A SAT-based Theorem Prover for Modal Logic

Tian Luan

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honour)
The Australian National University

June 2021

Final Version – 9 June 2021

Final Version – 9 June 2021

Except where otherwise indicated, this thesis and all work are my own original work.

Tian Luan
9 June 2021

to my family, my supervisor Professor Rajeev Gore

Final Version – 9 June 2021

Acknowledgments

I would like to thank my supervisor, Professor Rajeev Gore, for giving me the opportunity to research this project and for always being there to help me when problems arose, whether they were simple or complex, and for his patience in giving helpful advice and guiding me through the research. I would also like to thank my family and friends, whose support has motivated me to keep going.

Abstract

We present a theorem prover that uses the SAT-solver approach to automatic reasoning on satisfiability problems in modal logic K. This is achieved by converting the problem of modal logic into a classical logic part and a modal logic part that can be dealt with using additional transitions. For this purpose, we also implement algorithms for converting a formula of modal logic K into a set of modal clauses. When we compare our theorem provers with extant state-of-the-art theorem provers, we find that there is still a relative performance gap, but our prover has excellent potential. We analyse the performance of the theorem provers and make a hypothesis, and evaluate this hypothesis. We also propose future extensions to our research, where our research can be extended to multimodal logic, and we can also use a lower-level programming language to improve the performance; improving the algorithms in our study is also a possible research direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Description of Problem

When we have some knowledge, we can draw conclusions, make inferences or construct explanations based on that knowledge; this process is known as reasoning. Logical reasoning can force one to think about the relationships between facts; through logical reasoning, we can demonstrate how we can draw conclusions from known knowledge, which is often very rewarding [Harrison, 2009].

Automated reasoning is a branch of computer science that focuses on applying logical forms of reasoning to computational systems. An automated reasoning system should automatically make logical inferences toward the conclusion with a set of propositions, which we call assumptions [Portoraro, 2019]. Today automated reasoning is used in various applications, such as proving mathematical theorems, supporting reliable software development, and as an inference engine for AI systems [Sutcliffe and Suttner, 1998]. In a more practical sense, automated reasoning can detect and correct errors in proofs, automatically find new proofs, and find bugs in software development or improve software performance. For AI systems, it can also be used in general problem solver and robot planning Constable and Kreitz

. Modal logic is an extension of classical propositional logic, which includes modalities that express possibilities and necessities Garson [2021]. Strictly speaking, modal logic is exactly the study of the deductive behaviour of these two modalities. Modal logic is widely used in the philosophy of language, epistemology, metaphysics, and formal semantics [Sider, 2010]; it has also played an essential role in game theory, moral and legal theory, web design, and multiverse-based set theory and social epistemology [Van Benthem et al., 2010; Hamkins, 2012; Baltag et al., 2019].

In philosophical logic, many studies use modal logic to represent beliefs, desires, and mental attitudes. AI has been attempting to simulate human intelligence by constructing a way that can be implemented by artificial systems, especially computer-based systems, which means that formalization of mental attitudes such as belief and desires is necessary [Meyer and Veltman, 2007]. So, researchers will naturally use modal logic to represent these mental attitudes, and of course, there are many other ways to represent these attitudes, but this could also show why the AI community generally considers modal logic to be a valuable tool [Meyer and Veltman, 2007].

Automated theorem proving is a subfield of automated reasoning that focuses on using computer programs to prove mathematical theorems automatically [Knorr, 2018], or more narrowly, to use computers to determine the validity of a logical formula. A logical formula is valid if any assignment to its items makes the formula true, and it is invalid if there exists an assignment that makes the logical formula false. Automatic theorem proving has provided a significant impetus for the development of computer science; in terms of commercial applications, it has mainly provided the design and verification of integrated circuits [Loveland, 2016; Gallier, 2015; Wos et al., 1984]. In this area, recent research has focused on the satisfiability modulo theories (SMT) problem, which determines whether a logical formula is satisfiable [Barrett and Tinelli, 2018; Kaminski and Tebbi, 2013].

In order to describe more clearly what the validity of a logical formula is, we have given the following example, which comes from Versatile Math [Hartley], consider the following statement "If my computer crashes, then I will lose all my photos. I have not lost all my photos, so my computer has never crashed." We can determine whether the above statement is valid by using the truth table as follows steps, note that we are using classical propositional logic here, which will be defined in detail in Chapter 2 Basics of Modal Logic:

We can derive the following premises from the above statements:
$p \to q$: If my computer crashes, then I will lose all my photos.
$\neg q$: I have not lost all my photos.
We can derive the following conclusion from the above statements:
$\neg p$: My computer has never crashed.
Then we will have following logical formula which can represents the statements:
$((p \to q) \wedge \neg q) \to \neg p$

Table 1.1: Truth table of $((p \to q) \wedge \neg q) \to \neg p$

p	q	$\neg p$	$\neg q$	$p \to q$	$(p \to q) \wedge \neg q$	$((p \to q) \wedge \neg q) \to \neg p$
True	True	False	False	True	False	True
True	False	False	True	False	False	True
False	True	True	False	True	False	True
False	False	True	True	True	True	True

We can see from the table above that every case makes the formula $((p \to q) \wedge \neg q) \to \neg p$ true, so that means this formula is valid. This validity can also be derived from its satisfiability. A logical formula is satisfiable (SAT) if there is at least one assignment to its items that makes the whole formula true and is unsatisfiable (UNSAT) if none of the assignments can make the logical formula true. So the above process of determining validity through the truth table can also be determined simply and efficiently by the following steps: check whether $\neg A$ is satisfiable, and if $\neg A$ is satisfiable, there is a situation such that A is false, and then A is not valid. If $\neg A$ is unsatisfiable, no one situation makes A false, and obviously, A is valid. The

above steps allow us to convert the main problem of determining whether a formula A is provable into a problem of studying the satisfiability of formula $\neg A$, which we called the Boolean satisfiability problem. There are now two main approaches to solving the Boolean satisfiability problem: through semantic tables, the classical approach solves satisfiability problems. Another approach uses Satisfiability solvers (SAT-solvers). The second approach has developed rapidly in recent years, and many efficient and fast SAT-solvers have been developed, but all SAT solvers are designed for classical logic only.

1.2 Project Goals and Achievements

The project's main goal was to develop an efficient, simple and scalable theorem prover for modal logic K by using a high-performance SAT-solver. In their work intuit [Claessen and Rosén, 2015], Claessen and Rosén propose a new approach to solving intuitive propositional logic, one that involves an incremental SAT-solver, which we will cover in more detail in Chapter 3 Implementations, while they suggest that this approach can be extended to modal logic and they have not done this work yet. They inspire our theorem prover, and this theorem prover is to determine whether a formula of modal logic for axiom system K is provable by using a recursive algorithm and using an incremental SAT-solver. The work of Fiorentini, Goré and Graham-Lengrand presented a recursive version of intuit that proved to be very efficient [Fiorentini et al., 2019], so the central part of our algorithm mimics this algorithm and recursively using incremental SAT-solvers to determine the validity of the formulas for the modal logic K.

Claessen and Rosén's work [Claessen and Rosén, 2015] has mentioned the key idea of this algorithm, which is to convert a modal logic formula into three modal clause forms. Goré and Nguyen [Goré and Nguyen, 2009] also mention this process in their work; they proposed that a set of formulas can be converted into five types of modal clauses within quadratic time in any modal logic L. These five types of modal clause are essentially the same as the three type clauses proposed by Claessen and Rosén. In this project, we will use the modal clause form proposed by Claessen and Rosén, and in a future part, we will introduce the modal clause form in more detail. Therefore, based on the work of Gore and Nguyen [Goré and Nguyen, 2009], we modified certain parts to write an algorithm for modal clauses applicable to our project.

Our project also implements a parser that is important for the proof process. This parser converts the input into a postfix expression and transforms it into a special normal form called Negation Normal form to be covered details in Section 3.3 Negation Normal Form Algorithm Implementation, which is a prerequisite for conversion to a modal clause form.

Finally, we aimed to check that our implementation of the solver worked well and was meaningful relative to existing work, which is very important, so we compared the performance of our solver with some existing modal logic solvers such

as BDDTab [Goré et al., 2014], KSP Nalon et al. [2017] and Spartacus [Götzmann et al., 2010]. The tests were run on a standard benchmark [Balsiger et al., 2000], used different formulas under modal logic K and included results for satisfiable and unsatisfiable output, where the BDDTab benchmarking did not yield results due to a running bug when we run BDDTab, but the other tests were still relatively meaningful.

1.3 Thesis Outline

This article is divided into five chapters. The current Chapter 1 Introduction describes the problem that the research is primarily concerned with solving and the high-level motivation to solve it and provides an overview of automatic reasoning, automated theorem proving, and modal logic.

The second Chapter 2 Basics of Modal Logic gives a detailed description of modal logic, covering the syntax and semantic of modal logic, and also presents some related work on modal logic, including calculus and some other SAT-based solvers.

The third Chapter 3 Implementations presents a concrete implementation of the problem solving, including techniques on how to parse and convert a modal logic K formula into a Negation Normal Form, How to pre-process formulas to generate modal clauses, and the final algorithm for solving the main problem.

Chapter 4 Results provides an evaluation of our implemented solvers and compares the results with extant state-of-the-art solvers.

The final Chapter 5 Conclusion describes the possibilities and options for further expansion of our work in this area. It draws conclusions from the whole study process and results.

Chapter 2

Basics of Modal Logic

The concept of modality goes beyond mere truth or falsity; it embeds what we say or think in a larger conceptual space and refers to what could be or has been, should be or has been, or still could be. Modal expressions occur in natural language in various ways, ranging from necessity, possibility, and contingency to expressions of time, action, change, causation, information, knowledge, belief, commitment, permission, and so on. Modal logic is a general study of the formulation and reasoning about these concepts. Modal logic allows us to enhance our expressiveness while still maintaining the ability to determine truth values. We can say that Modal logic plays a very large role in artificial intelligence in many fields including philosophy of language, epistemology, metaphysics and formal semantics [Garson, 2021; Sider, 2010; Van Benthem et al., 2010; Hamkins, 2012; Baltag et al., 2019; Meyer and Veltman, 2007].

