Automated Reasoning for Artificial Intelligence

A SAT-based theorem prover for modal logic K

Darren Lawton

Supervisor: Professor Rajeev Goré

College of Engineering and Computer Science
The Australian National University

This report is submitted for
COMP8755: Individual Computing Project

May 2018

"When there are disputes among persons, we can simply say: Let us calculate, without further ado, to see who is right."

Gottfried Leibniz, 1685

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Darren Lawton
May 2018

Acknowledgements

I would like to thank Professor Rajeev Goré for allowing me the opportunity to work on this project and his patience throughout the entire process. He was always available to assist with any questions or issues that were encountered, and proactively sort ways to try and improve the results we ultimately obtained. I would also like to thank my family and friends for all the support throughout my studies at ANU.

Abstract

We present a sound and complete method for using a SAT-solver to implement an unlabelled tableau procedure for automated reasoning in modal logic K. At a fundamental level, we distil a modal problem into to a boolean satisfiability problem with additional theory for the remaining modal aspects. Furthermore, we use a modal clausal form to represent the problem.

In benchmarking the performance of our theorem prover to existing methods, we note poor relative performance. This is largely due to redundant exploration of the tableau tree. Our results indicate that using a SAT solver with an unlabelled tableau system may potentially not be an effective approach for searching a tableau tree in modal logic K.

Despite the poor performance, there still remains potential extensions to our research. Particularly in extending our theorem prover to incorporate tableau calculi for multimodal logics of belief.

Table of contents

Chapter 1

Introduction

Given an arbitrary statement, there are potentially many reasons as to why we might assign truth to it. It could be that we were told it is true by a friend, or that we have conducted rigorous experimentation that led us to this conclusion. Regardless of our method, if the reasoning is devoid of a logical underpinning, it is not infallible since assumptions can be left unjustified and inferences disputed [8]. Formal logic mitigates these limitations by defining a syntax to abstract away context, and providing a calculus to guide deduction.

Logical reasoning does not reveal new insights into a domain. However, it does force one to consider necessary relationships between facts, and highlights instances where possibly undesired states can enter into these relationships [8].

In a practical sense, we can distil a problem into a conclusion and set of assumptions. The conclusion represents a contention, whilst the assumptions represent the knowledge base of the problem domain. Logical reasoning solves the problem by proving that the conclusion can be inferred from the given assumptions by systematically applying the rules of deduction [20]. So naturally, given the syntactic nature of our inference rules, we program computers to derive these inferences for us.

1.1 Automated reasoning

An automated theorem prover is an algorithm that uses a set of logical deduction rules, of a given logic, so as to prove or verify the validity of a proposition. A deductive argument is said to be sound if and only if it can be shown that the conclusion is logically entailed within premises of the argument. If the logical deductions of the system are then perceived as being a formalisation of intended meaning (semantics), then the theorem prover becomes an algorithm of reasoning [4].

In practice, there are two key steps to automating reasoning [8]:

1. Express statement of theorems in a formal language

2. Use automated algorithmic manipulations on those formal expressions

In building an automated reasoning program we must provide an algorithmic description of a formal calculus for efficient implementation. The primary considerations of such an implementation involve defining the class of problems to be solved, specifying the rules of inference and how best to perform these computations efficiently [20]. In addition to efficiency, generality is a very desirable property for a theorem prover. Generality is the ability of the prover to handle a variety of logical systems with enough flexibility to allow users to define their own axioms and inference rules [15].

1.2 Automated theorem proving

Modal logic is an extension of classical propositional logic and is used as a formal way of representing and reasoning about notions of knowledge, belief and time. The direct applications of modal logic include linguistics [17], system verification [8], agent-based systems [16] and process analysis [12].

The design and implementation of systems and processes are fundamental to the operation of any organisation. For example, aircraft manufacturers must design highly complex systems which are then built into airplanes. The cost of such a system failing is immeasurable. Therefore, organisations need to ensure that systems will perform as required and that undesirable states will be unreachable within the context of the system. However, given the growing size and complexity of modern systems, the task of formal verification can be increasingly difficult. Automated reasoning aims to mitigate this complexity by automating the verification process [19].

Given a logical encoding of a system Γ, and a required property σ, a theorem prover aims to determine whether σ is a logical consequence of Γ, in some given logic L. That is, we determine whether the required property is always forced to be true, given the assumptions of the system. Unfortunately, such problems in modal logic are decidable with PSPACE-complete complexity [14].

Example: Consider a plane with an alarm that senses engine failure, specifically when an engine's output falls beneath a given threshold. Then considering the primitive propositions a (alarm on), f (alarm is faulty) and o (output of engine is too low), we can let Γ consist of the following assumption:

1. $\Box(a \to f \lor o)$: meaning, that in all related worlds, a triggered alarm implies that the alarm is faulty or the output of the engine is too low.

As a aircraft manufacturer, we might want to show that for every instance, a is a logical consequence of Γ. That is, the alarm always triggers when detected as faulty or engine output falls beneath a given threshold.

Automated reasoning is a mature area of research, in that there are many existing theorem provers for modal logic. Theorem provers such as FaCT++ [21] and LWB [9] are highly efficient but rely on complicated procedures, sophisticated data structures and optimised techniques to extract maximum performance [15]. That is, they are highly tailored to the logic systems that they are built for.

Recent work in this area has focused on satisfiability modulo theories (SMT) which reduce logics to boolean satisfiability problems by adding equality reasoning theories. This allows for more expressive reasoning simply by adding lightweight theories on top of existing algorithms. Instances of such theorem provers include InKreSAT [11] and BDDTAB [19], both of which perform very efficiently and at a state-of-the-art level (more detail in subsection 2.3.2).

1.3 Report Overview

Our objective in this report is to develop an elegant yet efficient algorithm for proving theorems in modal logic K by distilling a modal problem into to a boolean satisfiability problem with additional theory for the remaining modal aspects. The fundamental motivation of this idea is to leverage the enormous advances in SAT solving techniques, so as to build a simple, yet scalable theorem prover for modal logic K. Furthermore, we use a modal clausal form to represent the problem.

The current chapter provides a high-level motivation as to why automated reasoning maintains relevance given an increasingly complex world. We also introduced the basic tenet of automated theorem proving for modal logic.

Chapter two introduces the semantics and calculus of modal logic. Chapter three our method which introduces the algorithm underlying our implementation and the clausification technique we have used to preprocess input formulae. Optimisation issues relating to our implementation are discussed in chapter four. Chapter five provides an evaluation of our prover, including comparisons against existing state-of-the-art theorem provers for modal logic K. Finally, in chapter six we highlight potential opportunities for further work in this area, and our conclusion.

Chapter 2

Modal Logic

Classical propositional logic is truth-functional, in that a statement is either true or false. If we seek to add further expressiveness to such statements by way of first order logic, then we no longer have a decidable logic [5]. That is, any reasoning algorithm of first order logic cannot guarantee termination.

Modal logic allows us to enhance expressiveness, but importantly maintains decidability. This is achieved by abstracting context to different states where the modal aspect then captures the notion of transitioning between the states. Therefore we can represent a complex system as a relational frame constructed using worlds and relations [15]. In order to understand this deeper, we must first explore the syntax and semantics of modal logic. The following sections introduce the syntax, semantics and tableau procedure for modal logic per the work of Goré [6].

