Extending CEGARBox: An Efficient Theorem Prover for the 15 Normal Modal Cube Logics

Robert Neil McArthur

A report submitted for the course
COMP3770 Individual Research Project (6 units)
Supervised by: Rajeev Goré
The Australian National University

October 2021

Except where otherwise indicated, this report is my own original work.

Robert Neil McArthur
October, 2021

Acknowledgments

There are several people I would like to thank who have all helped me very significantly throughout this research project. First and most foremost I would like to extend my deepest appreciation to my supervisor Rajeev Goré. Raj's support to me has been beyond tremendous in the making of this project. I personally feel that he has gone above and beyond to support me academically in this project and personally. He is an amazing person, and I do feel that through this experience I have learnt so much from him. I have a lot to owe him.

Next, I would also like to thank a friend of mine and the author of the original CEGARBox, Cormac Kikkert. He has helped me out significantly in the understanding of his original algorithm so that I could effectively translate his work into C++ and has been very useful with his offerings of his unique personal insights into various optimisations which could be made, and ideas for supporting additional logics.

Finally, I would like to extend my sincere appreciation towards my parents Adrian and Luci McArthur. They have each sacrificed so much to give me the opportunities throughout my life to allow me to do what I'm doing today. They have been amazing role models to me, and I wouldn't be able to do anything like this without them.

Abstract

A recent work by Goré and Kikkert [2021] introduced CEGARBox - an efficient theorem prover for modal logics K, KT and S4. CEGARBox employs a counter-example guided abstract refinement strategy to explore a rooted tree-model where the classical propositional logic part of each Kripke world is evaluated by a SAT solver. CEGARBox was found in the work to be the best for those modal logics by, in some instances, orders of magnitude. As the current state-of-the-art theorem prover for these modal logics, this work aims to extend and improve CEGARBox in three ways. The first is to extend CEGARBox to handle the 15 normal modal logics that make up the modal cube [Garson, 2021a] by combining the axioms which make up reflexivity, symmetry, transitivity, seriality and euclideaness. To modify the procedure so that it can additionally handle multimodal logic. Finally is to improve the modularity of CEGARBox to allow for different underlying SAT solvers to be easily swapped out and to enhance the ease of implementing additional logics in the future. We should be able to do all of this while maintaining or increasing the efficiency of our prover. Additionally, while remaining unimplemented, this report provides the formalism needed to handle global assumptions. Experiments were conducted to measure the efficiency of our new prover against the original version and the only theorem prover to beat the original version in one benchmark. Correctness was also confirmed for a small number of test cases.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The realm of automated theorem proving has a long history for different logic systems. Classical propositional logic on its own is not very expressive. First order logic which adds for all and there exists operators to classical logic is very expressive, however is undecidable so it isn't very useful for theorem proving. Modal logic however, is a more expressive form of classical propositional logic with operators $\Box$ and $\Diamond$ encapsulating the ideas of necessity and possibility respectively. It sits in between classical propositional logic and first order logic in a sense that is much more expressive than the first but has almost universal decidability [Wu and Goré, 2019]. If we could make an efficient decision procedure for modal logic we could build an expressive intelligent agent that is able to reason automatically. That's where a recent work by Goré and Kikkert [2021] comes in, which introduced a theorem prover for modal logic called CEGARBox. CEGARBox was found to be the most efficient theorem prover for three different types of modal logic (K, T, and $S4$ which encapsulate different axioms we can apply to modal logic) in a monomodal context. This was sometimes by orders of magnitude. As the current state-of-the-art SAT solver for these modal logics the question becomes can we modify the algorithm in order to create an efficient theorem prover for all fifteen normal modal logics expressed in the modal cube. The purpose of this project is hence to formalise the changes that need to be made in order to handle these fifteen modal logics in a multimodal context. Additionally, we refactored the code into C++ and have structured it so that underlying SAT solvers can easily be swapped out in the future, and so that the method can be applied to even more logics. As extra work, we formalise the necessary changes needed in order for the algorithm to also handle global assumptions.

1.2 Report Outline

The work of this report is divided into 6 chapters. Chapter 2 begins by introducing the notation needed to understand the rest of the report. We also introduce the algorithm behind CEGARBox [Goré and Kikkert, 2021] in this chapter and discuss some other related work. Chapter 3 then formalises the work which needs to be

done in order to extend CEGARBox to deal with the 15 normal modal logics in modal cube in a multimodal context. Chapter 4 then formalises additional extensions and optimisations to CEGARBox including formalising global assumptions (though remaining unimplemented) in the refactoring of the code into C++. Chapter 5 then details the experiment which were run, before providing and critically analysing the results. Finally, we conclude in Chapter 6 and provide direction for future work.

Chapter 2

Background and Related Work

This chapter introduces the material notation required to understand the remainder of the report. This report assumes familiarity with classical propositional logic, together with modal logic and basic tableau calculi.

2.1 Background

2.1.1 Notation

We use a similar notation as used by Goré and Kikkert [2021] and extend to the multimodal case.

We consider multimodal logic with the modal operators $\Box_z$ and $\Diamond_z$ where $z \in \mathbb{Z}$ is the modality of these operators. A formula is defined by atoms $p \in Atm$ and the grammar $\varphi := \bot \mid \top \mid p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box_z \varphi \mid \Diamond_z \varphi$. Additionally we define $\varphi_1 \rightarrow \varphi_2$ to be $\neg\varphi_1 \vee \varphi_2$ and consequently $(\varphi_1 \leftrightarrow \varphi_2)$ to be $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_2)$. For readability we also express powers of these box and diamond clauses such that $\Box_z^n \varphi := \Box_z \Box_z^{n-1} \varphi$ with $\Box_z^0 \varphi := \varphi$, and of course $\Diamond_z^n \varphi := \Diamond_z \Diamond_z^{n-1} \varphi$ with $\Diamond_z^0 \varphi := \varphi$ [Goré, 2019].

A literal is either an atom p or its negation $\neg p$. A formula is in negation normal form when the only negations are in the form of literals and implications are removed.

A cpl-clause is a disjunction of literals. We call a formula of the form $\neg a \vee \Box_z b$ a box-clause of modality z and similarly $\neg c \vee \Diamond_z d$ a diamond-clause of modality z with a, b, c, d all literals. We more commonly represent these as $a \rightarrow \Box_z b$ and $c \rightarrow \Diamond_z d$ respectively.

These cpl-clauses, box-clauses and diamond-clauses together form our modal clauses - the set of clauses of a certain modality. We use $\mathcal{C}$ to express the union of these clauses within different modal contexts and access the individual cpl, box and diamond-clauses using $\mathcal{C}^{cpl}$, $\mathcal{C}^{\rightarrow\Box_z}$ and $\mathcal{C}^{\rightarrow\Diamond_z}$ respectively. Unlike in monomodal logic where it was easy to express these modalities and their relations with a sequence, instead we use a trie to express them. We use a semicolon delimited list to represent the modality of the modal contexts and subscript this on $\mathcal{C}$. For example, we write $\mathcal{C}_{[x^i;y^j;\ldots;z^k]}$ to express the cpl, box and diamond clauses contained within $\Box_x^i \Box_y^j \ldots \Box_z^k \mathcal{C}$ with a visualisation of this shown in Figure 2.1. We can express concatenation of the

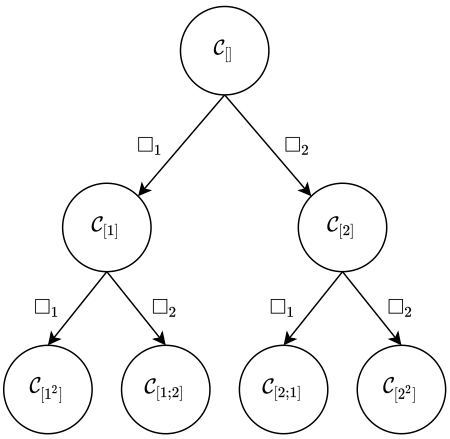


Figure 2.1: An example of how modal clauses are represented with their relations.

lists use by writing $[x^i; y^j; ...; z^k] + [n]$ which is simply $[x^i; y^j; ...; z^k; n]$ We write $\overrightarrow{\mathcal{C}}_s$ to represent the union of the modal clauses at modality $s = [x^i; y^j; ...; z^k]$ and all of its descendants in the trie.

A Kripke frame is a tuple $\langle W, R \rangle$ representing a directed graph over worlds W with relations R. A Kripke model is a triple $\langle W, R, \vartheta \rangle$ made of a Kripke frame $\langle W, R \rangle$ and a valuation ϑ assessing the truth of atoms in every world W. We use our rooted tree model shown in Figure 2.1 to help generate a valid Kripke model in our prover. Further, we say a box clause $a \rightarrow \Box_z b$ is fired with modality z in world w when a valuation $\vartheta(w)$ makes a true, with the analogous definition holding true for diamond clauses [Goré, 2019].