In this chapter, we first introduce the syntax of modal logic in Section 2.1 Syntax, formally defining how to form a modal logic formula. In Section 2.2 Semantics, we describe the semantics used by modal logic, which the Kripke model defines. Finally, in Section 2.3 Related Work, we present some existing work on theorem provers using modal logic. Please note that some examples and the definitions in this chapter are taken from Goré's work [Goré, 1999].

2.1 Synatx

First, we define the classical propositional logic as follows, this definition is from Achourioti [Achourioti]. Classical propositional logic consists of:

Propositional variables: $p, q, r \cdots$
Logical operators: $\neg$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
Parentheses: (,)

And the well-formed formula (wff) of classical propositional logic is recursively defined as:

1. A propositional variable is a wff.

2. If ϕ and ψ are wffs, then $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi)$ are wffs.
3. Nothing else is a wff.

Modal logic is an extension of classical propositional logic. In addition to using the notation of classical propositional logic, modal logic uses two unary connectives, $\Box$ (box) and $\Diamond$ (diamond). Box denotes a necessity modality, and diamond denotes a possibility modality; in more general terms, the Box operator means 'it is necessary that' while the Diamond operator means 'it is possible that '. Thus a formula of modal logic can be expressed recursively in the following form, noting that p is expressed here as any element from a countably infinite set of atomic propositions, and ϕ, ψ possibly with subscripts stand for arbitrary formulae (including atomic ones). We define modal logic as follows, here the definition and examples describe the modal logic syntax are from Goré [Goré, 1999]:

Atomic Fomulae: $p ::= p_0 \mid p_1 \mid p_2 \mid \cdots (Atm)$
Fomulae: $\phi ::= p \mid \phi \mid \Box\phi \mid \Diamond\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi (Fml)$

Here are some examples of modal logic formulas:

Examples: $\Box p_0 \rightarrow p_2$ $\Box p_3 \rightarrow \Box\Box p_1$ $\Box(p_1 \rightarrow p_2) \rightarrow ((\Box p_1) \rightarrow (\Box p_2))$

2.2 Semantics

The semantics of modal logic is defined using the Kripke Model, and similar to the use of truth tables in propositional logic, modal logic is evaluated using the Kripke Model. Following definitions and examples are from Goré [Goré, 1999].

In order to express clearly what a Kripke Model is, it is important first to introduce the most basic notion of a Kripke Frame. Kripke Frame is a pair $\langle W, R\rangle$, where W is a non-empty set of points/worlds/vertices and $R \subseteq W \times W$ is a binary relation over W. We need to introduce another concept about how to evaluate in Kripke Frame. We can use a map $\vartheta : W \times Atm \mapsto t, f$ which called valuation to determine the truth value (t or else f) of every atomic formula at every worlds. Based on the above definitions, Kripke Model is defined as extending Kripke Frame by a valuation function ϑ: $\langle W, R, \vartheta\rangle$. If we have a Kripke model $\langle W, R, \vartheta\rangle$ and some worlds $\omega \in W$, we can determine the truth value of a formula by recursively using ϑ function on its shape.
In modal logic, following evaluations are used to determine the truth value.

For classical connectives:

$$\vartheta(\omega, \neg\phi) = \begin{cases} t \;\; if & \vartheta(\omega, \phi) = f \\ f \;\; otherwise & \end{cases}$$

$$\vartheta(\omega, \phi \wedge \psi) = \begin{cases} t \ \ if & \vartheta(\omega,\phi) = t \ and \quad \vartheta(\omega,\psi) = t \\ f \ \ otherwise \end{cases}$$

$$\vartheta(\omega, \phi \vee \psi) = \begin{cases} t \ \ if & \vartheta(\omega,\phi) = t \ or \quad \vartheta(\omega,\psi) = t \\ f \ \ otherwise \end{cases}$$

$$\vartheta(\omega, \phi \rightarrow \psi) = \begin{cases} t \ \ if & \vartheta(\omega,\phi) = f \ or \quad \vartheta(\omega,\psi) = t \\ f \ \ otherwise \end{cases}$$

For modal connectives:

$$\vartheta(\omega, \Diamond\phi) = \begin{cases} t \ \ if \ some & v \in W \ has \quad \omega R v \ and \quad \vartheta(v,\phi) = t \\ f \ \ otherwise \end{cases}$$

$$\vartheta(\omega, \Box\phi) = \begin{cases} t \ \ if \ every & v \in W \ with \quad \omega R v \ has \quad \vartheta(v,\phi) = t \\ f \ \ otherwise \end{cases}$$

From the above semantics we can see that classical connectives are truth functional, but for modal connectives, truth of modalities depends on underlying R, which means it is not truth functional. And that is why modal logic has additional expressiveness. Here is an example of Kripke Model of Modal Logic from Goré [Goré, 1999]:

Example: If $W = \omega_0, \omega_1, \omega_2$ and $R = (\omega_0, \omega_1), (\omega_0, \omega_2)$ and $\vartheta(\omega_1, p_3) = t$ then $\langle W, R, \vartheta \rangle$ is a Kripke model as pictured below:

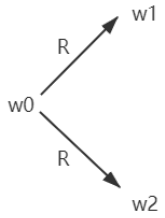

Figure 2.1: Kripke Model Example

In this example some valuations are:

$$\vartheta(\omega_0, \Diamond p_3) = t$$
$$\vartheta(\omega_0, \Box p_3) = f$$
$$\vartheta(\omega_1, \Box p_1) = t$$
$$\vartheta(\omega_1, \Box \neg p_1) = t$$
$$\vartheta(\omega_0, \Diamond \Box p_1) = t$$

Since the truth values of modal logic connectives are dependent on the underlying R, modal logic also introduces semantic force relations. Let K be the class of all Kripke models, and $M = \langle W, R, \vartheta \rangle$ a Kripke model. Let κ be the class of all Kripke frames and let $\mathscr{F}$ be a Kripke frame. Let Γ be a set of formulae, and ϕ be a formula. We have the following table which from Goré's work [Goré, 1999].

Table 2.1: Semantic Forcing Relation

Forces	We say	We write	When	$\bullet \nVdash \phi$
in a world	w forces ϕ	$w \Vdash \phi$	$\vartheta(w, \phi) = t$	$\vartheta(w, \phi) = f$
in a model	M forces ϕ	$M \Vdash \phi$	$\forall w \in W.w \Vdash \phi$	$\exists w \in W.w \nVdash \phi$

Classicality: either $\bullet \Vdash \phi$ or else $\bullet \nVdash \phi$ holds for $\bullet \in \{w, M, \mathscr{F}\}$
Let $\bullet \Vdash \Gamma$ stand for $\forall \phi \in \Gamma.\bullet \Vdash \phi$ ($\bullet \in \{w, M, \mathscr{F}\}$)

And Goré also define Logical Consequence, Validity and Satisfiability by the following definitions [Goré, 1999]:

Logical Consequence: $\Gamma \Vdash \phi$ iff $\forall M \in K.M \Vdash \Gamma \Rightarrow M \Vdash \phi$
Validity: ϕ is K-valid iff $\emptyset \Vdash \phi$
Satisfiability: ϕ is K-satisfiable iff $\exists M = \langle W, R, \vartheta \rangle \in K, \exists w \in W, w \Vdash \phi$

2.3 Related Work

There are now many existing theorem provers for proving theorems in modal logic K. There are two main approaches on which they have based: the traditional tableau-based methods, which are widely used in various inference systems to determine the satisfiability of formulas. Many state-of-the-art tableau-based algorithms for modal logic K, such as FaCT++ [Tsarkov and Horrocks, 2006], are highly optimised for the corresponding logic in a very targeted way and therefore have outstanding performance.

Another approach is based on the rapidly evolving SAT-solvers, such as InKer-SAT [Kaminski and Tebbi, 2013], which uses incremental SAT-solvers to solve the satisfiability problem for modal logics. In the work of Goré, Olesen and Thomson [Goré et al., 2014] proposed the use of binary decision diagrams (BDDs) instead of DPLL and constructed the BDDTab system, which can efficiently perform theorem proving in modal logic.

There are many other ways of solving modal logic formulas. One such method is the Automata-theoretic technique, which solves the satisfiability problem by the following process: construct an automaton for a given formula of modal logic K accepts some trees if and only if the formula is satisfiable. At the heart of this approach is a class of finite automata on infinite trees [Vardi and Wolper, 1986].

The resolution-based method is also a very effective approach. Nivelle, Schmidt, and Hustadt describe how resolution methods can be applied to modal logic by taking the standard translation approach and considering different resolution methods to provide decision procedure for the clauses. Ordered resolution and selection-based resolution are used in their procedure to solve multimodal logic problems [De Nivelle et al., 2000].

In Voronkov's work, a bottom-up decision procedure is proposed for modal logic K based on an "inverted" version of a sequent calculus. Their procedure introduces a new technique based on the analysis of tableau-based derivations in K for proving redundancy criteria; they also introduce a method based on so-called traces to formalize the inverse method. Their work also proves to be a very effective method for solving the satisfiability problem for modal logic K [Baader and Tobies, 2001].

Chapter 3

Implementations

In this chapter, the main focus of the description is on the whole design and implementation of the SAT-based modal logic K theorem prover, including the choice of language and library and the reason; description of the input format; construction of the parser and the required parsing form; algorithms for Negation Normal Form transformations and for generating modal clauses; main algorithms for formal proofs of modal logic K theorems. To present these algorithms clearly, we not only give the pseudocode of the algorithms in the following sections, but we also give some examples to describe the algorithms' operations.

In Section 3.1 Programming Language and Libraries, we focused on how the programming language was chosen, describing the advantages and disadvantages of the selected programming language and explains the reasons for choosing the language. In Section 3.2 Input Format and Parsing Algorithm Implementations, we describe in detail the input forms accepted by our solver and tell how to parse the formulas of modal logic K into the postfix form. In Section 3.3 Negation Normal Form Algorithm Implementation, we describe how the parsed formulas are transformed into Negation Normal Forms (NNF), which is a prerequisite for the next step of modal clausification. Section 3.4 Modal Clausification Algorithm Implementation provides a detailed description of modal clauses, describing how we convert formula in NNF form to a set of modal clauses. Section 3.5 Modal Logic Algorithm Implementation Implementation is the implementation of the core algorithm, describing how the algorithm verifies the validity of the modal logic K formulas.