2.1 Syntax

Modal logic extends classical propositional logic and accordingly is more expressive with two additional unary operators - box ($\Box$) and diamond ($\Diamond$).

A formula φ in modal logic is an expression built recursively out of atomic propositions and connectives as follows, where an atomic proposition is any element from a countably infinite set of atomic propositions:

$\mathrm{p} ::= \mathrm{p}_0 \mid \mathrm{p}_1 \mid \mathrm{p}_2 \mid \ldots$

$\varphi ::= \mathrm{p} \mid \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \varphi \to \varphi$

2.2 Semantics

The semantics of modal logic are defined by Kripke models [13]. The most basic notion of a model is a Kripke Frame $\mathfrak{F}$, which is a directed graph ⟨W, R⟩ where W is a non-empty set of worlds and R $\subseteq$ W x W is a binary relation over W. The valuation on a given $\mathfrak{F}$ is a map υ: W x *atomic proposition* $\mapsto$ t, f that provides the truth value on every atomic proposition at every world of W. A Kripke model M comprises both a $\mathfrak{F}$ and υ, such that M = ⟨W, R, υ⟩.

So given a model M and some world w $\in$ W, we can lift υ to compute the truth value of a non-atomic formula by recursion on its shape.
Semantics for classical propositional connectives:

$$\vartheta(w, \neg\varphi) = \begin{cases} t & \text{if } \upsilon(w,\varphi) = \text{f} \\ f & \text{otherwise} \end{cases} \tag{2.1}$$

$$\vartheta(w, \varphi \wedge \gamma) = \begin{cases} t & \text{if } \upsilon(w,\varphi) = \text{t and } \upsilon(w,\gamma) = \text{t} \\ f & \text{otherwise} \end{cases} \tag{2.2}$$

$$\vartheta(w, \varphi \vee \gamma) = \begin{cases} t & \text{if } \upsilon(w,\varphi) = \text{t or } \upsilon(w,\gamma) = \text{t} \\ f & \text{otherwise} \end{cases} \tag{2.3}$$

$$\vartheta(w, \varphi \to \gamma) = \begin{cases} t & \text{if } \upsilon(w,\varphi) = \text{f or } \upsilon(w,\gamma) = \text{t} \\ f & \text{otherwise} \end{cases} \tag{2.4}$$

Semantics for modal connectives:

$$\vartheta(w, \Box\varphi) = \begin{cases} t & \forall \text{v} \in \text{ W if wRv then } \upsilon(v,\varphi) = \text{t} \\ f & \text{otherwise} \end{cases} \tag{2.5}$$

$$\vartheta(w, \Diamond\varphi) = \begin{cases} t & \exists \text{v} \in \text{ W such that wRv and } \upsilon(v,\varphi) = \text{t} \\ f & \text{otherwise} \end{cases} \tag{2.6}$$

From these semantics we see that classical connectives remain truth functional, whereas the truth of modal connectives are dependent on the underlying R and hence not truth functional. This is where the additional expressiveness of modal logic is derived.

To this point, modal logic includes a semantic forcing relation $\Vdash$, which is defined on a world w, a model M and a given formula φ:

Table 2.1 Consequence relations

Forces	We Say	We Write	When	$\bullet \nVdash \varphi$
in a world	w forces φ	w $\Vdash \varphi$	v(w, φ) = t	v(w, φ) = f
in a model	M forces φ	M $\Vdash \varphi$	$\forall$w$\in$W. w $\Vdash \varphi$	$\exists$w$\in$W. w $\nVdash \varphi$

Extending these consequence relations, we can define both the logical consequence $\vDash$ and satisfiability, across any set of models K:

$$\Gamma \vDash \varphi \text{ iff } \forall M \in M \Vdash \Gamma \implies M \Vdash \varphi \tag{2.7}$$

$$\varphi \text{ is K-satisfiable iff } \exists M = \langle W, R, v\rangle \in K.w \in W.w \vDash \varphi \tag{2.8}$$

We say a formula φ is satisfiable in a model M if and only if there exists some world w $\in$ W in M at which φ is true. Furthermore a set of formulae Γ is forced by a model when each φ in Γ is satisfied by the model. We can use this fact to determine logical consequence in the following way:

$$\Gamma \vDash \varphi \text{ iff } \Gamma \wedge \neg\varphi \text{ is not K-satisfiable} \tag{2.9}$$

Therefore if Γ is an empty set, we can show that φ is K-valid by proving that φ is a logical consequence of the empty set. This is ultimately the goal of our theorem prover, to show the validity of any given φ.

Modal logic systems are generally characterised by restrictions on the reachability relation R. Accordingly each system comprises various sets of formulae which are valid within their systems. Modal logic K has no restrictions on R, and is the only logic considered in this report.

There are two main deductive systems for determining logical consequence in modal logic. The first is Hilbert calculi which aims to show a sound derivation of φ from Γ via syntactic manipulation. The second deductive system is tableau calculi, which we use in our implementation.

2.3 Tableau calculi for modal logic

Tableau-based methods are widely used deductive systems to show satisfiability, and thus logical consequence, in a modal logic (L). In this section, we present the procedure of unlabelled tableau as presented by Gore [6]. We then briefly discuss existing theorem provers for modal logic K which also utilise tableau based methods.

2.3.1 Tableau calculi

A tableau rule σ consists of a numerator N and a finite list of denominators possibly separated by vertical bars. The numerator is a finite set of formulae, as is each element in the denominator. Each rule is interpreted as "if the numerator is L-satisfiable, then some denominator is L-satisfiable".

Tableau proofs are achieved via refutation. That is, to prove that a formula φ is valid we need to show that its negation $\neg\varphi$ is L-unsatisfiable. According to the correspondence that we established above, if $\neg\varphi$ is L-unsatisfiable then φ is a logical consequence of an empty set and therefore K-valid.

A tableau procedure is represented as a tree for a given formula set $\neg\varphi$, where each child branch is obtained from their parent nodes by instantiating a tableau rule. The application of a rule can either be a deterministic expansion or a non-deterministic expansion. A deterministic (e.g. $\wedge$ operator) expansion begets a single child node, whereas a non-deterministic expansion (e.g. $\vee$ operator) results in two child nodes since there are multiple possibilities for truth assignment. Note, $\neg\varphi$ must be represented in negation normal form (subsection 3.1).

The tableau rules for modal logic K (assuming NNF) are as follows:

X, Y, Z are possibly empty multi-sets of formulae

Static rules:

$$\text{(id)}: \ \frac{p; \neg p; X}{\text{x}}$$

$$(\wedge): \ \frac{\varphi \wedge \psi; X}{\varphi; \psi; X}$$

$$(\vee): \ \frac{\varphi \vee \psi; X}{\varphi; X \mid \psi; X}$$

Transitional rule:

$$(\Diamond K): \ \frac{\Diamond\varphi; \Box X; Z}{\varphi; X} \ \forall\psi.\Box\psi \notin Z \text{ and } \Box X = \{\psi \mid \psi \in X\}$$

We let Δ be the set of tableau rules for a modal logic K. By applying rules from Δ to X, we can obtain a path to a leaf node Y. A branch in the tableau is closed if a leaf node is an instance of (id) - a contradiction. A tableau is closed if all its branches are closed. Conversely, a tableau is open if it is not closed.