2.1.2 CEGARBox

CEGARBox relies predominantly on two main features of modern SAT solvers - incremental clause learning and satisfiability under unit assumptions. Satisfiability under unit assumptions allows us to query a SAT solver as to whether its attached formula is satisfiable under such literal assumptions without modifying the SAT solver's state. Incremental clause learning allows us to, as the name suggests, progressively add clauses to the SAT solver. The main insight behind CEGARBox is that we can attach an individual SAT solver to each modal context while maintaining the accessibility relation via recursion as we move down our modal contexts. The core idea is as follows, we start at the root world $w_0 = \mathcal{C}_{[]}$ and assess the SAT solver on the cpl-clauses $\mathcal{C}^{cpl}_{[]}$. If that SAT solver returns unsatisfiable, then obviously the modal formula is unsatisfiable and we are done. Otherwise, if it finds a valuation $\vartheta(w_0)$ we begin to go through our diamond clauses. For each "fired" diamond $\vartheta(w_0)$ triggers, as defined in the previous section, we must create a new world. For each fired diamond we take the union of the "fired" box clauses call the same proof procedure with these assumptions for the next modal context. If the next modal context returns unsatisfiable, we inspect the unsatisfiable core X - the unit assumptions the SAT solver cannot

satisfy. We know then that either the literal c which triggered the diamond clause $(c \to \Diamond_1 d)$ must be false (otherwise we wouldn't need to create the world) or at least one of the literals which triggered a box clause $(a \to \Box_1 b)$ such that $b \in X$ must be false. Hence we can make use of the incremental SAT solver by adding the clause $\bigvee \{\neg c\} \cup \{\neg a | a \to \Box_1 b \wedge b \in X\}$ and rerunning the proof method with the new clause in the current modal context. If the children of a modal context is satisfiable for all triggered diamond clauses, then it is satisfiable and we return as such.

This method is shown in the algorithm taken from the paper below. `TrieNode` is used to keep track of the trie as depicted by figure 2.1. Each TrieNode has an incremental SAT solver `TrieNode.sat` initialised with the cpl-clauses for its respective modality. Each child of the `TrieNode` is accessed by `TrieNode.child(n)` for each modality n. The box and diamond clauses are accessed with `TrieNode.DiaCl` and `TrieNode.BoxCl` respectively.

Algorithm 1 `CEGARBox`(A, `TrieNode`)

{Inputs: A is a set of unit assumptions and `TrieNode` is at level i in our trie}
$t_0 \leftarrow$ `solve`(`TrieNode.sat`, A)
if t_0 = (unsat, A') **then**
  return Unsatisfiable(A)
else if $t_0 = (sat, \vartheta)$ **then**
  {Check box and diamond clauses that fire under classical valuation ϑ}
  for every $(c \to \Diamond_1 d) \in$ `TrieNode.DiaCl` with $c \in \vartheta$ **do**
    $B \leftarrow \{b \mid (a \to \Box_1 b) \in$ `TrieNode.BoxCl` and $a \in \vartheta\}$
    if `CEGARBox`$((d; B),$ `TrieNode.child(1)`$)$ = Unsatisfiable(X) **then**
      $C \leftarrow \{c\} \cup \{a \mid (a \to \Box_1 b) \in$ `TrieNode.BoxCl` and $a \in \vartheta$ and $b \in X\}$
      {Learn new clause $\varphi := \neg C$}
      $\varphi \leftarrow \bigvee_{l \in C} \neg l$
      `addClause`(`TrieNode.sat`, φ)
      return `CEGARBox`(A, `TrieNode`)
    end if
  end for
  return Satisfiable
end if

For an example of how this works consider the prover is asked to show the K axiom $\Box(p \to q) \to (\Box p \to \Box q)$ is valid. As we are assessing validity we take the negation $\Box(p \to q) \wedge \neg(\Box p \to \Box q)$ and put it into negated normal form $\Box(\neg p \vee q) \wedge (\Box p \wedge \Diamond \neg q)$. Through our normal forming process we get ${\mathcal{C}_{[]}}^{cpl} = \{a\}$, ${\mathcal{C}_{[]}}^{\to\Box} = \{a \to \Box b\}$, ${\mathcal{C}_{[]}}^{\to\Diamond} = \{\top \to \Diamond \neg q\}$, $\mathcal{C}^{cpl}_{[1]} = \{\neg b \vee \neg p \vee q, p\}$. We run the SAT solver in $\mathcal{C}_{[]}$ and get a to be true, and both b and $\neg q$ are fired. As a diamond clause is fired we create a world containing b and $\neg q$ in the context of $\mathcal{C}_{[1]}$. We run the SAT solver in $\mathcal{C}_{[1]}$ under these assumptions. The first cpl clause gives $\neg p$ under these assumptions but p also appears here so it is unsatisfiable with unsatisfiable core b and $\neg q$. Hence we go back

Name	Relation	Axiom	Frame Condition
T	Reflexivity	$\Box p \rightarrow p$	uRu
4	Transitivity	$\Box p \rightarrow \Box\Box p$	$uRv \wedge vRw \implies uRw$
B	Symmetry	$p \rightarrow \Box\Diamond p$	$uRv \implies vRu$
D	Seriality	$\Box p \rightarrow \Diamond p$	$\forall u \exists v(uRv)$
5	Euclideaness	$\Diamond p \rightarrow \Box\Diamond p$	$uRv \wedge uRw \implies vRw$

Table 2.1: The five basic axioms which make up the modal cube.

to $\mathcal{C}_{[]}$ and see what triggered these assumptions. Thus, either $\neg a$ is true, or $\bot$ is true so we learn a new clause $\neg a \vee \bot$. With a in the cpl clauses $\mathcal{C}_{[]}$ is obviously no longer satisfiable so we return unsatisfiable and the procedure terminates. Therefore as the negation is unsatisfiable the formula is valid.

2.1.3 Modal Cube

There are fifteen normal logics which form the so called modal cube [Garson, 2021b]. The above algorithm works for the normal modal logic K and in the paper was extended to assess the satisfiability of a formula under modal logics T (reflexivity) and $S4$ (reflexivity and transitivity together). The fifteen normal modal logics which make up the modal cube are formed by mixing and matching different axioms corresponding to different conditions on Kripke frames.

These axioms of modal logic are extremely useful in terms of expressibility. Without considering the axiom's duals, The reflexive axiom tells us that if p is necessary, then p is a fact. Transitivity tells us that if p is necessary, then it is necessary that p is necessary. Symmetry tells us if p is a fact, then it is necessary that p is possible. Seriality tells us if p is necessary, then p is possible. And finally, euclideaness tells us that if p is possible then it is necessary that p is possible. All of these axioms, and their combinations, add different restrictions which allow to us much more expressive reasoning.

By combining these axioms we generate the modal cube. In addition to K we have T (reflexive), $K4$ (transitive), KB (symmetric), D (serial), $K5$ (euclidean), $S4$ (reflexive and transitive), B (reflexive and symmetric), $D4$ (serial and transitive), DB (serial and symmetric), $D5$ (serial and euclidean), $K45$ (transitive and euclidean), $D45$ (serial, transitive and euclidean), $KB4 = KB5 = K4B5$ (symmetric and either transitive or euclidean) and $S5$ (reflexive and either both symmetric and transitive, or euclidean). Note that often adding additional axioms doesn't change the class of modal logic - for example reflexivity already implies seriality. Additionally, seriality with symmetry and transitivity implies reflexivity (and hence $S5$). The truly exciting thing that this work shows is that by preprocessing the final modal contexts and with slight additions to the proof search procedure in some contexts, we can handle all fifteen of these logics in a multimodal context.

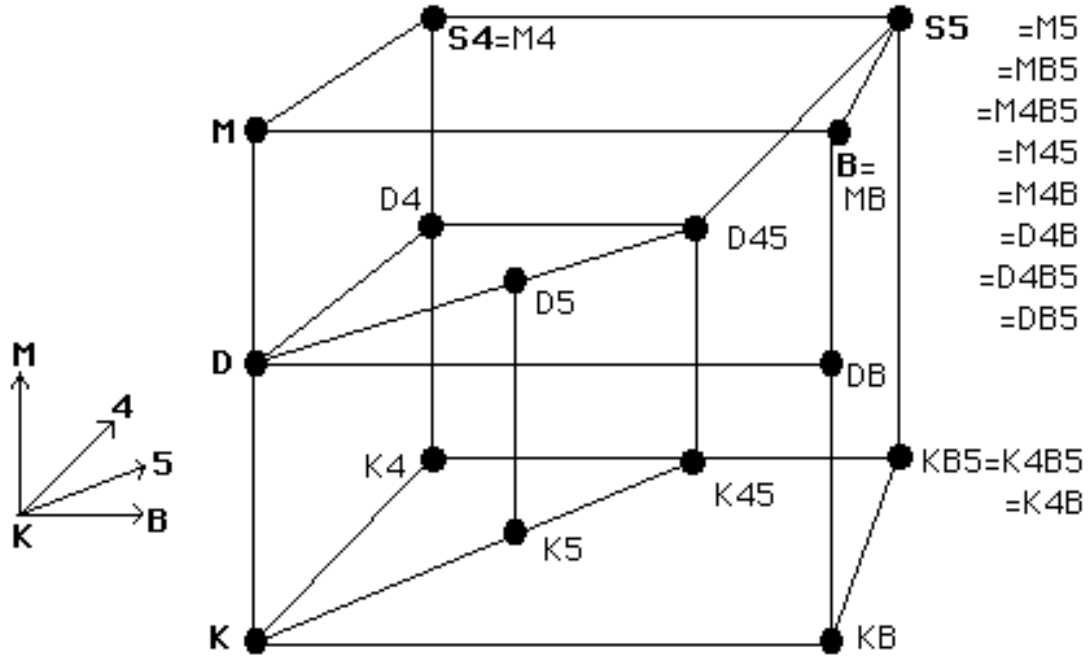


Figure 2.2: The 15 modal logics which make up the modal cube Garson [2021b]. We use T instead of M.