3.1 Programming Language and Libraries

For the choice of language for this development project, we considered the following points: As our modal logic K-theorem provers are based on SAT-solvers, it is of utmost importance that the language should provide an interface to use high-performance SAT-solvers. Secondly, the performance of the language also needs to be taken into account, as a crucial criterion for evaluating our theorem prover is speed. High-level languages do bring many operational convenience advantages, but this is often at the expense of performance, especially compared with more low-level languages.

The SAT-solver we chose is MiniSat; MiniSat is a minimalistic, open-source SAT-solver developed to help researchers and developers get started on SAT. It is easy to modify, design for integration, and most importantly, it is very efficient. We have therefore chosen it as the SAT-solver on which our algorithm relies.

Considering that our chosen language needs to be able to call MiniSat [Sörensson, 2010], we ultimately chose Python 3.7 as the development language. It is an interpretable, object-oriented high-level development language with many libraries. Python's interpretability as a high-level language causes its slow execution. So it isn't easy to optimise algorithms to increase speed in Python, which will cause us to sacrifice some performance. Nonetheless, Python has some advantages. Simplicity is certainly an advantage of Python, as we can usually do more with less code in it. The main reason that motivated us to choose Python was the many extension libraries that it offers. In particular, we noticed that Python provides an API to the MiniSat solver that we will be using.

The only additional library we used in our code was the PySat library, and the rest was written manually. PySat is the library that provides an API to MiniSat. It is a Python (2.7, 3.4+) toolkit, aiming to provide a simple and unified interface to several state-of-art Boolean satisfiability (SAT) solvers and a variety of cardinality and pseudo-Boolean encodings.

3.2 Input Format and Parsing Algorithm Implementations

Our theorem provers are designed to prove theorems of modal logic K. More explicitly, we wish to prove the validity of an formula of modal logic K. This is done by refutation, i.e. we prove that ϕ is valid by showing that $\neg\phi$ of is unsatisfiable, so at the beginning of the whole proof process, we need to negate the whole formula.

The first thing to introduce is the format of input that our program can accept. Obviously, when the input is wrong, the program will not get the correct answer. Here in our program the format of the input accepted by the program is as follows:

Set of propositions: $\{p1,\ p2,\ p3\cdots\}$
Proposition connectives and parentheses: $\{|,\ \&,\ ->,\ <->,\ -,\ (,\ )\}$
Modal connectives: $\{[],\ <>\}$
Please note that the parentheses must appear in pairs otherwise the parsing process will have error.

In our program, for classical propositional connectives, we use $|$ to represent $\vee$, $\&$ to represent $\wedge$, $->$ to represent $\rightarrow$, $<->$ to represent $\leftrightarrow$, $-$ to represent $\neg$. While for modal operators, we use $[]$ to represent $\Box$ and $<>$ for $\Diamond$. It is very important that the input formula contains the correct form so that our program can process it correctly.

Before checking whether a formula of Modal Logic K is valid, we have to convert the formula into a form that the program can handle. For a modal logic formula ϕ,

we represent it as a postfix notation string, also known as an abstract tree (AST). The reason for choosing to convert the formula ϕ to a postfix notation string is that in this way, the computer can handle different levels of symbols, and the postfix structure allows us to do some operations more efficiently by using data structures such as stacks. The following examples show the correct input form of some modal logic formulas and the parsed form of them.

Example 1: Consider the formula ϕ is $\Box(p_1 \rightarrow p_2)$:
(1)The correct input string for this formula should be: [](p1->p2)
(2)The parser will output: [p1,p2,->,[]]

Example 2: Consider another formula ϕ is $\Box(p_1 \wedge p_2) \rightarrow (\Diamond p_3 \vee p_4)$:
(1)The correct input string for this formula should be: [](p1&p2)-> (<>p3|p4)
(2)The parser will output: [p1,p2,&,[],p3,<>,p4,|,->]

As the above examples show, we convert the input of the correct form into a postfix expression. The shunting-yard algorithm achieves this parsing process, which is a method for parsing mathematical expressions in infix notation. It can produce a postfix notation string, known as Reverse Polish notation (RPN), or an abstract syntax tree (AST). Edsger Dijkstra invented the algorithm, and It is named the "shunting yard" algorithm because it is executed much like a railroad shunting yard.

Note that the traditional sense of the shunting-yard algorithm is designed for mathematical expressions, which means that it cannot handle unary operators such as $\neg$, and cannot deal with modal operators $\Box$ and $\Diamond$. To solve this problem, Andrew Wei have modified the shunting-yard algorithm to handle these exceptional cases and here we use this new version of shunting-yard algorithm [Wei]. In this study, we focus only on prefixed unary operators, so postfixed unary operators such as "!, ++, –" will not be discussed. The detailed pseudo-code for the shunting-yard algorithm is shown below:

Algorithm 1 shunting-yard$(String)$

```
1:  for token in String do
2:      if token is an atom then
3:          Push token onto output
4:      else if token is an unary operator then
5:          Push token onto stack
6:      else if token is a binary operator then
7:          while stack contains an operator that has equal/higher precedence than
    token do
8:              Pop stack and push the popped item onto output
9:          end while
10:         Push token onto stack
11:     else if token is a parenthesis then
```

```
        if lef-parenthesis ( then
            Push token onto stack
        else
            while the top of stack is not a left-parenthesis do
                Pop each item from stack and push each popped item onto output
            end while
            Pop stack
        end if
    end if
end for
Pop remainder of stack and push each popped item onto output
return output
```

3.3 Negation Normal Form Algorithm Implementation

The next step is to convert the input to a Negation Normal Form, which is a prerequisite for the algorithm that generates modal clauses. Goré and Nguyen's work on generating modal clauses proposed in Clausal Tableaux for Multimodal Logics of Belief mentioned this prerequisite [Goré and Nguyen, 2009]. Negation Normal Form means the negation operator ($\neg$) is only applied to variables and for other Boolean operators, the only allowed are conjunctions ($\wedge$) and disjunctions ($\vee$). In both classical and modal logic, each formula can be converted to this form by the following process: eliminating the implication and using De Morgan's laws to push the negation inwards, and finally, eliminating the double negation. The process can be represented as the following rules, please note that in this case $\Rightarrow$ actually means $\rightarrow$:

For propositional connectives:

$$\begin{aligned} A \Rightarrow B &\rightarrow \neg A \vee B \\ A \Leftrightarrow B &\rightarrow (\neg A \wedge \neg B) \vee (A \wedge B) \\ \neg(A \vee B) &\rightarrow \neg A \wedge \neg B \\ \neg(A \wedge B) &\rightarrow \neg A \vee \neg B \\ \neg\neg A &\rightarrow A \end{aligned}$$

For modal connectives:

$$\begin{aligned} \neg\Box A &\rightarrow \Diamond\neg A \\ \neg\Diamond A &\rightarrow \Box\neg A \end{aligned}$$

Here are some examples of converting the formula of modal logic K to a Negation Normal Form that can help explain the process better.

Example 1: Consider the formula ϕ is $\neg\Box(p_1 \to p_2)$:
First eliminate the implication, the formula ϕ would be $\neg\Box(\neg p_1 \vee p_2)$
Then push the negation inwards, the formula ϕ would be $\Diamond(\neg\neg p_1 \wedge \neg p_2)$
Finally eliminate the double negation, the formula ϕ would be $\Diamond(p_1 \wedge \neg p_2)$

Example 2: Consider another formula ϕ is $\Box(p_1 \wedge p_2) \to (\Diamond p_3 \vee p_4)$:
First eliminate the implication, the formula ϕ would be $\neg\Box(p_1 \wedge p_2) \vee (\Diamond p_3 \vee p_4)$
Then push the negation inwards, the formula ϕ would be $\Diamond(\neg p_1 \vee \neg p_2) \vee (\Diamond p_3 \vee p_4)$
Finally eliminate the double negation, the formula ϕ would remain unchanged.
In the actual implementation of the algorithm, we implement this conversion algorithm by recursively converting the formula of modal logic K to a Negation Normal Form, with the following detailed pseudocode.

Algorithm 2 NNF(I)

if I is a variable **then**
    return I
end if
if I has the form $\phi \leftrightarrow \psi$ **then**
    return $(\text{NNF}(\neg\phi) \wedge \text{NNF}(\neg\psi)) \vee (\text{NNF}(\phi) \wedge \text{NNF}(\psi))$
end if
if I has the form $\phi \to \psi$ **then**
    return $\text{NNF}(\neg\phi) \vee \text{NNF}(\psi)$
end if
if I has the form $\neg\phi$ **then**
    if ϕ is a variable **then**
        return $\neg\phi$
    else if ϕ has the form $\neg\psi$ **then**
        return $\text{NNF}(\psi)$
    else if ϕ has the form $\omega \wedge \psi$ **then**
        return $\text{NNF}(\neg\omega) \vee \text{nnf}(\neg\psi)$
    else if ϕ has the form $\omega \vee \psi$ **then**
        return $\text{NNF}(\neg\omega) \wedge \text{nnf}(\neg\psi)$
    else if ϕ has the form $\Box\psi$ **then**
        return $\Diamond\text{NNF}(\neg\psi)$
    else if ϕ has the form $\Diamond\psi$ **then**
        return $\Box\text{NNF}(\neg\psi)$
    else if ϕ has the form $\psi \to \omega$ **then**
        return $\text{NNF}(\psi) \wedge \text{NNF}(\neg\omega)$
    else if ϕ has the form $\psi \leftrightarrow \omega$ **then**
        return $(\text{NNF}(\neg\psi) \vee \text{NNF}(\neg\omega)) \wedge (\text{NNF}(\psi) \vee \text{NNF}(\omega))$
    end if
end if
if I has the form $\Box\phi$ **then**

30: **return** $\Box\text{NNF}(\phi)$
31: **else if** I has the form $\Diamond\phi$ **then**
32: **return** $\Diamond\text{NNF}(\phi)$
33: **else if** I has the form $\phi \vee \psi$ **then**
34: **return** $\text{NNF}(\phi)\vee\text{NNF}(\psi)$
35: **else if** I has the form $\phi \wedge \psi$ **then**
36: **return** $\text{NNF}(\phi)\wedge\text{NNF}(\psi)$
37: **end if**

3.4 Modal Clausification Algorithm Implementation

In this section, we will follow some of the definitions proposed in the work of Goré and Nguyen. First, a classical literal is a primitive proposition p or its negation($\neg p$), and we will use a, b, c to represent classical literal. At the same time, we have a new definition $\boxdot$, a $\boxdot$ can denote a sequence of box operators with the possibility of being empty, which we call $\boxdot$ modal context. $\boxdot[\phi_1, \cdots, \phi_k]$ denotes $\boxdot[\phi_1 \vee \cdots \vee \phi_k]$, which shows that the order of disjunctions does not matter and we can ignore them.

