation($\varphi$);
    if $\varphi_1$ = False
        then return False;
    if $\varphi_1 = \emptyset$
        then return True;
    $l$ := choose_literal($\varphi_1$);        // apply some heuristic to select $l$
    return DPLL($\varphi_1 \cup l$) or
        DPLL($\varphi_1 \cup \neg l$);
end;

function unit_propagation($\varphi$)
    while $\varphi$ contains a propositional unit clause $l$ do
        if $\varphi$ contains a propositional unit clause $\neg l$,
            return False;
        remove all clauses containing $l$ from $\varphi$;
        remove $\neg l$ from all clauses in $\varphi$ that contain it;
    return $\varphi$;
end;
```

The first part of the procedure is deterministic, the second is non-deterministic. The search is greatly affected by two factors – firstly by the ordering of variables and secondly by the choice of literals in the branching step. Extensive research has been done on the definition of branching rules to optimally pick these literals. The paper [81] provides an excellent overview of the heuristic methods that have been developed. For example, a heuristic called MOMS counts only occurrences of literals in minimum size clauses and selects one of these literals for the split. The MAXO heuristic selects the literal with the maximum number of occurrences in a formula. The paper notes that none of these heuristics is the best in all cases. To overcome this limitation, they propose a reinforcement-learning approach.

3.4.2 The KSAT solver

We now look at the design of the KSAT solver as applied to the modal logic K. Note that in the papers referred to, some of the results have been proved for the modal logic $K(m)$, but for simplicity, we restrict this discussion to K. Recall that propositional atoms are denoted as p, p_1, ..., q, q_1, ... and formulae as φ, φ_1, ..., ψ, ψ_1. We use $\alpha, \alpha_1, \ldots$ and $\beta, \beta_1, \ldots$ to construct modal atoms $\Box\alpha, \Box\alpha_1, \ldots$ and $\Box\beta, \Box\beta_1, \ldots$.

We require the following preliminary definitions.

Definition 3.4.4. *Given a modal formula φ in conjunctive normal form, a top-level atom in φ is either a propositional atom (Definition 2.2.7) or a modal atom (Definition 2.2.8) which occurs in φ under the scope of no boxes.*

Example 3.4.5. *Consider the modal formula*

$$\varphi = \neg\Box(p_1 \vee p_2) \wedge (p_1 \vee \Box p_3 \vee \Box\Box\Box(p_2 \vee \Box(p_3 \vee \Box p_4))).$$

Its top-level atoms are $\{\Box(p_1 \vee p_2), p_1, \Box p_3, \Box\Box\Box(p_2 \vee \Box(p_3 \vee \Box p_4))\}$ whereas, for example, p_3 and $\Box p_4$ are not top-level atoms as they occur under the scope of a box. ⊣

Definition 3.4.6. *A total truth assignment μ for a modal K formula φ is the set of literals*

$$\begin{aligned} \mu = \{&\Box\alpha_1, \ldots, \Box\alpha_n, \neg\Box\beta_1, \ldots, \neg\Box\beta_m, \\ &p_1, \ldots, p_r, \neg q_{r+1}, \ldots, \neg q_s\} \end{aligned} \tag{3.4}$$

such that every top-level atom of φ occurs either positively or negatively in μ.

This truth assignment can also be represented as

$$\mu = \textstyle\bigwedge_i \Box\alpha_i \wedge \bigwedge_j \neg\Box\beta_j \wedge \gamma$$

where

$$\gamma = \textstyle\bigwedge_{1\le k\le r} p_k \wedge \bigwedge_{r+1\le k\le s} \neg q_k$$

In the above, $\Box\alpha_i \in \mu$ means that $\Box\alpha_i$ is assigned the value *True* and $\neg\Box\beta_i \in \mu$ means that $\Box\beta_i$ is assigned the value *False*. Similarly, the p_i are assigned the value *True* and the q_i are assigned the value *False*.

Definition 3.4.7. *A total truth assignment μ for φ propositionally satisfies φ, written $\mu \models_p \varphi$, if and only if φ evaluates to True under the assignment μ.*

Definition 3.4.8. *A partial truth assignment μ for φ is a truth assignment to a proper subset of the top-level atoms of φ. A partial truth assignment propositionally satisfies φ if and only if all the total assignments for φ that extend μ propositionally satisfy φ.*

Definition 3.4.9. *The restricted truth assignment μ^r for a modal K formula φ is defined as*

$$\mu^r = \{\Box\alpha_1, \ldots, \Box\alpha_n, \neg\Box\beta_1, \ldots, \neg\Box\beta_m\}$$

and can also be represented as

$$\mu^r = \textstyle\bigwedge_i \Box\alpha_i \wedge \bigwedge_j \neg\Box\beta_j$$

Having provided these definitions, we now illustrate their application in the following example.

Example 3.4.10. Consider the modal formula φ, where

$$\begin{aligned}\varphi = \quad & (p_1 \vee \Box(p_2 \wedge p_3)) \wedge \\ & (p_4 \vee \neg p_1) \wedge \\ & (\Box(p_4 \vee p_5 \vee p_6) \vee \neg\Box(p_4 \vee p_5 \vee p_6) \vee p_3) \wedge \\ & (p_2 \vee \neg p_4)\end{aligned}$$

In terms of Definition 3.4.4, the top-level atoms in φ are $\{p_1, p_2, p_3, p_4, \Box(p_2 \wedge p_3), \Box(p_4 \vee p_5 \vee p_6)\}$. A total truth assignment will assign a truth value to each top-level atom – for example

$$\mu = \{p_1, p_2, \neg p_3, p_4, \neg\Box(p_2 \wedge p_3), \Box(p_4 \vee p_5 \vee p_6)\}$$

This truth assignment propositionally satisfies φ. This formula is also propositionally satisfied using the partial truth assignments

$$\mu = \{\Box(p_4 \vee p_5 \vee p_6),\, p_1,\, p_2,\, p_4\}$$

and

$$\mu = \{\Box(p_2 \wedge p_3),\, \Box(p_4 \vee p_5 \vee p_6),\, p_2,\, p_4\}$$

⊣

The next question that needs to be answered is how to relate modal satisfiability to propositional satisfiability. The first step in establishing this relationship is to approximate a modal formula φ as a propositional formula in its top-level atoms, which can then be solved by any state-of-the-art SAT solver. This approximation is achieved as follows.

Definition 3.4.11. *The propositional approximation of a modal formula φ, denoted $Prop(\varphi)$, is defined recursively as follows:*

$$\begin{aligned}
&Prop(p) := p\\
&Prop(\neg\varphi) := \neg Prop(\varphi)\\
&Prop(\varphi_1 \wedge \varphi_2) = Prop(\varphi_1) \wedge Prop(\varphi_2)\\
&Prop(\Box\varphi) = x_i[\Box\varphi] \textit{ where } x_i[\Box\varphi] \textit{ denotes a fresh propositional literal.}
\end{aligned}$$

Example 3.4.12. Consider the modal formula of Example 3.4.10. It can be approximated as the following propositional formula, in which each modal atom has been replaced with a new propositional atom p_i.

$$\begin{aligned}
Prop(\varphi) = \quad & (p_1 \vee p_5) \wedge\\
& (p_4 \vee \neg p_1) \wedge\\
& (p_6 \vee \neg p_6 \vee p_3) \wedge\\
& (p_2 \vee \neg p_4)
\end{aligned}$$

The propositional formula $Prop(\varphi)$ is then fed into a SAT solver and a truth assignment is returned for each of its variables. ⊣

Before we look at the KSAT algorithm, we look at two theorems developed and proved by Giunchiglia and Sebastiani [46].

Theorem 3.4.13. *A modal formula φ is K-satisfiable if and only if there exists a K-satisfiable truth assignment μ such that $\mu \models_p \varphi$.*

This means that the K-satisfiability of a formula φ can be reduced to determining the K-satisfiability of its truth assignments.

We now look at how to determine the satisfiability of the modal portion of φ (Definition 3.4.9) – that is, how do we determine the truth assignment of μ^r?

Theorem 3.4.14. *The restricted assignment μ^r is satisfiable if and only if the formula*

$$\varphi_j = \textstyle\bigwedge_i \alpha_i \wedge \neg\beta_j$$

is K-satisfiable for every $\neg\Box\beta_j$ occurring in μ^r.

Informally, we need to show that $\bigwedge_i \alpha_i$ is true at all worlds accessible from the current world. Now suppose we have $\neg\Box\beta_1$ and $\neg\Box\beta_2$ in the assignment μ^r. We need to show that there is *some* world accessible from the current world at which $\neg\beta_1$ is true and some world at which $\neg\beta_2$ is true – and these worlds do not need to be the same. Hence we need to test the satisfiability of $\bigwedge_i \alpha_i \wedge \neg\beta_1$ and $\bigwedge_i \alpha_i \wedge \neg\beta_2$ – effectively, we are generating two branches on the modal tree that has φ as its root.

Note that the modal depth of every formula φ_j is strictly smaller than the depth of the original formula φ.

We now proceed to look at the KSAT algorithm in detail. It consists of two basic steps:

Step 1: Propositional reasoning – finding a truth assignment μ for φ such that $\mu \models_p \varphi$.

Step 2: Checking the K-satisfiability of μ by generating the corresponding restricted assignment μ^r and determining the satisfiability of the φ_j formulae.

Algorithm 4. *The KSAT algorithm can be summarized in the following pseudo-code in which φ is a modal formula:*

```
function KSAT($\varphi$)
    return KSAT$_W$($\varphi, \emptyset$);
end;
```

```
function KSAT$_W(\varphi, \mu)$
    if $\varphi$ = True
        then return KSAT$_A(\mu)$;
    if $\varphi$ = False
        then return False;
    if {a propositional unit clause $(l)$ occurs in $\varphi$}
        then return KSAT$_W(assign(l, \varphi), \mu \cup \{l\})$;
    $l := choose\_literal(\varphi)$;
    return KSAT$_W(assign(l, \varphi), \mu \cup \{l\})$ or
            KSAT$_W(assign(\neg l, \varphi), \mu \cup \{\neg l\})$;
end;

function KSAT$_A(\{\Box\alpha_1, \ldots, \Box\alpha_N, \neg\Box\beta_1, \ldots, \neg\Box\beta_M,$
        $p_1, \ldots, p_r, \neg q_{r+1}, \ldots, \neg q_s\})$
    for each conjunct $\neg\Box\beta_j$ do
        $\varphi_j := \bigwedge_i \alpha_i \wedge \neg\beta_j$;
        if not KSAT$(\varphi_j)$
            then return False;
    return True;
end;
```

KSAT takes as input a modal propositional formula φ and returns a truth value asserting whether it is K-satisfiable or not. KSAT invokes KSAT$_W$ which is a variant of DPLL – in particular, the functions *assign* and *choose_literal* relate to the two phases of the DPLL procedure. However, *assign* now adds the truth value of the propositional unit clause l to the truth assignment μ. Truth assignments μ^r are then generated, whose K-satisfiability is recursively checked in the procedure KSAT$_A$. KSAT$_W$ assigns values to both propositional literals and modal literals. For the negative modal literals in μ^r, KSAT$_A$ generates a formula $\varphi_j = \bigwedge_i \alpha_i \wedge \neg\beta_j$ for each $\neg\Box\beta_j$ occurring in μ^r and then invokes KSAT with φ_j. This is repeated until either KSAT returns $False$ or no more $\neg\Box\beta_j$'s are available, in which case KSAT$_A$ returns *True*. Each such iteration decreases the modal depth of the formula. Essentially, this amounts to recursively generating an assignment μ that propositionally satisfies φ

and then testing whether μ is satisfiable. It can be seen that the depth of recursion is limited by the modal depth of the set of formulae.

KSAT is the result of the DPLL-application and $\neg\Box/\Box$-elimination by the application of the (K) tableau rule of Definition 3.1.4. For each $\neg\Box\beta_j$ in μ^r, KSAT$_A$ applies $\neg\Box$-elimination (that is, $\Diamond$-elimination), generating $\neg\beta_j$ in an implicit new world w' accessible from the current world w. Then, for every $\Box\alpha_i$ in μ^r, it applies $\Box$-elimination, adding α_i to w'. As a result, $\varphi_j = \alpha_1 \wedge \ldots \wedge \alpha_n \wedge \neg\beta_j$ must be tested for satisfiability at world w'. KSAT is then invoked with φ_j to evaluate its satisfiability at the world w'. If KSAT returns $True$ for each of these φ_js, then φ is satisfiable. If it returns $False$, φ is not satisfiable.

KSAT therefore proceeds in a depth-first manner, each time working on a single world. As no confusion can arise between worlds, there is no need to keep labels explicitly. This approach can easily be extended to deal with $K(m)$, as discussed by Giuchiglia and Sebastiani [46].

3.4.3 SAT-based modal solvers

Various versions of KSAT have been produced, each of which has different optimizations.

The basic algorithm discussed above was optimized in several ways. In the paper by Giunchiglia and Sebastiani [46], the following improvements are made:

- The modal atoms are sorted according to some order on the set of subformulae. After sorting, one can trivially eliminate clauses such as $\Box(p_1 \vee p_2) \vee \neg\Box(p_1 \vee p_2)$, which are tautologies.

- For each of the $\neg\beta_j$s, the common component $\bigwedge_i \alpha_i$ is re-evaluated. An alternative approach that enables $\bigwedge_i \alpha_i$ to be evaluated only once is as follows: All of the $\neg\beta_j$s are placed in a set B. A set of truth assignments that satisfy $\bigwedge_i \alpha_i$ is found. Each truth assignment is checked against each $\neg\beta_j$ and those $\neg\beta_j$ that are satisfiable are removed from the set B. If B is empty, μ^r is satisfiable.

- The satisfiability of an incomplete assignment is checked before the split introduced in *choose-literal*. This is called *early pruning*. For example, in the case of the partial assignment

$$\mu_1 = \Box(\neg p_1 \vee p_2 \vee p_3) \wedge \neg\Box(\neg p_1 \vee p_2 \vee p_3)$$

KSAT$_A$ will check the K-satisfiability of the formula

$$(\neg p_1 \vee p_2 \vee p_3) \wedge p_1 \wedge \neg p_2 \wedge \neg p_3$$

and find it unsatisfiable. Without this step, splitting using p_1, p_2 and p_3 respectively would have taken place. This saves a considerable amount of processing.

Giunchiglia et al. [42] later developed KSATC and KSATLISP (C and LISP versions respectively) that differ primarily on their splitting strategy. KSATC is built on a highly optimized DPLL algorithm that provides efficient data structures for literal assignment and the partial evaluation of formulae. Literals are assigned and unassigned dynamically inside a set of formulae simply by moving pointers, and this takes place in time proportional to the number of their occurrences. The DPLL algorithm also includes smart splitting heuristics – it splits on the literal that occurs most often in the shortest clauses. KSATLISP on the other hand splits on the atom occurring most often in the input formula. Of these, KSATLISP ends up spanning a bigger search space than KSATC.

Giunchiglia and Tacchella [43] subsequently developed a new SAT solver called *SAT, which is based on an existing SAT solver, namely SATO 3.2, one of the most efficient SAT checkers then available [135]. *SAT allowed for easy integration of SATO and makes use of a commercially available library of data types that provides highly optimized data structures. By following this approach, they were able to capitalize on several years of experience in building highly optimized data structures and algorithms for propositional satisfiability. While other modal logic solvers need to include optimization techniques, in this case, these techniques were already included in SATO. They did however apply early pruning and modal backjumping at the modal layer.

The KSAT solver was later enhanced to deal with the modal logic KT. However, it is unable to deal with any other modal logics.

Various benchmarks have been run comparing KSAT to other solvers. In Section 4.3.13 of the next chapter we look at its performance at the TANCS-2000 conference, the details of which are provided in [119] and [87]. With some of the data sets, its performance was however poor.

3.5 The CSP-based approach

Modal solvers for each of the approaches discussed so far were submitted and benchmarked at the TANCS-2000 conference. Subsequently, Brand et al. [18] developed a modal logic solver called KCSP that is applicable to the normal modal logic K. It translates a modal satisfiability problem into a layered set of *constraint satisfaction problems* and then solves them using the state-of-the-art constraint solver, ECL^iPS^e [130]. Hence, as is the case with KSAT, KCSP benefits indirectly from improvements made to the underlying solver, which in this case is a constraint solver.

KCSP follows a similar approach to that used by the KSAT algorithm but uses a constraint solver instead of a SAT solver. It encodes the modal formula into layers of subformulae of decreasing modal depth, each of which is processed as a finite constraint satisfaction problem. This is possible because of the tree model property of modal logics that has already been discussed in Section 2.7 – a similar approach was followed in the translation to first-order logic by Areces et al. [6] (Section 3.3.3).

Before we examine this approach, we need to look at constraint satisfaction problems (CSPs) in general, and how they are solved using constraint logic programming (CLP) languages. We provide a general overview of CLP languages that discusses their close ties with logic programming languages such as Prolog. Two important components of a CLP language are its constraint solver, which makes use of constraint propagation to prune the search space wherever possible, and its constraint store, which is used to store partial information about the constraints on variables. CLP languages include specialized algorithms and libraries that we briefly discuss. An overview of the ECL^iPS^e constraint logic programming language that is used by the KCSP solver is provided.

3.5.1 The constraint satisfaction problem

We begin by defining a constraint satisfaction problem – the class of problem into which a modal formula is to be converted.

A constraint satisfaction problem consists of a set of variables, a domain for each variable and a set of constraints. The variables can be assigned any value in the corresponding domain, with the limitation that the constraints on the variables need to be satisfied. The constraints therefore limit the scope of the variables.

Definition 3.5.1. *Suppose we have a set of variables* $X = \{x_1, x_2, \ldots, x_n\}$*, with each variable* x_i *having a domain* D_i*. Let* D *be the set of domains* $D_1, \ldots D_n$ *and* C *a set of constraints on* X*. Each constraint* $c \in C$ *is a pair* $c = \langle \sigma, \rho \rangle$ *where* σ*, the constraint scope, is a list of variables in* X *and* ρ*, the constraint relation, is a subset of the Cartesian product of their domains. Such a problem is a constraint satisfaction problem (CSP).*

A solution to the CSP instance $\langle X, D, C \rangle$ is a complete assignment such that every constraint c ∈ C, the restriction of the assignment to the scope σ_c, is satisfied.

Example 3.5.2. Consider the following example. We have a set of variables $X = \{x_1, x_2, x_3\}$, each of which has domain {1, ..., 5}. The constraints on these variables are $x_1 < x_2$ and $x_2 < x_3$. It is easy to see that by applying these constraints to the domains of the variables, we can restrict their domains to {1, 2, 3}, {2, 3, 4} and {3, 4, 5} respectively.

Consider the variable x_3. The values 1 and 2 are *inconsistent* with the constraints that $x_1 < x_2$ and $x_2 < x_3$ and hence are *pruned* from its domain. The values 3, 4 and 5 are however *consistent* with the constraints of the problem. ⊣

Many different approaches have been followed to solve problems of this nature, among which are theorem proving, integer programming, equation solving using Boolean algebra and constraint logic programming. A detailed overview of constraint programming is to be found in [5] and [109], with details of various application areas being provided in [117].

Search problems of this nature can be combinatorially complex and often cannot be solved. *Combinatorial problems* are those where a solution results from making a whole series of interdependent choices – the correctness or optimality of a given choice is not usually apparent until a number of other choices have been made, which may in turn depend on further choices. The result is that the correctness of the very first choice is usually not confirmed until the last choice has been made [128]. Such problems are known to be **NP**-complete [83].

We now look in particular at the approach taken to solve these problems that uses constraint logic programming.

3.5.2 Constraint logic programming

Overview

Constraint logic programming (CLP) combines the advantages of logic programming and constraint handling.

Logic programming languages are based on Horn-clause logic. They are declarative or rule-based languages. A program consists of clauses that are *facts* and *rules.*

We provide the following relevant definitions, following the logic programming convention to read implication from the right to the left.

Definition 3.5.3. *A clause* $p_1 \vee \ldots \vee p_r \leftarrow q_1 \wedge \ldots \wedge q_s$ *is called a rule if* $r \geq 1$ *and* $s \geq 1$*; it is called a fact if* $r \geq 1$ *and* $s = 0$*; and a goal if* $r = 0$ *and* $s \geq 1$*.* $p_1 \vee \ldots \vee p_r$ *is called the rule head and* $q_1 \wedge \ldots \wedge q_s$ *is called the rule body. Every* q_i *is called a subgoal.*

Every clause is a closed formula with universal quantification on variables in the formula – for example $p(x) \leftarrow q(x)$ *means that* $\forall x(p(x) \leftarrow q(x))$*.*

Definition 3.5.4. *A clause* $\pi : p_1 \vee \ldots \vee p_r \leftarrow q_1 \wedge \ldots \wedge q_s$ *is called a Horn clause if* $r \leq 1$ *and every subgoal is an atom. A logic program is a set of Horn clauses defined over some set of terms* L*.*

Horn-clause logic makes use of *resolution.* The resolvent of two Horn clauses is a Horn clause and the resolvent of a goal clause and a fact or rule is again a goal clause. This is the basis of logic programming. Prolog is an example of such a language. The reader is referred to texts such as [19] for full details of Prolog.

Example 3.5.5. The following is a simple Prolog program, which consists of one rule and two facts.

```
mother(X) :-                  // rule
    hasChild(X, Y).

hasChild(Bronwyn, Fred).      // fact
hasChild(Pam, Nicky).         // fact
```

The rule states 'If X is a mother, then X has a child Y'. Note that in Prolog, implication is denoted by ':-'.

A problem that can be posed to such a program is 'Is Bronwyn a mother?' and this will be stated as the *goal statement*

mother(Bronwyn).

The program will match hasChild(X, Y) with hasChild(Bronwyn, Fred) and will return the answer 'Yes'. $\dashv$

Logic programming applies *unification* of terms – unification is the reduction of an equation to an equivalent set of variable assignments, as can be seen in the example above. Unification of terms takes place over the *Herbrand universe*, which is the set of ground terms (terms without variables) of the language. In the example above, the ground terms are Bronwyn, Fred, Pam and Nicky.

Constraint logic programming preserves the basic syntax and computational properties of logic programming, in that facts and rules are defined in the same way. It then adds additional functionality which incorporates constraints and allows the evaluation of terms. In a logic programming language, because *unification* of terms takes place over the Herbrand universe, a statement such as '3 = 2 + 1' is evaluated as *False*. The CLP language on the other hand allows the *evaluation* of terms. In the integer domain, it will return *True* for the above statement. CLP languages operate over a number of domains. For example, in the domain of integers, it will allow constraints such as $3 + X + Y$ and $A \leq B$ to be evaluated, where X, Y, A and B each have an integer domain.

To process constraints, the CLP language makes use of a *constraint solver*, which allows *propagation* of constraints, the effect of which is to prune the search space (these concepts will shortly be expanded). Unification on the other hand, as used by logic programming, does not prune the search space – it makes use of a standard backtracking approach.

Examples of early CLP languages are CHIP [33] and CLP(R) [76]. They are constraint logic programming systems in which constraints are defined and solved using a constraint solver. CHIP allows constraints over finite domains, Booleans and rational numbers while CLP(R) allows constraints over real numbers. A general framework for CLP languages was formalized by Lassez et. al. and is referred to as the CLP scheme [82] and most CLP languages use this framework as a basis.

CLP languages have proved themselves in their ability to model and solve combinatorial problems. An important factor in this success is their ability to prune the search space during computation, thereby speeding up the execution of the program considerably.

The constraint solver

We now look at some of the aspects of how constraints are dealt with by the constraint solver.

A key innovation behind constraint programming is *constraint propagation*. In the case of the constraint $x = y + 1$, if a value is assigned to x, a value is automatically assigned to y. The constraint between any two objects or variables can be represented as an edge in a graph and as long as the graph is free of cycles, the propagation behavior is guaranteed to terminate [129].

CLP languages execute in two closely linked phases – the *inference phase* during which constraints are propagated, and the *search phase* which heuristically assigns values to variables. These two phases interact closely.

During the inference phase, consistency techniques such as *arc consistency* and *path consistency* are used to propagate constraints. Arc consistency ensures that any legal value in the domain of a single variable has a legal match in the domain of any other single variable. Path consistency ensures that any consistent solution to a two-variable subgraph can be extended to any third variable. These techniques enforce various forms of local consistency and add inferred problem constraints that are used to prune away inconsistent values and build up partial solutions.

This phase is followed by the application of a search method that traverses the space of partial solutions by assigning values to variables. The most common algorithm for performing systematic search is backtracking. Backtracking incrementally attempts to extend a partial solution that specifies consistent values for some of the variables towards a complete solution. This is done by repeatedly choosing a value for an unassigned variable which is consistent with the values in the current partial solution. When extension is impossible, the algorithm backtracks to make alternative choices. It typically backtracks to the last choice point. As an alternative, a lookback scheme can be deployed which decides how far to backtrack by analyzing the reasons for the dead end. This process is also referred to as *backjumping*. It records the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again. This is also referred to as *no-good learning*.

There are several approaches that can be followed to assign a value to a variable. A general technique is called *first-fail*, which chooses as the next variable to label the one with the smallest domain. Alternatively, the variable that occurs in most constraints can be chosen. Another approach is to make a binary chop of the domain, which means that the domain of a particular variable is cut into two halves and the value is assumed to be in one of them. The technique selected should be tailored to the data being dealt with.

As soon as a suitable value has been assigned by the search method, the inference method is reapplied to ensure consistency. These phases are repeated until either a consistent solution is found or it is established that there is no satisfactory solution. In each iteration, the domains of the remaining values are reduced further and further until one becomes empty or the remaining domains contain only one value or in some cases a range of values, in which case the problem is solved.

The inference phase is deterministic while the search phase is non-deterministic (based on heuristics) – the key aspect in the constraint logic approach is the tight integration between the deterministic constraint evaluation and the non-deterministic search process. Deterministic computations are performed as soon as possible during propagation and non-deterministic computations are used in the search phase only when there is no more propagation to be done. This approach exhibits a data-driven computation and can be characterized as 'constrain and generate'.

In Example 3.5.2, the domains of the variables were pruned (the inference phase). In order to progress in the search for a solution, the search phase needs to assign a value to one of the variables. Suppose we set $x_2 = 3$. The inference phase is then re-entered and the domains of the variables can be further pruned to $\{1, 2\}$, $\{3\}$ and $\{4, 5\}$ respectively. This process is repeated until a solution is found.

These techniques are described in further detail in [40, 25, 129]. Constraint logic programming has been successfully applied in many application areas, which include scheduling, circuit verification, real-time control systems and production sequences.

Further details of these and other application areas are provided in the papers [125, 128].

The constraint store

Another important aspect of a CLP language is the *constraint store* that is used to hold partial information about the constraints on variables. Variables can be stored as a range of values or as a linear equation or inequation. Constraint stores have been built to deal with different classes of constraints and with mathematical functions that are processed and checked for consistency using standard mathematical techniques. For example, the CLP(R) language deals with constraints over real numbers.

As already stated, the CLP program is syntactically a collection of *clauses* that are either *rules* or *facts*. The CLP program starts execution with the goal statements and an empty constraint store. The results of each step of its execution are stored as *computation states* that are represented as `Store @ Goals` – at each step, the current content of the constraint store and the remaining goals to be solved are stored. The aim of the program is to reduce the goals – a goal is reduced using a clause whose head matches the goal. Such a sequence of reduction steps is referred to as a *derivation*. A derivation terminates when there are no more goals to be reduced and the final constraint store is consistent. If the constraint store is inconsistent, the derivation fails. In the case of a failed derivation, the CLP program will abandon the current branch of the search tree and will backtrack until either a successful derivation is found or no more choices are possible, in which case the derivation fails.

Example 3.5.6. Consider a CLP program with the following sets of clauses:

```
findvalues(X, Y, Z) :-
    X < 5,
    Y < 5,
    Z < 5,
    calcY(X, Y),
    calcZ(Y, Z).

calcY(X, Y) :-
    Y is X + 1.
```

```
calcZ(Y, Z) :-
    Z is Y + 1.
```

The goal statement

```
findvalues(1, A, B).
```

will produce the following successful sequence of computational states in the constraint store:

(1) $\emptyset$ @ findvalues(1, A, B)

The goal is reduced by matching the goal clause findvalues(1, A, B) with findvalues(X, Y, Z). The goal is replaced by the body of the clause and the constraints in the body are added to the constraint store.

(2) X=1, A=Y, B=Z, Y $<$ 5, Z $<$ 5 @ calcY(1, Y), calcZ(Y, Z)

The goal is further reduced by matching the goal clause calcY(1, Y) with the clause calcY(X, Y) and setting A = Y = 2.

(3) X=1, A=2, B=Z, Y=2, Z $<$ 5 @ calcZ(2, Z)

The final goal clause calcZ(2, Z) is matched with calcZ(Y, Z) setting B = Z = 3.

(4) X=1, A=2, B=3, Y=2, Z=3 @ $\emptyset$

No goal clauses remain and values have been assigned to each variable.

$\dashv$

Specialized algorithms

Because of the complexities associated with constraint handling, various specialized algorithms have been developed in various CLP languages. We provide a few examples of such algorithms.

Propagation constraints – more commonly referred to as *constraint agents* – actively propagate new information to a constraint store and typically operate independently of each other. A special means of dealing with them is sometimes required. Because constraints alternate between suspended and waking states, in some cases, they need to be reactivated only when a special constraint is met. To control their behavior, the notion of a *guard* has been introduced. The guard, which is an additional primitive constraint, is put in place to ensure that the constraint agent is suspended until a necessary condition has been satisfied, after which the agent wakes up.

An example of a clause that has a guard and a constraint is the following:

```
calcY(X, Y) <==>
    var(Y)          // the guard ensures that Y has not been assigned
                    // a value
    |
    Y is X + 1      // the constraint
```

Another example of a specialized algorithm is *constraint entailment*, which is used to establish whether a single constraint is implied by a conjunction of constraints. The *implication combinator* is used to achieve local propagation of constraints by suspending goals when not enough information is available and then reactivating them when new information allows them to be reconsidered. The *cardinality combinator* is a declarative and relational operator that handles general forms of disjunctions and that can be used to enforce arc-consistency on any arbitrary finite domain constraints. It implements a principle known as the inference principle – ‘infer simple constraints from difficult ones’.

These and other algorithms are described in more detail in [126].

Specialized libraries

Constraint programming languages generally include specialized libraries. These libraries provide more efficient constraint propagation algorithms than the standard techniques, when applied to the specific classes the language deals with. They are designed to solve and find solutions in polynomial time. We look briefly at one such library – the CHR library – as it plays an important role in the K_KCSP solver.

Traditionally, the constraint solver has been a ‘black box’ that is hard to modify. To deal with this problem, additional constructs in the form of *constraint handling* *rules* (CHRs) have been added to constraint solvers. CHRs allow *user-defined* constraints and enable *simplification* and *propagation* over them. Simplification rules replace constraints with simpler constraints, while propagation rules add new constraints that may cause further simplification. These constraints are incrementally solved by the constraint solver.

CHIP was the first CLP language to introduce constructs for user-defined constraint handling. These were subsequently refined to give CHRs that are based on guarded rules. Further details of constraint handling rules are presented in [38] and [39].

3.5.3 The ECL^iPS^e constraint logic programming language

ECL^iPS^e incorporates all of the functionality already discussed, with the additional strength that it allows the programmer to use a combination of algorithms appropriate to the application at hand. In the work which follows, we will see some of this flexibility.

The platform offers a conceptual modeling language for specifying the problem clearly and simply, after which an appropriate algorithm can be selected to solve it. ECL^iPS^e offers many different constraint handling capabilities in different libraries. These libraries include the *ic* (interval constraint) library which is a hybrid integer / real interval arithmetic constraint solver, an *ic_symbolic* library which has a separate symbolic solver library, the *eplex* (mixed integer programming) library, the *propria* (generalized propagation) library that allows complex constraints and handles disjunction, and the *ech* library – which is its implementation of constraint handling rules (CHRs).

In ECL^iPS^e, constraint handling rules are multi-headed guarded clauses that consist of one or more heads, an optional guard, a body and an optional name. A rule relates heads and body, provided the guard is true. A rule can fire only if its guard succeeds, in which case the appropriate actions are applied to the head and body constraints.

The *simplification rule* has the form

SimplificationRule ::= [RuleName @] Head [, Head] <=> [Guard |] Body while the *propagation rule* has the form

PropogationRule ::= [RuleName @] Head [, Head] ==> [Guard |] Body

Note that the difference in the rules is represented by the syntax. To illustrate the use of the simplification rule, we include a simple example which explains the concept.

Example 3.5.7.

minimize_xy @ $values(X_1, Y_1)$, $values(X_2, Y_2)$ <=>

$verify\_values(X_1, X_2)$
$verify\_values(Y_1, Y_2)$
|
X is $min(X_1, X_2)$
Y is $min(Y_1, Y_2)$
$values(X, Y)$.

We have a rule named *minimize_xy*, two heads *values* that are constraints and the function *verify_values* that forms the guard. Only if this function executes correctly is the body executed. In the body, the minimum X and Y values are calculated and the constraint $values(X, Y)$ is then generated. In this case, the two constraints $values(X_1, Y_1)$ and $values(X_2, Y_2)$ are removed from the constraint store and the constraint $values(X, Y)$ is added. ⊣

If the above example was modified to represent the propagation rule, the constraints $values(X_1, Y_1)$ and $values(X_2, Y_2)$ would have remained in the store, and the new constraint $values(X, Y)$ would be added.

The *minimize_xy* rule is executed whenever a change occurs in any *values* constraint. The constraint store is searched for another *values* constraint and if one is found, the guard is executed and will either succeed or fail. If the guard succeeds, the body is executed; if it fails, the rule is delayed until a variable occurring in the head is touched, whereupon it is reactivated and the guard is re-evaluated.

In addition to user-defined constraint rules, it is possible to add conditional statements that can also delay the execution of goals. For example, we could have

$delay\ verify\_values(X,\, Y)$ if $var(X)$.
$delay\ verify\_values(X,\, Y)$ if $var(Y)$.

In this case, the guard function *verify_values* is delayed by a suspend solver provided in ECL^iPS^e if either X or Y have not been assigned a value.

ECL^iPS^e also provides many different search algorithms from which the user is able to select. These include branch-and-bound, depth-first search with backtracking and various heuristics searches.

It also contains specialized functions. For example, the *ic_symbolic* library has a function $atmost(N,\, L,\, V)$ which selects at most N elements with value V from a list L. Another useful function is $alldifferent(L)$ which fails if two elements in a list L are equal. Because such functions are preprogrammed, the user's task is considerably simplified.

ECL^iPS^e is a complex language with a vast amount of functionality from which the appropriate application can be built. It has recently been open-sourced and can be downloaded off the Internet [2].

3.5.4 The K_KCSP solver

We now look at the design of K_KCSP as applied to the modal logic K. Note that the modal satisfiability problem is **PSPACE**-complete, whereas, when the problem is stratified into layers and each layer is solved as a constraint satisfaction problem, the layers are **NP**-complete [18].

In this approach, the modal formula is encoded into layers where each layer occurs at a different world. The same approach is followed as was used with the SAT translation – modal atoms are treated as propositional atoms on the layer in which they occur. The modal formula of a particular layer is translated into a constraint satisfaction problem. Its top-level literals are identified (Definition 3.4.4) and a domain of $\{0,\, 1\}$ is assigned to each, as a variable must have a value of either *True* (1) or *False* (0). The constraints associated with the modal formula are then generated. Such constraints are easy to generate when the modal formula is in conjunctive normal form (Definition 2.2.21), as each disjunctive clause contributes a constraint to the

CSP.

Note that we will sometimes loosely refer to *modal atoms* as *literals*, as they are treated as propositional literals at each modal layer.

Example 3.5.8. Consider the CNF formula

$$(\neg p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee \neg p_2) \wedge \Box p_4$$

which has top-level atoms $\{p_1,\, p_2,\, p_3,\, \Box p_4\}$, each of which has domain $\{0,\, 1\}$.

The constraints on this formula are as follows:

- The disjunction $(\neg p_1 \vee p_2 \vee p_3)$ has the constraint that p_1 must be *False* or at least one of p_2 and p_3 must be *True* – hence, at least one of the following assignments is required: $p_1 = 0$, $p_2 = 1$, $p_3 = 1$.
- The disjunction $(p_1 \vee \neg p_2)$ has the constraint that we must have at least p_1 $True$ or p_2 $False$ – hence, at least one of the following assignments is required: $p_1 = 1$, $p_2 = 0$.
- The clause $\Box p_4$ has the constraint that it must be $True$ and so $\Box p_4 = 1$.

$\dashv$

A constraint solver will always return a *total assignment* to a constraint satisfaction problem. In the above example, the problem is solved with a total assignment of $\mu = \{p_1,\, \neg p_2,\, p_3,\, \Box p_4\}$. However, it can also be solved with a partial Boolean assignment of $\mu = \{p_1,\, p_3,\, \Box p_4\}$.

Because less computational effort is required to return a partial assignment, this is a preferred approach. Brand et al. [18] followed an approach in which the domain of each literal is set to $\{0, 1, u\}$ instead, with u being used to denote 'unknown', meaning that a value is not assigned to the associated literal. In the above example, when we return the assignment $\mu = \{p_1,\, p_3,\, \Box p_4\}$, a value has not been assigned to top-level atom p_2. In the implementation, u is assigned a value of 2. We will however continue to refer to u for clarity.

Recall that a modal formula φ must be approximated as a propositional formula in its top-level atoms before being processed by a SAT solver, and the same approximation must be applied in the case of the constraint solver. Definition 3.4.11 defines this approximation. Each modal atom $\Box\psi$ is replaced with $x_i[\Box\psi]$, where x_i denotes a unique propositional atom. A modal formula such as $\varphi = p \wedge \Box p \wedge \neg\Box p$ is approximated as $Prop(\varphi) = p \wedge x_1[\Box p] \wedge \neg x_2[\Box p]$, and effectively contains three different propositional atoms. The constraint solver will assign a value of 1 to $x_1[\Box p]$ and 0 to $x_2[\Box p]$. The next modal layer then contains p and $\neg p$. Note that this is the *internal representation* of the modal atom – in our discussions, we will continue to refer to $\Box p$ and so on, instead of $x_1[\Box p]$, which ensures clarity.

We now look at how a modal formula is encoded into different modal layers. We proceed as with the SAT-based solver, where a truth assignment μ is generated for the top-level literals of the relevant modal formula φ (Definition 3.4.6). If this assignment propositionally satisfies φ, that is, $\mu \models_p \varphi$, then in terms of Theorem 3.4.13, the modal formula will be satisfiable. The next modal layer will be satisfiable if the restricted truth assignment μ^r (Definition 3.4.9) is satisfiable, as per Theorem 3.4.14.

Brand et al. [18] based the K_KCSP algorithm on KSAT which we have already presented – we represent a condensed version of their K_KCSP algorithm and will use this representation for subsequent versions thereof.

Algorithm 5. The K_KCSP algorithm schema can be represented as follows:

```
function K_KCSP($\varphi$)
    $\varphi_{neg} = \neg\varphi$;
    $\varphi_{init} = to\_cnf(\varphi_{neg})$;
    K_CSP($\varphi_{init}$);
end;