2.2 Related work

There are several additional related works for general automated theorem proving for automated logic. One such general and slightly older theorem prover is TABLEAUX [Catach, 1991]. TABLEAUX works by initialising a single world in which all formulas hold. Then, a model is progressively generated around the one world using tableau rules in a tree based model. Internally, formulas are kept track of by the worlds they exist in and are labelled for whether they have yet been examined. The tableaux rules are broken into three sections. One type are the simplifications rules to help simplify formula and improve efficiency. Another type are transformation rules which label the formula and help to apply definitions based on the type of modal logic being examined. There are additionally three core rules for the prover - one which creates worlds through the diamond clauses, another which propagates box clauses, and a third which handles disjunctions. The theorem prover was tested for a variety of logics over different axioms but results are quite slower than today's standards. Another general theorem prover is Lotrec [Cerro et al., 2001]. Lotrec was built with the intention to allow users to explore different search based strategies for automated theorem proving. Lotrec allows users to define their own logic languages, with customisable tableau rules and search strategies (with a focus on modal description logics). As such it isn't heavily optimised. Even though future versions have helped to remove certain redundancies [Gasquet et al., 2005], its focus is still more on allowing users to play around with different search strategies and as such isn't as fast. Yet

another automated theorem prover is MOIN [Girlando and Straßburger, 2020] which is a theorem prover for the intuitionistic modal cube relying on nested sequents and recursively invokes rules based on the type of modal logic to generate a search tree to see if an input formula is derivable. It is a relatively successful implementation but is particularly slow on some axiomatic systems such as $ID45$ for relatively small formula sizes. There are many implementations for the more general modal logics however, most do sacrifice efficiency in return for their generality. CEGARBox is a highly efficient proving method for modal logic, and this report will show that by simply performing some preprocessing techniques we can achieve a highly efficient theorem prover for all fifteen modal logics in the modal cube.

2.3 Summary

This chapter introduced the notation that is necessary in order to be able to understand the remainder of the report. Importantly, we defined what a modal context is, how the CEGARBox algorithm works, and the axioms of the modal cube. We also briefly introduced some related work. In the next section we now show how to extend CEGARBox to handle the modal cube.

Chapter 3

Extensions to Modal Cube

This chapter formalises the changes which need to be made to CEGARBox in order to handle all 15 normal modal logics found in the modal cube in a multimodal context. We begin in Section 3.1 by discussing changes to the algorithm which allow us to handle multimodal logic. This is followed by Section 3.2 where we discuss how to handle the modal logics with one axiom excluding euclideaness. Section 3.3 is then used to discuss the combinations of axioms that involve reflexivity. We then consider the special case of euclideaness in Section 3.4 before discussing the extensions to the rest of the logics finally in Section 3.5.

3.1 Extending CEGARBox to Multimodal Logic

The extension needed to handle multimodal logic is trivial. Observe that we must satisfy the box and diamond clauses of each modality using the trie. Hence, by additionally grouping the box and diamond clauses by their modality such that they can be gathered by `TrieNode.BoxCl(n)` and `TrieNode.DiaCl(n)` with n of course being the modality, we may simply iterate through the modalities with a `TrieNode` being satisfiable if all of its children are satisfiable for each. We group the `TrieNode`'s future modalities in a list by `TrieNode.FutureMods` and say that a future modality, n, exists if there exists a diamond clause $(c \to \Diamond_m d)$ with $n = m$. The modification is shown in Algorithm 2.

Algorithm 2 `CEGARBox`(A, `TrieNode`)

{Inputs: A is a set of unit assumptions and `TrieNode` is at level i in our trie}
$t_0 \leftarrow$ `solve`(`TrieNode.sat`, A)
if t_0 = (unsat, A') **then**
  return Unsatisfiable(A')
else if $t_0 = (sat, \vartheta)$ **then**
  {Check box and diamond clauses that fire under classical valuation ϑ}
  for every future modality $n \in$ `TrieNode.FutureMods` **do**
    for every $(c \rightarrow \Diamond_n d) \in$ `TrieNode.DiaCl(n)` with $c \in \vartheta$ **do**
      $B \leftarrow \{b \mid (a \rightarrow \Box_n b) \in$ `TrieNode.BoxCl(n)` and $a \in \vartheta\}$
      if `CEGARBox`$((d; B),$ `TrieNode.child(n)`$)$ = Unsatisfiable(X) **then**
        $C \leftarrow \{c\} \cup \{a \mid (a \rightarrow \Box_n b) \in$ `TrieNode.BoxCl(n)`, $a \in \vartheta$ and $b \in X\}$
        {Learn new clause $\varphi := \neg C$}
        $\varphi \leftarrow \bigvee_{l \in C} \neg l$
        `addClause`(`TrieNode.sat`, φ)
        return `CEGARBox`(A, `TrieNode`)
      end if
    end for
  end for
  return Satisfiable
end if

3.2 Extensions to Modal Logics with One Axiom

By making modifications to our trie, and in some instances adding to our proof search, we are able to handle all of the axioms of modal cube. There are three aspects within the algorithm design which handle modalities which we need to consider - the modal contexts stored in each $\mathcal{C}_s$ - in itself stored with path s in the trie, their box clauses $\mathcal{C}_s^{\rightarrow\Box}$, and their diamond clauses $\mathcal{C}_s^{\rightarrow\Diamond}$. This section considers extensions to modal logics with one axiom except for the case of euclideaness.

3.2.1 Handling Reflexivity (T)

Reflexivity in a Monomodal Context

We must make modifications to our trie to encapsulate reflexivity. Goré and Kikkert [2021] showed the modifications we needed to be made to encapsulate the characteristic axioms $\Box\varphi \rightarrow \varphi$ and $\varphi \rightarrow \Diamond\varphi$.

Modal contexts: Considering a single modality, we know that if $\Box\varphi$, then φ. Thus, under a single modality we know that if C_t is a descendant of C_s (i.e. it has at least one additional box in its modality), then under reflexivity all of the clauses contained in C_t must be in C_s (or rather $C_t \subseteq C_s$). Thus, by starting at the leaf node of the trie and going upwards until we reach the root node, we may simply propagate the clauses contained in `TrieNode.child(1)` into `TrieNode`.

Box clauses: In every modal context C_s, for each box clause $(a \to \Box b)$ we additionally insert $(a \to b)$ into C_s to capture reflexivity.

Diamond Clauses: As each world is its own successor if there is a diamond clause $c \to \Diamond d$, and d is in the current valuation ϑ then we needn't create a new world as the diamond clause is already satisfied.

We can also go a step further from what was suggested by Goré and Kikkert [2021] by keeping track of the history of box and diamond clauses that have been fired when going down a path. When a world w_k is being evaluated with a set of assumptions from the fired box clauses and diamond clauses from another world w_p, we check the history of assumptions that have been made for all of its ancestors in the set W. If there is a world $w_h \in W$ such that the assumptions made in w_h is a superset of those made for w_k then we simply return satisfiable as instead of creating w_k we may have added the relation $w_p R w_h$. This procedure is called ancestor checking and is sound as all clauses contained in a current world must be contained by the ancestor by reflexivity.

Termination follows from the termination of K. Soundness is obvious. For completeness we can just take the reflexive closure of the model generated by our procedure.

Reflexivity in a Multimodal Context

Reflexivity in a multimodal context is slightly more difficult to handle. The main problem is that we instead of only needing to propagate one modal context at a time as in the monomodal case, we must also propagate up its ancestors (following the above procedure would put $\mathcal{C}_{[1]}$ in $\mathcal{C}_{[]}$ and $\mathcal{C}_{[1,2]}$ into $\mathcal{C}_{[1]}$ but not $\mathcal{C}_{[1,2]}$ into $\mathcal{C}_{[2]}$). Thus, we create a new function `overShadow` which takes in two tries and places the clauses in the second trie into the first. The algorithm is displayed in Algorithm 3 with a visual example being displayed in Figure 3.1.

Algorithm 3 `overShadow(BaseTrie, ShadowTrie)`

```
{Inputs: BaseTrie, a trie to impose the clauses of ShadowTrie onto}
Insert cpl clause, box clause and diamond clauses of ShadowTrie into BaseTrie
for modality n ∈ ShadowTrie.FutureMods do
    overShadow(BaseTrie.child(n), ShadowTrie.child(n))
end for
```

Then by starting at the leaf nodes and going upwards to the root, we can use `overShadow(TrieNode,TrieNode.child(n))` for each child to ensure that $\mathcal{C}_s = \overrightarrow{\mathcal{C}_s}$ as implied by reflexivity. Note that under this method $\mathcal{C}_{[1,1]}$ would be propagated to $\mathcal{C}_{[1]}$ twice. Once for the `overShadow` called directly on $\mathcal{C}_{[1]}$ and another through recursion on the subsequent call on $\mathcal{C}_{[]}$. We can improve the efficiency by ignoring the case in the recursive call where `overShadow` is called on the node's final modality.