In Goré and Nguyen's work on Clausal Tableaux for Multimodal Logics of Belief they propose that in any modal logic L, every set X of formulas can be converted into a set of modal clauses X', and they also give the procedure for this transformation process [Goré and Nguyen, 2009]. They define modal clause as the following forms: $\bot$, $\boxdot[a_1, \cdots, a_k]$, $\boxdot[a, \Box_i b]$, $\boxdot[a, \Diamond_i b]$, or $\boxdot\Diamond_i b$. In our study, since we are working on modal logic Rather than multimodal logic, our $\Box$ and $\Diamond$ operators will not have subscripts.

In Classen and Rosén's work on SAT Modulo Intuitionistic Implications [Claessen and Rosén, 2015], they considered that the idea of theorem provers they constructed for intuitionistic logic could be generalized to modal logic. For this purpose, they propose that all modal logic formulae should be transformed into a set of modal clauses of the following form: $\Box p$, $\Box(a \rightarrow \Box p)$, $\Box(a \rightarrow \Diamond p)$.

In our study, we have chosen to use the type of modal clause defined in Classen and Rosén's work [Claessen and Rosén, 2015], while we have further expressed the type of modal clause based on the definition proposed in Goré and Nguyen's work [Goré and Nguyen, 2009]. We define our modal clauses as following three types, and please note that the definitions are not in the form of NNF because they contain implications ($\rightarrow$):

Type 1 Clause: $\boxdot(p)$ where p is prmitive proposition

Type 2 Clause: $\boxdot(p \rightarrow \Box q)$ where p, q are prmitive propositions

Type 3 Clause: $\boxdot(p \rightarrow \Diamond q)$ where p, q are prmitive propositions

We convert a modal logic K ϕ that has been converted to NNF in the process of the previous sections to a modal clause by repeatedly applying some of the follow-

ing rules which were proposed by Goré and Nguyen [Goré and Nguyen, 2009]:

1. Replace the formula of the form $\boxdot(\phi \wedge \psi)$ with $\boxdot\phi$ and $\boxdot\psi$.

2. Replace the formula of the form $\boxdot[\phi_1, \cdots, \phi_k, \psi \vee \xi]$ with $\boxdot[\phi_1, \cdots, \phi_k, \psi, \xi]$.

3. Replace the formula of the form $\boxdot[a, \phi \wedge \psi]$ with $\boxdot[a, \phi]$ and $\boxdot[a, \psi]$.

4. If the formula has the form $\boxdot[a_1, \cdots, a_h, \phi_1, \cdots, \phi_k]$, where $\phi_1, \cdots, \phi_k$ are not classical literals, and k $\geq$ 2, or k = 1 and h > 1, then we add new primitive propositions $p_1, \cdots, p_k$ and use $\boxdot[a_1, \cdots, a_h, p_1, \cdots, p_k], \boxdot[\neg p_1, \phi_1], \cdots, \boxdot[\neg p_k, \phi_k]$ to represent the formula.

5. If the formula has the form $\boxdot[a, \nabla_i\phi]$, where ∇_i is either $\Box_i$ or $\Diamond_i$, and ϕ is not a classical literal, then we add new primitive poposition p and use $\boxdot[a, \nabla_i p]$ and $\boxdot\Box_i[\neg p, \phi]$ to represent the original formula.

We modified the above transformation process to make it work for the types of modal clauses we define, with the following modified pseudocode:

Algorithm 3 modalClausification

```
1:  procedure modalClausification(X)
2:      // List is a global parameter
3:      if X is a classical literal then
4:          return X
5:      end if
6:      if X has the form ⊡(φ ∨ ψ) then
7:          tmp_φ ← (modalClausification(φ)
8:          tmp_ψ ← (modalClausification(ψ)
9:          if tmp_φ and tmp_ψ are not list then
10:             return ⊡(tmp_φ ∨ tmp_ψ)
11:         else if tmp_φ is a list and tmp_ψ is not a list then
12:             res = ∅
13:             for i in tmp_φ do
14:                 add ⊡(i ∨ tmp_ψ) to res
15:             end for
16:             return res
17:         else if tmp_ψ is a list and tmp_φ is not a list then
18:             res = ∅
19:             for i in tmp_ψ do
20:                 add ⊡(i ∨ tmp_φ) to res
21:             end for
22:             return res
23:         else
```

```
24:             res = ∅
25:             for i in tmp_φ do
26:                 for j in tmp_ψ do
27:                     add ⧈(i ∨ j) to res
28:                 end for
29:             end for
30:             return res
31:         end if
32:     else if X has the form ⧈(φ ∧ ψ) then
33:         tmp_φ ← (modalClausification(⧈(φ))
34:         tmp_ψ ← (modalClausification(⧈(ψ))
35:         res = ∅
36:         if tmp_φ is a list then
37:             add all items in tmp_φ to res
38:         else
39:             add tmp_φ to res
40:         end if
41:         if tmp_ψ is a list then
42:             add all items in tmp_ψ to res
43:         else
44:             add tmp_ψ to res
45:         end if
46:         return res
47:     else if X has the form ⧈(∇φ) where ∇ is either □ or ◇ then
48:         if φ is a literal then
49:             add new atom t
50:             add ⧈(t → ∇φ) to List
51:         else
52:             add new atom t and add new name p_φ to φ
53:             add ⧈(t → ∇p_φ) to List
54:             tmp_φ ← modalClausification(φ)
55:             if tmp_φ is a list then
56:                 for i in tmp_φ do
57:                     add ⧈□(p_φ → i) to List
58:                 end for
59:             else
60:                 add ⧈□(p_φ → tmp_φ) to List
61:             end if
62:         end if
63:         return ⧈(t)
64:     end if
65: end procedure
66: procedure modalClausificationMain(X)
67:     // List is a global parameter
```

68: $List = \emptyset$
69: $X = \text{MODALCLAUSIFICATION}(X)$
70: add $List$ to X
71: **return** X
72: **end procedure**

To show more clearly how we transform a formula of modal logic K into modal clauses, we give an example:
Example: Consider the formula $(\Box(p \to q) \to (\Box p \to \Box q)$
First we need to negate it and convert it to Negation Normal Form, this will be done by last section's algorithm and we will get: $\Box(\neg p|q)$ & $\Box p$ & $\Diamond\neg q$
Then because we have & in our formula, so we will so we split them up and do a separate recursive process for each clause:

1. $\Box(\neg p|q)$
Here we can regard this clause as $\boxdot(\neg p|q)$ and we do not need to do any more processing.
We can also regard this clause as $\boxdot(\Box(\neg p|q))$ because $\boxdot$ can be a empty sequence of $\Box$, in this situation we will get a new atom a_1 and rename the $(\neg p|q)$ part as a_2, and we will go through lines 52-61 of the algorithm to arrive at the following clauses: a_1, $a_1 \to \Box a_2$, $\Box(\neg a_2|\neg p|q)$

2. $\Box p$
Here we can regard this clause as $\boxdot(p)$ and we do not need to do any more processing.
We can also regard this clause as $\boxdot(\Box p)$, in this situation we will go through lines 48-50 of the algorithm to add a new atom a_3 and then arrive at the following clauses: a_3, $a_3 \to \Box p$

3. $\Diamond\neg q$
We will go through lines 48-50 of the algorithm to add a new atom a_4 and then arrive at the following clauses: a_4, $a_4 \to \Diamond \not p$
So at last we will get a set of modal clauses, this set is: [a_1, a_3, a_4, $a_1 \to \Box a_2$, $\Box(\neg a_2|\neg p|q)$, $a_3 \to \Box p$, $a_4 \to \Diamond\neg p$]

3.5 Modal Logic Algorithm Implementation

Our theorem prover of modal logic K is based on an SAT-solver and follows a Satisfiability-Modulo-Theories (SMT) approach. The fact that the SMT-solving work can only be applied to classical logic with first-order theories means that we cannot use it directly on Modal Logic.

In Claessen and Rosén's work [Claessen and Rosén, 2015], they suggest that this approach is relevant to intuitionistic logic, and they also suggest that the SMT approach can also be applied to modal logic by using an incremental SAT-solver.

Our work is based on this idea, and the whole proving process can be described as a special case of the Counter-Example Guided Abstraction Refinement (CEGAR) loop [Clarke et al., 2000]. CEGAR loop is an automatic iterative abstraction-refinement methodology. The initial abstract model is generated by automatic analysis. This abstraction model could admit erroneous counterexamples and refine the abstract model by analyzing the counterexamples. The key point of our prover is that the modal logic is, in fact, an extension of the classical logic, which means that we can use as efficient an SAT-solver as possible for the classical part of the logic and use transition rules to handle the rest of the modal part.

In Fiorentini, Goré and Graham-Lengrand's A proof-theoretic perspective on SMT-solving for intuitionistic propositional logic paper, they propose a recursive version of intuit which proved to be very efficient, so that our theorem provers chose to learn this recursive version of the algorithm instead of the one in the original intuit [Fiorentini et al., 2019].

The core of our ModalSat theorem prover is the use of an existing high-performance SAT solver, MiniSat, which was chosen mainly because of its high efficiency. In general, SAT-solvers (in this project MiniSat) contain APIs for some of the following operations:

$$\textit{procedure newSolver}()$$

$$\textit{procedure addClause}(s, r)$$

$$\textit{procedure satProve}(s, A, q)$$

The *procedure newSolver*() will create a new, unconstrained SAT-solver.
The *procedure addClause*(s, r) add a classical clause r to a SAT-solver s as a contraint.
The *procedure satProve*(s, A, q) try to use assumptions A and all clauses that have been added to s so far to prove the goal q. When we call *procedure satProve*(s, A, q) we will get one of two results:

–$No(M)$, if no proof could be found. Here, M is a model that is found by the SAT-solver represented as a set of true atoms. The model M is guaranteed to satisfy all added clauses, all assumptions, but it makes the goal q false. So, we know $A \subset M$ and $q \notin M$.
–$Yes(A')$, if a proof could be found. Here, A' is the subset of the assumptions that were actually used in the proof of q from the clauses. So, we know that $A' \in A$.