function K_CSP($\varphi$)                    // succeeds if $\varphi$ is satisfiable
    $\varphi_{csp} = to\_csp(\varphi)$;
    $\mu := csp(\varphi_{csp})$;
    $\Theta = \bigwedge\{\alpha : \Box\alpha = 1$ is in $\mu\}$;
    for each $\Box\beta = 0$ in $\mu$ do
        K_CSP($\Theta \wedge \neg\beta$);    // backtrack if this fails
end;
```

The K_KCSP algorithm firstly negates the input formula and then converts it into CNF. The *sat* procedure of KSAT is replaced with two procedures – *to_csp* and *csp*. *to_csp* identifies the top level literals of φ, sets their domains to {0, 1, u} and defines the constraints on each clause, thereby defining the constraint satisfaction problem. The resulting formula is then passed to the ECL^iPS^e constraint solver with the call to *csp*. It either returns a solution or determines that the formula is unsatisfiable. If it cannot immediately return a solution, *csp* will backtrack through previous choice points to try to find a suitable solution. In the case where no solution is returned, it will have backtracked across all possible choice points.

Once a solution has been returned for the current modal layer, if it contains negative modal literals to which a value of 0 has been assigned, further processing is required. Each of these modal literals $\neg\Box\beta_j$ effectively generates a new branch of the modal tree. The conjunction of each β_j and the α_i literals having $\Box\alpha_i = 1$ form the modal formula that will be processed at the next modal layer. If there are no negative modal literals, the formula is satisfiable and no further processing is required. In effect, we are generating a modal tree with the modal formula at the root and the branches being created by the negative modal literals.

The effect of using a domain of {0, 1, u} is that we effectively have a partial assignment of truth values, which reduces the processing requirement – if all the $\neg\Box\beta_j$ literals at a modal layer can be assigned a value of u, no further processing will be required.

Example 3.5.9. Consider the following example to which the K_CSP function is applied.

$$\varphi = \neg\Box(p_1 \vee p_2) \wedge (\Box p_1 \vee \Box p_3 \vee \neg\Box\Box\Box(p_2 \vee \Box(p_3 \vee \Box p_4)))$$

1. At the first modal layer, this formula has top-level atoms $\{\Box\Box\Box(p_2 \vee \Box(p_3 \vee \Box p_4)), \Box(p_1 \vee p_2), \Box p_1, \Box p_3\}$, each of which has domain {0, 1, u}. The constraints are that $\Box(p_1 \vee p_2)$ must be 0 and that we need at least one of $\Box p_1 = 1$, $\Box p_3 = 1$ or $\Box\Box\Box(p_2 \vee \Box(p_3 \vee \Box p_4)) = 0$.

   The constraint solver could return the partial assignment

$$\mu = \{\neg\Box(p_1 \vee p_2), \Box p_1 = 1\}$$

2. Because we have $\neg\Box(p_1 \vee p_2)$, we move to the next modal layer with the formula $\neg(p_1 \vee p_2) \wedge p_1 = \neg p_1 \wedge \neg p_2 \wedge p_1$ which is unsatisfiable.

3. The constraint solver backtracks to the first modal layer and returns with the partial assignment

$$\mu' = \{\neg\Box(p_1 \vee p_2), \Box p_3\}$$

4. We once more have $\neg\Box(p_1 \vee p_2)$, so we move to the next modal layer with the formula $\neg(p_1 \vee p_2) \wedge p_3 = \neg p_1 \wedge \neg p_2 \wedge p_3$, which is now satisfiable. $\dashv$

We note the following about the algorithm. The constraint solver backtracks as soon as an unsatisfiable result is obtained and will progress through the entire tree until a satisfiable result is obtained – in which case the modal formula is *not valid*. If a satisfiable result is not obtained, the negation of the original formula is unsatisfiable, and hence the original modal formula is *valid*.

3.5.5 Constraint-based modeling

We have provided the details of the K_KCSP algorithm but have not yet discussed the details of how the constraints are defined and processed by the constraint solver.

Definition 3.5.10. *A modal formula φ can be represented in conjunctive normal form (Definition 2.2.21) as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$ where each ψ_i is of the form*

$$\bigvee\{p : p \in P^+\} \vee \bigvee\{\neg q : q \in P^-\} \vee \bigvee\{\Box\alpha : \Box\alpha \in B^+\} \vee \bigvee\{\neg\Box\beta : \Box\beta \in B^-\}$$

and P^+ and P^- contain propositional atoms, and B^+ and B^- contain modal atoms respectively.

A clause ψ_i is true if *at least one* literal in the sets P^+ and B^+ is true, *or at least one* literal in the sets P^- and B^- is false. This can be written as,

$$at\_least\_one(P^+ \cup B^+, 1) \vee at\_least\_one(P^- \cup B^-, 0) \tag{3.5}$$

However, constraint solvers find disjunctions difficult to deal with. This disjunction can be transformed into a conjunction by adding a link variable l to it. Equation (3.5) then becomes

$$at\_least\_one(P^+ \cup B^+ \cup \{l\}, 1) \wedge at\_least\_one(P^- \cup B^- \cup \{l\}, 0) \tag{3.6}$$

with the link variable l selecting implicitly which of the two constraints must hold. If $l = 0$, the constraint on the left becomes $at\_least\_one(P^+ \cup B^+ \cup \{0\},\ 1)$, or $at\_least\_one(P^+ \cup B^+,\ 1)$. The constraint on the right becomes $at\_least\_one(P^- \cup B^- \cup \{0\},\ 0)$, which is satisfiable. If $l = 1$, the constraint on the left becomes $at\_least\_one(P^+ \cup B^+ \cup \{1\},\ 1)$, which is satisfiable, while the constraint on the right becomes $at\_least\_one(P^- \cup B^- \cup \{1\},\ 0)$, or $at\_least\_one(P^- \cup B^-,\ 0)$.

The ECL^iPS^e *ic_symbolic* library contains a predefined constraint *atmost* in which the number of literals in the list can be set to be at most 1, as was discussed in Section 3.5.3. It can easily be adapted to deal with the requirement for an *at_least_one* constraint. As we stated in Section 3.5.3, one of the benefits of ECL^iPS^e is its many predefined algorithms and libraries.

Once the constraint satisfaction problem has been defined and fed to the ECL^iPS^e constraint solver, it deals with the constraints by *suspending* constraints which have variables to which a value has not been assigned. As soon as a variable is assigned a value, the constraint is activated and it is processed further.

Example 3.5.11. Suppose we have the constraints

$$at\_least\_one(\{p_1,\ p_2,\ \Box p_3,\ l_1\},\ 1) \wedge at\_least\_one(\{p_4,\ l_1\},\ 0)$$

Suppose p_1 is assigned a value of 1. In this case, the first constraint is activated and l_1 is set to 0 so that the second constraint is satisfied. All other constraints containing p_1 are also activated. At the same time, the domain of the literals p_2, $\Box p_3$ and p_4 can be set to u – having satisfied one of the literals, the remaining literals do not need to be assigned a definite value of 0 or 1.

Alternatively, if p_1 was assigned a value of 0, no further processing would take place and the constraints would once more be suspended.

$\dashv$

By setting as many literals to u as possible, a partial assignment is extended to become a total assignment. Because of the inherent backtracking power of a constraint solver, assignments of this nature can be reversed on backtracking and an alternate value assigned to the literals. In the example above, if the assignment $p_1 = 1$ is no longer feasible, the constraint solver will backtrack and the values assigned to the literals l_1, p_2, $\Box p_3$ and p_4 will be reversed.

Brand et al. took this concept further. They showed that, in order to reduce the processing requirement, it is preferable to assign the value u to as many modal literals as possible – if any positive or negative propositional literals in a clause is assigned a value of 1 or 0 respectively, all modal literals in the clause can be assigned values of u. If there are no possible assignments to propositional literals, it is then necessary for only one modal literal to be assigned a value of either 0 or 1. Furthermore, it is better to assign a value to a positive modal literal, as we only move to the next modal layer if we have a negative modal literal with values of 0.

Equation (3.6) can now be rewritten as:

$$\begin{array}{lclcl} \mathit{at\_least\_one}(P^+ \cup \{l_P^+\}, 1) & \wedge & \mathit{exactly\_one}(B^+ \cup \{l_B^+\}, 1) & \wedge \\ \mathit{at\_least\_one}(P^- \cup \{l_P^-\}, 0) & \wedge & \mathit{exactly\_one}(B^- \cup \{l_B^-\}, 0) & \end{array} \tag{3.7}$$

Note that the link variables l_P^+, l_B^+, l_P^- and l_B^- are distinct. If any one of these link variables is assigned a value of 1, the rest are set to 0. Hence, if a positive propositional literal in P^+ is assigned a value of 1, the constraint solver will set $l_B^+ = 1$, $l_P^- = 0$ and $l_B^- = 0$ so that Equation (3.7) is satisfied. At the same time, the remaining unassigned literals in this set of clauses are assigned a value of u.

The implementation details of these linking constraints is explained in [4]. The *ech* library, which implements constraint handling rules, was used to define and manage these user-defined constraints (see Section 3.5.3). Good results were returned with this implementation when compared with the KSATC solver, as listed in Table 3.1 [18]. Note that the details of the data sets used are provided in Section 4.2 and so are omitted here.

3.5.6 K_KCSP optimizations

We summarize some of the optimizations Brand et al. applied to the K_KCSP algorithm.

	KSATC	**K_KCSP**	**KSATC**	**K_KCSP**
	n	**n**	**p**	**p**
branch	8	11	8	21
d4	5	6	8	8
dum	21	17	11	11
grz	21	21	17	10
lin	3	21	21	21
path	8	9	4	4
ph	5	4	5	4
poly	12	16	13	9
t4p	18	6	10	8

Table 3.1: Comparative results of the KSATC and K_KCSP solvers

Optimization 1:

The first key optimization was to mark as many box formulae as possible as irrelevant which causes fewer subformulae to enter the subsequent layer. This was achieved with the change of domain of the literals to $\{0, 1, u\}$, which we have already covered above in detail.

Optimization 2:

In the basic algorithm, every time 1 is assigned to a formula $\neg\Box\varphi$, the subformula $\neg\varphi$ is first transformed into CNF before turning it into CSP form. Such a CNF conversion can lead to an explosion in the size of the formula. To remove this problem, $\neg\varphi$ is treated as follows: The constraint $\neg\varphi$ is satisfiable iff φ (which is a conjunction of clauses) has at least one unsatisfiable clause. Hence, the constraint corresponding to $\neg\varphi$ is treated as a disjunction of constraints, each standing for a negated clause. The disjunction is then converted into a conjunction with a set L of linking variables, one for each disjunct. By taking this approach, the constraint solver is able to provide a more efficient solution.

Optimization 3:

A subformula $\Box\psi$ of φ can occur several times in the modal formula at a particular modal layer and can occur both positively and negatively. Each occurrence is treated as a distinct propositional literal. To ensure consistency in the solution, an *additional constraint* is added which states that this subformula must have only one value.

Optimization 4:

Top-level input formulae are simplified using the simplification rules for propositional formulae, as defined in Rules 3.4.1 and 3.4.2.

3.5.7 CSP-based modal solvers

To the best of our knowledge, K_KCSP is the only CSP-based modal solver on which a paper has been published. The papers by Brand et al. [18, 17] provide details of the benchmark between K_KCSP and KSATC in which the Heuerding and Schwendimann test data sets were used. Because we will be discussing these data sets in detail in Section 4.2 of the next chapter, we omit further details. As can be seen in Table 3.1, K_KCSP returns very good results for some of the data sets; for others KSATC is better. Brand et al. concluded that the results K_KCSP returns are good, warranting a further study of this approach.

3.6 Alternative approaches to solving modal problems

We briefly outline some of the alternative approaches followed to solve the modal satisfiability problem. The reader is referred to the relevant papers for further details.

One approach has been to translate modal problems into automata-theoretic problems and automata-theoretic methods have then been used to obtain answers. An example of automata-theoretic techniques that were used to develop a decision procedure for the modal logic K was that developed by Pan, Sattler and Vardi [102]. Non-deterministic automata have size exponential in the size of the input formula. However, by making use of Binary Decision Diagrams (BDDs) a good solution can be obtained. An input formula is translated into a tree automaton that accepts all tree models and the automaton is then tested for non-emptiness. This approach returned good results. In [8], it was shown that an automata-based approach can be transformed into a tableau-based decision procedure.

Modal formulae have been translated into quantified Boolean formulae (QBFs). This was motivated by the recent development of QBF solvers (the SAT-2006 Ninth International Conference on Theory and Applications of Satisfiability Testing included a QBF Solver Evaluation) [15]. We provide a formal definition of QBFs.

Definition 3.6.1. *A quantified Boolean formula (QBF) is a propositional formula with a quantifier prefix whose variables are quantified existentially or universally. A quantified Boolean formula QBF formula has the form*

$$F = Q_1 X_1 \ldots Q_n X_n P(X_1, \ldots, X_n)$$

where $X_1, \ldots, X_n$ denote n mutually disjoint sets of variables that are quantified by $Q_1, \ldots, Q_n$ respectively. $Q_1, \ldots, Q_n$ alternate between universal $(\forall)$ and existential $(\exists)$ quantifiers while $P(X_1, \ldots, X_n)$ is the propositional part of the formula.

An example of a QBF is $\forall xy \exists abc(x+y'+a)(x+y+b+c)$ where $(x+y'+a)(x+y+b+c)$ is the propositional part, variables x and y are universally quantified and variables a, b and c are existentially quantified.

At the TANCS-2000 conference, some of the benchmark formulae were solved by a QBF-solver that solved all the problems within a second – far exceeding the performance of the modal systems [87]. A translation of modal formulae into QBF was attempted in [101], but it was found that the reduction is not polynomial.

Modal proof systems have been based on the connection method of Wallen [131]. It uses matrix proof systems for modal logics that reduce the task of checking a modal formula for validity to one of path checking. Complementary tests are performed by a specialized unification algorithm. The connection method was applied by Otten and Kreitz [100] to intuitionistic logic to develop a proof method that was subsequently implemented as the theorem prover ileanTAP [99].

3.7 Final remarks

In this chapter we have looked at the most popular approaches taken to solve the modal satisfiability problem and have found that a significant number of solvers have implemented the tableau approach. These solvers are highly optimized, part of the motivation of which comes from the need to find adequate solutions to the problems arising as a result of the information explosion on the Internet.

The majority of other solvers translate the modal logic into various other classes of problem, which has the benefit of being able to make use of the existing, well-developed solvers of the alternate class.

The translation into first-order logic has been motivated partly by the desire to understand why modal logics are decidable and first-order logics are not, particularly in view of the seeming similarity in their semantics. However, the solver MSPASS that implements the optimized functional translation gives good results.

The translations of modal formulae into layered propositional formulae that are solved using a SAT solver or a constraint solver were looked at. Both of these solvers – KSAT and K_KCSP – returned good results for the modal logic K. This can partly be attributed to the fact that SAT and CSP are both **NP**-complete, whereas the modal satisfiability problem is at best **PSPACE**-complete – a **PSPACE**-complete problem requires polynomial space to solve which can cause problems such as stack overflows on implementation. These solvers have however been limited to the modal logic K, with KSAT also being implemented for the modal logic KT.

A formal comparison of the performance of many of these solvers is to be found in [87], which lists the results obtained at the TANCS-2000 conference. The solvers submitted include MSPASS, *SAT, RACE, FaCT and DLP. The benchmark data used at this conference were *unbounded modal QBF formulae*. A complicating factor was that each solver was run on different hardware, using different compilers and software, which made comparison difficult. To address this problem, the results obtained for each solver were normalised and it was found that MSPASS performed the best, followed by DLP, FaCT and lastly RACE. The reader is referred to the papers [71, 51, 64, 103, 119] for further details.

Various other solvers have been developed to solve the modal satisfiability problem. We looked at some of these, most of which have been once-off attempts which have not been taken further.

In the next chapter, we translate modal KT and $S4$ formulae into constraint satisfaction problems and solve them by implementing a number of extensions and enhancements to the existing K_KCSP solver.

Chapter 4

KT and S4 modal satisfiability in a constraint logic environment

We now focus on the translation of *KT* and *S4* modal formulae into constraint logic problems and the enhancement of the K_KCSP solver developed by Brand et al. [18] to achieve this objective. This solver returned some good results when tested with the Heuerding / Schwendimann data sets for the modal logic *K* [11]. The question now arises as to whether it can produce equally good results for *KT* and *S4* modal formulae. These logics have been selected primarily because they are the logics implemented as a next step by other solvers – for example, the KSAT solver was enhanced to deal with *KT* modal formulae.

Furthermore, it would be of interest to investigate whether this enhanced solver can be adapted to deal with temporal logic problems, in which transitivity of the accessibility relation is a key requirement – time is by its very nature transitive.

Transitive and reflexive accessibility relations have also proved to be important in application areas of knowledge and belief where the modal logic $KT45$, which is also referred to as $S5$, is commonly used to reason about agents. In this logic, a reflexive accessibility relation is used to represent truth – 'an agent knows only true things', a transitive accessibility relation is used for positive introspection – 'an agent knows what it knows', and a euclidean accessibility relation is used for negative introspection – 'an agent knows what it does not knows'. The logic S5 is commonly chosen as the logic of knowledge while the logic KD45, in which D is a serial accessibility relation, is chosen as the logic of belief [73].

This chapter begins with a discussion of how tableau and sequent solvers deal with the *KT* and *S4* modal logics (due to the dissimilarity of the first-order translation approach, it is not considered). For all these solvers, the application of reflexive and transitive rules to a modal formula is problematic – the algorithm for *KT* loops within worlds, while the algorithm for *S4* produces infinite loops. Over and above these problems, as soon as the algorithms to implement reflexivity and transitivity are applied to a modal formula, its complexity increases considerably. Hence we first look at how other solvers have dealt with these issues.

The next step is to identify suitable benchmark data sets that will be used to test the efficiency of the prototypes we develop. We begin by looking at the background of data sets that have been developed for modal logics. There are currently two groups of data sets that we could make use of. Of these, we select the Heuerding / Schwendimann data sets [11] which consist of nine classes of satisfiable and unsatisfiable formulae. They are well structured with problems of increasing complexity.

The modal logic KT in which the accessibility relation is reflexive, is then analyzed. We find that once the reflexivity algorithm is applied to a modal formula, it is no longer in conjunctive normal form (Definition 2.2.21). The algorithm of Brand et al. requires formulae to be in CNF. However, the conversion back to CNF results in a marked increase in the number and size of the modal clauses. We therefore propose two prototypes – one in which the final formula is in conjunctive normal form and one in which it is not – and compare their respective performances.

The initial two prototypes implemented were not optimized or enhanced in any way, other than with the optimizations already implemented by Brand et al. The Heuerding / Schwendimann data sets for KT were used to benchmark them but it was found that the results obtained were not particularly good. I then motivate and apply a progression of enhancements and the improvements in the results are noted. The enhancements consist of improving the simplification process, early pruning, grouping clauses with the same literals, value assignments and caching. The final prototype generates results that compare favorably with those of the TANCS-1998 benchmarks – I motivate the validity of the comparison. Of the two prototypes tested, the one that does not maintain the modal formula in conjunctive normal form returns better results. Finally, two of the data set classes that do not return good results are analyzed – it does not seem that there is an enhancement that can be applied that would improve their performance.

We then look at the modal logic *S4*, which requires the implementation of reflexive and transitive accessibility relations. An analysis shows that we have the same looping behavior experienced by the tableau and sequent approaches and that, after the rules to implement reflexivity and transitivity have been applied, the original modal formula increases substantially in complexity and size. The detection of loops is simple because of the caching enhancement applied to the KT_KCSP prototype. The initial S4_KCSP prototype is based on the prototype in which the formulae are not maintained in CNF, as this conversion is even more problematic in this case. It does not return good results, particularly for one class of data sets. Simplification was revisited and applied to modal formulae within an NNF clause. Early pruning was reapplied after the simplification step. These were the only enhancements that made a notable difference. The results are analyzed and suggestions made on possible future improvements to this prototype.

4.1 Current approaches to satisfiability solving of modal logics KT and S4

We begin by looking at how KT and $S4$ have been dealt with in tableau and Gentzen sequent systems, as discussed in Sections 3.1 and 3.2. The commonality between these approaches is the fact that they build a modal tree with the modal formula at the root of the tree, whereas the translation to first-order formulae applies resolution to the translated formula (Section 3.3.2).

When solving modal satisfiability problems for KT and $S4$ in tableau and sequent systems, the proofs in general do not terminate. A proof is an attempt to build a Kripke counter-model – recall from Definition 2.2.4 that for a formula to be satisfiable, it must be true at some world in some model. Hence, in the counter-model, if *all possibilities* have been tried and a counter-model cannot be constructed, the formula is valid.

The tableau rules for the modal logics KT and $S4$ were provided in Definitions 3.1.7 and 3.1.8 of the previous chapter and their application results in non-termination as demonstrated below:

- Looping *inside a world* occurs with the application of the tableau rule (T) in the modal logic KT. Consider the formula $\Box\Box(\varphi_1 \vee \Box\varphi_2)$ and its tableau proof in Figure 4.1. The continued application of the (T) rule results in the tableau not terminating. Because we have to try *all* possibilities, the reapplication of this rule is theoretically necessary. Note that to simplify the tableau, we represent $(\varphi_1 \vee \Box\varphi_2)$ as φ_3 in the Figure in most cases.

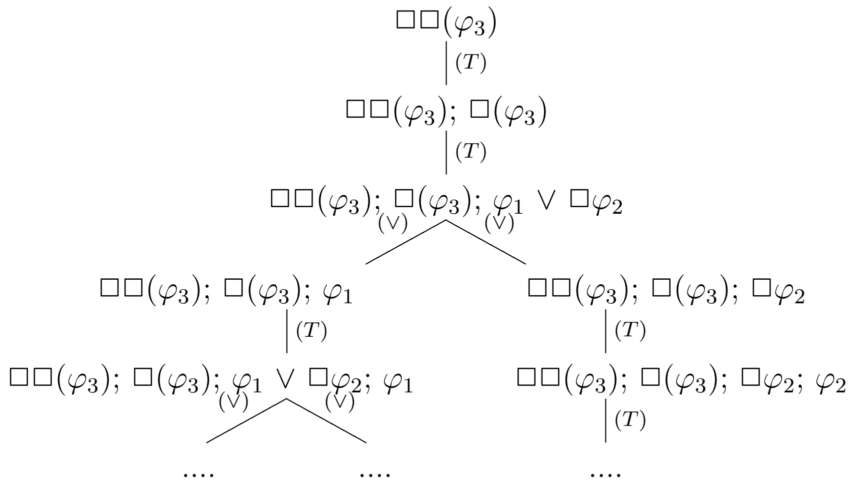

$\Box\Box(\varphi_3)$
(T)
$\Box\Box(\varphi_3);\ \Box(\varphi_3)$
(T)
$\Box\Box(\varphi_3);\ \Box(\varphi_3);\ \varphi_1 \vee \Box\varphi_2$
$(\vee)$
$(\vee)$
$\Box\Box(\varphi_3);\ \Box(\varphi_3);\ \varphi_1$
$\Box\Box(\varphi_3);\ \Box(\varphi_3);\ \Box\varphi_2$
(T)
(T)
$\Box\Box(\varphi_3);\ \Box(\varphi_3);\ \varphi_1 \vee \Box\varphi_2;\ \varphi_1$
$\Box\Box(\varphi_3);\ \Box(\varphi_3);\ \Box\varphi_2;\ \varphi_2$
$(\vee)$
$(\vee)$
(T)
....
....
....

Figure 4.1: Looping in the tableau for the formula $\Box\Box(\varphi_1 \vee \Box\varphi_2)$ in the modal logic KT

- Infinite looping occurs in a branch with the application of the tableau rules (T) and $(S4)$ in the modal logic $S4$. Consider the formula $\Box\Diamond\varphi$ and its tableau proof in Figure 4.2. The continued application of the tableau rules (T) and $(S4)$ results in a proof that does not terminate.

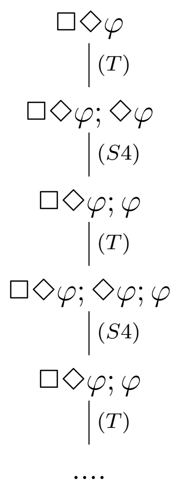

$\Box\Diamond\varphi$
(T)
$\Box\Diamond\varphi;\ \Diamond\varphi$
$(S4)$
$\Box\Diamond\varphi;\ \varphi$
(T)
$\Box\Diamond\varphi;\ \Diamond\varphi;\ \varphi$
$(S4)$
$\Box\Diamond\varphi;\ \varphi$
(T)
....

Figure 4.2: Looping in the proof of the formula $\Box\Diamond\varphi$ in the modal logic S4

Various strategies have been identified in tableau systems and in the sequent calculus to deal with non-termination. In the case of *KT*, one can control the reapplication of the same rule within a particular world. In the case of *S4*, the application of the (*S4*) tableau rule systematically produces a copy of the principal modal formula, and the formula does not disappear as it does in the case of the *KT* tableau. To solve the resulting nontermination issue of *S4*, a backtracking mechanism and a *loop checking* test are required. Before applying any rule to a formula, it is necessary to check that the rule has not already been applied. If a loop is detected, backtracking is necessary to ensure completeness of the proof search. Backtracking and loop checking can be expensive and so need to be reduced wherever possible – loop checking requires that the history of formulae already processed is stored.

The general approaches applied to create decision procedures for these logics include the following:

- An early approach was to keep the original tableau calculus and constrain the applicability of the rules in the case of *S4* [32]. A non-uniform ordering of tableau rules was imposed with a *non-exhaustive* backtracking mechanism. The periodicity test which checks for loops was performed only if a π-rule was applied to the current branch, which limited the application of the test (recall from Definition 3.1.11 that a rule applied to a $\Diamond$-formula is referred to as a π-rule).

- The rules of the Gentzen sequent calculus were revised and constraints were directly incorporated [61]. In the case of *KT*, the sequent rules were modified to split formulae into those not yet handled in this world and those already handled, which resulted in proofs that did not loop. In the case of *S4*, loop-checking was applied. Loop-checking however requires that history is stored, which can be expensive. The sequent rules were modified to include a limited history of all sequents on the branch in the past. This solution is efficient and easy to implement and also minimizes the stored history.

- The logic *S4* can be translated into a logic which has less complex proof procedures, as was proposed in [20]. A proof procedure for *S4* requires periodicity tests that a logic such as *KT* does not and since the translation of *S4* into *KT* can be done in deterministic polynomial time, this is a feasible option. In the case of KT, at any step in the expansion of a tableau, there are only a finite number of choices and the construction of any tableau terminates if one avoids the reapplication of the (T) rule.

- The accessibility relation can be represented explicitly, as was done in the single step tableau of [84] which was discussed in Section 3.1.2. To obtain decision procedures, the applicability of tableau rules is restricted – a rule is applied to a prefixed formula only if the formula has not already been reduced, which is a form of loop checking. Loop checking for transitive logics is thus replaced with a simple and effective termination check – a set of formulae already reduced is not reduced again.

These techniques were further developed in solvers such as DLP and FaCT, discussed in Section 3.1.4 of the previous chapter. In particular, the status of each node is cached which reduces search time when a similar node is encountered and its satisfiability status is already known. This increases the space requirement but results in significant gains in speed. When an unsatisfiable node is encountered and backtracking is required, backjumping over the nodes that did not contribute to the current clash is applied, thereby reducing the search space. The data structures used are also highly optimized.

4.2 Benchmark test data sets

Now that we have outlined how existing solvers deal with these logics, we look at the history of test data sets for modal logics and select those that will be extensively used in our benchmarks.

In 1996, Giunchiglia and Sebastiani [44] identified the need for formal test data sets for modal logics. At that time, there was a wide bibliography on problem sets and test-generating methods for propositional and first-order theorem proving, but nothing much for modal logic problems. They took the *fixed clause-length SAT test model* (as described in, for example, [90]) and modified it to provide for the modal logic $K(m)$. The fixed clause-length test generates random clauses of length 3, while varying the number of clauses L and the number of propositional literals N. The formulae generated are referred to as *3CNF well-formed formulae* (wffs).

Giunchiglia and Sebastiani [44] adapted this approach to generate $\mathrm{CNF}_{K(m)}$ wffs. They used clauses in which the number of literals per clause was fixed to 3 – thereby generating *3CNF$_{K(m)}$ formulae*. These were generated using 5 parameters – the modal depth d, the number of distinct boxes m, the number of top-level clauses L, the number of propositional literals N and the probability p with which any random $3\mathrm{CNF}_{K(m)}$ atom is propositional.

These data sets were used in the initial comparisons between their KSAT solver and various other solvers [46] and were the first formal modal logic data sets.

Hustadt and Schmidt [70] subsequently showed that these data sets had some shortcomings in that they produced a substantial amount of tautologous and contradictory subformulae and did not produce challenging unsatisfiable modal formulae. They analyzed the data sets produced and developed guidelines for the random generation of modal formulae. In particular, they recommended that the random generator should be modified to prevent repeated propositional literals inside the same clause. They also recommended that the probability variable p be removed. Giunchiglia et al. [42] then tested these assertions and made further recommendations to improve the quality of test data sets.

Heuerding and Schwendimann took a different approach to designing data sets [11, 60]. They defined the criteria such formulae must adhere to, namely that they must be sufficiently hard for forth-coming provers, the result of each formula must already be known (an excellent benefit for new provers), and the formulae must not take too long to prove. They defined nine classes of 21 provable and 21 not provable formulae for each of the logics *K*, *KT* and *S4* and placed these formulae on the Internet for easy access [74]. The formulae in each class become progressively more complex, enabling one to clearly determine the limits of one's solver.

Heuerding and Schwendimann [11] provide full details of how the formulae in each class are generated – we will describe only the generation of the *k_poly_p* formulae (this naming convention denotes *provable* formulae for modal K for the *poly*-class). They use as a baseline the formula

$$(p_1 \leftrightarrow p_2) \vee \dots \vee (p_{n-1} \leftrightarrow p_n) \vee (p_n \leftrightarrow p_1)$$

which says that if we have a polygon with n vertices and all the vertices are either black or white, then two adjacent vertices have the same color. If n is odd, this formula is provable. The data sets are generated as follows:

$$k\_poly\_p(n) = \begin{cases} poly(3n+1) & \text{if } n \bmod 2 = 0 \\ poly(3n) & \text{if } n \bmod 2 = 1 \end{cases}$$

where

$$poly(n) = \quad \Box^{n+1} \bigwedge_{i=1,\ldots,n+1}(p_i) \vee f(n,n) \vee \Box^{n+1} \bigwedge_{i=1,\ldots,n+1}(\neg p_{2i})$$

and

$$f(i,n) = \quad \Diamond(f(i-1,n) \vee \Diamond^i(p_n \leftrightarrow p_1)) \vee \Box\ p_{i+2}$$

The formulae for *k_poly_n* (n denotes *non-provable*) are generated with an even number of vertices, that is,

$$k\_poly\_n(n) = \begin{cases} poly(3n) & \text{if } n \bmod 2 = 0 \\ poly(3n+1) & \text{if } n \bmod 2 = 1 \end{cases}$$

and *poly* and f are defined as in *k_poly_p*.

These data sets were used as benchmarks in the TANCS-1998 competition [10]. An alternate series of benchmark data was used for the TANCS-2000 competition [87], these being modal quantified Boolean formulae (QBF formulae have been defined in Definition 3.6.1). Some of these TANCS-2000 test data sets however ran for over 3 hours before being timed out [6]. An analysis of empirical test methodologies and a comparison of the Heuerding / Schwendimann, 3CNF and QBF data sets and their results is provided in the paper [66].

Having considered the available benchmark data sets, the Heuerding / Schwendimann data sets were selected, as these formulae are not randomly generated and have *known* results. This is an important feature when developing a prototype for formulae as complex as modal formulae. These data sets are of increasing complexity, which gives a good understanding of the limitations of the prototype. For example, if the prototype cannot solve the fifth dataset, it will definitely not be able to solve any of the remaining data sets. A standard is set on the timing so that, if a prototype does not return a result within 100 CPU seconds, the formula is counted as unsolvable. This means that results are quickly verified. These data sets are therefore considered ideal for the work that follows.

4.3 Reflexivity and the KT_KCSP solver

Before adding reflexivity to the K_KCSP algorithm, we need to understand the specifics of the problem and describe how it will be implemented.

4.3.1 Basic Issues

Consider the following modal formula to which the reflexivity axiom or rule T (Definition 2.3.3) is applied *at the current modal layer.*

Example 4.3.1. Suppose we have

$$\varphi = p_1 \vee p_2 \vee \Box\Box\Box(p_3 \vee \Box p_4)$$

If we apply the reflexivity or (T) rule, which replaces $\Box\varphi$ with $\Box\varphi \wedge \varphi$, we get

$$p_1 \vee p_2 \vee (\Box\Box\Box(p_3 \vee \Box p_4) \wedge \Box\Box(p_3 \vee \Box p_4) \wedge \Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4)))$$

$\dashv$

We can immediately see that there are two challenges when reflexivity is applied – firstly the number of clauses in the formula is significantly increased and secondly, the formula is no longer in CNF (Definition 2.2.21).

We address these issues separately:

1. *Problem: The increase in the number of clauses*

   Example 4.3.2. We take as an example the formula $\varphi = \Box\Box\Box(p_3 \vee \Box p_4)$ and stratify it into modal layers. We apply the reflexivity rule to get:

   $$\begin{aligned} \text{Layer 1} \quad : \quad & \Box\Box\Box(p_3 \vee \Box p_4) \wedge \Box\Box(p_3 \vee \Box p_4) \wedge \Box(p_3 \vee \Box p_4) \wedge \\ & (p_3 \vee (\Box p_4 \wedge p_4)) \end{aligned}$$

   For this formula to be true at this world, we can set p_3 and the $\Box$-formulae to *True.* This formula is thus satisfiable at this modal layer and strictly speaking no further processing is required. However, the purpose of this example is to clarify the expansion of such a formula and so we proceed. We now move to the next modal layer where the formula becomes $\Box\Box(p_3 \vee \Box p_4) \wedge \Box(p_3 \vee \Box p_4) \wedge (p_3 \vee \Box p_4)$. We apply the reflexivity rule once more to get:

$$\begin{aligned} \text{Layer 2} \quad : \quad & (\Box\Box(p_3 \vee \Box p_4) \wedge \Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))) \wedge \\ & (\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))) \wedge \\ & (p_3 \vee (\Box p_4 \wedge p_4)) \end{aligned}$$

which can be simplified to $(\Box\Box(p_3 \vee \Box p_4) \wedge \Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4)))$. Once more, setting p_3 and the $\Box$-formulae to *True* ensures the satisfiability of the formula at this modal layer.

We move to the next modal layer where the formula becomes $\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee \Box p_4)$. We apply the reflexivity rule to get:

$$\text{Layer 3} \quad : \quad (\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))) \wedge (p_3 \vee (\Box p_4 \wedge p_4))$$

This formula can be simplified to $\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4)))$. We follow the same process of setting p_3 and the $\Box$-formulae to *True* to achieve satisfiability at this world. At the next modal layer, the formula becomes $(p_3 \vee \Box p_4)$. We apply the reflexivity rule to get:

$$\text{Layer 4} \quad : \quad (p_3 \vee (\Box p_4 \wedge p_4))$$

Once more, p_3 is set to $True$ to achieve satisfiability at this world.

In effect, we have satisfied this formula by creating a model in which p_3 is *True* at each of the four worlds in the model. We can see from the above that the strict application of the reflexivity axiom results in the repetition of clauses. We also see that the modal depth of the formula decreases at each modal layer. $\dashv$

An alternate approach is required to reduce the number of clauses generated – we propose the following Lemma.

Lemma 4.3.3. *For any frame $\mathcal{F} = (W, R)$, R is reflexive if and only if, for n > 0,*

$$\Box^n \varphi \to \varphi$$

is valid in the frame, where $\Box^n$ represents n occurrences of $\Box$.

Proof. Suppose we have an arbitrary frame $\mathcal{F}$ in which R is reflexive. By applying the T axiom (Definition 2.3.3) to $\Box^n\varphi$, we get $\Box^n\varphi \to \Box^{n-1}\varphi$. Similarly, $\Box^{n-1}\varphi \to \Box^{n-2}\varphi$ and finally, $\Box\varphi \to \varphi$. Thus, $\Box^n\varphi \to \varphi$ is valid in $\mathcal{F}$. Since $\mathcal{F}$ and n were arbitrarily chosen, $\Box^n\varphi \to \varphi$ is valid in any reflexive frame.

Conversely, suppose $\Box^n\varphi \to \varphi$ is valid in an arbitrary frame $\mathcal{F} = (W, R)$. By setting $n = 1$, we have that $\Box\varphi \to \varphi$ is valid, which is the T axiom. Therefore, R is reflexive (Definition 2.3.5). $\dashv$

To simplify the Lemma which follow, we will refer to $\Box^n\varphi \to \varphi$ as the KT' axiom.

We next need to prove soundness and completeness. However, before proceeding, we revisit the internal representation of a modal formula and look at how the modal literals are actually dealt with.

Recall that the K_KCSP algorithm assigns a unique propositional literal to each modal literal (Definition 3.4.11), which effectively means that modal literals are not processed at the current modal layer – all that happens is that they are assigned a value from the domain $\{0,\ 1,\ u\}$. To be more specific, a positive modal literal will be assigned a value of either 1 or u and only if it is assigned a value of 1 does it move to the next modal layer. For example, if we have a clause that contains $p \vee \Box p_1$, the propositional approximation of which is $p \vee x_1[\Box p_1]$, $x_1[\Box p_1]$ will typically be assigned the value u and will only be given the value 1 if p is $False$. If we have a modal unit clause $\Box p_2$, which has the propositional approximation $x_2[\Box p_2]$, it will be assigned a value of 1 and the next modal layer will have p_2 as a propositional literal (recall Definition 3.4.6 and Theorem 3.4.14).

When the T axiom is applied to $\Box^n\varphi$, the expansion is $\Box^n\varphi \wedge \Box^{n-1}\varphi \wedge \ldots \wedge \varphi$. This formula is represented internally as $x_1[\Box^n\varphi] \wedge x_2[\Box^{n-1}\varphi] \wedge \ldots \wedge \varphi$ where each x_i is a unique propositional literal. The constraint solver assigns the value of 1 or u to each x_i. Therefore, the only literal in this expansion which could be assigned the value $False$ is φ.

Lemma 4.3.4. *Applying the KT' axiom at each modal layer, to each occurrence of $\Box^n\varphi$, is a sound and complete strategy to ensure the reflexivity of R in the KT_KCSP algorithm.*

Proof. We need to show that the application of the KT' axiom at the each modal layer returns the same result as the application of the T axiom.

We begin by considering the current modal layer. The application of the KT' axiom to the clause $\Box^n\varphi$ results in it being replaced with $\varphi_1 = \Box^n\varphi \wedge \varphi$. The application of the T axiom results in it being replaced with $\varphi_2 = \Box^n\varphi \wedge \Box^{n-1}\varphi \wedge \ldots \wedge \varphi$.

When the constraint solver evaluates a modal formula, it assigns a value of 1 or u to each positive modal literal, as already discussed. This means that, when φ_1 and φ_2 are processed, they are both effectively reduced to φ. Hence, the constraint solver will return the same result at this modal layer.

At the next modal layer, if $n = 1$, we have φ from both φ_1 and φ_2. For $n > 1$, we have $\Box^{n-1}\varphi$ from φ_1 and the application of the KT' axiom gives $\varphi_1' = \Box^{n-1}\varphi \wedge \varphi$. We have $\Box^{n-1}\varphi \wedge \ldots \wedge \varphi$ from φ_2 and the application of the T axiom gives $\varphi_2' = \Box^{n-1}\varphi \wedge \ldots \wedge \varphi$. Again, since the constraint solver assigns a value of 1 or u to each positive modal literal, φ_1' and φ_2' are both effectively reduced to φ and so both scenarios return the same result.

The application of both axioms therefore returns the same result.

$\dashv$

Example 4.3.5. We apply Lemma 4.3.4 to Example 4.3.2 above to get:

Layer 1 : $\Box\Box\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))$

Layer 2 : $\Box\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))$

Layer 3 : $\Box(p_3 \vee \Box p_4) \wedge (p_3 \vee (\Box p_4 \wedge p_4))$

Layer 4 : $(p_3 \vee (\Box p_4 \wedge p_4))$

These formulae are satisfiable at each modal layer by setting p_3 to *True* – we have created the same model as in the previous example. ⊣

2. *Problem: The loss of CNF*

We need to address the problem of dealing with the loss of conjunctive normal form in the formula after Lemma 4.3.4 has been applied (recall that the K_KCSP algorithm (Section 3.5.4) is based on formulae that are in CNF, as is the DPLL SAT algorithm (Section 3.4.1)).

Example 4.3.6. After Lemma 4.3.4 has been applied to the formula $\varphi = p_1 \vee \Box p_2 \vee \Box p_3 \vee \Box p_4$, it becomes

$$(p_1 \vee (\Box p_2 \wedge p_2) \vee (\Box p_3 \wedge p_3) \vee (\Box p_4 \wedge p_4))$$

in which we have three clauses that are not in conjunctive normal form. When converted to CNF (Definition 2.2.21), it becomes

$$\begin{aligned}
&(p_1 \vee \Box p_2 \vee \Box p_3 \vee \Box p_4) \wedge\\
&(p_1 \vee p_2 \vee \Box p_3 \vee \Box p_4) \wedge\\
&(p_1 \vee \Box p_2 \vee p_3 \vee \Box p_4) \wedge\\
&(p_1 \vee p_2 \vee p_3 \vee \Box p_4) \wedge\\
&(p_1 \vee \Box p_2 \vee \Box p_3 \vee p_4) \wedge\\
&(p_1 \vee p_2 \vee \Box p_3 \vee p_4) \wedge\\
&(p_1 \vee \Box p_2 \vee p_3 \vee p_4) \wedge\\
&(p_1 \vee p_2 \vee p_3 \vee p_4)
\end{aligned}$$

Hence, three clauses not in CNF have been converted into $8 = 2^3$ clauses. The growth in the number of clauses has occurred due to the conversion of the formula into CNF. ⊣

Theorem 4.3.7. *Suppose we have a modal formula φ in conjunctive normal form, which contains a modal clause which has n positive modal literals. Suppose we apply Lemma 4.3.4 to this modal clause and then convert it back to CNF. The original modal clause will be replaced by 2^n modal clauses.*

Proof. The proof is by induction and is applied only to the modal clause. Let φ' be the modal clause with $\varphi' = \psi_1 \vee \Box\varphi_1 \vee \ldots \vee \Box\varphi_n$ where ψ_1 contains no positive modal literals.

In the base case, if $n = 0$, the clause will stay the same after Lemma 4.3.4 has been applied, generating $2^0 = 1$ clause.

Suppose that the theorem holds for some $n > 0$. That is, after Lemma 4.3.4 has been applied and the formula is converted to CNF, 2^n clauses are generated from these positive modal literals.

We must now show that the theorem holds for a clause φ' which has $n + 1$ positive modal literals. We can rewrite $\varphi' = \psi_1 \vee \Box\varphi_1 \vee \ldots \vee \Box\varphi_n$ as $\varphi' = \psi_1 \vee \psi_2$ where $\psi_2 = \psi_n \vee \Box\varphi_{n+1}$ and $\psi_n = \Box\varphi_1 \vee \ldots \vee \Box\varphi_n$. We first apply Lemma 4.3.4 to $\Box\varphi_{n+1}$ to get $\psi_2 = \psi_n \vee (\Box\varphi_{n+1} \wedge \varphi_{n+1})$.

By the inductive hypothesis, when Lemma 4.3.4 is applied to ψ_n and the result is converted to CNF, it becomes 2^n clauses – we denote each clause as ψ_1, …, ψ_{2^n}. By substitution, we now have

$$\psi_2 = (\psi_1 \wedge \ldots \wedge \psi_{2^n}) \vee (\Box\varphi_{n+1} \wedge \varphi_{n+1}).$$

Applying the distributive law (Definition 2.2.21) gives

$$\psi_2 = (\psi_1 \vee (\Box\varphi_{n+1} \wedge \varphi_{n+1})) \wedge \ldots \wedge (\psi_{2^n} \vee (\Box\varphi_{n+1} \wedge \varphi_{n+1}))$$

The re-application of the distributive law gives

$$\psi_2 = (\psi_1 \vee \Box\varphi_{n+1}) \wedge (\psi_1 \vee \varphi_{n+1}) \wedge \ldots \wedge (\psi_{2^n} \vee \Box\varphi_{n+1}) \wedge (\psi_{2^n} \vee \varphi_{n+1})$$

which is now in CNF and contains $2^n + 2^n = 2^{n+1}$ clauses. That is, the theorem holds.

$\dashv$

The theorem was proved for a single NNF clause which contains n positive modal literals. If we have m NNF clauses in a modal formula, each of which has n positive modal literals, the number of clauses generated will be $2^n m$. Hence, the application of the CNF rule causes an exponential increase in the number of clauses.

4.3.2 Approaches to the KT_KCSP algorithm

The work that follows expands the work done in the paper by Brand et al. [18] to include the modal logics KT and $S4$. The prototype of the K_KCSP algorithm which was kindly provided by Brand was applicable only to formulae in CNF and was applicable only to the modal logic K. I now discuss the enhancements I have made to deal with modal formulae not in conjunctive normal form, as well as the enhancements made to the prototype to make provision for the modal logics KT and $S4$. We begin with KT.

We have seen that after reflexivity has been applied to a modal formula and it is translated back into CNF, we end up with an exponential increase in the number of clauses. Therefore, I propose two alternate approaches:

1. As was the case with the K_KCSP prototype, convert the input formula to CNF. Thereafter, at each modal layer, apply Lemma 4.3.4 to the positive modal literals in the formula and convert the result back to CNF before processing by the constraint solver. This approach maintains the formula in CNF. I refer to this approach as KT_KCSP_CNF.

2. Do not convert the input formula or the formula after Lemma 4.3.4 has been applied into CNF. I refer to this approach as KT_KCSP_NoCNF.

4.3.3 The KT_KCSP_CNF algorithm

Algorithm 6. The KT_KCSP_CNF algorithm schema can be represented as follows:

```
function KT_KCSP_CNF($\varphi$)
    $\varphi_{neg} = \neg\varphi$;
    $\varphi_{init} = to\_cnf(\varphi_{neg})$;
    KT_CSP_CNF($\varphi_{init}$);
end;

function KT_CSP_CNF($\varphi$)              // succeeds if $\varphi$ is satisfiable
    $\varphi_{kt} = apply\_reflexivity(\varphi)$;
    $\varphi_{cnf} = to\_cnf(\varphi_{kt})$;
    $\varphi_{csp} = to\_csp(\varphi_{cnf})$;          // backtrack if this fails
    $\mu := csp(\varphi_{csp})$;
    $\Theta = \bigwedge\{\alpha : \Box\alpha = 1 \text{ is in } \mu\}$;
    for each $\Box\beta = 0$ in $\mu$ do
        KT_CSP_CNF($\Theta \wedge \neg\beta$);   // backtrack if this fails
end;
```

The algorithm proceeds as follows: The initial formula is negated and then converted to CNF. Thereafter, at each modal layer, after Lemma 4.3.4 has been applied, the formula is converted first to CNF and then to a constraint satisfaction problem. It is submitted to the ECL^iPS^e constraint solver, which either returns a truth assignment or backtracks.

The only difference between this and the K_KCSP algorithm (Section 3.5.4) is the application of Lemma 4.3.4 and the conversion of the result to CNF.

4.3.4 The KT_KCSP_NoCNF algorithm

We begin by considering an input formula and then look at an example that illustrates the benefits of not applying CNF. This example serves to introduce the approach we will be taking.

In the K_KCSP algorithm, the input formula was first converted into negation negative form (Definition 2.2.18) and then into CNF. We now propose the conversion of the input formula into only NNF. This conversion occurs only once.

In the following example, we convert a modal formula to CNF and then generate its partial truth assignments (Definitions 3.4.6 and 3.4.8). We compare this with the case where CNF is not applied. This serves to illustrate the approach we will use in the KT_KCSP_NoCNF prototype.

Example 4.3.8. Consider the modal formula

$$\varphi = \Box\psi \wedge (\neg\Box\psi_1 \vee (\neg\Box\psi_2 \wedge \neg\Box\psi_3))$$

which consists of modal literals.

When we convert this formula to CNF, we get

$$\varphi' = \Box\psi \wedge (\neg\Box\psi_1 \vee \neg\Box\psi_2) \wedge (\neg\Box\psi_1 \vee \neg\Box\psi_3)$$

There are several possible truth assignments that the constraint solver can return, these being:

(1) $\mu = \{\Box\psi, \neg\Box\psi_1\}$
(2) $\mu = \{\Box\psi, \neg\Box\psi_1, \neg\Box\psi_2\}$
(3) $\mu = \{\Box\psi, \neg\Box\psi_1, \neg\Box\psi_3\}$
(4) $\mu = \{\Box\psi, \neg\Box\psi_2, \neg\Box\psi_3\}$

If we do not convert the formula to CNF, we can instead verify the satisfiability of

$$\varphi_1 = \Box\psi \wedge \neg\Box\psi_1$$

and

$$\varphi_2 = \Box\psi \wedge \neg\Box\psi_2 \wedge \neg\Box\psi_3$$

with the proviso that φ_2 is processed only if φ_1 is not satisfiable.

The possible truth assignments the constraint solver will return in this case are

(5) $\mu = \{\Box\psi, \neg\Box\psi_1\}$
(6) $\mu = \{\Box\psi, \neg\Box\psi_2, \neg\Box\psi_3\}$

Now suppose that $\neg\psi_1$ is unsatisfiable (recall that $\neg\Box\psi_1$ becomes $\neg\psi_1$ at the next modal layer). If this was a complex formula in that it consisted of many modal layers, it could take considerable resources before establishing that it is unsatisfiable. In the case in which the formula was translated into CNF, $\neg\psi_1$ is processed three times, causing backtracking each time; in the second case, it gets processed only once. $\dashv$

The approach I propose is as follows: Instead of converting the formula into CNF, add a step in which we *selectively construct* the modal formula to convert into a constraint satisfaction problem and then pass to the constraint solver. In the event that this formula is not satisfiable, the constraint solver will backtrack and an alternate formula will be constructed for processing. This process will be repeated until either a satisfiable result is returned or no further processing is possible.

Prior to defining the algorithm, I provide an alternate way of structuring a modal formula.

Definition 4.3.9. *A NNF clause ψ in a modal formula φ can be represented as*

$$\psi = \psi' \vee \theta_1 \vee \ldots \vee \theta_n$$

where

$$\psi' = \bigvee\{l : l \in P\} \vee \bigvee\{\Box\alpha : \Box\alpha \in B^+\} \vee \bigvee\{\neg\Box\beta : \Box\beta \in B^-\}$$

and P is a set of propositional literals, B^+ and B^- are sets of modal atoms and θ_1, …, θ_n are NNF formulae (Definition 2.2.18).

Example 4.3.10. Consider the NNF clause

$$\psi = p_1 \vee p_2 \vee (p_3 \wedge (\Box p_4 \vee p_5)) \vee \neg\Box p_6$$

In terms of the above definition, it can be rewritten with $\psi' = p_1 \vee p_2 \vee \neg\Box p_6$, where $\theta_1 = (p_3 \wedge (\Box p_4 \vee p_5))$. $\dashv$

I now define the KT_KCSP_NoCNF algorithm.

Algorithm 7. The KT_KCSP_NoCNF algorithm schema can be represented as follows:

```
function KT_KCSP_NoCNF($\varphi$)
    $\varphi_{neg} = \neg\varphi$;
    $\varphi_{nnf} = to\_nnf(\varphi_{neg})$;
    KT_CSP_NoCNF($\varphi_{nnf}$);
end;