We apply the same rule for box clauses as well as diamond clauses and we have handled reflexivity in the multimodal context.

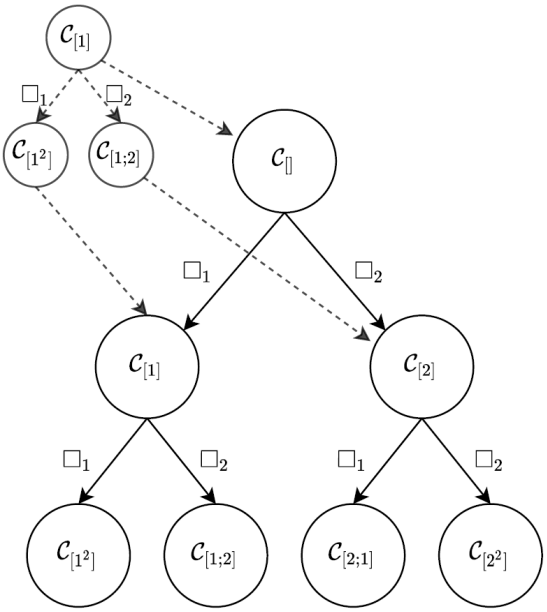


Figure 3.1: An example of overshadowing. The clauses of $\mathcal{C}_{[1]}$, and its descendants, are being imposed on $\mathcal{C}_{[]}$ as shown.

3.2.2 Handling Transitivity (K4)

Transitivity in a Monomodal Context

Goré and Kikkert [2021] discussed the theory behind the monomodal logic $K4$ but did not implement it in their prover. Transitivity is reflected by the axiom $\Box\varphi \rightarrow \Box\Box\varphi$. Let $\mathcal{C}_{[1]\rightarrow[1^k]}$ represent the union of the modal contexts $\mathcal{C}_{[1]}, \mathcal{C}_{[1^2]}, \ldots, \mathcal{C}_{[1^k]}$. As $\Box\varphi \rightarrow \Box\Box\varphi$ and remembering $\mathcal{C}_{[1^n]}$ represent clauses with modality $\Box^n$ We must therefore have in our final modal context for each level $\mathcal{C}_{[1]} = \mathcal{C}_{[1]\rightarrow[1^1]}, \mathcal{C}_{[1^2]} = \mathcal{C}_{[1^1]\rightarrow[1^2]}, \ldots, \mathcal{C}_{[1^k]} = \mathcal{C}_{[1^1]\rightarrow[1^k]}$ with k being equal to the maximum modal depth. Hence, we modify each modal context by, starting at the trie node of depth 1 below the root node, inserting the node into its child (i.e. we insert $\mathcal{C}_{[1]}$ into $\mathcal{C}_{[1^2]}$, before inserting $\mathcal{C}_{[1^2]}$ into $\mathcal{C}_{[1^3]}$ and so on).

For our box clauses, if a valuation of a world is such that a is true and there is a box clause $a \rightarrow \Box b$, then not only is b true in all immediately subsequent worlds but b is also true in all ancestors by transitivity. Hence we modify our box clauses as follows. For each box clause we insert rename b to be P_b (persistent b). We modify the modal context by inserting the cpl clause $a \rightarrow P_b$ and replace the box clause with $P_b \rightarrow \Box P_b$. Following this we add $P_b \rightarrow \Box P_b$ and $P_b \rightarrow b$ to all ancestors of our current modal context.

No changes are needed for our diamond clauses but there is the problem that transitivity may cause the search procedure to loop forever. For example consider the formula $\Diamond\top \wedge \Box\Diamond p$. In the final modal context $\mathcal{C}_{[1]}$ we have $\Diamond p$, meaning that under transitivity a naive implementation would cause the prover to loop forever generating modalities of infinite depth. To fix this we employ a variant of the ancestor

checking described in the reflexivity section. Once we reach the final modal context if there are any fired diamond clauses we gather the triggered assumptions, and re-evaluate the current modal context under these assumptions while keeping track of the history of assumptions we have made. If the assumptions are a subset of any of the assumptions of any of the worlds in our history then we assume that the world we are in is satisfiable by adding a link back to the relevant world. In our example what this looks like is in the root world we have $\Diamond\top$ so we need to generate a world with the assumption $\top$ in the next modal context. So we evaluate the $\mathcal{C}_{[1]}$ under the assumption $\top$. This is the final modal context as there are no subsequent ones, and we have another triggered diamond clause $\Diamond\top$. We store $\top$ in our history and re-evaluate the context under the triggered diamond clause p. As p is not a subset of any of the assumptions for the worlds in our history. we thus have another re-evaluate $\mathcal{C}_{[1]}$ in a new world as $\Diamond p$ is fired. Now p is in our history of assumptions so instead we could have linked back to the world containing the assumption p so we return satisfiable. This backpropagates through recursion and we discover that the formula is satisfiable. As the number of diamond clauses and box clauses are finite, this procedure is guaranteed to terminate as the number of different assumptions which could be made is in itself finite. Soundness follows from observing that the above transformations are sound. For completeness we simply take the transitive closure of our model.

Transitivity in a Multimodal Context

Transitivity in a multimodal context is very similar to the monomodal one. We have under transitivity that $\Box_n \to \Box_n\Box_n$. Hence, in a sense, different modalities act in a sense as circuit breakers when propagating clauses.

For our modal contexts hence, we want $\mathcal{C}_{[x_1^{n_1};x_2^{n_2};\ldots;x_k^{n_k}]}$ to propagate to all $\mathcal{C}_{[x_1^{m_1};x_2^{m_2};\ldots;x_k^{m_k}]}$ with $m_i \geq n_i$. In order to do just this we can make use again of our `overShadow` function described in Algorithm 3. Starting with the root `TrieNode`, if the child of modality n, `TrieNode.child(n)`, has itself a child of the same modality, we use `overShadow` to overlay `TrieNode.child(n)` onto `TrieNode.child(n).child(n)` (skipping sub-modality n as that is already the child). This has the effect of imposing $\overrightarrow{\mathcal{C}}_{[}n]$ onto $\overrightarrow{\mathcal{C}}_{[}n^2]$ and so forth going down the trie, implementing transitivity with respect to the modal contexts.

The case for our box clauses is the same. The only difference to extend this to the multimodal case is that we replace $a \to \Box_n b$ with $a \to P_b$ and $P_b \to \Box_n P_b$ and propagate the new box clause together with $P_b \to b$ down all children with modality n.

Ancestor checking is exactly the same once we reach a final modal context with respect to one specific modality (i.e. if the final box is of modality n and there is no child modal context with modality n). If there are any fired diamond clauses with respect to this modality, we like before, re-evaluate the modal context under the new assumptions while keeping track of history.

3.2.3 Handling Symmetry (KB)

Symmetry in a Monomodal Context

The axiom for symmetry is $p \to \Box\Diamond p$ and its dual $\Diamond\Box p \to p$.
First we prove that $(\Diamond\top \wedge \Box\Box\varphi) \to \varphi$ is valid under symmetry, or in other words if there exists a successor world then we can insert $\mathcal{C}_{[1^{k+2}]}$ into $\mathcal{C}_{[1^k]}$.

Lemma 1. *$(\Diamond\top \wedge \Box\Box\varphi) \to \varphi$ is valid under symmetry.*

Proof. Assume that the $(\Diamond\top \wedge \Box\Box\varphi) \to \varphi$ is invalid - that is there is a symmetric Kripke model where world w makes $\Diamond\top \wedge \Box\Box\varphi$ true but φ not and we will find a contradiction.
As $\Diamond\top$ is true, in w we must create a world v with wRv. Thus, also $\Box\varphi$ must be true in v. Then by symmetry vRw so therefore φ is true in w. But φ is false in w and we have found a contradiction.
Therefore, $(\Diamond\top \wedge \Box\Box\varphi) \to \varphi$ is valid under symmetry. QED

After rearranging this formula we see a slightly more useful way of writing it is $\Box\bot \vee (\Box\Box\varphi \to \varphi)$. This is all the information we need to handle our modal contexts in the case of symmetry.
For any world which is not the root world, $\Box\bot$ must be false as this world is a successor of another world and symmetry hold. Hence $\Box\Box\varphi \to \phi$ must be true. Therefore, for $\mathcal{C}_{[1^k]}$ we must propagate into the modal context $\mathcal{C}_{[1^{k+2m}]}$ for $m, k \geq 1$.
The root world however is slightly more complex. Here, either $\Box\bot$ must be true (there are no future worlds) or we must propagate into the root world all $\mathcal{C}_{[1^{2m}]}$ by symmetry. The trick to handle this is simple. Into the root world we insert a new box clause $s \to \Box\bot$ then into the root world we may insert for each clause $\varphi \in \mathcal{C}^{cpl}_{[1^{2m}]}$, $s \vee \varphi$. Similarly we apply a renaming process for the box and diamond clauses such that for each $(a \to \Box b) \in \mathcal{C}^{\to\Box}_{[1^{2m}]}$ we add $(x_a \to \Box b)$ to the box clauses and $(a \wedge \neg s) \to x_a$ to the cpl clauses. Similarly for the diamond clauses for each $(c \to \Diamond d) \in \mathcal{C}^{\to\Diamond}_{[1^{2m}]}$ we add $(x_c \to \Diamond b)$ to the diamond clauses and $(c \wedge \neg s) \to x_a$ to the cpl clauses.
Hence for the modal contexts, to more efficiently do this, effectively for $\mathcal{C}_{[1^k]}, k \geq 1$ we insert $\mathcal{C}_{[1^k+2]}, m \geq 1$ - propagating upwards starting at the leaf nodes. Then for $\mathcal{C}_{[]}$ we effectively insert $\Box\bot \vee \varphi$ for each cpl clause, box clause and diamond clause$\varphi \in \mathcal{C}_{[1^2]}$.
For the box clauses, if a `TrieNode` makes $a \to \Box b$ fire, then b must be true in the parent `TrieNode`. To encapsulate this we show that $\Box(a \to \Box b) \to (\neg b \to \Box\neg a)$ is valid on symmetric frames.