In our descriptions, we often refer to the SAT-solver as the incremental SAT-solver, which means that all clauses can be added to the SAT-solver but cannot be removed from it, and for different problems, we solve them utilizing different Assumptions. All clauses are therefore global and must hold throughout the whole proof.

From an overview, our algorithm follows an analytic tableaux process, where we construct a recursive depth-first search tree in a straightforward way to find if there is a model that satisfies the conditions, each node of the tree can be thought of as a world, and through modalities, we can transition to different worlds, and we will

check each world to ensure that there are no contradictions in that world. If there is a contradiction during the search, we will go back to the previous world and add the new constraints we have learned. Finally, if we can find a model that satisfies the conditions, then the model we have constructed is satisfiable (SAT), which means that the original formula of modal logic K is not valid. However, if we cannot find a model that satisfies the conditions at the end of the search, then the model is unsatisfiable (UNSAT), which means that the formula for the original modal logic K is valid. The above procedure allows us to determine the validity of the formula for a modal logic K. We will describe the whole algorithm in more detail in the following parts.

As a whole, our algorithm follows an analytic tableaux process, where we construct a recursive depth-first search tree in a straightforward way to find if there is a model that satisfies the conditions, each node of the tree can be thought of as a world, and through modalities, we can reach different worlds, and we will check each world forEach world is checked to ensure that there are no contradictions in that world. If there is a contradiction during the check, we go back to the previous world and add the new conditions we have learned, and finally, if we can find a model that satisfies the conditions, then the model we have constructed is satisfiable (SAT), which means that the original modal logic K is invalid. However, if we cannot find a model that satisfies the conditions at the end of the search, then the model is unsatisfiable (UNSAT), which means that the formula for the original modal logic K is invalid. The above procedure allows us to confirm the validity of the formula for a modal logic K. We will describe the whole algorithm in more detail next.

Here we need to use the concept of modal depth, which specifies "how far" we need to go in the Kripke Model when checking the validity of a modal logic formula. In more detail, in modal logic, we can only access the world with accessible relations through modalities, so modal depth actually refers to the distance of the transition we need to go when checking validity.

The algorithm first requires us to convert the formula of the modal logic K to be checked into a set of modal clauses, a step already described in the previous section, where we record the modal depth for each modal clause, which will be useful in the future for determining the worlds to which different clauses belong. Please note that the modal depths are calculated relative to the initial world.

Algorithm 4 shows the underlying part of our algorithm. Here we will add all the clauses belonging to the classical logic to the SAT-solver. This step will provide an initial world, and if there is a contradiction in the initial world, then the whole algorithm will be terminated, as it means that we cannot find any model that satisfies it. Otherwise, our initial world will be a model M returned by SAT-solver that satisfies all of our classical logic clauses. If a clause does not contain modal operators ($\Box$ or $\Diamond$) and the modal depth is 0, it means that the clause belongs to classical logic. The underlying part of the algorithm also makes a judgement on the final result returned. Here we look for classical logic clauses by determining the modal depth.

Algorithm 4 ModalSatMain

```
1: procedure ModalSatMain(X_0)
2:     // s_0, X_0 are global parameters
3:     s_0 ← newSolver()
4:     for φ ∈ X_0 do
5:         addClause(s_0, φ) such that φ = a_1 ∨ ··· ∨ a_n or φ is an atom
6:     end for
7:     r ← ModalSat(X_0, ∅, s_0, 0)
8:     if r = False then
9:         return SAT
10:    else
11:        return UNSAT
12:    end if
13: end procedure
```

Algorithm 5 shows how we use some transition rules to process the rest of the modal logic clauses; the parameter "*boxDepth*" represents the modal depth, which is a constant. In the first recursion, we process all the classical logic formulae mentioned before, which generates the initial world. Here we need to pay particular attention to a property of modalities: the $\Box$ operator does not force us to have a new world, whereas the $\Diamond$ operator forces at least one world to satisfy the requirement, so we can use the $\Diamond$ operator to transition into a new world.

Based on the initial world, we will have several type 3 clauses ($\Diamond$) that are "activated", we generate a new world for each "activated" type 3 clause and add to that world all the type 2 clauses that can reach this world and all type 1 clauses with the same depth as the current world. We would check the current world with a new SAT-solver, and if there is a contradiction, we have learned something new, and we will return to the previous world and add this knowledge as constraints to the SAT-solver and check again. For each world, we recursively check all type 3 clauses as in the above steps.

When we cannot find a satisfying model during the search process, we get the UNSAT result, and by refuting, we can finally conclude that the formula of the original modal logic K is valid. If the whole search process ends and we find a satisfying model, then we get the SAT result and eventually, the formula of the original modal logic K is not valid.

Algorithm 5 ModalSat

```
1: procedure ModalSat(X, A, s, boxDepth)
2:     r_0 ← satProve(s, A)
3:     if r_0 = Yes(A′) then
4:         return True, A′
5:     else
6:         for k_0 = ⧈[a → ◇d] ∈ X such that a ∈ M do // r0 = No(M)
7:             s_1 ← newSolver()
8:             B ← {d}
```

```
            for k_1 = ⧈[b → □e] ∈ X such that b ∈ M and the depth = boxDepth do
                B ← B ∪ {e}
            end for
            for k_2 = ⧈[a_1, ··· , a_n] such that the depth = boxDepth + 1 do
                addClause(s_1, a_1 ∨ ··· ∨ a_n)
            end for
            r_1 ← ModalSat(X, B, s_1, boxDepth + 1)
            if r_1 = True then
                for k_1 = ⧈[y → ◇z] or ⧈[y → □z] such that z ∈ A_1, k_1 ∈ X, y ∈ M,
    depth = boxDepth  do // r_1 = True, A_1
                    φ ← φ ∨ ¬y such that depth = boxDepth
                end for
                addClause(s, φ)
                r2 ← ModalSat(X, A, s, boxDepth)
            end if
        end for
    end if
    return False
end procedure
```

To better demonstrate the whole proving process, here is an example, which is as follows:

Example:
Here we still consider the formula $(\Box(p \to q) \to (\Box p \to \Box q))$, which is same as last section's example.

First, we will negate it and convert it to Negation Normal Form, and then we need to transmit it to a set of modal clauses. These steps were done before, and we will get: [a_1, a_3, a_4, $a_1 \to \Box a_2$, $\Box(\neg a_2|\neg p|q)$, $a_3 \to \Box p$, $a_4 \to \Diamond\neg p$]

Then we will use SAT-solver on all classical clauses, here is a_1, a_3 and a_4, this gives us a Model = { a_1, a_3, a_4 }, this is the initial world. From the initial world, we will "active" the type 3 clause $a_4 \to \Diamond\neg p$ and add type 1 clause $\Box(\neg a_2|\neg p|q)$ as contraints. So we will call a new SAT-solver with {a_2, $\neg p$, $\neg q$, p } constraints, and we will get UNSAT as the result due to p and $\neg p$.

We therefor learn $\neg(a_1$ & a_3 & $a_4)$ and restart the original SAT-solver. But it has already has a_1, a_3, a_4 as true, so adding the learned clauses will give us UNSAT. And then we will know that the original modal logic formula $(\Box(p \to q) \to (\Box p \to \Box q))$ is valid by refutation.

Chapter 4

Results

In this chapter, we provide an evaluation of our theorem provers, including a description of the methods and Benchmarks we used to evaluate and a comparison of the performance of existing state-of-the-art theorem provers for modal logic K. Ultimately, we give conclusions based on these evaluations.

Section 4.1 Benchmarks describes our chosen benchmark and its features and also gives some examples of benchmarks. In Section 4.2 Results we present the results based on the above benchmark tests and compare these results with some other state-of-the-art modal logic theorem provers.

4.1 Benchmarks

To check the performance of the theorem provers we constructed, we used Benchmarks generated by Balsiger, Peter and Heuerding, Alain and Schwendimann, Stefan, which were generated following the following guidelines [Balsiger et al., 2000]:

1. The benchmarks should have provable as well as not provable examples.

2. The benchmarks should have various structures of examples.

3. Some of the benchmark formulas are hard enough for forthcoming provers.

4. The result of each formula should already be known today.

5. A prover cannot use simple 'tricks'to help solve the problems.

6. Applying the benchmark test to a prover does not take too much time.

7. Testers can summarize the results.

The benchmark we have chosen here only applies to the part of modal logic K. This is because our theorem provers only apply to modal logic K. In detail, the benchmarks tested are divided into t4p, poly, ph, path, lin, grz, dum, d4, branch, each with

not provable and provable examples. Every valid and not valid type has 21 concrete examples, which grow in difficulty. We performed all examples in the following environment: a 2.60GHz CPU with 4GB of memory. We used 30 seconds as the limit per formula, and this time included converting the input to a form that the program could handle, parsing to suffix expressions, converting to NNF, converting to modal clauses, and proving the validity of the modal logic K theorems.

4.2 Results

We have tested mainly against BDDTab, spartacus, KSP and our theorem provers. All these can handle formulas of modal logic K and have state-of-the-art performance. All theorem provers use default settings, except for KSP, which uses the "-fsub" setting. Due to buggy runtime issues, our test of BDDtab has not produced results, but tests of other theorem provers and comparisons of our theorem provers are still of some interest.