function KT_CSP_NoCNF($\varphi$)                    // succeeds if $\varphi$ is satisfiable
    $\varphi_{kt} = apply\_reflexivity(\varphi)$;
    $\varphi_{formula} = construct\_formula(\varphi_{kt})$;
    $\varphi_{csp} = to\_csp(\varphi_{formula})$;
    $\mu := csp(\varphi_{csp})$;                     // backtrack if this fails
    $\Theta = \bigwedge\{\alpha : \Box\alpha = 1 \text{ is in } \mu\}$;
    for each $\Box\beta = 0$ in $\mu$ do
        KT_CSP_NOCNF($\Theta \wedge \neg\beta$);     // backtrack if this fails
end;
```

The input formula is first converted into negation normal form (Definition 2.2.18) and Lemma 4.3.4 is applied, returning φ_{kt}. The new function *construct_formula* proceeds as follows. Each clause in φ_{kt} is grouped as per Definition 4.3.9. The formula $\varphi_{formula}$ is constructed as the conjunction of the ψ' components of each NNF clause. If there is no ψ' component in an NNF clause, the θ_1 component is used instead. This formula is then converted into a constraint satisfaction problem and fed to the constraint solver. If the constraint solver cannot find a solution, it will backtrack to *construct_formula* and a new formula will be built using the remaining θ_i clauses.

This is demonstrated in the following example.

Example 4.3.11. Consider the modal formula

$$\varphi = \Box\psi_3 \wedge (\neg p_1 \vee (p_4 \wedge \Box\psi_1)) \wedge (p_2 \vee (\neg p_2 \wedge \Box\psi_2))$$

which becomes

$$\varphi' = \Box\psi_3 \wedge \psi_3 \wedge (\neg p_1 \vee (p_4 \wedge \Box\psi_1 \wedge \psi_1)) \wedge (p_2 \vee (\neg p_2 \wedge \Box\psi_2 \wedge \psi_2))$$

after Lemma 4.3.4 has been applied.

In terms of Definition 4.3.9, the first clause has $\psi'_1 = \Box\psi_3$, the second has $\psi'_2 = \psi_3$, the third has $\psi'_3 = \neg p_1$ and $\theta_{21} = (p_4 \wedge \Box\psi_1 \wedge \psi_1)$ and the fourth has $\psi'_3 = p_2$ and $\theta_{31} = (\neg p_2 \wedge \Box\psi_2 \wedge \psi_2)$. The function *construct_formula* successively generates the following formulae to submit to the constraint solver.

(1) $\varphi_1 = \Box\psi_3 \wedge \psi_3 \wedge \neg p_1 \wedge p_2$
(2) $\varphi_2 = \Box\psi_3 \wedge \psi_3 \wedge \neg p_1 \wedge (\neg p_2 \wedge \Box\psi_2 \wedge \psi_2)$
(3) $\varphi_3 = \Box\psi_3 \wedge \psi_3 \wedge (p_4 \wedge \Box\psi_1 \wedge \psi_1) \wedge p_2$
(4) $\varphi_4 = \Box\psi_3 \wedge \psi_3 \wedge (p_4 \wedge \Box\psi_1 \wedge \psi_1) \wedge (\neg p_2 \wedge \Box\psi_2 \wedge \psi_2)$

The constraint solver processes the first of these formulae and only if it is not satisfiable will it backtrack and process the next formula. $\dashv$

We can see from the above example that this approach displays the same exponential behavior as the conversion to CNF – if the modal formula consisted of three NNF clauses, each of which had a θ_1-component, *construct_formula* would generate eight formulae in total. However, as we have seen in Example 4.3.8, there are benefits to selectively constructing the modal formula to submit to the constraint solver.

4.3.5 The initial KT_KCSP prototypes

The prototypes discussed in the previous two sections were tested using the Heuerding-Schwendimann *KT* data sets (discussed on page 103). Any data set that takes more than one hundred CPU-seconds to return a result is discounted. Wherever all 21 data sets have been solved, it is indicated by the '>' symbol. The results obtained for each class are listed in Table 4.1. In order to obtain a clearer understanding of the results, they are classified in the second table in terms of the number of data sets solved in each of 5 categories. For the n-data sets, the CNF prototype solved between 0 and 5 data sets for 4 classes while being unable to solve 21 data sets for any class. For the p-data sets, all 21 data sets were solved for only one class, this being *kt_poly_p*. Ideally, the higher the numbers in the last column, the more successful the solver is.

The KT_KCSP_CNF prototype was unable to solve any data sets in the *kt_grz_n*, *kt_grz_p* and *kt_t4p_p* classes. An analysis shows that this is due to the large number

	CNF	NoCNF	CNF	NoCNF
	n	**n**	**p**	**p**
kt_branch	12	3	18	2
kt_45	7	8	9	8
kt_dum	11	14	3	5
kt_grz	**0**	>	**0**	9
kt_md	5	5	5	4
kt_path	10	10	2	2
kt_ph	7	7	3	4
kt_poly	2	2	>	>
kt_t4p	1	3	**0**	3

		0-5	6-10	11-15	16-20	21
CNF	n	4	3	2	–	–
	p	6	1	–	1	1
NoCNF	n	4	3	1	–	1
	p	6	2	–	–	1

Table 4.1: Initial results of the KT_KCSP prototypes

of clauses generated by the CNF conversion – in fact, after 10 minutes, no result was returned for the first data sets. KT_KCSP_NoCNF on the other hand solved 21, 9 and 3 data sets respectively for each of these classes. For the data sets of the *kt_branch* class, KT_KCSP_CNF returned significantly better results than KT_KCSP_NoCNF – in this case, the CNF conversion is beneficial. For the *kt_path* and *kt_poly* classes, both prototypes solved the same number of data sets and for the remaining classes, there are not significant differences in their performance.

Note that in most cases a smaller number of problems were solved for p-data sets than for n-data sets.

The KT_KCSP_CNF and KT_KCSP_NoCNF prototypes used so far have been based on the code of Brand, with the addition of Lemma 4.3.4 in both cases and the removal of the conversion to CNF in the second prototype. In order to improve these results, the prototypes need to be further enhanced. These enhancements are detailed in the remainder of this chapter. They have, to my knowledge, not been applied in a constraint logic environment before.

4.3.6 KT – Enhancement 1 – Propositional and modal simplification

We saw in the previous chapter (Sections 3.1.4 and 3.4.1) that simplification results in significant improvements in performance for all the solvers developed – this is an obvious area to address. The K_KCSP prototype includes simplification at a propositional level – we now look at whether any further improvements are possible.

Unit subsumption and unit resolution (Rules 3.4.1 and 3.4.2) are both applied to modal formulae which are in conjunctive normal form. In the KT_KCSP_NoCNF prototype, a modal formula is no longer in CNF and hence simplification is *not* applied to clauses, simply because the clauses are no longer programmatically in the correct format. We consider the implications thereof in the next example.

Example 4.3.12. Consider the modal formula

$$\begin{aligned}
\varphi = \quad & p_{100} \wedge \\
& \neg p_{101} \wedge \\
& \Box((\neg p_{101} \vee p_{100}) \wedge \\
& \quad (\neg p_{102} \vee p_{101}) \wedge \\
& \quad (\neg p_{100} \vee ((\neg p_0 \vee \Box(\neg p_{100} \vee p_0)) \wedge (p_0 \vee \Box(\neg p_{100} \vee \neg p_0)))) \wedge \\
& \quad (\neg p_{101} \vee ((\neg p_1 \vee \Box(\neg p_{101} \vee p_1)) \wedge (p_1 \vee \Box(\neg p_{101} \vee \neg p_1)))) \wedge \\
& \quad (p_{101} \vee \neg p_{100} \vee ((\neg\Box(p_{102} \vee \neg p_{101} \vee \neg p_1)) \\
& \qquad \wedge (\neg\Box(p_{102} \vee \neg p_{101} \vee p_1)))))
\end{aligned}$$

When Lemma 4.3.4 is applied to it, we get

$$\begin{aligned}
\varphi_1 = \quad & p_{100} \wedge & (1)\\
& \neg p_{101} \wedge & (2)\\
& (\Box((\neg p_{101} \vee p_{100}) \wedge \\
& \quad (\neg p_{102} \vee p_{101}) \wedge \\
& \quad (\neg p_{100} \vee ((\neg p_0 \vee \Box(\neg p_{100} \vee p_0)) \wedge (p_0 \vee \Box(\neg p_{100} \vee \neg p_0)))) \wedge \\
& \quad (\neg p_{101} \vee ((\neg p_1 \vee \Box(\neg p_{101} \vee p_1)) \wedge (p_1 \vee \Box(\neg p_{101} \vee \neg p_1)))) \wedge \\
& \quad (p_{101} \vee \neg p_{100} \vee ((\neg\Box(p_{102} \vee \neg p_{101} \vee \neg p_1)) \\
& \qquad \wedge (\neg\Box(p_{102} \vee \neg p_{101} \vee p_1)))))) \wedge
\end{aligned}$$

$$
\begin{aligned}
&(\neg p_{101} \vee p_{100}) \wedge && (3)\\
&(\neg p_{102} \vee p_{101}) \wedge && (4)\\
&(\neg p_{100} \vee (((\neg p_0 \vee ((\neg p_{100} \vee p_0) \wedge (\Box(\neg p_{100} \vee p_0)))) \wedge \\
&\quad ((p_0 \vee ((\neg p_{100} \vee \neg p_0) \wedge (\Box(\neg p_{100} \vee \neg p_0))))))) \wedge && (5)\\
&(\neg p_{101} \vee (((\neg p_1 \vee ((\neg p_{101} \vee p_1) \wedge (\Box(\neg p_{101} \vee p_1)))) \wedge \\
&\quad ((p_1 \vee ((\neg p_{101} \vee \neg p_1) \wedge (\Box(\neg p_{101} \vee \neg p_1))))))) \wedge && (6)\\
&(p_{101} \vee \neg p_{100} \vee ((\neg\Box(p_{102} \vee \neg p_{101} \vee \neg p_1)) \wedge \\
&\qquad (\neg\Box(p_{102} \vee \neg p_{101} \vee p_1)))) && (7)
\end{aligned}
$$

By applying unit subsumption and unit resolution to the NNF clauses, this formula can be simplified to

$$
\begin{aligned}
\varphi_2 = \quad & p_{100} \wedge \\
& \neg p_{101} \wedge \\
& (\Box((\neg p_{101} \vee p_{100}) \wedge \\
& \quad (\neg p_{102} \vee p_{101}) \wedge \\
& \quad (\neg p_{100} \vee ((\neg p_0 \vee \Box(\neg p_{100} \vee p_0)) \wedge (p_0 \vee \Box(\neg p_{100} \vee \neg p_0)))) \wedge \\
& \quad (\neg p_{101} \vee ((\neg p_1 \vee \Box(\neg p_{101} \vee p_1)) \wedge (p_1 \vee \Box(\neg p_{101} \vee \neg p_1)))) \wedge \\
& \quad (p_{101} \vee \neg p_{100} \vee ((\neg\Box(p_{102} \vee \neg p_{101} \vee \neg p_1)) \\
& \qquad \wedge (\neg\Box(p_{102} \vee \neg p_{101} \vee p_1)))))) \wedge \\
& \neg p_{102} \wedge && (8)\\
& (\neg p_0 \vee ((\neg p_{100} \vee p_0) \wedge (\Box(\neg p_{100} \vee p_0)))) \wedge && (9)\\
& (p_0 \vee ((\neg p_{100} \vee \neg p_0) \wedge (\Box(\neg p_{100} \vee \neg p_0)))) \wedge && (10)\\
& (\neg\Box(p_{102} \vee \neg p_{101} \vee \neg p_1)) \wedge && (11)\\
& (\neg\Box(p_{102} \vee \neg p_{101} \vee p_1)) && (12)
\end{aligned}
$$

The justification for the simplification is as follows: For φ_1 to be $True$, we must have (p_{100}) $True$ and (p_{101}) $False$. Thus, clause (3) is $True$ and can be omitted. In the case of the clause (4), since (p_{101}) is $False$, we must have $(\neg p_{102})$ $True$ and so remove (p_{101}) from it to give clause (8). Clause (5) has $\neg p_{100}$ $False$ and so it is removed to give clauses (9) and (10). Clause (6) has $\neg p_{101}$ $True$ and so this clause is $True$ and can be removed. Finally, clause (7) has $(p_{101} \vee \neg p_{100})$ $False$ – after these are removed, we are left with clauses (11) and (12).

Formula φ_1 has now been considerably simplified – the only choice points we are left with at the current modal layer are in clauses (9) and (10). Hence, *construct_formula* will now generate at most four formulae for the constraint solver. Before simplification was applied, we had five choice points in clause (5), five in clause (6) and two in clause (7). Hence, *construct_formula* could generate up to fifty formulae (5 x 5 x 2). $\dashv$

It is easily seen from the above example that unit subsumption and unit resolution can also be applied to NNF clauses.

We begin by reconsidering the unit subsumption and unit resolution rules applied in the DPLL SAT procedure, as specified in Rules 3.4.1 and 3.4.2 of the previous chapter, and extend these rules as follows.

Enhancement 1. *After Lemma 4.3.4 has been applied to a modal formula φ, we apply the following rules:*

1. *For every clause ψ that is either a propositional unit clause (Definition 2.2.9) or a unit modal literal (Definition 2.2.11), unit subsumption is applied to every other NNF clause (Definition 2.2.18) containing ψ and such clauses are removed, provided that ψ does not occur in a modal formula within the NNF clause.*

2. *For every clause ψ that is either a propositional unit clause or a unit modal literal, unit resolution is applied and $\neg\psi$ is removed from every other NNF clause in which it occurs, provided that $\neg\psi$ does not occur in a modal formula within the NNF clause.*

This process is repeated until no further simplification is possible.

We have extended the rules of the DPLL SAT procedure to include NNF clauses as well as unit modal literals. The proof of the modified rules is as follows.

Justification. We prove unit subsumption and then unit resolution.

1. Suppose we have a modal formula $\varphi = \psi \wedge (\psi \vee \psi_1) \wedge \psi_2$ where ψ is a propositional unit clause or a unit modal literal, ψ_1 is an NNF clause in which ψ does

not occur and ψ_2 is the conjunction of any number of clauses. We can rewrite φ as $\varphi = ((\psi \wedge \psi) \vee (\psi \wedge \psi_1)) \wedge \psi_2$ by the distribution law (Definition 2.2.21). This is equivalent to $\varphi = \psi \wedge \psi_2$ by the rules of propositional logic. Hence, clause $(\psi \vee \psi_1)$ can be excluded from the formula.

2. Suppose we have a modal formula $\varphi = \psi \wedge (\neg\psi \vee \psi_1) \wedge \psi_2$ where ψ is a propositional unit clause or a unit modal literal, ψ_1 is an NNF clause in which ψ does not occur and ψ_2 is the conjunction of any number of NNF clauses. We can rewrite φ as $\varphi = ((\psi \wedge \neg\psi) \vee (\psi \wedge \psi_1)) \wedge \psi_2$ by the distribution law (Definition 2.2.21). This is equivalent to $\varphi = \psi \wedge \psi_1 \wedge \psi_2$ by the rules of propositional logic – hence, the clause $(\neg\psi \vee \psi_1)$ can be replaced with ψ_1.

$\dashv$

Note that in the case of the KT_KCSP_NoCNF prototype, this enhancement is not applied to modal formulae within an NNF clause. Suppose we have the formula

$$\varphi = p_1 \wedge (p_2 \vee (p_3 \wedge (p_1 \vee q)))$$

Unit subsumption is *not* applied to the modal formula $(p_3 \wedge (p_1 \vee q))$ – no simplification is applied to φ.

The improvements obtained are as follows (we list the prototype used, the class and the additional data sets that have been solved):

	Additional data sets solved
CNF	*kt_grz_n*(1 – 21)
	kt_t4p_n (1 – 2)
	kt_45_p (10 – 11)
	kt_dum_p (4 – 6)
	kt_t4p_p (1)
NoCNF	*kt_branch_n*(4 – 10)
	kt_branch_p(3 – 21)
	kt_45_p(9)
	kt_t4p_p(4)

This enhancement has resulted in significant improvements in the *kt_grz_n* data sets in the case of KT_KCSP_CNF as well as in the *kt_branch_p* data sets in the case of KT_KCSP_NoCNF. However, KT_KCSP_CNF is still unable to solve any of the *kt_grz_p* data sets.

The modal formula in Example 4.3.12 comes from data set *kt_branch_n*(1). Before simplification was applied, the KT_KCSP_CNF prototype solved 12 of the *kt_branch_n* data sets while the KT_KCSP_NoCNF prototype solved only 3; after simplification was applied, there was no change in the results of the KT_KCSP_CNF prototype whereas the KT_KCSP_NoCNF prototype now solves 10 data sets. The two prototypes have similar timings up to the tenth data set that they solve in 59.80 CPU seconds and 65.89 CPU seconds respectively.

4.3.7 KT – Enhancement 2 – Early pruning

Lemma 4.3.4 and then simplification are applied to each modal formula – to be more specific, they are applied to the initial modal formula and then to each of the $\varphi_j = \bigwedge_i \alpha_i \wedge \neg\beta_j$ formulae. An analysis of some of the data sets that do not perform well shows that the following scenario can occur.

Example 4.3.13. Suppose we have the modal formula

$$\varphi = \Box p \wedge \Box\psi_1 \wedge \neg\Box(\neg p \vee \psi_2) \wedge \neg\Box(p \vee \psi_3)$$

where the ψ_i are of any complexity.

To prove its satisfiability at the next modal layer, we need to verify the satisfiability of each $\varphi_j = \bigwedge_i \alpha_i \wedge \neg\beta_j$, as per Theorem 3.4.14. At the next modal layer, we have $\bigwedge_i \alpha_i = p \wedge \psi_1$ from the positive modal literals and $\beta_1 = (\neg p \vee \psi_2)$ and $\beta_2 = (p \vee \psi_3)$ from the negative modal literals.

We first process $\bigwedge_i \alpha_i \wedge \neg\beta_1 = (p \wedge \psi_1) \wedge (p \wedge \neg\psi_2)$. If this is satisfiable, we process $\bigwedge_i \alpha_i \wedge \neg\beta_2 = (p \wedge \psi_1) \wedge (\neg p \wedge \neg\psi_3)$, which we immediately find is unsatisfiable.

Formulae ψ_1 and ψ_2 could be highly complex in which case a lot of resources are expended on the first formula before the second is found to be unsatisfiable. Visually, it is easy to see that if we have p occurring at the next modal layer from the positive modal literals and $\neg p$ from any of the negative modal literals, we would need to backtrack. $\dashv$

We propose and implement the following enhancement:

Enhancement 2. *Let* $\varphi' = \psi_1 \wedge \bigwedge_j \neg\beta_j$ *where* $\psi_1 = \bigwedge_i \alpha_i$. *If a propositional unit clause* l *in* $\bigwedge_j \neg\beta_j$ *also occurs in* ψ_1, *force a backtrack to the previous modal layer.*

Justification. Suppose some β_j contains a propositional unit clause l and suppose $\bigwedge_i \alpha_i$ also contains l. We then have $\bigwedge_i \alpha_i \wedge \neg\beta_j = l \wedge \ldots \wedge \neg l \wedge \ldots$, which we can immediately see is unsatisfiable. $\dashv$

When this enhancement was applied to the two prototypes, no improvement occurred in any of the n-data sets. The improvements obtained in the p-data sets are as follows:

	Additional data sets solved
CNF	*kt_45_p*(12-14)
	kt_dum_p(7)
	kt_ph_p(4)
	kt_t4p_p(2)
NoCNF	*kt_dum_p*(6-7)

Although not many additional data sets were solved, in general the processing time of the data sets was reduced. The data sets of the *kt_grz_p* class for the KT_KCSP_CNF prototype remained unsolved.

4.3.8 KT – Enhancement 3 – Grouping of clauses

A further analysis shows that in some cases, propositional literals can occur in disjoint clauses.

Example 4.3.14. Consider the following modal formula.

$$
\begin{aligned}
&(\Box(\neg\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \vee p_2)) \wedge && (1)\\
&(\neg\Box(\neg\Box(((\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2)) \wedge \\
&\quad (\neg\Box(\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2)))) \vee \\
&\quad ((\Box(\neg p_2 \vee \Box p_2)) \wedge \neg p_2) \vee \Box((\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \wedge \\
&\quad (\neg\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \vee \Box(\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2))))) \vee \\
&\quad ((\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \wedge (\neg\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \vee \\
&\quad \Box(\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2))))) \vee ((\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \wedge \\
&\quad (\neg\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2) \vee \Box(\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2))))) \wedge && (2)\\
&(\Box(\neg\Box(\neg p_1 \vee \Box p_1) \vee p_1)) \wedge && (3)\\
&(\neg\Box p_1) \wedge && (4)\\
&(((\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2)) \wedge (\neg\Box p_2)) \vee \Box\neg\Box\neg p_0) \wedge && (5)\\
&(\Box(\neg\Box(\neg p_3 \vee \Box p_3) \vee p_3)) \wedge && (6)\\
&(\neg\Box p_3) \wedge && (7)\\
&(((\Box(\neg\Box(\neg p_2 \vee \Box p_2) \vee p_2)) \wedge \neg\Box p_2) \vee \Box\neg p_0) && (8)
\end{aligned}
$$

which contains eight clauses. We have p_1 occurring only in clauses (3) and (4) and p_3 occurring only in clauses (6) and (7). Hence, we can group the clauses in this formula as {(1), (2), (5), (8)}, {(3), (4)} and {(6), (7)} and process each group separately. ⊣

Enhancement 3. *Suppose we have a modal formula φ. Let $\gamma = \{p_1, \ldots, p_n\}$ be the set of propositional atoms p_i occurring in φ, where p_i can occur at any modal layer in φ. We then group the clauses in φ as follows:*

$$\varphi = \psi_1 \wedge \ldots \wedge \psi_m$$

where each ψ_i is a conjunction of NNF clauses and for each $p_k \in \gamma$, if p_k occurs in ψ_i, then p_k does not occur in any other ψ_j, where $j \neq i$.

By determining the satisfiability of each ψ_i, we determine the satisfiability of φ.

Justification. The satisfiability of a set of clauses ψ_i depends on the values of their propositional literals. Because each ψ_i has distinct literals, their satisfiability or otherwise is independent of each other. Furthermore, if each ψ_i is satisfiable, φ is satisfiable; if one of the ψ_is is not satisfiable, φ is not satisfiable. ⊣

When this enhancement was applied to the two prototypes, once again no improvement occurred in the n-data sets. The improvements obtained in the p-data sets were as follows:

	Additional data sets solved
CNF	*kt_grz_p*(1 – 3)
NoCNF	*kt_45_p*(10 – 13)
	kt_grz_p(10)

KT_KCSP_CNF is now able to solve three data sets in the *kt_grz_p* class.

4.3.9 KT – Enhancement 4 – Value assignment in unit clauses

Let us look at the *top-level propositional literals* at a particular modal layer. A propositional unit clause with a positive propositional literal p must have a value of 1 to be satisfiable; a propositional unit clause with a negative propositional literal $\neg p$ must have a value of 0. When the constraint satisfaction problem for these literals is constructed, we can therefore limit their domains to $\{1\}$ and $\{0\}$ respectively, instead of $\{0, 1, u\}$.

We can extend this concept further: any top-level propositional literal that *only* occurs positively or that *only* occurs negatively in the modal formula can also have its domain set to $\{1\}$ or $\{0\}$ respectively. It is only in cases where we have both p and $\neg p$ in a formula that we need to specify a domain of $\{0, 1, u\}$.

Note that if we have p and $\neg\Box p$ occurring at a particular modal layer, we do not take $\neg\Box p$ into consideration as it is not a propositional literal.

Now consider a clause which contains a positive propositional literal p to which a value of 1 has been assigned. We can in fact replace the literal with the value 1 to give clause $(\psi_1 \vee 1)$, which is satisfiable. This clause therefore does not need to be passed to the constraint solver. By reducing the number of clauses in the constraint satisfaction problem, we reduce the number of choice points for the constraint solver.

Note that this is *not* the same as unit subsumption – we are now dealing with propositional literals that are not propositional unit clauses.

We illustrate this with the following example.

Example 4.3.15. Suppose $\varphi = (p_1 \vee \neg\Box p_5) \wedge (p_2 \vee \neg\Box p_6) \wedge \Box p_3 \wedge \neg\Box(p_3 \vee p_4)$ is processed by the KT_KCSP prototype. It will return the following truth assignments, each of which is unsatisfiable at the next modal layer:

1. $\mu = \{p_1, p_2, \Box p_3, \neg\Box(p_3 \vee p_4)\}$. The modal formula at the next modal layer is $p_3 \wedge \neg(p_3 \vee p_4)$, which is unsatisfiable and so the constraint solver backtracks.
2. $\mu = \{\neg\Box p_5, p_2, \Box p_3, \neg\Box(p_3 \vee p_4)\}$.
3. $\mu = \{p_1, \neg\Box p_6, \Box p_3, \neg\Box(p_3 \vee p_4)\}$.
4. $\mu = \{\neg\Box p_5, \neg\Box p_6, \Box p_3, \neg\Box(p_3 \vee p_4)\}$.

Suppose we follow the suggestion above. In the formula φ, we have p_1 and p_2 occurring only positively. Since they can be assigned a value of 1, the clauses in which they occur can be removed. The formula that is passed to the constraint solver then becomes

$$\varphi = \Box p_3 \wedge \neg\Box(p_3 \vee p_4)$$

and now only one truth assignment is returned, this being

$$\mu = \{\Box p_3, \neg\Box(p_3 \vee p_4)\}$$

which is unsatisfiable at the next modal layer. $\dashv$

Enhancement 4. *Suppose we have a modal formula φ. Let $\gamma = \{p_1, \ldots, p_{n1}, \neg q_1, \ldots, \neg q_{n2}\}$ where if $p_i \in \gamma$, then p_i occurs only positively in φ and if $q_j \in \gamma$, then q_j occurs only negatively in φ.*

We apply the following rule to the clauses in φ:

For each clause ψ in φ, if ψ contains a propositional unit clause l and $l \in \gamma$, then this clause is removed from φ.

Justification. If a propositional literal occurs only positively in the modal formula φ, it must be assigned a value of 1 from its domain of $\{0, 1, u\}$. The NNF clause ψ it occurs in becomes $\psi = 1 \vee \psi_1$ where ψ_1 contains the remaining disjuncts of the clause. No further evaluation of literals is required as the clause is $True$. Hence it can be removed from the modal formula.

If a propositional literal occurs only negatively in the modal formula φ, it must be assigned a value of 0. The NNF clause ψ it occurs in becomes $\psi = \neg 0 \vee \psi_1$, and no further evaluation of disjuncts is required as the clause is $True$. $\dashv$

In this case, the results of both the n- and p-data sets were improved. The improvements were as follows:

	Additional data sets solved
CNF	*kt_branch_n*(11)
	kt_grz_p(1 – 21)
	kt_path_p(3 – 12)
NoCNF	*kt_branch_n*(11)
	kt_grz_p(11 – 21)
	kt_path_p(3 – 12)

This enhancement had a major effect on the *kt_grz_p* and the *kt_path_p* data sets in both prototypes as all have now been solved. This is an indication that reducing the number of clauses passed to the constraint solver has a significant impact.

4.3.10 KT – Enhancement 5 – Caching

The *kt_poly_n* and *kt_t4p_p* data sets still return poor results. In the *kt_poly_n* class, only the first two data sets returned a result within the required timeframe for both the KT_KCSP_CNF and KT_KCSP_NoCNF prototypes – in fact, the third data set was aborted after running for more than 10 minutes.

An analysis of the *kt_poly_n*(2) data set shows the following pattern: At the second modal layer, there are six negative modal literals, which means we will have six branches at layer 3. At layer 3, the constraint solver returns the same positive box and negative modal literals for three of these branches, which means that the same formula is processed three times at the next layer. At layer 4, a similar pattern is found. Hence a lot of reprocessing of the same formulae is taking place. This scenario can be represented as follows:

Modal layer 2:	Positives – ψ, Negatives – ψ_1, ψ_2, …, ψ_6
Modal layer 3.1:	Positives – ψ_1, Negatives – ψ_2
Modal layer 3.2:	Positives – ψ_1, Negatives – ψ_2 (a repeat of Layer 3.1)
Modal layer 3.3:	Positives – ψ_1, Negatives – ψ_2 (a repeat of Layer 3.1)
Modal layer 3.4:	Positives – ψ_3, Negatives – ψ_4
Modal layer 3.5:	Positives – ψ_5, Negatives – ψ_6
Modal layer 3.6:	Positives – ψ_7, Negatives – ψ_8
Modal layer 4.1.1:	Positives – ψ_{10}, Negatives – ψ_{11}
Modal layer 4.1.2:	Positives – ψ_{10}, Negatives – ψ_{11} (a repeat of Layer 4.1.1)
Modal layer 4.1.3:	Positives – ψ_{10}, Negatives – ψ_{11} (a repeat of Layer 4.1.1)
…	…

We can see from the above that the modal formula $(\psi_1 \wedge \neg\psi_2)$ is generated at modal layers 3.1, 3.2 and 3.3. Similarly, the modal formula $(\psi_{10} \wedge \neg\psi_{11})$ is generated at modal layers 4.1.1, 4.1.2 and 4.1.3. This means that the modal formula $(\psi_{10} \wedge \neg\psi_{11})$ will be processed nine times in total.

An analysis of the *kt_t4p_p* class shows the following pattern: In the *kt_t4p_p*(2) data set, at layer 3, we have four positive and five negative modal literals. The first three branches are processed and then the fourth, which contains the negative modal literal $\neg\beta_j$, turns out to be unsatisfiable. The constraint solver backtracks to layer 2 and generates a new solution that consists of the same four positive modal literals and four of the previous five negative modal literals, one of which is $\neg\beta_j$. Again, the modal formula containing $\neg\beta_j$ is unsatisfiable. The constraint solver backtracks and continues to generate minor variations of this same scenario. This process involves a significant amount of reprocessing.

This scenario can be represented as follows:

Modal layer 3:	Positives – ψ, Negatives – ψ_1, …, ψ_4, ψ_5
Modal layer 4.1:	$\psi \wedge \neg\psi_1$ – satisfiable
Modal layer 4.2:	$\psi \wedge \neg\psi_2$ – satisfiable
Modal layer 4.3:	$\psi \wedge \neg\psi_3$ – satisfiable
Modal layer 4.4:	$\psi \wedge \neg\psi_4$ – unsatisfiable – so backtrack to layer 2

Modal layer 3:	Positives – ψ, Negatives – ψ_1, …, ψ_4, ψ_6
Modal layer 4.1:	$\psi \wedge \neg\psi_1$ – satisfiable
Modal layer 4.2:	$\psi \wedge \neg\psi_2$ – satisfiable
Modal layer 4.3:	$\psi \wedge \neg\psi_3$ – satisfiable
Modal layer 4.4:	$\psi \wedge \neg\psi_4$ – unsatisfiable – so backtrack to layer 2
Modal layer 3:	Positives – ψ, Negatives – ψ_1, …, ψ_4, ψ_7

…

The reprocessing in both the above scenarios can be avoided by implementing a caching mechanism to store the formulae processed. We will need to differentiate between those that are satisfiable and those that are unsatisfiable.

We therefore propose the following enhancement.

Enhancement 5. *A cache* Γ *is created in which modal formulae and their respective status are stored –* Γ *contains entries of the form <<modal formula>, <status>>. Before passing a modal formula* φ *to the constraint solver, the cache is checked to see whether the formula has already been processed. The pseudo-code for this process, in which* Γ *is a global variable, is as follows:*

```
function check_cache($\varphi$)
    for each <$\varphi_{stored}$, $status_{stored}$> in $\Gamma$ do
        if $\varphi_{stored} = \varphi$ then
            if $status_{stored}$ is True then return True
                else backtrack;
        if $status_{stored}$ = True and $\varphi \subset \varphi_{stored}$
            return True;                                    (1)
        if $status_{stored}$ = False and $\varphi \supset \varphi_{stored}$
            backtrack;                                      (2)
    add <$\varphi$, False> to $\Gamma$;
    return False;
end;
```

In the KT_KCSP_CNF and KT_KCSP_NoCNF algorithms, the pseudo-code

```
$\varphi_{csp}$ = to_csp($\varphi_{formula}$);
$\mu$ := csp($\varphi_{csp}$);                    // backtrack if this fails
$\Theta = \bigwedge\{\alpha : \Box\alpha = 1$ is in $\mu\}$;
for each $\Box\beta = 0$ in $\mu$ do
    KT_CSP_xx($\Theta \wedge \neg\beta$);          // backtrack if this fails
```

is replaced with the following:

```
$\varphi_{csp}$ = to_csp($\varphi_{formula}$);
if check_cache($\varphi_{csp}$) is False then
    $\mu$ := csp($\varphi_{csp}$);                            // backtrack if this fails
    $\Theta = \bigwedge\{\alpha : \Box\alpha = 1$ is in $\mu\}$;
    for each $\Box\beta = 0$ in $\mu$ do
        KT_CSP_xx($\Theta \wedge \neg\beta$);                  // backtrack if this fails
    update <$\varphi_{csp}$, False> in $\Gamma$ to <$\varphi_{csp}$, True>;
```

where xx = 'CNF' in the case of the KT_KCSP_CNF algorithm and 'NoCNF' in the case of the KT_KCSP_NoCNF algorithm.

This enhancement first checks to see whether φ has been cached. If it has and has been marked as satisfiable, no further processing is required. If it has been marked as unsatisfiable, a backtrack is forced.

If φ has not yet been processed, it checks to see if it is a subformula of any modal formula φ' that has been cached. If this is the case and φ' has been marked as satisfiable, no further processing is required (refer to (1) below for the proof). Otherwise, if a subformula of φ has been cached and marked as unsatisfiable, a backtrack is forced (refer to (2) below for the proof).

If no information is available in the cache for φ, it is added to the cache with a status of unsatisfiable. It is then processed as usual and only if it is found to be satisfiable is its status in the store updated.

Justification. Suppose we have two modal formulae φ_1 and φ_2 and we know the satisfiability of φ_2.

1. Suppose φ_1 is a subformula of φ_2. Then $\varphi_2 = \varphi_1 \wedge \psi_1 \wedge \ldots \wedge \psi_n$ where ψ_i are NNF clauses.

   If φ_2 is satisfiable, each of its clauses are satisfiable, including φ_1. If φ_2 is not satisfiable, any of its clauses can be unsatisfiable and so the satisfiability of φ_1 needs to be determined.

2. Suppose φ_2 is a subformula of φ_1. Then $\varphi_1 = \varphi_2 \wedge \psi_1 \wedge \ldots \wedge \psi_n$ where ψ_i are NNF clauses.

   If φ_2 is satisfiable, because the status of the ψ_i clauses in φ_1 is not known, the satisfiability of φ_1 needs to be determined. If φ_2 is not satisfiable, then φ_1 is also not satisfiable.

$\dashv$

When this enhancement was implemented, considerable improvement occurred in the n-data sets, with some improvements in the p-data sets. These improvements can be summarized as follows:

	Additional data sets solved
CNF	$kt\_45\_n(8-21)$
	$kt\_dum\_n(12-21)$
	$kt\_md\_n(6)$
	$kt\_path\_n(9-21)$
	$kt\_poly\_n(3-9)$
	$kt\_t4p\_n(3-21)$
	$kt\_45\_p(15-21)$
	$kt\_dum\_p(8-21)$
	$kt\_path\_p(13-21)$
	$kt\_t4p\_p(3-12)$
NoCNF	$kt\_45\_n(9-21)$
	$kt\_dum\_n(15-21)$
	$kt\_md\_n(6)$
	$kt\_path\_n(9-21)$
	$kt\_poly\_n(3-10)$
	$kt\_t4p\_n(4-21)$
	$kt\_45\_p(14-21)$
	$kt\_dum\_n(8-21)$
	$kt\_md\_p(5)$
	$kt\_path\_p(13-21)$
	$kt\_t4p\_p(5-21)$

However, in the class *kt_branch_n*, the 11th data set can no longer be solved within the 100 cpu-second time limit. I attribute this to the overhead of the cache – this is the only class negatively impacted.

4.3.11 Final results of the KT_KCSP prototypes

The actual results that have now been obtained are provided in Table 4.2.

We can clearly see that the only classes that remain problematic are *kt_md*, *kt_ph*,

	CNF	**NoCNF**	**CNF**	**NoCNF**
	n	**n**	**p**	**p**
kt_branch	10	10	18	>
kt_45	>	>	>	>
kt_dum	>	>	>	>
kt_grz	>	>	>	>
kt_md	6	6	5	5
kt_path	>	>	>	>
kt_ph	7	7	4	4
kt_poly	9	10	>	>
kt_t4p	>	>	12	>

		0-5	**6-10**	**11-15**	**16-20**	**21**
CNF	n	–	4	–	–	5
NoCNF	n	–	4	–	–	5
CNF	p	2	–	1	1	5
NoCNF	p	2	–	–	–	7

Table 4.2: Final results of the KT_KCSP prototypes

kt_branch_n and *kt_poly_n*. The results of the *p* data sets are better in the case of the KT_KCSP_NoCNF prototype.

There has been considerable improvement when these results are compared with those in Table 4.1 – so we can conclude that both prototypes are effective, with the KT_KCSP_NoCNF prototype remaining more effective than the KT_KCSP_CNF prototype.

4.3.12 The exponential nature of the KT results

We next look at some of the actual results as we need to understand the exponential nature of this class of problem.

We look at the results of the *kt_branch_n* and *kt_md_n* classes for the KT_KCSP_CNF prototype.

- The *kt_branch_n* results are as follows:

Data set	Timing (CPU-seconds)	Exponential factor
1	0.00	–
2	0.03	–
3	0.09	–
4	0.19	2.11
5	0.45	2.37
6	1.13	2.51
7	2.95	2.61
8	8.64	2.93
9	28.17	3.26
10	95.14	3.50

I define a term – the *exponential factor* – that is calculated by taking the timing of the current data set and dividing it by the timing of the previous data set. We can see from these timings that this factor steadily increases and as a result, we can project that the 11th data set will take in excess of 360 CPU-seconds to be solved and that the 12th data set will take in excess of 1320 CPU-seconds. Hence we have exponential behavior.

- The *kt_md_n* results are as follows:

Data set	Timing (CPU-seconds)	Exponential factor
1	0.00	–
2	0.00	–
3	0.02	–
4	0.09	4.5
5	0.87	9.7
6	86.52	99.5

In this case the exponential factor is very high – the next data set will take in excess of 8500 CPU-seconds to solve.

These results give an idea of what is involved in improving the results of a particular class – we would need a considerable improvement for the results of the *kt_branch_n* class to include the 12th data set. In the case of the *kt_md_n* class, one would need a paradigm shift in the algorithm to enable the prototype to solve the 7th data set within the 100 CPU-second time limit.

Clearly, the smaller the number of data sets solved, the higher the exponential factor is – if we had a class for which 17 data sets were solved, the slope of the exponential graph would be lower than the slope of a class for which only 2 data sets were solved.

We will now look at the results obtained in previous benchmarks and then consider some of the reasons for the poor performance of the *kt_md* and *kt_ph* classes.

4.3.13 Comparative benchmark results of the KT data sets

It is interesting to compare results obtained by different solvers. Such a comparison is challenging, since different solvers use different compilers and operating systems, and the benchmarks themselves are run using different hardware configurations. If we compare benchmarks that were run a few years apart, the increase in CPU power will strongly favor the more recent results. However, in the case where the problem is **NP**-complete or worse, this difference will not be as marked – if we refer to the results discussed above, we would not expect newer hardware to be able to solve the 8th data set of the *kt_md_n* class, given that we are using the same solver.

To understand the impact of using more powerful hardware, we look at the results of the K_KCSP solver, as recorded in [18] in 2003, and the results obtained running the same data sets on a typical 2007 PC configuration (we do not provide the details of the configuration as that is irrelevant in this context). Brand et al. made use of Linux; our tests were run in the Windows environment. The comparison is provided in Table 4.3.

The results for some of the data sets have improved – for example, in the case of the *k_dum_n* class, we now solve all of the data sets instead of only 17. However, if we look at the *k_d4_n* class, we now solve only one additional data set – the exponential

Hardware of:	**2004**	**2007**	**2004**	**2007**
	n	**n**	**p**	**p**
branch	11	14	>	>
d4	6	7	8	10
dum	17	>	11	13
grz	>	>	10	11
lin	>	>	>	>
path	9	12	4	5
ph	4	4	4	4
poly	16	>	9	11
t4p	6	7	8	11

Table 4.3: Comparative results of the K_KCSP solver using the Heuerding / Schwendimann K data sets

	FaCT	**DLP**	**KSAT**	**FaCT**	**DLP**	**KSAT**
	n	**n**	**n**	**p**	**p**	**p**
kt_45	>	>	5	>	>	5
kt_branch	4	11	7	6	16	8
kt_dum	>	>	12	11	>	7
kt_grz	>	>	>	>	>	9
kt_md	5	>	4	4	3	2
kt_path	3	>	5	5	6	2
kt_ph	7	18	5	6	7	4
kt_poly	7	6	2	>	6	1
kt_t4p	2	>	1	4	3	1

Table 4.4: Results of the FaCT, DLP and KSAT solvers using the Heuerding / Schwendimann KT data sets

nature of the timing has been retained. Hence, if we look at the results of previous benchmarks, in cases where the timings were exponential, we can still expect to see exponential behavior.

The benchmark results of the TANCS-1998 competition using the Heuerding / Schwendimann *KT* data sets, as obtained for the KSAT, DLP and FaCT solvers [67], are listed in Table 4.4.

We can firstly see from these results that the behavior of each solver is not consistently the same. We can see that all the solvers had difficulties with the *kt_branch* and *kt_ph* classes, with most of them having problems with the *kt_poly_n*, *kt_md_p*, *kt_path_p* and *kt_t4p_p* classes. The KT_KCSP_NoCNF prototype had similar difficulties with the *kt_ph*, *kt_md* and *kt_branch* classes (Table 4.2). Hence we can conclude that this prototype performs favorably.

4.3.14 An analysis of the kt_ph class

We demonstrate the problem faced with these data sets by looking in detail at the data set *kt_ph_p*(2).

After reflexivity and simplification have been applied at the first modal layer, a subset of the formula obtained contains the following:

(1) $((p_{102} \wedge \Box p_{102}) \vee p_{101}) \wedge$
(2) $(\neg p_{302} \vee \neg\Box p_{102}) \wedge$
(3) $(\neg p_{202} \vee \neg\Box p_{102}) \wedge$
(4) $(\neg p_{302} \vee \neg p_{202}) \wedge$
(5) $(\neg p_{301} \vee \neg p_{201}) \wedge$
(6) $(\neg p_{301} \vee \neg p_{101}) \wedge$
(7) $(\neg p_{201} \vee \neg p_{101}) \wedge$
(8) $(p_{202} \vee p_{201}) \wedge$
(9) $(p_{302} \vee p_{301}) \wedge$

We cannot apply Enhancement 4 as we have propositional literals that occur both positively and negatively. We could however establish the satisfiability of this formula as follows:

- Let $p_{101} = 1$ (1). We must then have $p_{301} = 0$ (6) and $p_{201} = 0$ (7) that in turn forces $p_{202} = 1$ (8) and $p_{302} = 1$ (9) . However, clause (4) cannot now be satisfied. Hence, $p_{101} \neq 1$.

- Let $p_{101} = 0$ that gives $p_{102} = 1$ and $\Box(p_{102}) = 1$ (1). We must then have $p_{302} = 0$ (2) and $p_{202} = 0$ (3). This in turn forces $p_{301} = 1$ (9) and $p_{201} = 1$ (8). However clause (5) cannot now be satisfied. Hence, the modal formula is unsatisfiable.

The above analysis quickly returns a result. However, the constraint solver solves this problem by generating the following assignments:

1. $\mu = \{p_{101}, \neg p_{302}, \neg p_{202}, \neg p_{301}, \neg p_{201}\}$, and since clause (8) is *False* with this assignment, the constraint solver backtracks.

2. $\mu = \{p_{101}, \neg p_{302}, \neg p_{202}, \neg p_{201}, \neg p_{301}\}$ – it backtracks.

3. $\mu = \{p_{101}, \neg\Box(p_{102}), \neg p_{302}, \neg p_{301}, \neg p_{201}, p_{202}\}$ – it backtracks.

4. ... and so on ...

KT_KCSP solves this particular data set in 0.00 CPU-seconds. However, for more complex formulae such as the *kt_ph_p*(5) data set, because there are many more clauses and possible combinations to consider, it takes a much longer time.

We saw that some of the solvers of the previous chapter apply a heuristic and assign values to the single propositional literals. If we followed this approach, we could set $p_{101} = 1$ and then simplify the formula. Applying the rules of Enhancement 1, we would get

$$\begin{array}{ll}
(1) & (\neg p_{302} \vee \neg\Box(p_{102})) \wedge \\
(2) & (\neg p_{202} \vee \neg\Box(p_{102})) \wedge \\
(3) & (\neg p_{302} \vee \neg p_{202}) \wedge \\
(4) & (\neg p_{301} \vee \neg p_{201}) \wedge \\
(5) & \neg p_{301} \wedge \\
(6) & \neg p_{201} \wedge \\
(7) & (p_{202} \vee p_{201}) \wedge \\
(8) & (p_{302} \vee p_{301}) \wedge
\end{array}$$

Further simplification would then give us

$$\begin{array}{ll}
(1) & (\neg p_{302} \vee \neg\Box(p_{102})) \wedge \\
(2) & (\neg p_{202} \vee \neg\Box(p_{102})) \wedge \\
(3) & (\neg p_{302} \vee \neg p_{202}) \wedge \\
(4) & \neg p_{301} \wedge \\
(5) & \neg p_{201} \wedge
\end{array}$$

(6) $p_{202} \wedge$

(7) $p_{302} \wedge$

which is clearly unsatisfiable (as can be seen from clauses (3), (6) and (7)). ECL^iPS^e would then backtrack and the same simplification exercise would be carried out with $p_{101} = 0$ – we would not get to the stage of building the constraint satisfaction problem and submitting it to the constraint solver.

This enhancement has not been included because the problem is not solved as a constraint satisfaction problem – the question that needs to be answered is whether such a solution is acceptable in this context, as our purpose has been to establish the limitations of using a constraint solver. I have not felt this to be appropriate.

4.3.15 An analysis of the kt_md class

For this class, in the case of the *kt_md_p* data sets, the first two problems are trivial and take no time to solve. The third data set goes to modal layer seven and its clauses include

$(\Box(\Box(\neg\Box(\neg\Box(\neg\Box(\neg\Box(p_1))))))) \wedge$
$(\neg\Box(\neg\Box(\neg\Box(\neg\Box(p_1))))) \wedge$
$(\Box(\neg\Box(\neg\Box(p_1 \vee \Box(\neg\Box(\neg\Box(\Box(\neg p_1)))))))) \wedge$
$(\neg\Box(\neg\Box(p_1 \vee \Box(\neg\Box(\neg\Box(\Box(\neg p_1))))))) \wedge$
$(\Box(\neg\Box(\neg\Box(\Box(\neg p_1))))) \wedge$
$\ldots$

It is very difficult to identify any pattern in these clauses and simplification and caching do not make any difference – the clauses contain a random mix of positive and negative box modalities. Table 4.4 shows that other solvers experienced the same problem with this class.

4.4 Transitivity and the S4_KCSP solver

Before adding transitivity to the KT_KCSP prototype, we look at the complexity of the problem and describe how we will solve it.

4.4.1 Basic Issues

To illustrate the issues we are dealing with when the transitivity and reflexivity rules are applied to a formula, we first consider the following two examples.

Example 4.4.1. Consider the modal formula $\Box\Box p$. We take this through several modal layers to illustrate the effect of the application of the reflexive and transitive rules. We apply Lemma 4.3.4, $\Box^n\varphi \to \varphi$, for reflexivity and the axiom, $\Box\varphi \to \Box\Box\varphi$, for transitivity (Definition 2.3.3).

Layer 1:	$\Box\Box p$	
	$\Box\Box\Box p \wedge \Box\Box p \wedge p$	(application of reflexivity and transitivity)
Layer 2:	$\Box\Box p \wedge \Box p$	(modal formula at the next modal layer)
	$\Box\Box\Box p \wedge \Box\Box p \wedge \Box p \wedge p$	(application of reflexivity and transitivity)
Layer 3:	$\Box\Box p \wedge \Box p \wedge p$	(modal formula)
	$\Box\Box\Box p \wedge \Box\Box p \wedge \Box p \wedge p$	(application of reflexivity and transitivity)

The modal formula of Layer 3 repeats at all subsequent modal layers – we have the same looping behavior experienced by the tableau and sequent solvers. ⊣

The following is a more complex example. This example is taken from the *s4_ipc_p* data set. We do not show *all* the branches the constraint solver generates, as many of these loop and it would complicate the example unnecessarily to include them.

Example 4.4.2. Consider the following modal formula which occurs at Layer 1:

Layer 1:

$$\begin{aligned}\varphi = \;& \Box(\neg\Box(\neg\Box p_1 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3))) \wedge \\ & \Box(\neg\Box(\neg\Box p_2 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3))) \wedge \\ & \Box(\neg\Box(\neg\Box p_3 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3)))\end{aligned}$$

To simplify the discussion, we rename these clauses as follows:

$$\begin{aligned}
\psi_1 &= (\neg\Box(\neg\Box p_1 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3)))\\
\psi_2 &= (\neg\Box(\neg\Box p_2 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3)))\\
\psi_3 &= (\neg\Box(\neg\Box p_3 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3)))
\end{aligned}$$

so that the formula can be rewritten as $\varphi = \Box\psi_1 \wedge \Box\psi_2 \wedge \Box\psi_3$. When the reflexivity and transitivity rules are applied to φ, we get

$$\begin{aligned}
\psi_{rt} = \quad & \Box\Box\psi_1 \wedge \Box\psi_1 \wedge \psi_1 \wedge \\
& \Box\Box\psi_2 \wedge \Box\psi_2 \wedge \psi_2 \wedge \qquad (1)\\
& \Box\Box\psi_3 \wedge \Box\psi_3 \wedge \psi_3
\end{aligned}$$

which is passed to the constraint solver.

Layer 2: The constraint solver returns positives modal literals:

$$\alpha_{L2} = \quad \Box\psi_1 \wedge \psi_1 \wedge \Box\psi_2 \wedge \psi_2 \wedge \Box\psi_3 \wedge \psi_3$$

and negative modal literals:

$$\begin{aligned}
\beta_1 &= \quad (\neg\Box p_1 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3))\\
\beta_2 &= \quad (\neg\Box p_2 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3))\\
\beta_3 &= \quad (\neg\Box p_3 \vee (\Box p_1 \wedge \Box p_2 \wedge \Box p_3))
\end{aligned}$$

Note that the β_is originate from the ψ_i clauses of modal layer 1. The constraint solver now processes each $\alpha_{L2} \wedge \neg\beta_i$ in turn.

It begins with $\varphi_2 = \alpha_{L2} \wedge \neg\beta_1 = \alpha_{L2} \wedge \Box p_1 \wedge (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3)$, which we rewrite as $\varphi_2 = \alpha_{L2} \wedge \Box p_1 \wedge \psi_{n1}$ where $\psi_{n1} = (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3)$ (2).

Applying the reflexive and transitive rules to φ_2 gives

$$\psi_{rt} \wedge \Box\Box p_1 \wedge \Box p_1 \wedge p_1 \wedge \psi_{n1} \qquad (3)$$

where ψ_{rt} is defined as in (1) above. This formula is then passed to the constraint solver.

Layer 3: The constraint solver returns positive modal literals:

$$\alpha_{L3} = \alpha_{L2} \wedge \Box p_1 \wedge p_1$$

and negative modal literals β_1, β_2, β_3 as above, and another negative modal literal, β_4, which can have values $\neg p_1$, $\neg p_2$ and $\neg p_3$. The first result will have $\beta_4 = \neg p_1$. If the algorithm backtracks, $\beta_4 = \neg p_2$ and the final backtrack will have $\beta_4 = \neg p_3$.

The constraint solver begins by processing $\varphi_3 = \alpha_{L3} \wedge \neg\beta_1 = \alpha_{L3} \wedge \Box p_1 \wedge (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3)$, which we rewrite as $\varphi_3 = \alpha_{L3} \wedge \Box p_1 \wedge \psi_{n1}$, with ψ_{n1} defined as in (2) above.

Applying the reflexive and transitive rules to φ_3 gives $\psi_{rt} \wedge \Box\Box p_1 \wedge \Box p_1 \wedge p_1 \wedge \psi_{n1}$ – which is a repeat of (3) above. This branch will now loop and is therefore satisfiable (a result we will prove later).

The constraint solver next processes $\varphi_3' = \alpha_{L3} \wedge \neg\beta_2 = \alpha_{L3} \wedge \Box p_2 \wedge (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3)$, which we rewrite as $\varphi_3' = \alpha_{L3} \wedge \Box p_2 \wedge \psi_{n1}$, where ψ_{n1} is defined as in (2) above.

Applying reflexivity and transitivity to φ_3' gives $\psi_{rt} \wedge \Box\Box p_1 \wedge \Box p_1 \wedge p_1 \wedge \Box\Box p_2 \wedge \Box p_2 \wedge p_2 \wedge \psi_{n1}$, which is passed to the constraint solver.

Layer 4: The constraint solver returns positive modal literals:

$$\alpha_{L4} = \alpha_{L2} \wedge \Box p_1 \wedge p_1 \wedge \Box p_2 \wedge p_2$$

and negative modal literals β_1, β_2, β_3 as above, and β_5 which can have values $\neg p_1$, $\neg p_2$ and $\neg p_3$ respectively.

When we process $\alpha_{L4} \wedge \neg\beta_1$ and $\alpha_{L4} \wedge \neg\beta_2$, we loop as already shown.

The constraint solver next processes $\varphi_4 = \alpha_{L4} \wedge \neg\beta_3 = \alpha_{L4} \wedge \Box p_3 \wedge (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3) = \alpha_{L2} \wedge \Box p_1 \wedge p_1 \wedge \Box p_2 \wedge p_2 \wedge \Box p_3 \wedge (\neg\Box p_1 \vee \neg\Box p_2 \vee \neg\Box p_3)$, which is unsatisfiable.

Since we have found an unsatisfiable branch in which there are no choice points to backtrack to, we conclude that the formula at Layer 1 is unsatisfiable. The alternative values for β_4 and β_5 are irrelevant as unsatisfiability was detected before they were processed. $\dashv$

We can see the following from the above example:

- Positive modal literals repeat at each modal layer (α_{L2} was generated from φ and occurs as a positive modal literal from Layer 2 onwards).

- The modal depth of the formula does not change, unlike in the case of reflexivity where it decreased at each modal layer – each α_{L2} has a modal depth of 3 which is the modal depth of the initial formula φ.

- For some of the branches, the modal formula at one modal layer repeats at the next modal layer.

To deal with modal *S4* formulae, we introduce the following lemmas.

Lemma 4.4.3. *For any frame $\mathcal{F} = (W, R)$, R is reflexive and transitive if and only if, for $n > 0$,*

$$\Box^n \varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$$

is valid in the frame, where $\Box^n$ represents n occurrences of $\Box$.

Proof. Suppose we have an arbitrary frame $\mathcal{F}$ in which R is reflexive and transitive. For $n = 1$, the validity of $\Box\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$ follows from a single application of the reflexivity rule $\Box\varphi \rightarrow \varphi$ and the transitivity rule $\Box\varphi \rightarrow \Box\Box\varphi$.

For $n > 1$, we apply the reflexivity rule to $\Box^n\varphi$ to get $\Box^n\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$, which is valid in $\mathcal{F}$. Since $\mathcal{F}$ and n were arbitrarily chosen, $\Box^n\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$ is valid in any frame which is reflexive and transitive.

Conversely, suppose $\Box^n\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$. Setting $n = 1$ gives $\Box\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$. This can be simplified as follows :

$$\begin{aligned} & \Box\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi) \\ \equiv\ & \neg\Box\varphi \vee (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi) \\ \equiv\ & (\neg\Box\varphi \vee \Box\Box\varphi) \wedge (\neg\Box\varphi \vee \Box\varphi) \wedge (\neg\Box\varphi \vee \varphi) \\ \equiv\ & (\Box\varphi \rightarrow \Box\Box\varphi) \wedge True \wedge (\Box\varphi \rightarrow \varphi) \end{aligned}$$

– the transitive and reflexive rules respectively. $\dashv$

To simplify the Lemma which follows, we will refer to $\Box^n\varphi \rightarrow (\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$ as the $S4'$ axiom.

Lemma 4.4.4. *Applying the $S4'$ axiom at each modal layer to each occurrence of $\Box^n\varphi$ is a sound and complete strategy to ensure reflexivity and transitivity of R in the S4_KCSP algorithm.*

Proof. We need to show that the application of the $S4'$ axiom at the each modal layer returns the same result as the application of the T and 4 axioms.

We begin by considering the current modal layer. The application of the $S4'$ axiom to the positive modal literal $\Box^n\varphi$ results in it being replaced with $\varphi_1 = \Box\Box\varphi \wedge \Box\varphi \wedge \varphi$. The application of the T and 4 axioms results in it being replaced with $\varphi_2 = \Box^{n+1}\varphi \wedge \Box^n\varphi \wedge \ldots \wedge \varphi$.

When the constraint solver evaluates a modal formula, it assigns a value of 1 or u to each positive modal literal, as already discussed on page 107. This means that, when φ_1 and φ_2 are processed, they are both effectively reduced to φ. Hence, the constraint solver will return the same result at this modal layer.

At the next modal layer, we have $\Box\varphi \wedge \varphi$ from φ_1 and the application of the $S4'$ axiom gives $\varphi_1' = \Box\Box\varphi \wedge \Box\varphi \wedge \varphi$ once more. We have $\Box^n\varphi \wedge \ldots \wedge \varphi$ from φ_2 and the application of the T and 4 axioms gives $\varphi_2' = \Box^{n+1}\varphi \wedge \Box^n\varphi \wedge \Box^{n-1}\varphi \wedge \ldots \wedge \varphi$ once more. Since $\varphi_1' = \varphi_1$ and $\varphi_2' = \varphi_2$, the same argument as before holds.

The application of both sets of axioms therefore returns the same result.

$\dashv$

Lemma 4.4.5. *Suppose we have a modal formula φ at modal layer n and suppose φ is also the modal formula generated at modal layer n+1. No further processing of this branch is required as it is satisfiable.*

Proof. Suppose we have a modal formula φ occurring both at modal layer n and at modal layer $n+1$. The fact that we have a modal formula at layer $n+1$ means that the formula φ at layer n is propositionally satisfiable. If φ was not propositionally satisfiable, the constraint solver would have backtracked and no formula would have been generated for modal layer $n+1$.

The modal formula φ would similarly be generated at modal layer $n+2$. Since φ is generated by the constraint solver at each successive modal layer, φ is satisfiable in an infinite model. $\dashv$

Note that this lemma is not applicable to modal KT formulae as at each modal layer, the modal depth of the formula is reduced by one – we will therefore not have the same modal formula φ occurring at two successive modal layers.

Now that we have determined the lemma to apply for reflexive and transitive accessibility relations, and have proved that a repeating formula is satisfiable, we can define the S4_KCSP algorithm.

4.4.2 The S4_KCSP algorithm

This algorithm differs from the KT_KCSP_NoCNF algorithm only in that Lemma 4.4.4 is applied to the modal formula at each modal layer instead of Lemma 4.3.4. I did not use the KT_KCSP_CNF algorithm as we now have $(\Box\Box\varphi \wedge \Box\varphi \wedge \varphi)$ replacing each positive modal literal. This is a conjunction of three variables – instead of generating 2^n additional clauses, we would generate 3^n additional clauses if we kept the formulae in conjunctive normal form (refer to Theorem 4.3.7).

Algorithm 8. The S4_KCSP algorithm schema can be represented as follows:

```
function S4_KCSP($\varphi$)
    $\varphi_{neg} = \neg\varphi$;
    $\varphi_{nnf} = to\_nnf(\varphi_{neg})$;
    S4_CSP($\varphi_{nnf}$);
end;

function S4_CSP($\varphi$)                                  // succeeds if $\varphi$ is satisfiable
    $\varphi_{s4} = apply\_reflexivity\_transitivity(\varphi)$;
    $\varphi_{formula} = construct\_formula(\varphi_{s4})$;
    $\varphi_{csp} = to\_csp(\varphi_{formula})$;
    $\mu := csp(\varphi_{csp})$;                            // backtrack if this fails
    $\Theta = \bigwedge\{\alpha : \Box\alpha = 1$ is in $\mu\}$;
    for each $\Box\beta = 0$ in $\mu$ do
        S4_CSP($\Theta \wedge \neg\beta$);                 // backtrack if this fails
end;
```

	n	**p**
s4_branch	8	>
s4_45	2	1
s4_grz	>	>
s4_ipc	8	>
s4_md	8	10
s4_path	9	10
s4_ph	7	4
s4_s5	5	5
s4_t4p	10	13

	0-5	**6-10**	**11-15**	**16-20**	**21**
n	2	5	1	–	1
p	3	1	2	–	3

Table 4.5: Initial results of the S4_KCSP prototype

4.4.3 The initial S4_KCSP prototype

The results obtained by this initial S4_KCSP prototype are listed in Table 4.5.

The results of the *s4_45_n* and *s4_45_p* data sets are particularly bad – the algorithm solves the first *p*-data set in under 10 CPU-seconds, while the second takes more than 100 CPU-seconds to solve (the exponential factor is huge in this case). Note that the *p*-data sets generally return better results than the *n*-data sets.

4.4.4 S4 – Enhancements 6 and 7 – Simplification revisited

When the *s4_45* data sets were examined, we found the following:

Example 4.4.6. Modal formulae can contain clauses such as

$$(p_4 \vee (p_2 \wedge (\Box p_1 \vee \Box p_0 \vee \Box p_5))) \wedge \Box p_0 \wedge \Box p_1 \tag{4.1}$$

For clarity, we have three clauses, the second and third of which are $\Box p_0$ and $\Box p_1$. When Lemma 4.4.4 is applied to this formula, it becomes

$$
\begin{aligned}
&(p_4 \vee (p_2 \wedge \\
&\quad ((\Box\Box p_1 \wedge \Box p_1 \wedge p_1) \vee \\
&\quad (\Box\Box p_0 \wedge \Box p_0 \wedge p_0) \vee \\
&\quad (\Box\Box p_5 \wedge \Box p_5 \wedge p_5)))) \wedge \\
&(\Box\Box p_0 \wedge \Box p_0 \wedge p_0) \wedge \\
&(\Box\Box p_1 \wedge \Box p_1 \wedge p_1)
\end{aligned}
$$

Simplification of such a formula is no longer straightforward, particularly when you bear in mind that this is a *simple* example.

The rules of simplification (Enhancement 1) have so far been applied to unit clauses that are either single propositional literals or single modal literals *after* Lemma 4.4.4 has been applied to the formula. If we were to apply simplification *before* applying Lemma 4.4.4, we would have a simpler formula to deal with.

Applying subsumption to (4.1) and applying it to the modal formula $(\Box p_1 \vee \Box p_0 \vee \Box p_5)$ within the NNF clause gives

$$(p_4 \vee p_2) \wedge \Box p_0 \wedge \Box p_1$$

given that $\Box p_0$ and $\Box p_1$ must be $True$. This is a far simpler formula to deal with. This simplification is verified in Enhancements 6 and 7. ⊣

The enhancement which follows may seem obvious - we are however verifying that simplication *before* the application of Lemma 4.4.4 is feasible.

Enhancement 6. *For each unit modal literal ψ (Definition 2.2.11) in a modal formula φ, we apply the rules of unit subsumption and unit resolution to every other clause containing ψ before Lemma 4.4.4 is applied to φ.*

Justification. We prove unit subsumption and unit resolution separately, as before.

1. Unit subsumption:

   Suppose we have a modal formula $\varphi = \psi \wedge (\psi \vee \psi_1) \wedge \psi_2$ where ψ is a unit modal literal, ψ_1 is an NNF clause and ψ_2 consists of any number of clauses.

   (a) Suppose $\psi = \Box p$. When we apply Lemma 4.4.4 to φ, we get

$$\varphi' = (\Box\Box p \wedge \Box p \wedge p) \wedge ((\Box\Box p \wedge \Box p \wedge p) \vee \psi_1') \wedge \psi_2'$$

where ψ_1' and ψ_2' are the expansions of ψ_1 and ψ_2 respectively.

If we first apply simplification to φ, we have $\Box p \wedge \psi_2$, which becomes

$$\varphi'' = (\Box\Box p \wedge \Box p \wedge p) \wedge \psi_2'$$

after Lemma 4.4.4 has been applied.

Propositional simplification establishes that $\varphi' \equiv \varphi''$.

(b) Suppose $\psi = \neg\Box p$. When we apply Lemma 4.4.4 to φ, we get

$$\varphi' = \neg\Box p \wedge (\neg\Box p \vee \psi_1') \wedge \psi_2'$$

Similarly, if we first apply simplification to φ, we have $\neg\Box p \wedge \psi_2$ which becomes $\varphi'' = \neg\Box p \wedge \psi_2'$ after Lemma 4.4.4 has been applied. Propositional simplification establishes that $\varphi' \equiv \varphi''$.

2. Unit Resolution:

   Suppose we have a modal formula $\varphi = \psi \wedge (\neg\psi \vee \psi_1) \wedge \psi_2$.

   (a) Suppose $\psi = \Box p$. When we apply Lemma 4.4.4 to φ, we get

$$\varphi' = (\Box\Box p \wedge \Box p \wedge p) \wedge (\neg\Box p \vee \psi_1') \wedge \psi_2'$$

If we first apply simplification to φ, we have $\Box p \wedge \psi_1 \wedge \psi_2$, which becomes

$$\varphi'' = (\Box\Box p \wedge \Box p \wedge p) \wedge \psi_1' \wedge \psi_2'$$

after Lemma 4.4.4 has been applied.

Propositional simplification establishes once more that $\varphi' \equiv \varphi''$.

(b) Suppose $\psi = \neg\Box p$. When we apply Lemma 4.4.4 to φ, we get

$$\varphi' = \neg\Box p \wedge ((\Box\Box p \wedge \Box p \wedge p) \vee \psi_1') \wedge \psi_2'$$

If we first apply simplification to φ, we have $\neg\Box p \wedge \psi_1 \wedge \psi_2$, which becomes $\varphi'' = \neg\Box p \wedge \psi_1' \wedge \psi_2'$ after Lemma 4.4.4 has been applied.

Propositional simplification establishes once more that $\varphi' \equiv \varphi''$.

Hence, we can apply simplification using the unit modal literals *before* Lemma 4.4.4 is applied to the modal formula to get the same result. $\dashv$

We extend unit subsumption and unit resolution as follows.

Enhancement 7. *We apply the following two simplification rules to every propositional unit clause and unit modal literal ψ in a modal formula φ:*

1. *Unit subsumption is applied to every other clause containing ψ as follows:*

   A formula $\varphi_1 \wedge (\psi_1 \vee (\psi \wedge \psi_2)) \wedge \psi$, in which φ_1 consists of the conjunction of any number of NNF clauses and ψ_1 and ψ_2 are NNF clauses, is replaced with $\varphi_1 \wedge (\psi_1 \vee \psi_2) \wedge \psi$.

2. *Unit resolution is applied to every other NNF clause containing $\neg\psi$ as follows:*

   A formula $\varphi_1 \wedge (\psi_1 \vee (\neg\psi \wedge \psi_2)) \wedge \psi$, whose variables are defined as in 1. above, is replaced with $\varphi_1 \wedge \psi_1 \wedge \psi$.

Justification. In both cases, when ψ is $False$, the formula is unsatisfiable and its replacement formula is also unsatisfiable. Hence we prove the case where ψ is $True$.

1. Unit subsumption:

   Since ψ is $True$, $(\psi \wedge \psi_2)$ is equivalent to $(True \wedge \psi_2)$ or ψ_2 and we can replace the formula with $\varphi_1 \wedge (\psi_1 \vee \psi_2) \wedge \psi$.

2. Unit resolution:

   Since ψ is $True$, $(\neg\psi \wedge \psi_2)$ is $False$ and so we need to verify the satisfiability of ψ_1 – we can replace the formula with $\varphi_1 \wedge \psi_1 \wedge \psi$.

$\dashv$

We summarize these enhancements as follows:

- Enhancement 6 is applied to single modal literals before the application of Lemma 4.4.4. Recall that it is already being applied after Lemma 4.4.4.

- Enhancement 7 is applied to the modal formulae within an NNF clause before and after the application of Lemma 4.4.4.

Enhancement 6 is obvious with hindsight - this is the sort of discovery which occurs in the process of optimizing an algorithm.

When these enhancements were applied to the *S4_KCSP* solver, the following improvements occurred:

Additional data sets solved
s4_45_n(3)
s4_path_n(10 – 20)
s4_ph_n(8)
s4_s5_n(6)
s4_45_p(2 – 12)
s4_md_p(11 – 12)
s4_path_p(11 – 19)

In this case, the *s4_path* data sets benefited the most from this enhancement.

4.4.5 S4 – Enhancement 8 – Early pruning revisited

The *s4_45_n* data sets were analyzed again as their results were still not good. It was found that, after simplification, at some of the modal layers, the simplified formula now contained $p_0 \wedge \neg p_0$.

The modal formula generated for a new modal layer is already checked to ensure that it does not contain complementary propositional literals; however, so far this test has not been carried out *after* the simplification step.

Enhancement 8. *Once a modal formula has been fully simplified, the formula is checked to ensure that it does not contain complementary literals.*

The justification for this enhancement is self-evident.

The resultant improvements are as follows:

Additional data sets solved
s4_45_n(4 – 14)
s4_md_n(9)

The timings of some of the other classes were improved, but not sufficiently to result in any change in the number of data sets solved.

4.4.6 Final results of the S4_KCSP prototype

The final results from the S4_KCSP solver are listed in Table 4.6.

	n	**p**
s4_branch	8	>
s4_45	14	12
s4_grz	>	>
s4_ipc	8	>
s4_md	9	12
s4_path	20	19
s4_ph	8	4
s4_s5	6	5
s4_t4p	12	13

	0-5	**6-10**	**11-15**	**16-20**	**21**
n	–	5	2	1	1
p	2	–	3	1	3

Table 4.6: Final results of the S4_KCSP prototype

The data sets which have benefited the most from the enhancements are the *n* classes. The results of the s4_45 class have also considerable improved.

4.4.7 The exponential nature of the S4 results

The worst class for the KCSP_S4 prototype is *s4_ph_p*. In this case, we have a very high exponential factor – the third data set takes 0.08 CPU-seconds while the fourth takes 49.94 CPU-seconds, giving an exponential factor of 624.25. Using this factor, we project that the fifth data set will take at least 32,000 CPU-seconds to solve.

	FaCT	**DLP**	**S4_KCSP**	**FaCT**	**DLP**	**S4_KCSP**
	n	**n**	**n**	**p**	**p**	**p**
s4_branch	4	8	8	4	10	>
s4_45	>	>	14	>	>	12
s4_grz	>	>	>	2	9	>
s4_ipc	4	>	8	5	10	>
s4_md	4	>	9	8	3	12
s4_path	1	>	20	2	3	19
s4_ph	4	18	8	5	7	4
s4_s5	2	>	6	>	3	5
s4_t4p	3	>	10	5	>	13

Table 4.7: Results of the FaCT and DLP solvers using the Heuerding / Schwendimann S4 data sets

Hence, in order to improve the results, we would need to make a significant change to the algorithm.

4.4.8 Comparative benchmark results of the S4 data sets

In Table 4.7, we list the results of the TANCS-1998 competition which were obtained for the *S4* data sets for the DLP and FaCT solvers [67] and add the results of the S4_KCSP solver to simplify the comparison process.

We can see from these results that my results are considerably better than the FaCT solver in most cases. However, FaCT was able to solve all *s4_45* data sets, whereas S4_KCSP solved only 14 and 12 of these data sets respectively.

When we compare my results with the DLP results, we find that in general, DLP returns better results for n-data sets, while S4_KCSP returns better results for the p-data sets. However, DLP solved all the *s4_45* data sets and solved significantly more of the *s4_ph* data sets. It is without a doubt the superior solver.

4.5 Final analysis of the constraint logic approach

To summarize what has been achieved, I have been able to add reflexivity and transitivity to the accessibility relation of the modal logics KT and $S4$ and solve formulae of these logics in the constraint logic environment, with reflexivity being the easier implementation. I have developed a prototype that does not require the modal formulae to be in conjunctive normal form and that gives good results.

These results were however not easy to achieve as the analysis process was complex due to the complex nature of the benchmark data. In order to optimize the prototypes, it was necessary to carry out a detailed analysis of the data sets that returned poor results, and then determine in what way they could be optimized. I tried many approaches that did not return good results and were subsequently abandoned.

There were tremendous advantages to using the Heuerding / Schwendimann data sets. Firstly, I knew whether the modal formula was satisfiable or not, which assisted greatly in the debugging process. Secondly, the data sets were comprehensive enough to thoroughly test the prototype – the quality of the data sets ensured correct results. One must bear in mind that when the output is $Valid$ or $Not\ Valid$, it becomes very difficult to analyze erroneous results.

When the timing of the data sets was analyzed, it was found that it was affected by the number of branches in the tree and their depth, the amount of backtracking and the number of choice points in a clause. The enhancements therefore focused on reducing these factors.

One of the strengths of the constraint logic approach is its ability to constrain the domain of the variables in the modal formula. Firstly, in the K_KCSP prototype, by setting the domain of the propositional literals to $\{0,\ 1,\ u\}$, Brand et al. enabled partial assignments to be returned. Secondly, in the enhancements I have introduced, good results were returned by setting the domain of positive propositional unit clauses to 1 and the domain of negative propositional unit clauses to 0 (Enhancement 4 in Section 4.3.9). My idea was taken a step further by setting the domain of propositional literals that occurred either only positively or only negatively to $\{1\}$ and $\{0\}$ respectively and then removing these clauses completely. In this way, the search space was significantly reduced. The tableau and sequent solvers cannot duplicate this approach.

The constraint satisfaction problem is **NP**-complete, whereas the modal satisfiability problem is **PSPACE**-complete. The timings of some of the classes were exponential; however no space problems were experienced during the benchmarking exercise, confirming this difference.

It is interesting to compare the enhancements made to the KCSP prototypes with those applied to the DLP and FaCT solvers (Section 3.1.4). In particular, these techniques include lexical normalization, simplification, storing the satisfiability status of each formula in a cache, using highly optimized data structures and the implementation of backjumping, which involves bypassing nodes that do not contribute to the current clash.

In the KT_KCSP and S4_prototype, extensive simplification was implemented, without which the results would not have been good. The concept of caching was implemented and formulae were stored together with their satisfiability status, which contributed significantly to the good results. Backtracking automatically takes place in a constraint solver – whenever a failure is encountered, it automatically reverts to the *last* choice point, which could be several modal layers away from the point of failure. However, this approach is not as sophisticated as backjumping.

The two enhancements that were *not* implemented were lexical normalization and highly optimized data structures. Lexical normalization, which stores common sub-formulae, facilitates the easy identification of sub-formulae such as φ and $\neg\varphi$. To implement it, well-defined data structures are required and so these are natural candidates for implementation together. The way in which modal formulae in the KCSP prototypes are represented and the way in which they are manipulated is far from optimal – one of the drawbacks of their representation is the cumbersome way in which lists are dealt with in a logic programming language.

Because some of my enhancements were essentially tableau optimizations, the prototypes I have developed are in fact hybrid tableau constraint-based solvers. I have thus been able to benefit from the optimizations of tableau and constraint solvers, which I think makes this an interesting approach.

A shortcoming of the approach I followed was that I limited myself to the Heuerding / Schwendimann data sets, although, as already discussed, there were huge benefits in doing so. If time had not precluded it, it would have been interesting to test the performance of the prototypes against the QBF data sets which were used at the TANCS-2000 conference. In all probability, testing the prototypes against these data sets would have highlighted further areas of improvement. When I was looking for ways to improve my results, I tended to focus on a particular class of data set and then developed an enhancement specific to it. However, I often found that when I implemented the enhancement, the results of other data sets unexpectedly improved. For example, Enhancement 1 was based on an analysis of the *kt_branch* data sets. However, the *kt_grz_n* data sets benefited unexpectedly in that I was able to solve all 21 data sets! I therefore feel that running such a prototype against as many different test sets as possible can only improve it.

Further enhancement of the S4_KCSP prototype was hampered by the representation of the data. Hence, this is an area which can be recommended as a future research project.

4.6 Final remarks

The KT_KCSP and S4_KCSP prototypes returned good results as we could see by comparing their results with those obtained by other solvers. Although these solvers were benchmarked in 2000, because of the exponential nature of the modal satisfiability problem, these results are still meaningful. Although the speed of hardware has improved significantly over the past seven years, in cases where, for instance, only 6 data sets in a class were solved in 2000, we cannot expect more than 8 data sets to now be solved (Section 4.3.12). This class of problem is **PSPACE**-complete which is where it remains. Comparison with these benchmarks has shown that the results of the KT_KCSP and S4_KCSP prototypes are highly competitive, with the KT_KCSP prototype being more so.

The enhancements applied returned good results for the KT_KCSP prototypes. However, in the case of the S4_KCSP prototype, it became much more difficult to enhance, particularly because of the cumbersome data structures – the implementation of Enhancement 7 (Section 4.4.4) was challenging and took a lot of time and effort. The implementation of lexical normalization is almost an impossibility in these prototypes because of the data structures. At this point, the best option would therefore be a rewrite of the prototype, given the requirement for optimized data structures.

As an aside, the prototypes were tested on the then latest HP hardware (July 2007) which had 2 x 2.2 GHz AMD Opteron Processors (dual core). The improvement in the results was insignificant. This suggests that, in order to benefit from this new technology, a concurrent algorithm is required. This was, I felt, an unexpected stumbling block as I was expecting to benefit from using the latest and most powerful hardware.

The results I obtained support the feasibility of this implementation and therefore further enhancements are recommended as a future research project.

Chapter 5

Conclusions and further work

We have looked at how the modal satisfiability problem can be solved in a constraint logic environment. We have extended the solver developed for the modal logic K by adding reflexivity and transitivity to the accessibility relation, which has resulted in two new prototypes.

We first look briefly at an application area of modal logic, this being temporal reasoning and identify this as an area in which the S4_KCSP prototype could be implemented. Temporal reasoning problems occur in a vast number of application areas, as time is a dimension which is an integral part of our lives. Scheduling by its very nature has an embedded time dimension, as events need to occur in a certain order. Many application areas that are addressed in computer science need to make use of time-based information for reporting, calculations, projections and so on. Banking applications, for example, are an area that we all are familiar with and that have a well defined time dimension. In the medical field, vast amounts of time-oriented clinical data need to be collected and analyzed to identify trends in disease, to monitor patients and to record responses to new drugs. Such data can be used to determine a diagnosis and to prescribe therapy and such an analysis is impossible without a time dimension [116]. An overview of temporal logic, its application areas and the temporal constraint satisfaction problem has already been provided in Section 2.8.3 – a further discussion follows which relates to the prototypes we have developed.

We then look very briefly at other application areas in which these prototypes could be used. This is however a brief and cursory investigation and serves to present some idea of further research areas.

This is followed by looking at ways in which the prototypes can be further enhanced.

5.1 Modal temporal logic

We have seen in Section 2.8.3 that the temporal constraint satisfaction problem (TCSP) has been solved using Allen's interval algebra. It makes use of a temporal representation based on time intervals and expresses these time intervals using a set of 13 interval relations. We also saw that a translation of this interval algebra into a temporal modal logic that has two modalities has been defined [41]. Since much research has been carried out into solving TCSPs using the interval algebra, this is a worthwhile area to take further. Modal temporal logic is able to deal with more powerful temporal problems than the interval algebra and is more expressive than the interval algebra, making this a useful extension. It also requires the accessibility relation to be transitive, which we have already implemented in the S4_KCSP prototype.

In order to understand some of the complexities of implementing this translation, we repeat the following example.

Example 5.1.1. The interval algebra problem of Example 2.8.3 is translated into the modal temporal logic, defined in Algorithm 1, as follows:

The path constraints

p_1 *overlaps* p_2; p_1 *starts* p_2

p_2 *meets* p_3

are translated into

$$((\Diamond(p_1 \wedge p_2) \wedge \Diamond(p_1 \wedge \neg p_2 \wedge \Diamond_F p_2) \wedge \Diamond(p_2 \wedge \neg p_1 \wedge \Diamond_P p_1)) \vee$$
$$(\Box(\neg p_1 \vee p_2) \wedge \Diamond(p_2 \wedge \neg p_1) \wedge \Box(\neg p_1 \vee \neg p_2 \vee \Box_P(\neg p_2 \vee p_1)))) \wedge$$
$$\Box((\neg p_2 \vee \neg p_3) \wedge (\neg p_2 \vee \Diamond_F p_3)) \wedge \neg\Diamond(\neg p_2 \wedge \neg p_3 \wedge \Diamond_P p_2 \wedge \Diamond_F p_3)$$

⊣

We observe the following from the temporal modal formula above, particularly with reference to the S4_KCSP prototype:

- The formula does not have the intuitive content of the interval algebra representation.

- The S4_KCSP prototype returns a result of $Valid$ or $Not\ Valid$. If the formula is $Valid$, we are not going to know anything about the relation between p_1 and p_3. In Example 2.8.3, the relation between p_1 and p_3 was established to be *before*. The S4_KCSP prototype will not however be able to return this sort of result.

- The S4_KCSP prototype will first process the clauses equivalent to {*overlaps*, *meets*} and then the clauses equivalent to {*starts*, *meets*}. If we had three relations on the first edge and three on the second edge, it would process 6 sets of relations. The path consistency algorithm on the other hand calculates the relations between p_1 and p_3 based on the relations between p_1 and p_2 and those between p_2 and p_3 – a totally different approach. However, it has the disadvantage that the composition of these relations can result in a more complex set of relations. For example, if we have the relation *before* between p_1 and p_2 and *finishes* between p_2 and p_3, the composition operation will return {*before, during, overlaps, meets, starts*} as the set of relations between p_1 and p_3.

- We now have modalities $\Box$, $\Box_P$, $\Box_F$, $\Diamond$, $\Diamond_P$ and $\Diamond_F$ – the prototype will need to be modified to deal with more than one modality.

It is clear that it will not be a trivial matter to modify the S4_KCSP prototype to deal with these temporal modal formulae. However, this is a potential future research area which will be worth pursuing further.

5.2 Additional application areas

We have not so far considered the implementation of the prototypes in the area of description logic. The majority of tableau solvers have been developed specifically to deal with description logic and so it would be a worthwhile exercise to investigate this further. Because of the correspondence between modal logic and description logic, as formalized in Definition 2.8.2, this is a relatively straightforward modification. FaCT, DLP and RACE all provide for the description logic $\mathcal{ALC}$ and support transitive roles. FaCT and RACE handle qualified number restrictions and graded modalities. RACE also provides for ABoxes, which correspond to restricted use of nominals in modal logics [87]. Number restrictions, which are described on page 25, can easily be implemented in both prototypes.

The modal logic $S5$ is used to reason about knowledge and belief and can be used to reason about multi-agent systems. This is an important research area, particularly because of current developments on the Web. The S4_KCSP prototype could be extended to deal with it by adding the symmetric axiom, B (Definition 2.3.3).

5.3 Further areas of improvement to the prototypes

Further enhancement to the S4_KCSP prototype was hampered by the complexity and inflexibility of its data structures. Furthermore, the original K_KCSP solver has been extensively modified in a somewhat ad-hoc manner. Hence, before any of the suggested application areas are looked into, it is necessary to either re-design the prototypes and optimize the data structures, or implement the algorithm in an alternate constraint-based system. The *SAT solver, discussed in Section 3.4.3, makes use of a commercially available library of data types that provides highly optimized data structures. This approach significantly reduced their development cycle. A possible alternative to a full re-development of the prototypes in a logic programming environment would be to use a product such as ILOG CP, which is commercially available constraint-based optimization software. Further details of ILOG are available on their web-site [1]. A useful paper which provides an introduction to the development of ILOG is [108].

The pros and cons of these two options would need to be carefully evaluated and considered.

One area which lends itself to further optimization in both the KT_KCSP_NoCNF and S4_KCSP prototypes is the following. Recall that the input modal formula is converted into NNF and then a formula is constructed which is fed into the constraint solver (Definition 4.3.9 and Algorithm 7). If no solution is found, the algorithm backtracks and the formula is reconstructed. The initial construction selects a propositional literal or a modal atom from each NNF clause whenever possible and only if neither is available does it select an NNF formula. However, no selection criteria were applied to these NNF formulae. This means that an NNF formula of high complexity might be selected which would seriously increase the search space.

5.4 Final remarks

In this dissertation, we identified the modal satisfiability problem as an area of interest. We began by looking at its complexity and found that, in the worst case, these problems are **NSPACE**-complete.