Lemma 2. *$\Box(a \to \Box b) \to (\neg b \to \Box\neg a)$ is valid on symmetric frames.*

Proof. Assume $\Box(a \to \Box b) \to (\neg b \to \Box\neg a)$ is not valid under symmetry. Then, there is a Kripke model with a world w which makes $\Box(a \to \Box b)$ true and $(\neg b \to \Box\neg a)$ false. Thus, in this world b is false and $\Diamond a$ is true. Hence the world, w, has a successor v so that uRv which makes a and $(a \to \Box b)$ both true. This means that in v, $\Box b$ is true and by symmetry b is true in w. But b is false in w and hence we have

found a contradiction. Therefore $\Box(a \to \Box b) \to (\neg b \to \Box\neg a)$ is valid on symmetric frames. QED

Therefore, for each box clause $(a \to \Box b) \in \mathcal{C}^{\to\Box}_{[1^{k+1}]}$ we insert $(\neg b \to \Box\neg a)$ into $\mathcal{C}^{\to\Box}_{[1^k]}$. Note this has the effect of, in the proof search procedure if $a \to \Box b$ is fired in a `TrieNode`, then b must be true in the parent node otherwise the parent node would make a false.
Finally for the diamond clauses, we needn't do anything special. That being said, we can improve the efficiency of our proof search by checking if the assumptions that re to be made in our current world are valid in the parent `TrieNode` to reduce the depth of our search.
Termination again follows from the termination of K. Soundness follows by observing that our transformations are indeed sound. For completeness we can take the symmetric closure of the Kripke Model.

Symmetry in a Multimodal Context

Handling symmetry in a multimodal context, is again, slightly more complex. However, yet again, we can make use of a modified variant of the `overShadow` function. The main complicating factor is for cases such as $\mathcal{C}_{[1^2;2^2]}$ which need to be propagated to $\mathcal{C}_{[2^2]}$ in the case that $[]_2\bot$ is false in the root world. To handle this, like before we start at the leaf nodes and work our way up the trie. If the `TrieNode`'s predecessor is of the same modality as that of two levels deep, then we can simply insert $\mathcal{C}_{[\ldots;m^{k+2}]}$ into $\mathcal{C}_{[\ldots;m^k]}$ $(k \geq 1)$. However, if the modality is different, i.e. we are trying to insert $\mathcal{C}_{[\ldots;n;m^2]}$ into $\mathcal{C}_{[\ldots;n]}$, then like in the monomodal case, either $\Box_m\bot$ is true, or the formulas should be inserted. So, just as in the monomodal case for the root world we insert $\Box_m\bot \vee \varphi$ for each cpl, box and diamond clause $\varphi \in \mathcal{C}_{[\ldots;n;m^2]}$. Then for each ancestor of $\mathcal{C}_{[\ldots;n]}$ such that the initial modality is different to m, call the path s, we extend the ancestor with the clauses of all $\varphi \in \mathcal{C}_{[\ldots;n;m^2]+s}$ by rerunning the clausification procedure from $\mathcal{C}_{[\ldots;n]}$ with $\Box_m\bot \vee \Box_s\varphi$. This essentially creates variables in the trie keeping track of the paths of the trie that is visible to the current model.
Handling box clauses is almost exactly the same as in the monomodal case. For each $(a \to \Box_m b) \in \mathcal{C}_{s+[n]}$ we insert $(\neg b \to \Box_m\neg a)$ into $\mathcal{C}_{s+[n]}$ if $n = m$. This ensures, as in the monomodal case, that the box clauses are satisfied for parent `TrieNode`'s under symmetry.
Similarly for the diamond clauses. We include the improvement to the efficiency of the proof search by checking if the assumptions that are to be made from our current world are valid in the parent node if the modality of the assumptions is equal to the modality of the link connecting them.

3.2.4 Handling Seriality (D)

Seriality is trivial in both the monomodal and multimodal context. Seriality states that every world must have a successor. The idea is, for every $\mathcal{C}_s$ we add $\Diamond_m\top$ for

each modality m expressed in the formula. Note that as the trie is finite, we can consider the future worlds of the leaf nodes of the trie to only contain their fired box and diamond clauses. These worlds themselves have no box or diamond clauses and hence we can trivially make them reflexive, satisfying the seriality condition.

3.3 Modal Logics involving Reflexivity

We now consider the modal logics with more than one axiom satisfying reflexivity.

3.3.1 Handling Reflexivity and Transitivity (S4)

Reflexivity and Transitivity in the Monomodal Context

Goré and Kikkert [2021] showed the following for $S4$. The axioms for reflexivity and transitivity respectively are $\Box\varphi \to \varphi$ and $\Box\varphi \to \Box\Box\varphi$. In a monomodal context reflexivity guarantees that every formula is unboxed to the root world so that $\mathcal{C}_{[]}$ is equal to $\mathcal{C}_{[]\to[1^k]}$ where k is the maximum modal depth. Similarly, $\mathcal{C}_{[1]}$ is equal to $\mathcal{C}_{[1]\to[1^k]}$ by the same reasoning. Then, by transitivity every successor to $\mathcal{C}_{[1]\to[1^k]}$ must in itself be equal to $\mathcal{C}_{[1]\to[1^k]}$. Therefore, in a monomodal setting, we can represent the information needed for $S4$ with two trie nodes - $\mathcal{C}_{[]\to[1^k]}$ and its successor $\mathcal{C}_{[1]\to[1^k]}$. We can achieve this form by first running the same preprocessing for reflexivity, then running the preprocessing for transitivity up to the second node. Then, when handling the fired diamond clauses for $\mathcal{C}_{[1]\to[1^k]}$ we apply the strong ancestor checking rule as used in reflexivity (which is equivalent to the transitivity rule here as the root world has no assumptions and the trie is only one level deep, but comes more in use for the multimodal context).

Reflexivity and Transitivity in the Multimodal Context

Reflexivity and transitivity in the multimodal context is only slightly more complex, only that the modal depth is not limited to two levels. For example, consider $\Box_1\Box_2\Box_1\Box_2\varphi$. Reflexivity guarantees that $\Box_1\varphi$ is true, and hence by transitivity $\Box_1^n\varphi$. However, while transitivity implies all the formula in $\mathcal{C}_{[1]}$ are in $\mathcal{C}_{[1^n]}$, there is no such guarantee from $\mathcal{C}_{[1]}$ to $\mathcal{C}_{[1;2]}$. The rule is that $\mathcal{C}_{[\ldots,m]} = \overrightarrow{\mathcal{C}}_{[\ldots,m]} = \overrightarrow{\mathcal{C}}_{[\ldots,m^k]}$ by reflexivity and then transitivity. This means that the successor of a trie node with the same modality as the final modality of the trie node is itself! Otherwise, we follow a normal trie pattern. Within a modal logic with two modalities 1 and 2, one can view the final trie structure as an alternating path - i.e. the final modal context $\mathcal{C}_s$ with $s = [m_1, m_2, \ldots, m_n]$ such that $m_i \neq m_{i+1}$. Hence we may simply run the preprocessing for reflexivity first, then transitivity (noting it is not necessary to handle reflexivity for the persistent boxes the transitivity processing creates, i.e. for $P_b \to \Box P_b$ we needn't add $P_b \to P_b$, nor do we need to backpropagate the right hand side of persistence, $\Box(P_b \to b)$, as it is already satisfied through the backpropagation of the original $a \to \Box b$ and hence $a \to b$ via reflexivity). We run the same proof

system with the strong ancestor checking defined for reflexivity. Note that transitivity does not break the stronger reflexivity ancestor checking rule because of the persistent box clauses (if some ancestor world satisfies the assumptions then so must the ancestor's descendants as the box clauses are defined to be persistent).