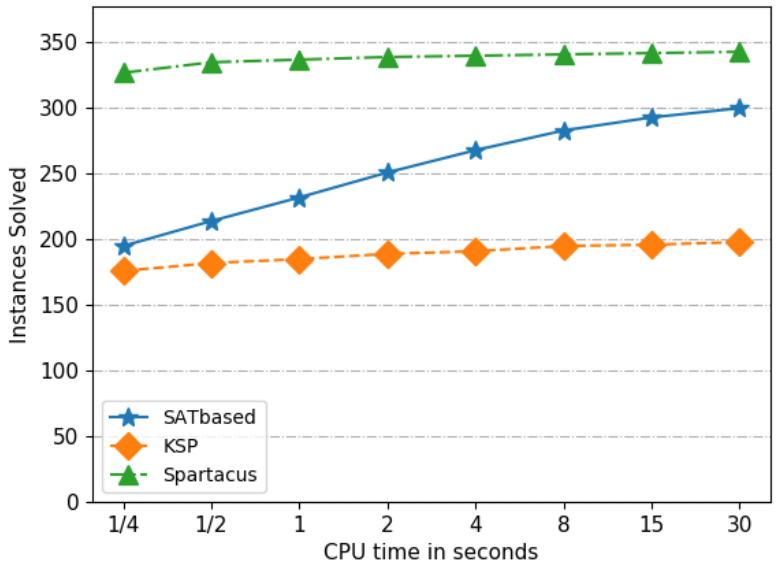

Figure 4.1: LWB benchmarking results

Figure 4.1 shows the results of our prover and Spartacus, KSP on top of the benchmarks collection, and this image shows the number of formulas that can be solved by different prover as time increases. There are 18 categories in the benchmarks, each with 21 instances and a total of 378 formulas. Spartacus performed best in the benchmarks, and our prover outperformed KSP's performance, although there was a gap with the most Spartacus, but the number of theorems solved by our prover within

the 30s time limit increased with time and gradually approached that of Spartacus. In fact, in some cases, our prover performs as well or better than Spartacus.

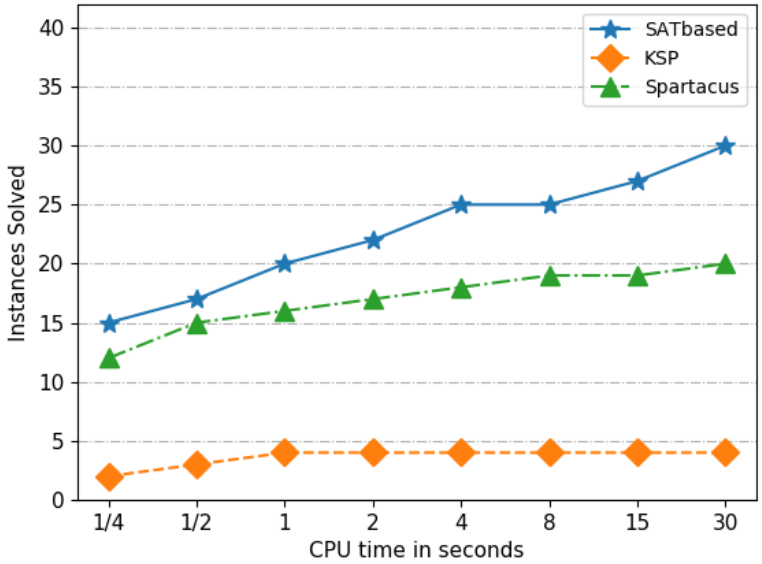

Figure 4.2: LWB benchmarking results for branch subclass

Figure 4.2 shows the results in the branch category of the benchmarks, which contains 21 provable and 21 non-provable examples each, in total 42 instances. As seen from the figure, our theorem prover outperforms the Spartacus prover on benchmark tests in the subclass branch and significantly better than the KSP prover in this subclass.

Table 4.1 shows the results in more detail. We can see that the performance of our prover is not so good as state-of-the-art prover such as spartacus, but still performs better on some benchmarks, and in many ways, our prover has better performance than another theorem prover, KSP.

Our SAT-based theorem prover performs well on the subclasses grz, lin, poly and t4, with all 21 results being arithmetically available in the required time (30s). Also, in the subclasses branch_p, d4n_p, dum, the performance is relatively good. They can all calculate more than 17 results in the allotted time. However, our prover does not perform as well for the other subclasses, and on subclass d4n_n, our solver only produces five results.

Compared to the KSP theorem prover, our prover performs relatively well, except for the subclass d4n_n, where our solver outperforms KSP significantly in all subclasses. Compared to the Spartacus theorem prover, our theorem prover outperforms Spartacus in the subclasses branch and ph_p, but in all other subclasses, the SAT-based theorem prover performs worse than or equal to Spartacus.

To check that our theorem prover is outputting the correct result, we print the

Subclass	SATbased	KSP	Spartacus
k_branch_n	12	1	9
k_branch_p	18	3	11
k_d4n_n	5	5	21
k_d4n_p	21	21	21
k_dum_n	17	21	21
k_dum_p	17	21	21
k_grz_n	21	7	21
k_grz_p	21	21	21
k_lin_n	21	1	21
k_lin_p	21	21	21
k_path_n	10	7	21
k_path_p	11	8	21
k_ph_n	12	2	21
k_ph_p	9	3	8
k_poly_n	21	3	21
k_poly_p	21	11	21
k_t4_n	21	21	21
k_t4_p	21	21	21
total	300	198	343

Table 4.1: Results on LWB benchmarks for Modal Logic K

results of the operations, and since the answers to these benchmarks are known in advance, we can compare the results of the operations with these benchmarks, and they should be the same. Note that our prover cannot produce results for all benchmarks due to memory problems and maximum loops limits, but the benchmarks that produce results are all correct.

Although our theorem prover is inferior to the state-of-the-art modal logic K-theorem prover in some performance aspects, there is still much room for optimisation, given the performance sacrificed by the high-level language we use, so this result is acceptable as it represents excellent potential for our prover.

Chapter 5

Conclusion

In this chapter, we discuss future directions for expansion of the study and draw conclusions based on the process and experimental results carried out.

In Section 5.1 Future Work, we present what areas we can extend our study to and how we can improve our theorem provers' performance in the future. In Section 5.2 Conclusion, we draw conclusions based on whole project process and the experimental results, we also analyse them and make assumptions about the reasons for the results.

5.1 Future Work

For the future work of this study, we consider the following points:

1. Calculate the time complexity of our algorithms. Computing this measurement will help us to estimate the performance of our algorithms better. We can use the time complexity to evaluate the efficiency of our prover, so in future work, we will need to do a theoretical complexity analysis about the time complexity of our algorithm.

2. Rewrite our work in a more low-level language. Since our programming language of choice is Python, a high-level language that sacrifices performance, using a more low-level programming language such as C and Haskell could effectively improve the performance of the theorem prover. Whereas C has the advantage of quickly implementing algorithms and data structures, facilitating the fast computation of programs, Haskell has the advantage that it is compiled in advance, which means that its compiler is very good at optimising and generating efficient executables. Especially when we note that Niklas Sörensson provides a kind of experimental version of MiniSat for C, while Haskell also has a bundle that call Minisat, so rewriting is possible using these two languages.

3. There is still much room for optimisation of our algorithms, such as further simplifying the NNF, optimising the data structures we use or active selection of the generated worlds in the main Proving algorithm. However, these ideas are very

rudimentary and have not been further thought through, so they need to be considered more deeply.

4. The algorithm we use for generating modal clauses comes from the Clausal Tableaux for Multimodal Logics of Belief paper by Goré and Nguyen and is designed for multimodal logics, so we can also extend our work to extensions to make it applicable to multimodal logic.

5.2 Conclusion

This report aims to construct an SAT-solver-based theorem prover for modal logic K. This prover makes use of the existing high-performance incremental SAT-solver. To implement this theorem prover, we first negate the original formula and use the shunting-yard algorithm as a parser to transform the input modal logic K formula into a postfix expression. And then, we convert this formula into a Negation Normal Form. We convert the Negation Normal Form into three different classes of modal clauses, and finally, we develop an algorithm that gives the ability to handle formulas of modal logic K to SMT method by separating modalities and classical. This algorithm constructs a recursive depth-first search tree, which is refined to try to generate a satisfying model to eventually determine the original formula's validity.

We then compared our theorem provers with existing state-of-the-art theorem provers, and although the performance of our theorem prover still falls short of the best and most advanced modal logic theorem prover, it still shows much potential. We believe that the reason for the relatively low performance of our theorem prover is because we take a simple and primitive approach to generating worlds and checking them, but this inference is not definitive, as we cannot arbitrarily conclude this due to the inefficiency of memory and the performance sacrifices made by high-level programming languages.

Bibliography

Achourioti, T. Some basics of classical logic. https://www.platformwiskunde.nl/wp-content/uploads/2017/09/Slides_VC2014_Dora.pdf. (cited on page 5)

Baader, F. and Tobies, S., 2001. The inverse method implements the automata approach for modal satisfiability. In *International Joint Conference on Automated Reasoning*, 92–106. Springer. (cited on page 9)

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 (2000), 297–317. (cited on pages 4 and 25)

Baltag, A.; Christoff, Z.; Rendsvig, R. K.; and Smets, S., 2019. Dynamic epistemic logics of diffusion and prediction in social networks. *Studia Logica*, 107, 3 (2019), 489–531. (cited on pages 1 and 5)

Barrett, C. and Tinelli, C., 2018. Satisfiability modulo theories. In *Handbook of Model Checking*, 305–343. Springer. (cited on page 2)

Claessen, K. and Rosén, D., 2015. Sat modulo intuitionistic implications. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 622–637. Springer. (cited on pages 3, 16, and 19)

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H., 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, 154–169. Springer. (cited on page 20)

Constable, R. and Kreitz, C. Automated reasoning. (cited on page 1)

De Nivelle, H.; Schmidt, R. A.; and Hustadt, U., 2000. Resolution-based methods for modal logics. *Logic Journal of the IGPL*, 8, 3 (2000), 265–292. (cited on page 9)

Fiorentini, C.; Goré, R.; and Graham-Lengrand, S., 2019. A proof-theoretic perspective on smt-solving for intuitionistic propositional logic. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, 111–129. Springer. (cited on pages 3 and 20)

Gallier, J. H., 2015. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications. (cited on page 2)

Garson, J., 2021. Modal Logic. In *The Stanford Encyclopedia of Philosophy* (Ed. E. N. Zalta). Metaphysics Research Lab, Stanford University, Summer 2021 edn. (cited on pages 1 and 5)

Goré, R., 1999. Tableau methods for modal and temporal logics. In *Handbook of tableau methods*, 297–396. Springer. (cited on pages 5, 6, 7, and 8)

Goré, R. and Nguyen, L. A., 2009. Clausal tableaux for multimodal logics of belief. *Fundamenta Informaticae*, 94, 1 (2009), 21–40. (cited on pages 3, 14, 16, and 17)