Tableau procedures can be considered a tree search problem. In that, we want to expand each tableau branch in order to reveal a potential satisfying model. If such a model is found, we have a proof that X is K-satisfiable. Otherwise, we have a closed tableau and X is determined to be K-unsatisfiable.

Therefore, to prove the validity of formula φ, we must show that the tableau of $\neg\varphi$ is closed. Explicitly, we prove that $\neg\varphi$ is not K-satisfiable and accordingly φ is always true.

2.3.2 Tableau based theorem provers

There are many existing tableau based algorithms for modal logic that perform very well against benchmarks and accordingly, are used in applications of modal logic.

Traditional provers are designed as all-encompassing reasoning engines that are highly specialised and purpose-built to reason within a set of logics. An instance of such a theorem prover is FaCT++ which is considered a state of the art reasoners in this field. FaCT++ is implemented in C++ and uses many optimisation techniques that have been adapted from propositional logic such dependency-directed backtracking, boolean constant propagation and ordering heuristics [21].

Recent work in this area has focused on satisfiability modulo theories which reduce a given logic to boolean satisfiability problems by adding equality reasoning theories. A SAT solver can then be used to decide the satisfiability of formulas in these logics [3].The main benefits of such approaches are simplicity and scalability, as lightweight theories can be built on top of existing SAT solvers or similar structures.

Using such an approach, Kaminiski and Tebbi develop a theorem prover (InKreSAT) for modal logic by using an incremental SAT solver and an explicit representation of the tableau rules [11]. Specifically, InKreSAT relies on a labelled tableau system which requires labelled formulae and tableau rules that operate on labelled formulae. A label represents a world w of the Kripke model, to which w $\in$ W. A labelled formula is a structure of the form w:φ where w is a label and φ is a formula. It has the semantic meaning that the formula φ is true in the world w [15].

Using binary decision diagrams (BDDs), Olesen [19] presents a method for implementing unlabelled tableau calculus for modal logic K. Given the recursive nature of unlabelled tableau calculus, the tableau expansion does not require labelling of expanded formula to maintain which formulae must be forced in which worlds.

Our implementation relies on unlabelled tableau calculus for derivation, meaning there is no explicit representation of the reachability relation since these properties are built into the rules of deduction [6]. Furthermore, our saturation phase relies on a SAT solver which has very similar internal mechanics to that of BBDs.

Chapter 3

A SAT-Based Method

As introduced, we aim to develop an elegant, scalable and automated method for proving theorems in modal logic K. Explicitly, we want to prove the validity of a given modal formula. Therefore to show validity, we want to show that the negated formula is not satisfiable. The key insight to our reasoning algorithm is the fact that modal logic is an extension of classical propositional logic. Therefore we try to relegate as much of the required deduction to a SAT solver, and use the tableau transition rule for the remaining modal aspects of the formula [2].

Our implementation is written in Python, and being an interpreted high-level language, we immediately sacrifice some computational efficiency. That said, the premise of our research is to rely on the optimisation and efficiency of an existing SAT solver, therefore any additional computation cost to our wrapper should be immaterial. We also note that Python does provide a rich library and more importantly has an API for the SAT solver we use in our implementation.

The underlying algorithm for our prover is recursive in structure, but the overall design is very simple. The result is a robust theorem prover, however shows poor performance on standard benchmarks.

The remaining sections of this chapter discuss each aspect of our algorithm in detail.

3.1 Modal clausal form

Since our algorithm finds a proof by refutation, before completing the following transformation to modal clausal form, we first negate the input formula and then let it equal to φ. Transformation into sequent representation was applied to tableau calculi by Nguyen [18], who also showed that a clausal form can give better space bounds and improved decision procedures for some modal logics. Additionally, a clausal form also allows us to easily distil the propositional components of φ which is important to our implementation. The following sequence of functions are called to transform φ into clausal form:

3.1.1 Lexing and parsing

Lexical analysis and parsing of φ
We now represent φ as an abstract syntax tree (AST) for further simplification and transformation.

3.1.2 Negation normal forming (NNF)

Convert simplified φ to NNF
To minimise the number of inference rules required to decompose a formula set, we first generally convert the formula into NNF before commencing any proof procedure. A formula is in NNF if and only if all occurrences of $\neg$ appear in front of atomic formulae only and there are no occurrences of $\rightarrow$ and $\leftrightarrow$. Accordingly, we recursively distribute negations over subformulae using the rules in table 3.1.

$\varphi \rightarrow \upsilon = \neg\varphi \vee \upsilon$	$\neg(\varphi \rightarrow \upsilon) = \varphi \wedge \neg\upsilon$
$\neg(\varphi \wedge \upsilon) = \neg\varphi \vee \neg\upsilon$	$\neg(\varphi \vee \upsilon) = \neg\varphi \wedge \neg\upsilon$
$\neg\neg\varphi = \varphi$	$\neg\Diamond\varphi = \Box\neg\varphi$
$\neg\Box\varphi = \Diamond\neg\varphi$	

Table 3.1 Negation normal forming rules

3.1.3 Simplification

Simplify AST(φ)
This step is a quasi optimisation to promote early inconsistencies, but also simplify the concepts of the expression [3]. Simplifications include removing constants (e.g. a $\wedge$ False -> False) and eliminating redundancy in expressions (e.g. $\neg\neg$ a -> a). These rules are summarised in table 3.2.

$$\begin{array}{ll} \varphi \wedge \top = \varphi & \varphi \wedge \bot = \bot \\ \varphi \vee \top = \top & \varphi \vee \bot = \varphi \\ \varphi \rightarrow \top = \top & \top \rightarrow \varphi = \varphi \\ \bot \rightarrow \varphi = \top & \top \rightarrow \bot = \bot \end{array}$$

Table 3.2 Basic boolean simplification rules

3.1.4 Modal clausal forming

Now that our formula is in NNF, we can begin our transformation into modal clausal form [7]. We use a,b,c to denote atomic literals and $\boxdot$ to denote a possibly empty sequence of modal box operators, which we can think of as an universal modality. We refer to $\boxdot$ as the modal context from herein. A modal clause is a formula of the form $\bot$, $\boxdot[a_1,..,a_k]$, $\boxdot[a,\Box b]$, $\boxdot[a,\Diamond b]$ or $\boxdot\Diamond b$; where $\boxdot[\varphi_1,..,\varphi_k]$ is written to denote $\boxdot[\varphi_1 \vee .. \vee \varphi_k]$.

Per Goré and Nguyen [7], for any modal logic L, every set X of formulae can be transformed in quadratic time to a set of modal clauses X' such that X is L-satisfiable iff X' is L-satisfiable.

In the context of our prover, we repeatedly apply the following transformations to NNF(φ) in order to achieve a modal clausal form:

1. If of the form $\boxdot[\gamma \wedge \psi]$ then replace by $\boxdot\gamma$ and $\boxdot\psi$.

2. If of the form $\boxdot[\gamma_1,..,\gamma_k,\psi \vee \xi]$ then replace by $\boxdot[\gamma_1,..,\gamma_k,\psi,\xi]$.

3. If of the form $\boxdot[a_1,..,a_h,\gamma_1,..,\gamma_k]$ where $\gamma_1,..,\gamma_k$ are complex sub-formula, and $k \geq 2$, or $k = 2$ and $h > 1$, then replace by $\boxdot[a_1,..,a_h,p_1,..,p_k]$, $\boxdot[\neg p_1,\gamma_1]$ .. $\boxdot[\neg p_k,\gamma_k]$ where $p_1,..,p_k$ are new atomic literals.