3.3.2 Handling Reflexivity and Symmetry (B)

The rules for handling modal logic B are far easier to understand than that of KB, simply because we needn't worry about backpropagating the modal contexts under symmetry as they have already been done and satisfied by reflexivity! Hence, we begin by running our reflexivity preprocessing procedure. Then we need only satisfy the box clauses to handle symmetry. The process is exactly the same - for each $(a \rightarrow \Box_m b) \in \mathcal{C}_{s+[n]}$ we insert $(\neg b \rightarrow \Box_m \neg a)$ into $\mathcal{C}_s$ if $m = n$. Note in this instance we don't need to rerun reflexivity with respect to these newly inserted box clauses because $(\neg b \rightarrow \neg a)$ is merely the contrapositive of $(a \rightarrow b)$ which has already been inserted through reflexivity. Further, nor do we need to backpropagate $(\neg b \rightarrow \Box_m \neg a)$ as reflexivity has already backpropagated $(a \rightarrow \Box_m b)$ and the symmetry rule thus handles that automatically.

3.3.3 Handling Reflexivity, Symmetry and Transitivity (S5)

As was discussed in the above section, reflexivity already handles the modal context preprocessing required for symmetry. Thus, we may begin by merely applying the preprocessing for $S4$ (reflexivity and transitivity). The only thing which thus needs to be changed for $S5$ is looking at the box clauses. Under persistence every box clause is of the form $P_b \rightarrow \Box_m P_b$. Applying the symmetry rule, we then get $\neg P_b \rightarrow \Box_m \neg P_b$. Under a single modality, this has the effect of having either P_b is true everywhere or false everywhere which is exactly what we want for $S5$ as this causes (under a single modality) all possible worlds to be connected everywhere. Hence, by modifying our $S4$ procedure to include the clause $\neg P_b \rightarrow \Box_m \neg P_b$ in addition to $P_b \rightarrow \Box_m P_b$ symmetry is hence satisfied.

3.4 The Case of Euclideaness

Euclideaness is a special single axiom because of its close relationship with $S5$, we consider this case semi-independently.

3.4.1 Handling Euclideaness (K5)

The frame condition for euclideaness is $uR_m v \wedge uR_m w \implies vR_m w$ for modality m. First notice that this relationship puts no additional constraints on the root world as under the rooted tree model it would only appear on the left hand side of the accessibility relation in $uR_m v \wedge uR_m w$ and the right hand side does not involve u. Let's consider the root world to be x and suppose the existence of a successor world

y such that xR_my. We claim that y obeys reflexivity, symmetry and transitivity under modality m.
For the reflexivity of y, observe that xR_my. Hence, because of euclideaness yR_my so y is reflexive. Now suppose y has a successor z. Because yR_my and yR_mz we have by euclideaness zR_my and hence y is symmetric. Finally suppose y has a successor z_1 which has a successor z_2. Because z_1R_my by the successor of y being symmetric and $z_1R_mz_2$ yR_mz_2 and so y is transitive.
Further as y's successors with respect to modality m also obey euclideaness, they must too be reflexive, symmetric and transitive. Therefore, the successors of the root world under euclideaness with modality m obey $S5$ with respect to modality m.
Additionally, the axiom associated with euclideaness is $\Diamond_m p \to \Box_m \Diamond_m p$. What this tells us is if we have a diamond clause $c \to \Diamond_m p$, we can satisfy the axiom by creating a new literal x and inserting $c \to \Box_m x$ along with $x \to \Box_m x$ into the current modal context $\mathcal{C}_s$, and into $\mathcal{C}_{s+[m]}$ we may insert $x \to \Diamond_m p$. As the next modal context must satisfy $S5$ with respect to modality m it is handled with the rules according to $S5$. We propagate this preprocessing downwards for modalities not handled by $S5$.
Combining these things together, in order to handle euclideaness, we first apply the diamond preprocessing described above. Then, for each $\mathcal{C}_{[m]}$, we run the $S5$ preprocessing only with respect to modality m. For any other successor of different modality $n \neq m$ $\mathcal{C}_{[m]}$ has we run the euclidean preprocessing again making the next level with respect to modality n (making $\mathcal{C}_{[m,n]}$ $S5$ with respect to modality n), and so on and so forth until we reach the leaf nodes.
For running the solver hence under euclideaness, we follow the exact same process. Applying the reflexivity ancestor rule for the $S5$ worlds when expanding down the modality it is $S5$ with respect to (ignoring earlier worlds such as the root world). In fact, because every $S5$ world with respect to modality m is a successor with respect to that modality, the formula in each of its successors of modality m is guaranteed to be identical due to transitivity. Thus the ancestor rule only applies to the history of assumptions for itself!

3.5 Other Modal Logics with More than One Axiom

3.5.1 Seriality and Transitivity (D4)

Seriality and transitivity is extremely trivial under one modality. We simply insert $\Diamond\top \wedge \Box\Diamond\top$ before applying the transitivity preprocessing which hence guarantees every world has a successor.
Under multiple modalities, it is still trivial, but the above trick doesn't work - for example it doesn't guarantee $\mathcal{C}_{[1;2]}$ has a successor. In this case we simply iterate over every level of the trie and insert $\Diamond_m\top$ for every expressed modality m. Then, we merely apply the transitivity preprocessing and use the transitivity prover. The ancestor checking from the prover thus guarantees that every world has a successor.
We can slightly improve the efficiency of the diamond insertion process for seriality by observing if we have inserted a diamond of modality n into $\mathcal{C}_{s+[n]}$ we needn't

insert a diamond of modality n into $\mathcal{C}_{s+[n^k]}$ ($k \geq 2$) as it will already be put there by transitivity.

3.5.2 Seriality and Symmetry (DB)

Seriality and symmetry, like reflexivity and symmetry is a lot easier to implement than symmetry on its own. Because every world has a successor with respect to every modality, $\Box_m \bot$ is always false. Hence the symmetry rule simply becomes $\Box_m \Box_m \varphi \to \varphi$ meaning that we can use our `overShadow` function to impose each $\overrightarrow{\mathcal{C}}_{[\ldots;n;m^2]}$ onto $\mathcal{C}_{[\ldots;n]}$. We apply the symmetry box clause rule and then we must simply insert $\Diamond_m \top$ into every modal context for every modality m. Note that seriality is guaranteed because if a world with no box clause or diamond clauses is created we can make it trivially reflexive and the number of box and diamond clauses is finite. We can slightly increase the efficiency of the diamond insertion by observing that under symmetry, the modal context $C_{s+[n]}$ always has a successor of modality n C_s.

3.5.3 Seriality and Euclideaness (D5)

Seriality and euclideaness is also relatively trivial. We can begin by running the euclidean preprocessing first. Then inserting $\Diamond_m \top$ into every modal context for every modality m. As discussed earlier, the next level is $S5$ with respect to modality m so the next level obeys seriality with respect to that modality. The process is propagated downwards and any empty modal context is made trivially reflexive with respect to all modalities.

3.5.4 Transitivity and Euclideaness (K45)

Transitivity and euclideaness is once again quite trivial. We merely run the preprocessing for transitivity first before applying the preprocessing for euclideaness. Again, using ancestor checking to guarantee termination.

3.5.5 Seriality, Transitivity and Euclideaness (D45)

Given a trivial solution to transitivity and euclideaness, adding seriality is also quite trivial. Simply put, we run the preprocessing to $K45$ first before adding $\Diamond_m \top$ for every modality m to every trie node. Once again using ancestor checking to guarantee termination and making worlds without box or diamond clauses reflexive with respect to each modality to satisfy seriality.

3.5.6 Symmetry with Transitivity or Euclideaness (KB4=KB5=K4B5)

This is logic is very close to $S5$, yet isn't exactly. In fact, given with the additional constrain of seriality we get $S5$. However, to handle $KB4 = KB5 = K4B5$ we consider the symmetric and transitive case. We begin by applying the preprocessing for transitivity as per the normal rules. Then, we apply the procedure given by

symmetry for $KB4$. Note that doing it in this way has the additional effect of propagating $\mathcal{C}_{[1]}$ into $\mathcal{C}_{[]}$ (with the $\Box_m \bot$) of course which is what we want as $KB4$ is reflexive with respect to m. We then handle the box clauses exactly as in $KB4$ observing that this has the effect of having if P_b from $P_b \rightarrow \Box_m P_b$ is true in the root node, then P_b is true everywhere with modality m, otherwise is true nowhere with modality m.

3.6 Summary

This chapter formalised the changes that need to be made in order to handle all fifteen normal modal logics found in the modal cube. It was shown that we can handle each modal logic just by simply preprocessing the trie. This is excellent news because it means that the core proving algorithm which caused the large jump in efficiency over other modal logic theorem provers is the same in almost all cases! The exceptions being in logics including reflexivity having rules for early termination, and as for the logics which involve loop checking such as $S4$ CEGARBox was still found to be far more efficient than any other theorem prover as shown by Goré and Kikkert [2021]. We next formalise additional extensions in the subsequent chapter.

Chapter 4

Additional Extensions and Optimisations

This chapter introduces several additional extensions to CEGARBox. Global assumptions refer to a set of formula Γ which must be true in every possible world [Goré, 2019]. Hence, we begin in Section 4.1 where we discuss the theory that is needed for CEGARBox to handle global assumptions in a multimodal context (though remaining unimplemented). We then follow in Section 4.2 providing the justification for refactoring the program from Haskell into C++ and additional features that it contains. We final discuss further optimisations that were made in Section 4.3 before summarising.