Goré, R.; Olesen, K.; and Thomson, J., 2014. Implementing tableau calculi using bdds: Bddtab system description. In *International Joint Conference on Automated Reasoning*, 337–343. Springer. (cited on pages 4 and 8)

Götzmann, D.; Kaminski, M.; and Smolka, G., 2010. Spartacus: A tableau prover for hybrid logic. *Electronic Notes in Theoretical Computer Science*, 262 (2010), 127–139. (cited on page 4)

Hamkins, J. D., 2012. The set-theoretic multiverse. *The Review of Symbolic Logic*, 5, 3 (2012), 416–449. (cited on pages 1 and 5)

Harrison, J., 2009. *Handbook of practical logic and automated reasoning*. Cambridge University Press. (cited on page 1)

Hartley, M., Lochman. Versatile math. http://hartleymath.com/versatilemath/read. (cited on page 2)

Kaminski, M. and Tebbi, T., 2013. Inkresat: modal reasoning via incremental reduction to sat. In *International Conference on Automated Deduction*, 436–442. Springer. (cited on pages 2 and 8)

Knorr, M., 2018. *Automated Reasoning*, 101–108. Springer New York, New York, NY. ISBN 978-1-4939-7131-2. doi:10.1007/978-1-4939-7131-2_110188. https://doi.org/10.1007/978-1-4939-7131-2_110188. (cited on page 2)

Loveland, D. W., 2016. *Automated Theorem Proving: a logical basis*. Elsevier. (cited on page 2)

Meyer, J.-J. and Veltman, F., 2007. 18 intelligent agents and common sense reasoning. In *Studies in Logic and Practical Reasoning*, vol. 3, 991–1029. Elsevier. (cited on pages 1 and 5)

Nalon, C.; Hustadt, U.; and Dixon, C., 2017. Ksp: A resolution-based prover for multimodal k, abridged report. In *IJCAI*, vol. 17, 4919–4923. (cited on page 4)

Portoraro, F., 2019. Automated Reasoning. In *The Stanford Encyclopedia of Philosophy* (Ed. E. N. Zalta). Metaphysics Research Lab, Stanford University, Spring 2019 edn. (cited on page 1)

Sider, T., 2010. *Logic for philosophy*. Oxford University Press, USA. (cited on pages 1 and 5)

Sörensson, N., 2010. Minisat 2.2 and minisat++ 1.1. *A short description in SAT Race*, 2010 (2010). (cited on page 12)

Sutcliffe, G. and Suttner, C., 1998. The tptp problem library. *Journal of Automated Reasoning*, 21, 2 (1998), 177–203. (cited on page 1)

Tsarkov, D. and Horrocks, I., 2006. Fact++ description logic reasoner: System description. In *International joint conference on automated reasoning*, 292–297. Springer. (cited on page 8)

Van Benthem, J.; van Benthem, J. F.; van Benthem, J. F.; Mathématicien, I.; and van Benthem, J. F., 2010. *Modal logic for open minds*. Center for the Study of Language and Information Stanford. (cited on pages 1 and 5)

Vardi, M. Y. and Wolper, P., 1986. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32, 2 (1986), 183–221. (cited on page 8)

Wei, A. the-shunting-yard-algorithm. https://www.andr.mu/logs/the-shunting-yard-algorithm/. (cited on page 13)

Wos, L.; Overbeck, R.; Lusk, E.; and Boyle, J., 1984. Automated reasoning: introduction and applications. (1984). (cited on page 2)

Appendix A README file for software

Here is a copy of README file for the SAT-based theorem prover in the following page.

---------- SATbased Prover ----------

This is a SAT-based Theorem Prover for Modal Logic K written by Tian Luan.

---------- Environments ----------

This prover is based on Python 3.8.5 and the only additional library is PySat, to run the solver you need to intall it first, this can be installed by following command:

```
$ pip install python-sat
```

---------- Usage ----------

```
python SAT.py "fml"
```

This SAT-based Prover will read one parameter (required) as a modal logic formula, then it
will return whether this formula is provable (valid) or not. The whole process behind it
is first negate the input and then determine whether the negation is satisfiable.

The prover accepts formuae in the following syntax:

```
fml ::= '(' fml ')' ( parentheses )
 | 'true' ( truth )
 | 'false' ( falsehood )
 | '-' fml ( negation )
 | '<>' fml ( diamonds )
 | '[]' fml ( boxes )
 | fml '&' fml ( conjunction )
 | fml '|' fml ( disjunction )
 | fml '->' fml ( implication )
 | fml '<->' fml ( equivalence )
 | a | b | c | .... ( classical literals )
```

---------- Using Example ----------

```
$ python SAT.py "[](p->q)->([]p->[]q)"
```

This will return: Input Formula is Provable

---------- Authors ----------

Tian Luan

---------- Acknowledgement ----------

Professor Rajeev Gore

Appendix B Code Implementations

Some codes of main algorithms in this prover are contained on the next pages.

Shunting-yard Algorithm

```
def shunting_yard(string):
    current_token = ''

    output_queue = []
    stack = []

    it = iter(range(0, len(string)))
    for i in it:
        ch = string[i]
        operator_char = (ch in ["-", "&", "|", "^", ">", " ", "(", ")", "[", "]", "<
                                                    ", "*"])

        if current_token and operator_char:
            output_queue.append(current_token)
            current_token = ''

        elif current_token and i == len(string) - 1:


            if not operator_char:
                current_token += ch

            output_queue.append(current_token)
            current_token = ''

        if ch in ["-", "&", "|", "^", '>', '<', '[', ']', '*']:

            # don't let unary '-' operator pop a binary operator from the stack
            # give parens a high precedence
            if stack and ch != "-" and stack[-1] != '(' and ch != "[" and ch != "<":
                output_queue.append(stack.pop())

                if output_queue[-1] == "-" and stack and stack[-1] != '(':
                    output_queue.append(stack.pop())
                if output_queue[-1] == "[]" and stack and stack[-1] != '(':
```

```
                output_queue.append(stack.pop())
            if output_queue[-1] == "<>" and stack and stack[-1] != '(':
                output_queue.append(stack.pop())

        if ch == ">" and string[i + 1] != ">" and string[i - 1] != ">":
            print("Error: single '>' is not allowed")
            return
        if ch == "<" and string[i + 1] != ">":
            print("Error: single '<' is not allowed")
            return
        if ch == "[" and string[i + 1] != "]":
            print("Error: single '[' is not allowed")
            return

        if ch == ">" and string[i + 1] == ">":
            stack.append(">>")
            next(it)
        elif ch == "<" and string[i + 1] == ">":
            stack.append("<>")
            next(it)
        elif ch == "[" and string[i + 1] == "]":
            stack.append("[]")
            next(it)
        elif ch == "*":
            stack.append("<->")
        else:
            stack.append(ch)

    if ch == "(":
        stack.append("(")

    if ch == ")":
        while True:
            next_operator = stack.pop()
            if next_operator == "(":
                break
            output_queue.append(next_operator)

    if ch not in ["-", "&", "|", "^", '>', "(", ")", " ", "<", "[", "]", "*"]:
        current_token += ch

while stack:
    output_queue.append(stack.pop())

return output_queue
```

Negation Normal Form Algorithm

```
def convertNNFMain(input):
    global phi
    phi = input.copy()
    return convertNNF()

phi = []
def convertNNF():
    global phi
    operators = ["&", "|", "-", ">>", "[]", "<>", "<->"]
    if phi:
        arg = phi.pop()
    else:
        return
    if arg not in operators:
        if arg == "true":
            return [True]
        elif arg == "false":
            return [False]
        return [arg]

    if arg == "<->":
        temp = phi.copy()
        tmp = convertNNF()
        tmp1 = convertNNF()
        phi = temp
        phi = phi + ["-"]
        tmp2 = convertNNF()
        phi = phi + ["-"]
        tmp3 = convertNNF()
        return tmp + tmp1 + ["&"] + tmp2 + tmp3 + ["&"] + ["|"]
    elif arg == ">>":
        tmp = convertNNF() # right arg
        phi = phi + ["-"]
        tmp1 = convertNNF() # left arg
        return tmp + tmp1 + ["|"]
    elif arg == "[]":
        return convertNNF() + ["[]"]
    elif arg == "<>":
        return convertNNF() + ["<>"]
    elif arg == "-":
        new_arg = phi.pop()
        if new_arg == "true":
            return [False]
        elif new_arg == "false":
            return [True]
        elif new_arg not in operators and new_arg != "true" and new_arg != "false":
            return [new_arg, "-"]
        elif new_arg == "-":
            return convertNNF()
        elif new_arg == "&":
            phi = phi + ["-"]
            tmp = convertNNF()  #r
```

```
            phi = phi + ["-"]
            tmp1 = convertNNF() #l
            return tmp1 + tmp + ["|"]
        elif new_arg == "|":
            phi = phi + ["-"]
            tmp = convertNNF()
            phi = phi + ["-"]
            tmp1 = convertNNF()
            return tmp1 + tmp + ["&"]
        elif new_arg == ">>":
            phi = phi + ["-"]
            tmp = convertNNF()
            tmp1 = convertNNF()
            return tmp1 + tmp + ["&"]
        elif new_arg == "<->":
            temp = phi.copy()
            tmp = convertNNF()
            tmp1 = convertNNF()
            phi = temp
            phi = phi + ["-"]
            tmp2 = convertNNF()
            phi = phi + ["-"]
            tmp3 = convertNNF()
            return tmp + tmp1 + ["|"] + tmp2 + tmp3 + ["|"] + ["&"]
        elif new_arg == "[]":
            phi = phi + ["-"]
            return convertNNF() + ["<>"]
        elif new_arg == "<>":
            phi = phi + ["-"]
            return convertNNF() + ["[]"]
    else:
        return convertNNF() + convertNNF() + [arg]


```

Modal Clause Class

```
from convertNNF import convertNNFMain

class modalClause:
    def __init__(self):
        self.clause = []     # clause
        self.boxDepth = 0     # modal depth
        self.clauseInModalContent = []

def popModalContext(phi):
    arg = phi[-1]
    if arg != "[]":
        return phi
    else:
        return popModalContext(phi[:-1])

def calDepth(clause):
    box_depth = 0
    for arg in clause[::-1]:
        if arg != "[]":
            return box_depth
        else:
            box_depth = box_depth + 1
    return box_depth

def constructModalClause(X):
    res = []
    if X:
        for clause in X:
            tmp = popModalContext(clause)
            if "[]" not in tmp and "<>" not in tmp:
                clause = convertNNFMain(clause)
                tmp = convertNNFMain(tmp)
            new = modalClause()
            new.clause = clause
            new.boxDepth = calDepth(clause)
            new.clauseInModalContent = tmp
            res.append(new)
    return res