4. If of the form $\boxdot[a,\nabla\gamma]$ where ∇ is either $\Box$ or $\Diamond$, and γ is a complex sub-formula, then replace by $\boxdot[a,\nabla p]$ and $\boxdot\Box[\neg p,\gamma]$ where p is a new atomic literal. We also note that in the absence of a, we simply let a$\mapsto \bot$ and include it in the original formula.

5. If of the form $\boxdot[a,\gamma \wedge \psi]$, then replace by $\boxdot[a,\gamma]$ and $\boxdot[a,\psi]$.

The above transformation makes at most n replacements, where n is the sum of the lengths of the each formula in NNF(φ) [7]. Testing was completed, by use of an external theorem prover, to ensure that our entire transformation preserved the if and only if satisfiability relation of a given ϕ and its modal clausal form. Using the theorem prover we checked whether $\neg\phi$ was not valid, meaning that ϕ was satisfiable. If ϕ was satisfiable, we then needed to check that $\neg mcf(\phi)$ was also deemed not valid by the prover.

3.2 SAT solver

A key aspect of our theorem prover is the use of a SAT solver. For the purpose of our implementation, we have used Z3, a SAT solver developed by Microsoft Research [3]. The primary motivation for using Z3 was the fact that it is highly optimised and also provides an API for Python. The extent to which we use the SAT solver is to retrieve satisfying models, if possible, based on a set of propositional constraints. This will become more evident in section 3.3.1.

3.3 Proving Algorithm

At a fundamental level, our algorithm follows a tableau procedure, in that we simply construct a search tree to find a satisfying model. This search is conducted in a recursive depth-first manner, with the termination conditions being a satisfying model was found (i.e. an open tableau branch) or closed tableau. The algorithm separates this tableau procedure into two phases, a saturation phase (3.3.1) and a transition phrase (3.3.2). The saturation phase achieves a downward-saturated and consistent set; meaning all classical connectives, for a given world, have been expanded and there are no contradictions in the formulae. To prevent destroying any proceeding closures in our tableau, we only apply the transitional rule to downward-saturated and consistent sets [6].

Given the transformation of φ to modal clausal form, it is represented as a dictionary when passed the prover. Each key in the dictionary represents the depth of a modal context in φ. Values are then mapped to each key based on modal depth W and are represented as a set of four possibly empty lists constructed as following:

- A is a list of formulae with the form $\boxdot[a_1, .., a_{\mathrm{k}}]$
- IB is a list of formulae with the form $\boxdot[a, \Box b]$
- ID is a list of formulae with the form $\boxdot[a, \Diamond b]$
- D is a list of formulae with the form $\boxdot\Diamond b$

For the purpose of our implementation the keys in the dictionary are always relative to the "starting world". To reflect modal jumps we track modal depth W and this value is used to calculate the effective modal context for each formulae in the dictionary. Simply, the effective modal context of a formulae is the modal context relative to the "starting world" (dictionary key) less modal depth.

Example:
Let $\varphi = \neg(\Box(a \to b) \to (\Box a \to \Box b))$.
Therefore, nnf(φ) = $\Box(\neg a \lor b) \land (\Box a \land \Diamond\neg b)$
Finally, mcf(φ) = $[\Diamond\neg b], \Box[\{\neg a, b\}, \{a\}]$

By applying our dictionary representation to mcf(φ), we get:
key: 0 ; value: D: $\{\Diamond\neg b\}$
key: 1 ; value: A: $\{\{\neg a, b\}, \{a\}\}$
Where each key represents the depth of a modal context (i.e. length of nested boxes).

From our main function we call the saturation function first at a modal depth of 0. Since this is proof by refutation, if the algorithm returns satisifiable then the input formula is not valid. Otherwise, the input formula is valid.

3.3.1 Saturation function

In a traditional tableau construction, during a saturation phase, we would only instantiate static tableau rules - (id), ($\land$) and ($\lor$). However, per Kripke semantics, we see that these rules are all contained within a single world, hence are positional in nature. Therefore, we allow the SAT solver to implicitly apply all static rules to achieve a downward-saturated and consistent set. By extension, we also observe that there are no modal aspects in the saturation phase.

Figure 3.1 shows our algorithm for the saturation phase of our tableau construction. In this algorithm, modal depth is analogous to the number of modal jumps from our "starting world". Based on this modal depth, we extract the sets A, IB, ID and D where effective modal context is 0.

The function aggregates the propositional constraints in A (see section 3.3) and the set of forced modal propositions from a directly related prior world. The aggregated set of constraints are then added to an instance of the SAT solver. If the SAT solver can find a satisfying valuation, it returns the satisfying model for that world. We then pass this valuation to the transition function to ensure all modal aspects, in IB, ID and D, are satisfied in this current world.

```
'''
M:  Dictionary representation of modal clausal form per above.
Key: modal depth (i.e. length of modal context)
Value: [A, IB, ID, D]
X: Dictionary of active modal literals from previous world
V: Dictionary of valuations at each modal depth.
W: Modal depth (initially 0)
'''
def saturation_function(M, X, V, w):
    if M[w] is None and X is None:
                return satisifiable
    else:
                A_w, IB_w, ID_w, D_w <- M[w]
                sat_constraints = set()
                sat_constraints <- add {X[w], A_w}

                valuation <- call_sat_solver(sat_constraints)
                if valuation is satisfiable:
                        V[w] <- valuation
                        return transition_function(M, X, V, w)
                else:
                        return unsatisifiable
```

Fig. 3.1 Saturation of propositional connectives

If the SAT solver cannot find a satisfying valuation, this is an instance of (id) and the branch will be closed.

3.3.2 Transition function

The transition function receives a valuation V, which is a downward-saturated and consistent set. However, we still need to check the formulae in IB, ID and D (see section 3.3), to ensure that any active modal propositions are satisfied.

Figure 3.2 shows our algorithm for the transition phase of our tableau construction. Formulae in lists IB and ID are of the form $\boxdot[p, \Box b]$ and $\boxdot[p, \Diamond b]$ respectively. Therefore if p is satisfied in V (or can be without contradiction), then each of the clauses in IB and ID are satisfied. If p is not satisfied, then the modal propositions are active and will need to be forced in the model. By construction, diamond modal propositions in D are active, and therefore must also be forced in the model.

```
def transition_function(M, X, V, w):
        w1 = w + 1
        A_w, IB_w, ID_w, D_w <- M[w]

        ''' get_active returns set of active modalities based
        on whether each clause in ID_w or IB_w is (or can be)
        satisified by V[w] '''
        active_box_literals <- get_active(IB_w, V[w])
        active_diamond_literals <- get_active(ID_w, V[w]), D_w

        if active_diamond_literals is None: return satisifiable

        for diamond in active_diamond_literals:
            X[w] <- add(diamond, active_box_literals)
                branch = saturation_function(M, X, V, w1)
                if branch == unsatisifiable:
                        sat_constraints <- add {X[w], A_w}
                        ''' get new valuation for w: OR-branch '''
                        valuation = call_sat_solver(sat_constraints)
                        if valuation is satisifiable:
                                V[w] <- valuation
                                return transition_function(M,X,V,w)
                        else: return unsatisifiable
        return satisifiable
```

Fig. 3.2 Transition by diamond modalities

If there are no active diamond propositions, then the branch is satisfied since any box propositions are vacuously true. Conversely, if there are indeed active diamond propositions, we apply the following modified transition rule:

$$(\Diamond K^{*}):\quad \frac{\Diamond\varphi_1;..;\Diamond\varphi_i;\Box X;Y;Z}{\varphi_1;X;Y^{*}\;||\;..\;||\;\varphi_i;X;Y^{*}}\;\forall\psi.\Box\psi\notin Z \text{ and } \Box X=\{\psi\mid\psi\in X\}$$

For notational clarity, Y is the set of formulae where the effective modal context of the formulae is non-empty. In Y^{*} one box has been stripped from the modal context for each formulae [1].