4.1 Global Assumptions

We denote our global assumptions to be a set of modal formula Γ. When assessing whether φ is satisfiable with global assumptions Γ a naive solution would be to insert Γ into every world. However, this may cause infinite loops - for example consider when $\Gamma = \{\Diamond_1 p\}$. The key insight we make is to observe the parallelity between global assumptions and transitivity. In fact, global assumptions are even stronger than transitivity obeying $\varphi \rightarrow \Box_m \varphi$ for each modality m with $\varphi \in \Gamma$. The proof of such is trivial by contradiction assuming the existence of a world.

Lemma 3. *Global assumptions obey $\varphi \rightarrow \Box_m \varphi$ for each modality m with $\varphi \in \Gamma$*

Proof. Suppose instead in a world $\varphi \rightarrow \Box_m \varphi$ is not true where $\varphi \in \Gamma$ - i.e. we have φ is true and $\neg\Box_m \varphi$ true. This means by moving the implication inwards that $\Diamond_m \neg\varphi$ is true. So we must create a successor world satisfying $\neg\varphi$. But as $\varphi \in \Gamma$ φ is true in this world so we have $\varphi \wedge \neg\varphi$ which is obviously a contradiction. Hence, $\varphi \rightarrow \Box_m \varphi$ is valid for each modality m with $\varphi \in \Gamma$. QED

To encapsulate this in the CEGARBox algorithm we create a new trie with modal contexts $\mathcal{G}$ from the same normal forming process as $\mathcal{C}$. We then use our `overShadow` function in order to impose $\mathcal{G}$ onto every trie node of $\mathcal{C}$ and run the CEGARBox algorithm accordingly. If we ever need to create a world which contains

only global assumptions, under a single modality we can keep expanding down the global assumptions until we have the union of all global assumptions in a trie node. From there we simply run the ancestor checking algorithm repeatedly on that trie node as is for transitivity to evaluate its satisfiability. The case is slightly more complex for multiple modalities. For example consider $\Gamma = \{\Box_1 p, \Box_1\Box_2 q\}$. In this instance no trie node will contain both p and q, hence we need something slightly better. Consider the length of the maximum modal depth of the global assumptions trie - in our example this is 2 (from $\Box_1\Box_2 q$). Then every trie node of maximal depth represents the clauses that are infinitely repeating. In our example there are 4 - any node ending with $\Box_2\Box_2$ is empty, both $\Box_2\Box_1$ and $\Box_1\Box_1$ contain p and $\Box_1\Box_2$ contains q. Every successor to each of these trie nodes will be one of these four options. So instead of keeping track of the history of assumptions that are made as for transitivity, we store the history of assumptions paired with each trie node of maximal depth. If we expand into one of these trie nodes with the assumptions in that trie node's history we return satisfiable for that case as instead we could have added a relation to that previous node. This is guaranteed to terminate as the size of assumptions which can be made and possible recurring worlds are each finite.

4.2 Refactoring to C++

A large part of the time I spent working on this project was dedicated to refactoring CEGARBox from Haskell into C++. This was done for several main reasons. One of the main ones is simply that more people are competent in C++ as opposed to Haskell as it is much more widely used (myself certainly included). As the current state-of-the-art modal SAT solver, for the longevity of the project and to allow more people to contribute to it, that was one contributing factor to the decision. Further, while Haskell is still great performance-wise, C++ is still considered the gold standard for performance and for such a performance-driven problem it does make more sense to lean towards C++. Most other SAT solvers of this nature are also written in C/C++ including BDDTab, FACT++ and K_SP to name a few [Goré et al., 2014; Tsarkov and Horrocks, 2006; Nalon et al., 2017]. A consequence of this is that we may easily swap out the underlying SAT solver each node of the trie relies upon. In this work we rely on MiniSAT [Sorensson and Een, 2005] but provide an abstract class for any other SAT solver to implement. This can allow for easy comparisons of the effect the underlying SAT solver has on the performance of CEGARBox. Additionally, we provide an abstract class for each proving method. This allows us to easily expand the code to handle even further logics with relative ease. One more justification was that for some extremely large problems, the Haskell version ran out of memory, this was while the C++ version was able to run the problem without error.

4.2.1 Problems with Parsing

Initially, a recursive parser was designed to handle the input formulas. It was found through experimentation that for particularly large input formulas with the order of tens of thousands of nested brackets, the stack overflowed causing a segmentation fault. In order to fix this, instead the recursive methods were simulated iteratively using a stack data structure to keep track of function calls and return values. It was found that this did not sacrifice the efficiency of the prover, nor did the problem appear further.

4.3 Further Optimisations

One optimisation which was stated in the paper for CEGARBox [Goré and Kikkert, 2021] was that all literals were biased to be false in the SAT solver so that we can minimise the number of diamond and box clauses that fire. This was commented out in the code for the original CEGARBox but one can actually go further than this. Instead we can simply bias all literals on the left hand side of the box and diamond clause to be false, rather than every literal. That way the importance of the box and diamond literals not firing is increased, and hence, the number of worlds which need to be created when determining satisfiability should be minimised.
Additionally, we make use of bitsets in order to memoise the solutions for each trie node. This allows us to make use of bit operations to quickly determine whether a set of assumptions being made is a subset of some other assumption which have already returned satisfiable. That way, we minimise the number of calls to SAT solvers which need to be made, and hence also the number of times nodes need to be explored.

4.4 Summary

The above chapter explored additional optimisations that were made to CEGARBox. We formalised how global assumptions could be handled under the current algorithm, justified the refactoring to C++, and additionally made additional optimisations on top of what has already been done. We next verify the performance of our modal SAT solver.

Chapter 5

Methodology and Results

This section details the experiment which were conducted to verify the efficiency of the new prover. Section 5.1 details the design of the experiment before the results are provided in Section 5.2. We critically analyse these results in Section 5.3 and finally discuss justifications for correctness in Section 5.4.

5.1 Experiment Design

The original CEGARBox implementation [Goré and Kikkert, 2021] was found to be the best theorem prover on all experimented benchmarks by an order of magnitude, except for the MQBF benchmarks [Massacci and Donini, 2000] where it was beaten by the K_SP theorem prover [Nalon et al., 2017] by a relatively small margin compared to CEGARBox's lead on all other provers.
Further, in the original paper [Goré and Kikkert, 2021], and as was earlier mentioned, it was stated that the underlying SAT solver biased all literals to be false in order to reduce the number of diamond and box clauses that fire. This code was commented out in the final version.
Hence, it was decided to set up an experiment with two key purposes in mind. The first purpose was to ensure that the shift to C++, and other optimisations, lead to a gain in the efficiency of the solver. The second key purpose was to compare the efficiency of the new version against that of K_SP. If the C++ version leads to a gain in efficiency over the two Haskell versions, and is more efficient than K_SP, we should be able to claim that it is the most efficient theorem prover for the so far tested logics.
The MQBF benchmarks [Massacci and Donini, 2000] were used and the experiment was conducted for modal logics K, so we can compare our efficiency against K_SP and the Haskell implementations, and T so we can test our efficiency on additional modal logics. Due to some time constraints (caused by late modifications to our prover which added slightly more overhead for ideally greater efficiency), only modal logics K and T were tested in this report. Work comparing efficiencies for the other thirteen modal logics are ongoing and the early experimentation is showing promising results.
The experiments were conducted on a virtual machine with an Intel(R) Xeon(R) CPU E5-2640 v4 CPU clocked at 2.40GHz as well as 8GB of RAM. Each problem had a

timeout of 40 seconds and results were compared to ensure correctness. Our version was compiled under the C++17 std with the O3 optimisation flag. Version 0.1.3 of the K_SP prover was used under the ordered configuration.

5.2 Results

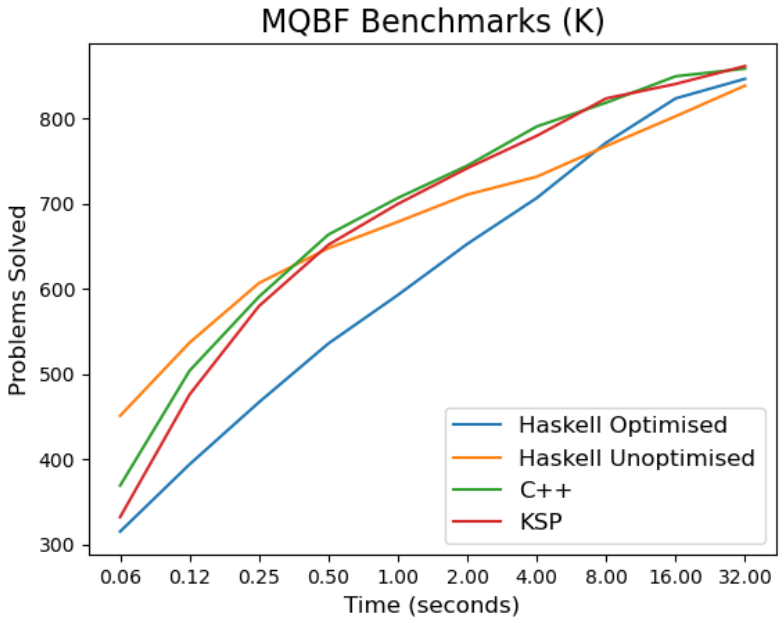
The time taken for each prover to solve each benchmark within the time limit was recorded. The graphs shown in Figure 5.1 show the results under the modal logics K and T and express the number of individual problems that can be solved within varying time limits. Additionally, Table 5.1 and Table 5.2 show explicitly for modal logics K and T respectively how many problems were solved by each prover at certain time intervals. Further, we recorded the average time needed for each prover to solve the problems it did without timing out.