```

Modal Clausification Algorithm

```
from convertNNF import convertNNFMain

counts = 0
def getNewAtom():
    global counts
    new_atom = 'p' + str(counts)
    counts = counts + 1
    return new_atom

phi = []
List = []
def modal_clausification(box_depth = 0):
    global List
    global phi
    opreators = ["[]", "<>", "&", "|"]
    if phi:
        arg = phi.pop()
    else:
        return
    if arg not in opreators:
        if arg == "-":
            new_arg = phi.pop()
            res = [new_arg, "-"]
        else:
            res = [arg]
        for i in range(box_depth):
            res = res + ["[]"]
        return res

    if arg == "[]":
        new_arg = phi[-1]
        # if new_arg not in opreators or new_arg == "[]":
        #     tmp = modal_clausification(box_depth + 1)
        #     return tmp
        if new_arg not in opreators:
            new_atom = getNewAtom()
            t = [new_atom]
            tmp = modal_clausification()
            tmp = [new_atom] + tmp + ["[]", ">>"]
            for i in range(box_depth):
                t = t + ["[]"]
                tmp = tmp + ["[]"]
            List.append(tmp)
        else:
            new_atom = getNewAtom()
            t = [new_atom]
            tmp = modal_clausification(box_depth + 1)
            if type(tmp[0]) != list:
                if len(tmp) != box_depth + 2:
                    new_name = getNewAtom()
                    res = [new_atom, new_name, "[]", ">>"]
                    res1 = [new_name] + tmp[:-(box_depth + 1)] + [">>", "[]"]
```

```
                    for i in range(box_depth):
                        t = t + ["[]"]
                        res = res + ["[]"]
                        res1 = res1 + ["[]"]
                    List.append(res)
                    List.append(res1)
                else:
                    res = [new_atom, tmp[0], "[]", ">>"]
                    for i in range(box_depth):
                        t = t + ["[]"]
                        res = res + ["[]"]
                    List.append(res)
            else:
                new_name = getNewAtom()
                res = [new_atom, new_name, "[]", ">>"]
                for i in tmp:
                    if len(i) == box_depth + 2:
                        res1 = [new_name, "-", i[0], "|", "[]"]
                    else:
                        res1 = [new_name] + i[:-(box_depth + 1)] + [">>", "[]"]
                    for i in range(box_depth):
                        res1 = res1 + ["[]"]
                    List.append(res1)
                for i in range(box_depth):
                    t = t + ["[]"]
                    res = res + ["[]"]
                List.append(res)
        return t
    elif arg == "<>":
        new_arg = phi[-1]
        new_atom = getNewAtom()
        t = [new_atom]
        if new_arg not in opreators:
            tmp = modal_clausification()
            tmp = [new_atom] + tmp + ["<>", ">>"]
            for i in range(box_depth):
                t = t + ["[]"]
                tmp = tmp + ["[]"]
            List.append(tmp)
        else:
            tmp = modal_clausification(box_depth + 1)
            if type(tmp[0]) != list:
                if len(tmp) != box_depth + 2:
                    new_name = getNewAtom()
                    res = [new_atom, new_name, "<>", ">>"]
                    res1 = [new_name] + tmp[:-(box_depth + 1)] + [">>", "[]"]
                    for i in range(box_depth):
                        t = t + ["[]"]
                        res = res + ["[]"]
                        res1 = res1 + ["[]"]
                    List.append(res)
                    List.append(res1)
                else:
                    res = [new_atom, tmp[0], "<>", ">>"]
```

```
                    for i in range(box_depth):
                        t = t + ["[]"]
                        res = res + ["[]"]
                    List.append(res)
            else:
                new_name = getNewAtom()
                res = [new_atom, new_name, "<>", ">>"]
                for i in tmp:
                    if len(i) == box_depth + 2:
                        res1 = [new_name, "-", i[0], "|", "[]"]
                    else:
                        res1 = [new_name] + i[:-(box_depth + 1)] + [">>", "[]"]
                    for i in range(box_depth):
                        res1 = res1 + ["[]"]
                    List.append(res1)
                for i in range(box_depth):
                    t = t + ["[]"]
                    res = res + ["[]"]
                List.append(res)
        return t
    elif arg == "|":
        tmp = modal_clausification(box_depth)
        tmp1 = modal_clausification(box_depth)
        res = []
        if type(tmp[0]) != list and type(tmp1[0]) != list:
            if box_depth>0:
                res = tmp[:-box_depth] + tmp1[:-box_depth] + ["|"]
            else:
                res = tmp + tmp1 + ["|"]
            for i in range(box_depth):
                res = res + ["[]"]
        elif type(tmp[0]) == list and type(tmp1[0]) != list:
            for i in tmp:
                if box_depth > 0:
                    result = i[:-box_depth] + tmp1[:-box_depth] + ["|"]
                else:
                    result = i + tmp1 + ["|"]
                for j in range(box_depth):
                    result = result + ["[]"]
                res.append(result)
        elif type(tmp[0]) != list and type(tmp1[0]) == list:
            for i in tmp1:
                if box_depth > 0:
                    result = i[:-box_depth] + tmp[:-box_depth] + ["|"]
                else:
                    result = i + tmp + ["|"]
                for j in range(box_depth):
                    result = result + ["[]"]
                res.append(result)
        else:
            for i in tmp:
                for j in tmp1:
                    if box_depth > 0:
                        result = i[:-box_depth] + j[:-box_depth] + ["|"]
```

```
                    else:
                        result = i + j + ["|"]
                    for m in range(box_depth):
                        result = result + ["[]"]
                    res.append(result)
        return res
    elif arg == "&":
        res = []
        tmp = modal_clausification(box_depth)
        if type(tmp[0]) == list:
            res = res + tmp
        else:
            res.append(tmp)
        tmp1 = modal_clausification(box_depth)
        if type(tmp1[0]) == list:
            res = res + tmp1
        else:
            res.append(tmp1)
        return res

def modal_clausification_main(X):
    global phi
    phi = X.copy()
    res = modal_clausification()
    if type(res[0]) == list:
        return res + List
    else:
        List.append(res)
        return List


```

Modal Sat Algorithm

```
from pysat.solvers import Minisat22
from convertToInput import convertToInputMain

def ModalSatMain(X0):
    global s0
    s0 = Minisat22()    # s0 <- newSolver()
    # add all atoms and constraints which box depth is 0
    for phi in X0:
        if "[]" not in phi.clause and "<>" not in phi.clause:
            s0.add_clause(convertToInputMain(phi.clause))
    r = ModalSat(X0, [], s0, 0)
    if r == False:
        return "SAT"
    else:
        return "UNSAT"

def ModalSat(X, A, s, boxDepth):
    r0 = s.solve(A) # r0 <- satProve(s, A)
    sub_A = s.get_core() # A'
    M = s.get_model()
    if r0 == False:
        return True, sub_A
    else:
        for k0 in X:
            if k0.boxDepth == boxDepth and "<>" in k0.clauseInModalContent:
                a = convertToInputMain([k0.clauseInModalContent[0]])[0]
                d = convertToInputMain(k0.clauseInModalContent
                                       [1:k0.clauseInModalContent.index('<>')])[0]
                if a in M:
                    s1 = Minisat22()
                    B = [d]
                    for k1 in X:
                        if k1.boxDepth == boxDepth and "[]" in \
                                k1.clauseInModalContent:
                            b = convertToInputMain([k1.clauseInModalContent[0]])[0]
                            e = convertToInputMain(k1.clauseInModalContent
                                    [1:k1.clauseInModalContent.index('[]')])[0]
                            if b in M:
                                B.append(e)
                    for k2 in X:
                        if k2.boxDepth == boxDepth + 1 and "[]" not in \
                                k2.clauseInModalContent and "<>" not in \
                                k2.clauseInModalContent:
                            s1.add_clause(convertToInputMain(k2.clauseInModalContent
                                                             ))
                    r1 = ModalSat(X, B, s1, boxDepth + 1)
                    if r1:
                        sub_A1 = r1[1]
                        if sub_A1:
                            phi = []
                            z = convertToInputMain(k0.clauseInModalContent
                                                   [1:k0.clauseInModalContent.
```

```
                                                index("<>")])[0]
                        y = convertToInputMain([k0.clauseInModalContent[0]])[0]
                        if z in sub_A1 and y in M:
                            phi.append(-y)
                        for k1 in X:
                            if "[]" in k1.clauseInModalContent and \
                                    k1.boxDepth == boxDepth:
                                z = convertToInputMain
                                (k1.clauseInModalContent
                                [1:k1.clauseInModalContent.index("[]")])[0]
                                y = convertToInputMain
                                ([k1.clauseInModalContent[0]])[0]
                                if z in sub_A1 and y in M:
                                    phi.append(-y)
                        s.add_clause(phi)
                        r2 = ModalSat(X, A, s, boxDepth)
                        return r2
                    else:
                        return True, None
    return False

```

Abstract Call of Prover

```
from ConvertTestFormat import convert_format
from shuntingYard import shunting_yard
from convertNNF import convertNNFMain
from convertAtoms import convertAtoms_x, convertAtoms_v
from modalClausification import modal_clausification_main
from modalClauses import constructModalClause
from modalSat import ModalSatMain

def Prove(formula):
    formula = convert_format(formula)
    formula = "-("+formula+")"
    # print("formula", formula)
    parsing_fml = shunting_yard(formula)
    # print("parsing",parsing_fml)
    nnf_fml = convertAtoms_v(convertNNFMain(parsing_fml))
    # print("negation normal form", nnf_fml)
    if True in nnf_fml:
        modal_clauses = convertAtoms_x(modal_clausification_main(nnf_fml),True)
    else:
        modal_clauses = convertAtoms_x(modal_clausification_main(nnf_fml))
    # print("modal clauses", modal_clauses)
    input_clauses = constructModalClause(modal_clauses)
    res = ModalSatMain(input_clauses)
    if res == "UNSAT":
        return "Provable"
    else:
        return "Not Provable"


```