Each denominator is passed to the saturation function at an incremented modal depth. This in effect achieves a modal jump with AND-branching. That is, if any of

[1]For the purpose of implementation, this reduction to modal context is based on the increment made to modal depth.

the denominators are not satisfiable then the entire branch closes. On this closure, we backtrack and seek a new satisfying valuation at the prior modal depth, which we directly pass into the transition function. This process of finding a new valuation dictates the OR-branching for our tableau tree, as only one of these models needs to result in a branch that remains open.

3.4 Formal proof of prover

To prove correctness of our algorithm we need to formally show the following:

3.4.1 Termination

Termination requires that our algorithm terminates given any input, this also shows decidability. The two termination conditions of our tableau are 1) an open tableau branch is found, and 2) the tableau is closed.

In relation to condition 1, for a tableau branch to be open, means that every node along a branch, including the leaf node, must be satisfiable. A leaf node will not enforce a modal jump, therefore we should have a saturated and consistent set with no active diamond propositions. We can see from figure 3.2 that this is indeed a termination condition of our algorithm that recursively passes the satisfied result to the top level prover. Furthermore, we note that each application of the ($\Diamond K^*$) rule reduces the maximal-modal degree of the formula, therefore ensuring this termination condition.

In relation to condition 2, a closed tableau is one in which every branch is closed. This means that the leaf node of every branch is an instance of (id) - i.e. unsatisfiable. Figure 3.1 shows that this is indeed a termination condition of our saturation phase when SAT solver cannot find a satisfying valuation. The algorithm then closes the (id) branch, and backtracks to a new OR-branch. OR-branching is exhaustive since there are only a finite number of satisfying valuations for a fixed set of constraints.

3.4.2 Soundness

Soundness requires that our rules of inference are truth-preserving. Since our algorithm is a proof by refutation, this means if we can derive a closed tableau for $\neg\varphi$, then we must be able to show that φ is logically valid with respect to the semantics of modal logic K [20].

The static rules (subsection 2.3.1) of our tableau procedure are propositional in nature, therefore are inherently truth-functional. Assuming φ contains no modal aspects, the SAT solver would handle the entire inference, which we can assume is sound. Now we must simply prove the soundness of our transition rule ($\Diamond K^*$).

Lemma: If $\{\Diamond\varphi_1;..;\Diamond\varphi_i;\Box X;Y;Z\}$ is K-satisfiable then $\{\varphi_1;X;Y^*\}$ || .. || $\{\varphi_i;X;Y^*\}$ is K-satisfiable.

Proof: Suppose $\{\Diamond\varphi_1;..;\Diamond\varphi_i;\Box X;Y;Z\}$ is K-satisfiable, where X, Y, Z are possibly empty multisets of formulae. Then it follows that:

- There exists a Kripke model M $= \langle W,R,v\rangle$ and some $w \in W$ with $w \Vdash \Diamond\varphi_1;..;\Diamond\varphi_i;\Box X;Y;Z$
- This means that there exists some $v \in W$ where vRw and $w \Vdash \Diamond\varphi_j$, *for each j = 1 .. i.* We now observe the following forced relations:
  - $v \Vdash \varphi_j$ and;
  - $v \Vdash X$ and;
  - $v \Vdash Y^*$ being the set of formulae where modal context is stripped of one box. Y* is forced inductively.

Therefore $\{\varphi_1;X;Y^*\}$ || .. || $\{\varphi_i;X;Y^*\}$ is K-satisfiable and our rules of inference, specifically ($\Diamond K^*$), are a sound. ∎

3.4.3 Completeness

Completeness requires that if φ is a logical consequence of its assumptions with respect to the semantics of modal logic K, then φ should also be derivable from these assumptions. In application, this means that if our theorem prover returns φ as being not valid by finding an open tableau for $\neg\varphi$, then φ must indeed not be valid.

To show this, we build a Kripke model with a "starting world" w$_0$ that satisfies $\neg\varphi$. As introduced in section 3.3, we first convert $\neg\varphi$ to modal clausal form and store it as dictionary Y. Each key in Y represents a modal depth; for example, at w$_0$ the modal depth would be 0 and all formulae mapped to 0 would have an empty modal context.

Proof: Supposing no tableau branches close for $\neg\varphi$, we then consider the following systematic procedure:

- Stage 0: In the saturation function, let w be set of formulae where effective modal context is empty, and let C be set of formula where effective modal context non-empty.

- Stage 1: Using the SAT solver, derive a downward-saturated and consistent node w^* by finding a satisfying valuation for the propositional components of w.

- Stage 2: Now in the transition function, let $\Diamond\gamma_1;..;\Diamond\gamma_\mathrm{i}$ be all <u>active</u> diamonds in the w^*. We also let X be the set of <u>active</u> box propositions. So w^* now looks like: $\Diamond\gamma_\mathrm{j}, \Box X; C; Z$ for each i in 1 .. i. To force the modal aspects of the node, we apply $(\Diamond K^*)$ to each diamond i - this includes incrementing the modal depth by one and recursing to the saturation function. This recursive step is analogous to a modal jump to $\mathrm{v_i}$ where $\mathrm{v_i R w_i}$.

We repeat these steps for each node $\mathrm{v_j} = (\gamma_\mathrm{j}; X; C)$ until termination - see subsection 3.4.1. Finally we let W be the set of all -nodes (downward-saturated and consistent nodes), where $\mathrm{w}^* \lhd \mathrm{v}^*$. This constitutes a model graph $\langle W, \lhd \rangle$ which by Hintikka is a sufficient condition to showing $\neg\varphi$ is satisfiable [6]. ∎

Chapter 4

Optimisation

The naive implementation of our theorem prover shows poor performance, where we measure performance by timing efficiency. Therefore, optimisation is required in order to achieve performance competitive with existing provers. However, in application, such optimisation proved to be difficult given the underlying structure of our implementation.

By using depth-first search, we expand each branch of our tableau completely before we begin expanding the next. To find an open tableau, we simply need to find the first open tableau branch, which represents a satisfying model for the negated formula. Conversely, to conclude that the tableau is closed, we need to expand all branches and show that each leaf is an instance of (id). Therefore to minimise the computational requirement, we want to prioritise the expansion of branches with a higher likelihood of remaining open and quickly fail on branches with potential contradictions. To this point, the key issue that our implementation encounters is thrashing.

4.1 Thrashing

Thrashing occurs during tableau search when branches with the same underlying formulae are explored. This can have significant impacts on the performance of a theorem prover, particular where the branching factor of a formula is high. Hustadt and Schmidt [10] propose a backtracking technique so as to mitigate this issue. Essentially they propose that on an unsatisfiable node, the algorithm backtracks to the node of the tableau tree which is relevant to the failure of the given branch. Explicitly, this means that if the algorithm expands an instance of (id), it must then backtrack to the node prior to the node responsible for allowing the eventual instantiation of the (id) leaf. This, of course, requires the maintenance of assumption sets which contain all the formulae and rules which have contributed to the closure of a branch.