We carried out an in-depth survey of the most successful solvers that have been developed to date. We found that the majority of solvers were based on tableau systems, although the Logics Workbench LWB, which has been based on sequent systems, and the MSPASS solver, which has been based on the translation of modal logics into first-order logic, are also powerful solvers. The solvers LWB, FaCT, DLP, RACE, RACER and MSPASS all support a transitive accessibility relation. Of these, FaCT, DLP, RACE and RACER support the description logic $\mathcal{ALC}$, while MSPASS supports the description logic $\mathcal{ALB}$ [97]. The solver *SAT, which is based on the translation of modal logic into layered propositional satisfiability problems does not support transitivity – it only supports reflexivity [119]. The KCSP solver, which is based on the translation of modal logic into layered constraint satisfaction problems only supports the modal logic K.

The translation into first-order logic differs from the other solvers in two ways. Firstly, it seems to have been motivated primarily by the desire to understand why modal logic is decidable, whereas first-order logic is not. Secondly, it is the only solver out of those discussed above which does not build a modal tree. However, the associated MSPASS solver has turned out to be competitive and for some data sets of the TANCS-2000 conference, it returned the best results.

The tableau-based solvers are highly competitive and to achieve their good results, it was necessary to apply extensive optimizations. Firstly, the modal formula is reduced as much as possible by applying the rules of unit subsumption and unit resolution, thereby minimizing the branches of the modal tree. When a branch is unsatisfiable, the solver backjumps over as many nodes as possible, thereby avoiding unnecessary processing. The results of formulae already processed are stored, again avoiding unnecessary reprocessing. Finally, highly optimized data structures are used.

We focused on the translation of modal formulae into a layered set of constraint satisfaction problems. This choice was made because this is a new approach and has only been applied to the modal logic K. The obvious extensions were to add reflexive and transitive accessibility relations, as this is the approach followed by other solvers. Adding reflexivity to the existing K_KCSP solver did not initially return good results. Various enhancements were implemented, some of which improved the results significantly and others which resulted in minor improvements. For example, Enhancement 2 resulted in only 5 additional data sets being solved by the KT_KCSP_CNF prototype and only 1 by the KT_KCSP_NoCNF prototype, although the timings in general marginally improved. Enhancement 5 on the other hand resulted in an additional 7 *classes* of data sets being fully solved by the KT_KCSP_CNF prototype and an additional 8 *classes* by the KT_KCSP_NoCNF prototype!

The full enhancement process included a number of the techniques applied by tableau solvers. It was possible to implement these because of the Prolog-like language embedded in ECL^iPS^e. Hence, we have a hybrid tableau constraint-based solver which includes the strengths of both.

Adding a transitive accessibility relation was more complex, mainly because of the way in which the data is represented. The caching approach was further refined – the algorithm is an improvement on that implemented for the KT_KCSP solver. Simplification was taken to its limits with subsumption and resolution now being applied within modal formulae in an NNF clause, and simplification was applied before transitivity and reflexivity. Again, neither of these were included in the code of the KT_KCSP solver.

Ultimately, we have good results in a prototype which needs to be rewritten in order to optimize its data structures. However, the efficiency or otherwise of the code plays a very small role in the timing of the results – it is the size of the modal tree which has the greatest effect. Reducing the number of branches is far more important than optimizing the code – improvements in performance were always attributable to pruning the search space and this is the area which must be focused on. Optimizing the data structures will enable further enhancements to be easily implemented. An example of such an enhancement is lexical normalization which enables the identification of φ and $\neg\varphi$.

We have identified two definite areas in which the prototypes can be deployed, these being a simple temporal modal logic into which formulae of the interval algebra can be converted, and the description logic $\mathcal{ALC}$ which has been included in most of the other solvers.

In conclusion, the good results obtained with the KT_KCSP and S4_KCSP prototypes indicate that this research area needs to be further considered.

Bibliography

[1] ILOG CP. Available at http://www.ilog.com/products/cp/, retrieved September 2007.

[2] The ECLiPSe Constraint Programming System. Available at http://eclipse.crosscoreop.com/, retrieved September 2007.

[3] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] K. R. Apt and S. Brand. Schedulers for rule-based constraint programming. In *Proceedings of the 2003 ACM Symposium on Applied Computing, Melbourne, FL*, pages 14–21. ACM Press, 2003.

[5] K. R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, USA, 2007.

[6] C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-Based Heuristics in Modal Theorem Proving. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany*, pages 199–203. IOS Press, 2000.

[7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[8] F. Baader, J. Hladik, C. Lutz, and F. Wolter. From Tableaux to Automata for Description Logics. *Fundamenta Informaticae*, 57(2-4):247–279, 2003.

[9] S. Baase and A. van Gelder. *Computer Algorithms: Introduction to Design and Analysis (2nd ed.)*. Addison-Wesley Longman, Inc., Boston, MA, 1999.

[10] P. Balsiger and A. Heuerding. Comparison of Theorem Provers for Modal Logics - Introduction and Summary. In H. C. de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oisterwijk, The Netherlands*, volume 1397 of *Lecture Notes in Computer Science*, pages 25–26. Springer-Verlag, London, UK, 1998.

[11] P. Balsiger, A. Heuerding, and S. Schwendimann. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, 2000.

[12] B. Beckert and R. Góre. Free-variable Tableaux for Propositional Modal Logics. In D. Galmiche, editor, *Proceedings of the International Conference on*

Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France, volume 1227 of *Lecture Notes In Computer Science*, pages 91–106. Springer-Verlag, London, UK, 1997.

[13] B. Beckert and R. Hähnle. Analytic Tableaux. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, pages 11–41. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[14] M. Ben-Ari. *Mathematical Logic for Computer Science.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, Hempel Hempstead, UK, 1993.

[15] A. Biere and C. P. Gomes, editors. *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA*, volume 4121 of *Lecture Notes in Computer Science.* Springer-Verlag, 2006.

[16] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, Cambridge, UK, 2001.

[17] S. Brand, R. Gennari, and M. de Rijke. Constraint Programming for Modeling and Solving Modal Satisfiability. In F. Rossi, editor, *Proceedings of Principles and Practice of Constraint Programming, Kinsale, Ireland*, volume 2833 of *Lecture Notes in Computer Science*, pages 795–800. Springer-Verlag, London, UK, 2003.

[18] S. Brand, R. Gennari, and M. de Rijke. Constraint Methods for Modal Satisfiability. In K. R. Apt, F. Fages, F. Rossi, P. Szeredi, and J. Váncza, editors, *Recent Advances in Constraints, International Workshop on Constraint Solving and Constraint Logic Programming, Budapest, Hungary*, volume 3010 of *Lecture Notes in Computer Science*, pages 66–86. Springer-Verlag, Berlin, Germany, 2004.

[19] I. Bratko. *Prolog (3rd ed.): Programming for Artificial Intelligence.* Addison-Wesley Longman, Inc., Boston, MA, 2001.

[20] S. Cerrito and M. Mayer. A Polynomial Translation of S4 into T and Contraction-Free Tableau for S4. *Logic Journal of the IGPL*, 5(2):287–300, 1997.

[21] B. F. Chellas. *Modal Logic : An Introduction.* Cambridge University Press, Cambridge, UK, 1980.

[22] L. Chittaro and A. Montanari. Temporal Representation and Reasoning in Artificial Intelligence: Issues and Approaches. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):47–106, 2000.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[25] J. Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, 1990.

[26] S. Cook. The Complexity of Theorem Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio*, pages 151–158. ACM Press, New York, USA, 1971.

[27] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[28] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):201–215, 1960.

[29] H. de Nivelle and M. de Rijke. Deciding the Guarded Fragments by Resolution. *Journal of Symbolic Computation*, 35(1):21–58, 2003.

[30] H. de Nivelle, R. A. Schmidt, and U. Hustadt. Resolution-Based Methods for Modal Logics. *Logic Journal of the IGPL*, 8(3):265–292, 2000.

[31] L. del Cerro, D. Fauthoux, O. Gasquet, A. Herzig, D. Longin, and F. Massacci. Lotrec: The Generic Tableau Prover for Modal and Description Logics. In R. Góre, A. Leitsch, and T. Nipkow, editors, *Proceedings of the 1st International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *Lecture Notes In Computer Science*, pages 453–458. Springer-Verlag, London, UK, 2001.

[32] S. Demri. Uniform and Non Uniform Strategies for Tableaux Calculi for Modal Logics. *Journal of Applied Non-Classical Logics*, 5(1):77–98, 1995.

[33] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 693–702. ACM Press, New York, USA, 1988.

[34] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. MIT Press, 1990.

[35] R. Epstein and W. Carnielli. *Computability (2nd ed.).* Wadsworth/Thomson Learning, Belmont, CA, 2000.

[36] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, MA, 1995.

[37] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library.* Kluwer Academic Publishers, Boston, MA, 1983.

[38] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends, Châtillon-sur-Seine, France*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, 1995.

[39] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.

[40] T. Frühwirth, A. Herold, V. Küchenhoff, T. le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint Logic Programming - An Informal Introduction. In G. Comyn, N. E. Fuchs, and M. Ratcliffe, editors, *Logic Programming in Action, Zurich, Switzerland*, volume 636 of *Lecture notes in Computer Science*, pages 3–25. Springer-Verlag, 1992.

[41] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-dimensional Modal Logics: Theory and Applications*, volume 148 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science, Amsterdam, The Netherlands, 2003.

[42] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. SAT vs Translation Based Decision Procedures for Modal Logics: A Comparative Evaluation. *Journal of Applied Non-Classical Logics*, 10(2):145–172, 2000.

[43] E. Giunchiglia and A. Tacchella. System Description: *SAT A Platform for the Development of Modal Decision Procedures. In D. A. McAllester, editor, *International Conference on Automated Deduction, Pittsburgh, PA*, volume 1831 of *Lecture Notes in Computer Science*, pages 291–296. Springer-Verlag, 2000.

[44] F. Giunchiglia, M. Roveri, and R. Sebastiani. A New Method for Testing Decision Procedures in Modal and Terminological Logics. In L. Padgham, E. Franconi, M. Gehrke, D. L. McGuinness, and P. F. Patel-Schneider, editors, *Proceedings of the 1996 International Workshop on Description Logics, Cambridge, MA*, volume WS-96-05 of *AAAI Technical Report*, pages 119–123. AAAI Press, 1996.

[45] F. Giunchiglia and R. Sebastiani. Building Decision Procedures for Modal Logics from Propositional Decision Procedure - The Case Study of Modal K. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ*, volume 1104 of *Lecture Notes in Computer Science*, pages 583–597. Springer, London, UK, 1996.

[46] F. Giunchiglia and R. Sebastiani. Building Decision Procedures for Modal Logics from Propositional Decision Procedures. The Case Study of Modal K(m). *Information and Computation*, 162(1-2):158–178, 2000.

[47] R. Goldblatt. *Logics of Time and Computation (2nd ed.).* CSLI Publications, Stanford, CA, 1992.

[48] R. Goldblatt. Mathematical Modal Logic - a View of its Evolution. *Journal of Applied Logic*, 1(5–6):309–392, 2003.

[49] R. Góre. Tableau Methods for Modal and Temporal Logics. In M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Kluwer Academic Publishers, 1999.

[50] V. Haarslev and R. Möller. RACE System Description. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. F. Patel-Schneider, editors, *Proceedings of the 1999 International Workshop on Description Logics, Linköping, Sweden*, volume 22, pages 130–132. 1999.

[51] V. Haarslev and R. Möller. Consistency Testing: The RACE Experience. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Computer Science*, pages 57–61. CEUR-WS.org, London, UK, 2000.

[52] V. Haarslev and R. Möller. RACER System Description. In R. Gor, A. Leitsch, and T. Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning, Siena,Italy*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer-Verlag, London, UK, 2001.

[53] J. Y. Halpern. Reasoning about Knowledge: A Survey. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (Volume 4): Epistemic and Temporal Reasoning*, pages 1–34. Oxford University Press, Oxford, UK, 1995.

[54] J. Y. Halpern. The Effect of Bounding the Number of Primitive Propositions and the Depth of Nesting on the Complexity of Modal Logic. *Artificial Intelligence*, 75(2):361–372, 1995.

[55] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada*, pages 50–61. ACM Press, New York, USA, 1984.

[56] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54(3):319–379, 1992.

[57] E. Hemaspaandra. Modal Satisfiability is in Deterministic Linear Space. In P. Clote and H. Schwichtenberg, editors, *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic, Fischbachau, Germany*, volume 1862 of *Lecture Notes in Computer Science*, pages 332–342. Springer-Verlag London, UK, 2000.

[58] A. Heuerding, G. Jäger, S. Schwendimann, and M. Seyfried. Propositional Logics on the Computer. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Schloß Rheinfels, Germany*, volume 918 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, London, UK, 1995.

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

[60] A. Heuerding and S. Schwendimann. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *Technical Report IAM-96-015*, 1996.

[61] A. Heuerding, M. Seyfried, and H. Zimmermann. Efficient Loop-Check for Backward Proof Search in Some Non-classical Propositional Logics. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Terrasini, Italy*, volume 1071 of *Lecture Notes in Computer Science*, pages 210–225. Springer-Verlag, London, UK, 1996.

[62] J. Hintikka. *Knowledge and Belief*. Cornell University Press, New York, USA, 1962.

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

[64] I. Horrocks. Benchmark Analysis with FaCT. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Computer Science*, pages 62–66. Springer-Verlag, London, UK, 2000.

[65] I. Horrocks and P. F. Patel-Schneider. Optimizing Description Logic Subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[66] I. Horrocks, P. F. Patel-Schneider, and R. Sebastiani. An Analysis of Empirical Testing for Modal Decision Procedures. *Logic Journal of the IGPL*, 8(3):293–323, 2000.

[67] I. Horrocks and P.F Patel-Schneider. FaCT and DLP: Automated reasoning with analytic tableaux and related methods. In H. C. de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oisterwijk, The Netherlands*, volume 1397

of *Lecture Notes in Computer Science*, pages 27–30. Springer-Verlag, London, UK, 1998.

[68] J. Hudelmaier. An O(n log n)-Space Decision Procedure for Intuitionistic Propositional Logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.

[69] U. Hustadt and R.A.Schmidt. On Evaluating Decision Procedures for Modal Logic. In M. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, Nagoya, Japan*, volume 1, pages 202–209. Morgan Kaufmann Inc., 1997.

[70] U. Hustadt and R. A. Schmidt. An Empirical Analysis of Modal Theorem Provers. *Journal of Applied Non-Classical Logics*, 9(4):479–522, 1999.

[71] U. Hustadt and R. A. Schmidt. MSPASS: Modal Reasoning by Translation and First-Order Resolution. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Aritifical Intelligence*, pages 67–71. Springer-Verlag, London, UK, 2000.

[72] U. Hustadt, R. A. Schmidt, and C. Weidenbach. Optimised Functional Translation and Resolution. In H. C. de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oisterwijk, The Netherlands*, volume 1397 of *Lecture Notes in Computer Science*, pages 36–37. Springer-Verlag, London, UK, 1998.

[73] M. Huth and M. Ryan. *Logic in Computer Science: Modeling and Reasoning about Systems.* Cambridge University Press, New York, USA, 2000.

[74] G. Jaeger, P. Balsiger, A. Heuerding, and S. Schwendimann. K, KT, S4 test data sets. Available at http://www.iam.unibe.ch/∼lwb/benchmarks/benchmarks.html, retrieved September 2007.

[75] G. Jaeger, P. Balsiger, A. Heuerding, and S. Schwendimann. LWB software. Available at http://www.lwb.unibe.ch/, retrieved September 2007.

[76] J. Jaffar, P. J. Stuckey, S. Michaylov, and R. H. C. Yap. An abstract machine for CLP(R). *ACM SIGPLAN Notices*, 27(7):128–139, 1992.

[77] J. A. W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, University of California, Los Angeles, CA, 1968.

[78] S. Kripke. Semantic considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94. 1963.

[79] P. B. Ladkin and A. Reinefeld. Fast Algebraic Methods for Interval Constraint Problems. *Annals of Mathematics and Artificial Intelligence*, 19(3-4):383–411, 1997.

[80] R. E. Ladner. The Computational Complexity of Provability in Systems of Modal Propositional Logic. *SIAM Journal on Computing*, 6(3):467–480, 1977.

[81] M. Lagoudakis and M. Littman. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing, Boston, MA*, volume 9. Elsevier Science, 2001.

[82] J. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In *Foundations of Logic and Functional Programming, Trento, Italy*, volume 306 of *Lecture Notes In Computer Science*, pages 67–113. Springer-Verlag, New York, USA, 1988.

[83] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[84] F. Massacci. Strongly Analytic Tableaux for Normal Modal Logic. In A. Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction, Nancy, France*, number 814 in Lecture Notes In Computer Science, pages 723–737. Springer-Verlag, New York, USA, 1994.

[85] F. Massacci. Design and Results of the Tableaux-99 Non-classical (Modal) Systems Comparison. In N. V. Murray, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Saratoga Springs, NY*, volume 1617 of *Lecture Notes in Computer Science*, pages 14–18. Springer-Verlag, London, UK, 1999.

[86] F. Massacci. Single Step Tableaux for Modal Logics. *Journal of Automated Reasoning*, 24(3):319–364, 2000.

[87] F. Massacci and F. M. Donini. Design and Results of TANCS-2000 Non-classical (Modal) Systems Comparison. In R. Dyckhoff, editor, *Proceedings of International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Computer Science*, pages 52–56. Springer-Verlag, London, UK, 2000.

[88] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87(1-2):343–385, 1996.

[89] G. Mints. Gentzen-type systems and resolution rules. Part I. Propositional logic. In P. Martin-Löf and G. Mints, editors, *Proceedings of the International Conference on Computer Logic, Tallinn, USSR*, volume 417 of *Lecture Notes in Computer Science*, pages 198–231. Springer-Verlag, London, UK, 1990.

[90] D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and Easy Distributions of SAT Problems. In W. R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA*, pages 459–465. MIT Press, 1992.

[91] B. Nebel. Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the Efficiency of Using the ORD-Horn Class. In W. Wahlster, editor, *12th European Conference on Artificial Intelligence, Budapest, Hungary*, pages 38–42. John Wiley and Sons, Chichester, UK, 1996.

[92] B. Nebel and H. Bürckert. Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.

[93] A. Nerode and R. A. Shore. *Logic for Applications*. Springer-Verlag, New York, USA, 1997.

[94] L. A. Nguyen. A New Space Bound for the Modal Logics K4, KD4 and S4. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science, Szklarska, Poland*, volume 1672 of *Lecture Notes in Computer Science*, pages 321–331. Springer-Verlag, London, UK, 1999.

[95] A. Nonnengart. First-Order Modal Logic Theorem Proving and Functional Simulation. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambéry, France*, volume 1, pages 80–85. Morgan Kaufmann Inc., 1993.

[96] H. Ohlbach. Translation Methods for Non-Classical Logics - an Overview. *Bulletin of the IGPL*, 1(1):69–90, 1993.

[97] H. Ohlbach, A. Nonnengart, M. de Rijke, and D. Gabbay. Encoding Two-Valued Non-Classical Logics in Classical Logic. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1403–1486. Elsevier Science, Amsterdam, The Netherlands, 2001.

[98] H. Ohlbach and R. A. Schmidt. Functional Translation and Second-Order Frame Properties for Modal Logics. *Journal of Logic and Computation*, 7(5):581–603, 1997.

[99] J. Otten. ileanTAP: An Intuitionistic Theorem Prover. In D. Galmiche, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, volume 1227 of *Lecture Notes in Computer Science*, pages 307–312. Springer-Verlag, London, UK, 1997.

[100] J. Otten and C. Kreitz. A Connection Based Proof Method for Intuitionistic Logic. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Schloß Rheinfels, Germany*, volume 918 of *Lecture Notes in Computer Science*, pages 122–137. Springer-Verlag, London, UK, 1995.

[101] G. Pan, U. Sattler, and M. Vardi. BDD-Based Decision procedures for the Modal Logic K. In B. Konev, R. Schmidt, and S. Schulz, editors, *Journal of Applied Non-Classical Logics*, volume 16, pages 169–208. Elsevier Science, 2006.

[102] G. Pan, U. Sattler, and M. Y. Vardi. BDD-Based Decision Procedures for K. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction, Copenhagen, Denmark*, volume 2392 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, London, UK, 2002.

[103] P. F. Patel-Schneider. TANCS-2000 Results for DLP. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Computer Science*, pages 72–76. Springer-Verlag, London, UK, 2000.

[104] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, Providence, RI*, pages 46–67. IEEE Computer Society, 1977.

[105] A. N. Prior. *Time and Modality.* Oxford University Press, Oxford, UK, 1957.

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

[107] A. N. Prior. *Papers on Time and Tense.* Oxford University Press, Oxford, UK, 1969.

[108] J. F. Puget. A C++ Implementation of CLP. In *Proceedings of the 2nd Singapore International Conference on Intelligent Systems, Singapore.* 1994.

[109] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science, New York, USA, 2006.

[110] K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia*, pages 466–471. Morgan Kaufmann Inc., 1991.

[111] R. A. Schmidt. Decidability by Resolution for many Modal Logics. *Journal of Automated Reasoning*, 22(4):379–396, 1999.

[112] R. A. Schmidt and D. Tishkovsky. Multi-agent Logics of Dynamic Belief and Knowledge. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 38–49. Springer-Verlag, London, UK, 2002.

[113] M. Schmidt-Schauß and G. Smolka. Attributive Concept Descriptions with Complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[114] E. Schwalb and R. Dechter. Processing Disjunctions in Temporal Constraint Networks. *Artificial Intelligence*, 93(1-2):29–61, 1997.

[115] E. Schwalb and L. Vila. Temporal Constraints: A Survey. *Constraints*, 3(2-3):129–149, 1998.

[116] Y. Shahar. Dimension of Time in Illness: An Objective View. *Annals of Internal Medicine*, 132(1):45–53, 2000.

[117] H. Simonis and M. Dincbas. Chapter 15: Propositional Calculus Problems in CHIP. In F. Benhamou and A. Colmerauer, editor, *Constraint Logic Programming: Selected Research*, pages 269–285. MIT Press, Cambridge, MA, 1993.

[118] A. Szalas. Temporal logic of programs: a standard approach. In L. Bolc and A. Szalas, editors, *Time and logic: a computational approach*, pages 1–50. UCL Press Ltd., London, UK, 1995.

[119] A. Tacchella. Evaluating *SAT on TANCS 2000 Benchmarks. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, St. Andrews, Scotland*, volume 1847 of *Lecture Notes in Computer Science*, pages 77–81. Springer-Verlag, London, UK, 2000.

[120] J. Thornton, M. Beaumont, A. Sattar, and M. Maher. Applying Local Search to Temporal Reasoning. In *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning, Manchester, UK*, page 94. IEEE Computer Society, Washington, DC, 2002.

[121] A. Turing. Computability and λ-Definability. *Journal of Symbolic Logic*, 2, 1937.

[122] P. van Beek. Reasoning about Qualitative Temporal Information. *Artificial Intelligence*, 58(1-3):297–326, 1992.

[123] P. van Beek and D. W. Manchak. The Design and Experimental Analysis of Algorithms for Temporal Reasoning. *Journal of Artificial Intelligence Research*, 4:1–18, 1996.

[124] J. van Benthem. Temporal patterns and modal structure. *Logic Journal of the IGPL*, 7(1):7–26, 1999.

[125] P. van Hentenryck and V. A. Saraswat. Strategic Directions in Constraint Programming. *ACM Computing Surveys*, 28(4):701–726, 1996.

[126] P. van Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence*, 58(1-3), 1992.

[127] M. Vardi. Why is Modal Logic so Robustly Decidable? In P. G. Kolaitis N. Immerman, editor, *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop, Princeton University*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–184. American Mathematical Society, 1996.

[128] M. G. Wallace. Survey: Practical Applications of Constraint Programming. *Constraints Journal*, 1(1), 1996.

[129] M. G. Wallace. Constraint Logic Programming. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 2002.

[130] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. *ICL Systems Journal*, 12(1):159–200, 1997.

[131] L. A. Wallen. Matrix Proof Methods for Modal Logics. In J. Dermott, editor, *10th International Joint Conference on Artificial Intelligence, Milan, Italy*, pages 917–923. Morgan Kaufmann Inc., 1987.

[132] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER version 0.42. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ*, volume 1104, pages 141–145. Springer-Verlag, London, UK, 1996.

[133] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems A Modern Approach to Distributed Artificial Intelligence*, pages 27–77. MIT Press, 2000.

[134] M. Zakharyaschev, K. Segerberg, M. de Rijke, and H. Wansing. The Origins of Modern Modal Logic. In M. Zakharyaschev, K. Segerberg, M. de Rijke, and H. Wansing, editors, *Advances in Modal Logic 2, Uppsala, Sweden*. CSLI Publications, 1998.

[135] H. Zhang. SATO: an Efficient Propositional Prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction, North Queensland, Australia*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275. Springer-Verlag, London, UK, 1997.

Index