In terms of our implementation this method of backtracking, or similar, is not possible given the use an unlabelled tableau procedure and the fact that we do not cache the formulae and rules from preceding nodes and branches. Therefore we cannot extract sufficient information from an contradicting leaf in order to guide our search. As an example, say we find a contradiction at a modal depth of 1, our algorithm will then backtrack to a modal depth of 0 and seek a new satisfying valuation (i.e. new OR-branch at "starting world"). If such a valuation is found, the prover will transition to a modal depth of 1 for each active diamond. Despite the constraints at this modal depth being potentially very similar to the previous OR-branch (at the same depth), there is no available information that we can use to determine whether this branch should be immediately closed. Therefore the SAT-solver will naively generate every OR-branch which leads to redundant expansion. The transition rule is the only aspect of the tableau that we have complete control over. Accordingly, we look to adapt a backtracking technique suitable to our implementation.

4.2 Modal deactivation

By analysing the structure of our tableau, we see that there are two types of branching; OR and AND branches. OR-branches are generated by the SAT solver, hence limiting our control over the sequence of OR-branching. That said, we can influence valuations via the constraints we add to the SAT solver. AND-branches are instantiated by the transition rule, which we apply to every active diamond modality. However, given the structure of formulae in our ID (implied diamonds) or IB (implied boxes) sets, the activity of these modal literals are dictated by the valuations provided by the SAT solver.

By using a modified backtracking technique, we attempt to promote potentially open branches in our tableau by blocking modal literals that we deem responsible for contradictions in proceeding worlds. This is achieved by the following procedure:

1. If a given world (w1) returns unsatisfiable, we identify the modal literals responsible for each contradiction in that world. In the case that the set of responsible modal literals is empty, the tableau is closed as there is an unavoidable contradiction at the given depth for all branches.

2. If the responsible modal literals are from the sets ID or IB, then this means they were activated from an implication clause in a preceding world (w). Therefore,

we create a set C, comprising the antecedents of each relevant modal implication clause.

3. Finally we backtrack to w, where we add C as a soft constraint [1] to the SAT solver. The aim being to block the activity of the responsible modal literals in w.

Ultimately, thrashing remains a critical problem for implementation that we cannot easily avoid. Even by promoting positive branches as best we could, the performance of our theorem prover remains poor in comparison to existing theorem provers. A detailed discussion of performance is continued in chapter 5.

[1] Soft constraints are those we would like to be true, but are not necessary in order to satisfy the model.

Chapter 5

Evaluation

This chapter provides an evaluation of our prover, including comparison against existing state-of-the-art theorem provers for modal logic K. However first we introduce the benchmarks we have used and provide a description of our test environment.

5.1 Benchmarks

To determine the performance of our implementation we use LWB benchmarks as constructed by Balsiger [1]. These benchmarks are designed precisely to compare various methods for proof search in modal logic K. Per Balsiger, the key considerations that underpin the construction of these benchmarks are:

- Formulas are a combination valid and not-valid
- Formulas are of differing shapes and structures
- Benchmark formulas are difficult enough for future implementations
- The outcome of formula proofs are deterministic
- Artificial optimisations do not enhance provability
- Applying the benchmarks to a prover does not take an unreasonable amount of time
- Results are easily summarised

In total there are nine subclasses of provable and non-provable formula, with each class having 21 instances of formula that increase in difficulty. Given the considerations above, the LWB benchmarks are the standard by which theorem provers for modal logic are compared.

5.2 Test Environment

Our testing was run on an intel 2.60GHz CPU with 8GB of memory. We used a time limit of 60 seconds per formula which included parsing, transformation to modal clausal form and proof search of the tableau.

5.3 Experimental results

We evaluate the performance of our prover against BDDtab, InKreSAT and FaCT++; all of which achieve state-of-the-art performance. Table 5.1 shows the number of instances, per problem subclass, that each prover solves. It is clear from these results that our prover does not compete with any of these existing methods.

Table 5.1 Results on LWB benchmarks for Modal Logic K

Subclass	**SAT-based**	**BDDtab**	**InKreSAT**	**FaCT++**
branch_n	9	16	13	10
branch_p	16	21	16	9
d4_n	2	21	21	21
d4_p	3	21	21	21
dum_n	10	21	21	21
dum_p	0	21	21	21
grz_n	0	21	21	21
grz_p	0	21	21	21
lin_n	14	21	21	21
lin_p	1	21	21	21
path_n	9	21	10	21
path_p	12	21	10	21
ph_n	4	9	21	13
ph_p	3	9	9	7
poly_n	21	21	21	21
poly_p	21	21	21	21
t4p_n	1	21	21	21
t4p_p	0	21	21	21

Our implementation performs well on subclasses *branch* and *poly.* Whereas, performance is extremely poor on subclasses *grz* and *t4p*. In subclass *grz* we fail to find a single proof within the defined time limit. In general, however, our theorem prover shows sub par performance when compared to the other methods. Given the nature of our implementation and the use of a higher level scripting language, these poor results are not entirely surprising.

The inefficiencies of our algorithm mainly stem from the fact that we are unable to optimally guide the branching of our tableau, particularly in relation to redundant branches. This is evidenced by the non valid problems for subclass *grz*, whereby an open branch can be found at a low modal depth (in smaller problems), however since our prover is stuck thrashing on one side of the tableau, it runs out of time to find the open branch on the other side. We note, that by using an iterative depth-first approach we were able to find proofs in *grz_n*, however this was at a greater detriment to other subclasses.

Using a SAT solver to generate the OR-branches of the tableau appears to hinder performance. This is due to an inability to influence the order in which valuations are generated and therefore need to exhaustively expand each branch without the ability to promote branches with a higher likelihood of remaining open. This is compounded by the fact that we have to manually constrain the SAT solver each time we require a new satisfying valuation of a world previously expanded. Subclasses such as *branch* and *poly* have limited OR-branching factors, hence our prover performs very well in these instances.

By using a binary decision diagram (BBD) in place of a SAT solver, BDDtab is able to achieve much greater performance given the compact data structure and control over how the BBD is refined when finding a new satisfying valuation on the closure of a branch. Since a BBD is represented as a directed acyclic graph, when it is refined, the subformula in the closed leaf is removed from the set of satisfying valuations. Accordingly, all potential OR-branches containing that instance of subformula are removed as well, therefore eliminating many more branches than the one originally closed [19]. Similarly, given the use of a labelled tableau calculi and a single instance of a SAT solver, InKreSAT is able to maintain an agenda of unexpanded formulae at each world hence allowing greater control over when and which branches are expanded [11]. Ultimately, our SAT-based method is inefficient given the naive manner in which we are forced to generate OR-branches.

Figures 5.1 and 5.2 compare the actual total time taken by our implementation (MCF transformation and proving algorithm) and BBDtab, for the problems in subclass *poly* - not valid (n) and valid(p). Despite being able to solve all 42 instances in this subclass, there is an evident and increasing discrepancy in the amount of time required by our theorem prover compared to BBDtab. The time complexity of our algorithm appears to grow at an exponential rate. We also note that, on average, the modal clausal transformation takes 12.80% of the total time per problem in subclass *poly*. However, this percentage proportion increases steadily as the problem sizes increased. Therefore, it is very evident that our inefficiencies stem from our proving algorithm.

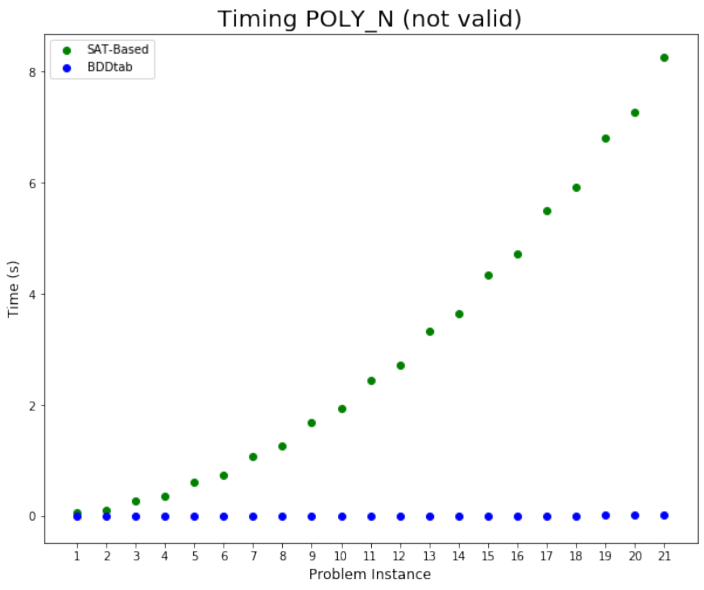

Fig. 5.1 Not valid instances from subclass poly

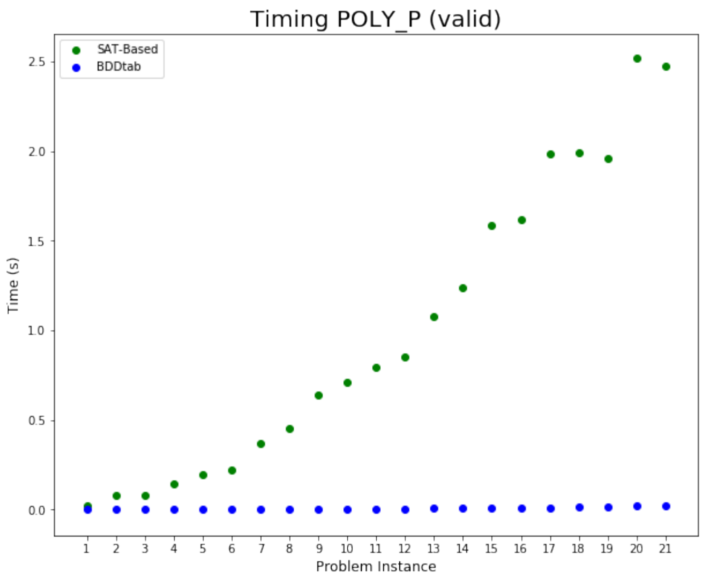

Fig. 5.2 Valid instances from subclass poly

Chapter 6

Conclusion

6.1 Further Work

Despite the poor performance of our naive SAT-based theorem prover, there remain several ideas for further improvement and experimentation in this area:

- **Application to multimodal logics of belief**
  Per Goré and Nguyen [7], we transform input formula into a modal clausal form. However, in their research, the intended use of this sequent form was to develop a set of clausal tableau calculi for multimodal logics designed for reasoning about multi-degree belief in multi-agent systems. Therefore there exists scope to extend our proving algorithm to incorporate these tableau calculi.

- **Application to Modal Logic S4**
  Claessen proposed an SAT-based theorem prover for Intuitionistic Logic that achieved very good results [2]. Given the accessibility relations of Intuitionistic Logic (reflexive, transitive and persistent), a single instance of an SAT solver was required throughout an entire proof because constraints could incrementally be added. Whilst Modal Logic S4 does not require a persistence condition, box modalities are still global constraints. As such, there is an immediate reduction in the amount of work that our SAT-solver would need to do, which could have a positive impact on performance.

- **Re-implement in a low level programming language**
  Our initial hypothesis was that by relying on a highly optimised SAT-solver to do the bulk of computation the resulting performance of our prover would be acceptable. However, this was not the case. Using python to implement our

algorithm certainly compounded any inefficiencies. Therefore scope certainly exists to improve performance by reimplementation in a lower level programming language.

- **Algorithm optimisation**
  Our current implementation of this SAT-based prover is rather naive. Whilst various backtracking techniques were considered, they were deemed infeasible given an inability to control and maintain interactions between clauses that were passed to the SAT solver. Therefore methods to optimise OR-branching, and avoid the expansion of redundant branches given knowledge of realised contradictions, could be further explored in order to extract better performance. A potential consideration could be the use of caching, to store causes responsible for contradictions.

6.2 Conclusion

The objective of this report was to develop and implement a theorem prover for modal logic K that leveraged the efficiencies of an existing SAT solver. For this purpose, we developed an algorithm that isolated the propositional and modal aspects of a tableau proof, and utilised a SAT solver for the propositional components of the proof search. Furthermore, we also show that algorithm that underpins our theorem prover is sound and complete.

Whilst memory efficient, the resulting performance of our theorem prover is very poor when compared to other state-of-the-art theorem provers. Our results indicate that using a SAT solver with an unlabelled tableau system may potentially not be an effective approach for searching a tableau tree in modal logic K. This is due to the fact that we are forced to generate our branching in a very naive manner. However, our ability to conclude ineffectiveness of this SAT-based method is not definite given our choice of programming language and potential for algorithm optimisation. Despite this, given our use of modal clausal forms, there remain avenues to pursue this research further in the implementation of tableau calculi for multimodal logics of belief [7].

References

[1] Balsiger, P., Heuerding, A., and Schwendimann, S. (2000). A benchmark method for the propositional modal logics k, kt, s4. *Journal of Automated Reasoning*, 24(3):297–317.

[2] Claessen, K. and Rosén, D. (2015). Sat modulo intuitionistic implications. In Davis, M., Fehnker, A., McIver, A., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 622–637, Berlin, Heidelberg. Springer Berlin Heidelberg.

[3] de Moura, L. and Bjørner, N. (2008). Z3: an efficient smt solver. 4963:337–340.

[4] Everitt, T. (2010). Automated theorem proving. Master's thesis, Stockholm University.

[5] Gabbay, D. M. and Guenthner, F. (2013). *Handbook of Philosophical Logic: Volume 17*. Springer Publishing Company, Incorporated.

[6] Goré, R. (1999). *Tableau Methods for Modal and Temporal Logics*, pages 297–396. Springer Netherlands, Dordrecht.

[7] Goré, R. and Nguyen, L. A. (2009). Clausal tableaux for multimodal logics of belief. *Fundam. Inf.*, 94(1):21–40.

[8] Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

[9] Heuerding, A., Jäger, G., Schwendimann, S., and Seyfried, M. (1996). The logics workbench lwb: a snapshot. *Euromath Bulletin*, 2:177–186.

[10] Hustadt, U. and Schmidt, R. A. (1998). Simplification and backjumping in modal tableau. In de Swart, H., editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 187–201, Berlin, Heidelberg. Springer Berlin Heidelberg.

[11] Kaminski, M. and Tebbi, T. (2013). Inkresat: modal reasoning via incremental reduction to sat.

[12] Koehler, J., Tirenni, G., and Kumaran, S. (2002). From business process model to consistent implementation: A case for formal verification methods. In *EDOC*.

[13] Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94.

[14] Ladner, R. E. (1977). The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing.*

[15] Li, Z. (2008). *Efficient and Generic Reasoning for Modal Logics.* PhD thesis, The University of Manchester.

[16] Marek, V. W., Shvarts, G. F., and Truszczynski, M. (1991). Modal nonmonotonic logics: Ranges, characterization, computation. *J. ACM*, 40:963–990.

[17] Moss, L. S. and Tiede, H.-J. (2007). 19 applications of modal logic in linguistics. In Blackburn, P., Benthem, J. V., and Wolter, F., editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 1031 – 1076. Elsevier.

[18] Nguyen, L. A. (1999). A new space bound for the modal logics k4, kd4 and s4. In Kutyłowski, M., Pacholski, L., and Wierzbicki, T., editors, *Mathematical Foundations of Computer Science 1999*, pages 321–331, Berlin, Heidelberg. Springer Berlin Heidelberg.

[19] Olesen, K. (2012). Using bdds to implement propositional modal tableaux. Master's thesis, The Australian National University.

[20] Portoraro, F. (2014). Automated reasoning. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, winter 2014 edition.

[21] Tsarkov, D. and Horrocks, I. (2006). Fact++ description logic reasoner: System description. In Furbach, U. and Shankar, N., editors, *Automated Reasoning*, pages 292–297, Berlin, Heidelberg. Springer Berlin Heidelberg.

Appendix A

Independent study contract

The final project description was to implement a sound and complete SAT-based theorem prover for modal logic K. The theorem prover was to show validity of an input modal formula. The prover was to be implemented in Python and make use of an existing SAT-solver. For this purpose, we decided to use Z3-solver by Microsoft. Another key requirement of the implementation was transformation of the modal problem into modal clausal form.

A signed copy of the independent study contract is contained on the following two pages. We note that the main deviations from this initial contract relate to the programming language and SAT solver used in the implementation.

Australian National University

INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID: u5453384

SURNAME: Lawton **FIRST NAMES:** Darren

PROJECT SUPERVISOR (*may be external*): Rajeev Goré

COURSE SUPERVISOR (*a RSCS academic*): Peter Strazdins

COURSE CODE, TITLE AND UNIT: COMP8755 Individual Computing Project, 12 Units

SEMESTER ☐ S1 ☒ S2 YEAR: 2016/17

PROJECT TITLE:
Automate reasoning using various modal logics for AI.

LEARNING OBJECTIVES:

- Gain an in depth understanding of propositional and modal logic.
- Understand the underlying workings of MiniSat. Additionally, learn OCaml programming language.
- Implement a theorem prover for modal logic, and evaluate results using relevant benchmarks.

PROJECT DESCRIPTION:
The project is to implement a theorem prover for modal logic, to show that "A logically implies B" in Logic L, where L is a parameter.

Since modal logic is a superset of classical propositional logic, we propose to transpose the modal formulae into modal clausal form. Given the reduction to clausal form, we can use an existing SAT solver (namely MiniSat) to then solve the overall problem. The transposition, and communication with MiniSat will be completed in a master program, which will be written in OCaml.

Furthermore, we will need to prove the soundness and completeness of our methodology.

Australian
National
University

ASSESSMENT (as per course's project rules web page, with the differences noted below):

Assessed project components:	**% of mark**	**Due date**	**Evaluated by:**
Report: name style: ______ (e.g. research report, software description..., no less than 45% weight assigned)	45		(examiner) ?Dirk?
Artefact: name kind: ______ (e.g. software, user interface, robot..., no more than 45% weight assigned)	45		(supervisor) ~~B~~ Raj
Presentation:	10		(course convenor)

MEETING DATES (IF KNOWN):
Weekly meetings as deemed necessary. once a week.

STUDENT DECLARATION: I agree to fulfil the above defined contract:

[illegible] .... 23/5/2017 ....
Signature Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

[illegible] .... 23/5/2017 ....
Signature Date

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course coordinator approval)

....
Signature Date

SECTION D (Projects coordinator approval)

....
Signature Date

Appendix B

Software artefacts

Our theorem prover has been implemented using Python, and requires installation of the Z3 python package. The software structure has been separated into two main packages:

1. Clausifier: lexical analysis, parsing and transforming the input formula into modal clausal form
2. Prover: constructs the tableau proof and returns the result to the top level caller

Both these packages are contained within the src folder and every sub module within these packages have been written by me for the purpose of this project. The top level calling function is the main.py file, which is also in the src folder.

Both the clausifier and prover were tested using benchmark formula, as described in subsection 5.1, for modal logic K. The test scripts are saved within each package folder along with a single set of results.

The test method for our clausification functions are detailed in subsection 3.1.4. We note that this test script relies on an external theorem prover which must accept formula represented in the same syntax per the README file included in Appendix C. For the purpose of our testing we used BBDtab - this software has not been included within our submitted artefacts.

To test the correctness of our prover, we simply analysed the output per the benchmark problems saved in the submitted artefacts folder. The desired result of each benchmark problem is known from the outset, therefore so we compared the output of our theorem prover to known result in order to determine correctness. There are 378 problem instances in the LWB benchmarks.

Our testing and implementation was completed on a standard desktop using a linux operating system. The specifications of the desktop are an intel 2.60GHz CPU, with 8GB of memory. Python3.6 was the version of compiler used.

Appendix C

README file for software

A copy of the README file for this software is contained on the following page.

```
# SATtab
A SAT-Based Theorem Prover for Modal Logic K.

### Prerequisites
To run this software you will need to install the ply and z3-solver packages.
Depending on your Python version, these packages can be installed using the following comma
nds:
        * pip3 install ply
        * pip3 install z3-solver

### Using the software
Usage: python3 src/main.py [-v]

The prover will read one line of standard input as a modal logic formula, and
will return whether this formula is provable or not. That is, it will negate
the input formula, and test whether this negation is satisfiable.

There are several further options:
-v              Output verbose basic statistics about the internal workings of the program.

The prover accepts formulae in the following syntax:

fml ::= '(' fml ')'                     ( parentheses )
        | 'True'                        ( truth )
        | 'False'                       ( falsehood )
        | '~' fml                       ( negation )
        | '<' id '>' fml | '<>' fml     ( diamonds )
        | '[' id ']' fml | '[]' fml     ( boxes )
        | fml '&' fml                   ( conjunction )
        | fml '|' fml                   ( disjunction )
        | fml '=>' fml                  ( implication )
        | fml '<=>' fml                 ( equivalence )
        | id                            ( classical literals )

where identifiers (id) are arbitrary nonempty alphanumeric sequences
(['A'-'Z' 'a'-'z' '0'-'9']+)

## Running the benchmarks
There are several benchmark sets that are available in the 'benchmarks' folder. The problem
 files ending with .k are located within each subfolder and are represented in the correct
syntax.

A benchmark testing script can be run as following:
        * python3 src/prover/testing/testing_prover.py

## Authors

* **Darren Lawton**

## License

This project is licensed under the MIT License

## Acknowledgments

* Professor Rajeev Gore
```