5.3 Discussion

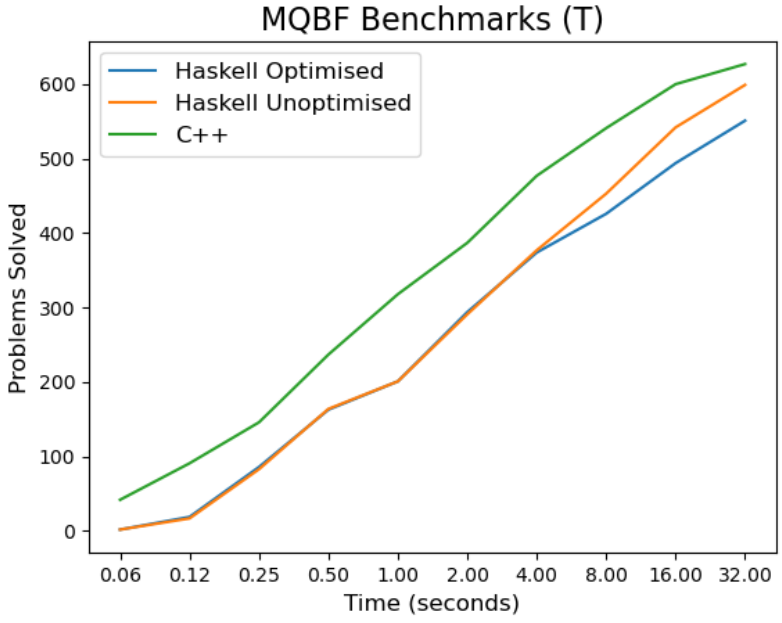
The results show that for modal logic T, our C++ version is consistently faster than its optimised and unoptimised Haskell counterparts. After one second our C++ version had solved over 50% more problems than either of the Haskell versions as shown in Table 5.2. Interestingly, the "optimised" Haskell appears to perform the worst in each of these cases. This is upon initial thought slightly surprising as setting all literals to be false should theoretically minimise the number of boxes and diamonds which need to be fired. However, this may be because by setting all literals to be false, this would likely make many clauses in each underlying SAT solver false. The SAT solver doesn't therefore know which variables to prioritise changing to true and hence it would in many cases simply add more unnecessary overhead to the prover.
The results for modal logic K are very promising. In the figure, the unoptimised Haskell version initially performs the best before it is quickly overtaken by both the K_SP version and our C++ version. A possible explanation for the initial lead of the

Prover	Problems Solved (1s)	Problems Solved (16s)	Problems Solved (32s)
Haskell Unoptimised	593	824	847
Haskell Optimised	679	803	839
C++	**707**	**850**	859
$\boldsymbol{K_SP}$	700	841	**862**

Table 5.1: The number of MQBF benchmarks (K) each prover was able to solve within different time limits.



(a) The number of K problems where a solution is found within the specified time.



(b) The number of T problems where a solution is found within the specified time.

Figure 5.1: The results over the MQBF benchmarks for modal logics K and T.

Prover	Problems Solved (1s)	Problems Solved (16s)	Problems Solved (32s)
Haskell Optimised	201	494	551
Haskell Unoptimised	201	542	599
C++	**318**	**600**	**627**

Table 5.2: The number of MQBF benchmarks (T) each prover was able to solve within different time limits.

Prover	Haskell Optimised	Haskell Unoptimised	C++	K_SP
Average Time for K (s)	2.72	2.30	1.22	1.51
Average Time for T (s)	6.08	6.16	3.96	–

Table 5.3: The average time each prover took to solve every MQBF benchmark it did within the time limit.

unoptimised version could be that it has very limited overhead as it only considers one modality and doesn't bias any literals. As we move to bigger problems however, we gain more and more efficiency from our preprocessing and jump in front. Surprisingly our C++ version and K_SP are consistently on par with one another solving a very similar number of problems as shown in Table 5.1. To compare performance further we analyse the average time it took for each prover to solve the problems within the time limit. Table 5.3 shows that on average, It took the C++ version 0.3 seconds less than K_SP to solve each problem. We hence conclude that our C++ version is slightly superior but still very close to that of K_SP on the MQBF benchmarks. Goré and Kikkert [2021] showed that in every other benchmark CEGARBox dominates all other state-of-the-art theorem provers. As our version dominates the Haskell implementations, and the Haskell unoptimised implementation was the most efficient theorem prover for modal logics K and T by orders of magnitude except in this one case, we can now conclude that CEGARBox is the most efficient theorem prover for modal logics K and T (though further experimentation is needed to help solidify this claim).

That being said there are several limitations to the results that were gathered, firstly these results are only for one set of benchmarks - different benchmarks by design could cause slightly different behaviours towards the provers leading to different results. Additionally, the efficiency of the C++ prover for other logics is not tested here. This was largely due to time constraints after making modifications to the prover later on in the semester. That being said, the other provers were run on some large problems for testing and appeared to behave in a comparable time.

5.4 Correctness

Due to time constraints we did not test the prover for other modal cube logics against different provers of that type. That being said, in order to help build confidence for correctness we built a small test set of formulas which are valid, satisfiable and unsatisfiable in each of these logics in a multimodal context. The results confirmed that the provers worked in those small number of test cases.

5.5 Summary

The above section detailed the experiments that were run, gave and analysed the results, and provided justification for correctness in the other cases. Our implementation. It was found that our prover dominates the Haskell implementation in both modal logics K and T. Additionally, our C++ version is pretty much on par with K_SP, the only theorem prover to beat the Haskell implementation in these benchmarks. Our C++ version had a faster average solve time however so we concluded that as the Haskell implementation is better than every other state-of-the-art theorem prover for K and T except in this benchmark, and our version dominates the Haskell implementation and is slightly better than K_SP here, then our version of CEGARBox is the most efficient theorem prover for modal logics K and T (with more experimentation ongoing to solidify this claim). We gave several justifications for correctness and in the next chapter we conclude and provide direction for future work.

Chapter 6

Conclusion

This work presented an extension of the efficient theorem prover CEGARBox to all fifteen normal modal logics expressed in the modal cube in a multimodal context. We also refactored the entire algorithm so that and underlying SAT solvers can easily be swapped out in the future, and prover for additional modal logics added. Additionally we formalised the necessary changes that are needed in order to implement global assumptions. The results show that the changes which were made improved the efficiency of CEGARBox overall - and is on par or slightly better than the only prover to beat CEGARBox in one benchmark. As our implementation also dominated the Haskell implementation we concluded that our theorem prover is the most efficient for modal logics K and T (with more experiments ongoing currently). We also justified correctness for the other modal logics with a small number of test cases. With such a large leap in both efficiency and generalness that this report has outlined, we are another step closer to being able to create an efficient and expressive intelligent agent that can reason automatically. There are many avenues for future work still with this state-of-the-art modal SAT solver.

6.1 Future Work

The most obvious area for future work would be to compare the efficiency of the prover for the other 13 logics not tested against other modal SAT solvers - especially in a multimodal context as well (even if running them on modified K benchmarks as they do not exist for all fifteen). It was decided for this report however to not test this and instead dedicate time in order to optimise the program behind the scenes in the code. Running the program on large input files however shows very promising early results with regards to the efficiency for these extra logics and this is something that will be tested over summer.
Another obvious area for future work would be to implement the global assumptions as were described in this paper. This wasn't done in the program as it was not a large priority. That being said, with the formalisation done it should be quite straightforward to implement with the way the code is setup.
The final and most exciting area for future work to me however, is the extension of this work to tense logic. Tense logic is an extension of modal logic with the

additional modal operators ⧫ and its dual ■. $\blacklozenge\varphi$ is used to say there exists a past world which makes φ true, and $\blacksquare\varphi$ as all past worlds make φ true. With the regular $\Box$ and $\Diamond$ symbols referring to future states. There is a distinct similarity between tense logic and the rules already described for symmetry - for example if $(a \rightarrow \blacksquare b)$ is true in a modal context then in the predecessor modal context $(\neg b \rightarrow \Box\neg a)$ must be true. I will be continuing this work over summer with the main intention of fully formalising the methodology behind CEGARBox towards tense logic.

Bibliography

Catach, L., 1991. Tableaux: A general theorem prover for modal logics. *Journal of Automated Reasoning*, 7, 4 (Dec 1991), 489–510. doi:10.1007/BF01880326. https://doi.org/10.1007/BF01880326. (cited on page 7)

Cerro, L.; Fauthoux, D.; Gasquet, O.; Herzig, A.; Longin, D.; and Massacci, F., 2001. Lotrec: The generic tableau prover for modal and description logics. vol. 2083, 453–458. doi:10.1007/3-540-45744-5_38. (cited on page 7)

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

Garson, J., 2021b. Modal Logic. In *The Stanford Encyclopedia of Philosophy* (Ed. E. N. Zalta). Metaphysics Research Lab, Stanford University, Summer 2021 edn. (cited on pages ix, 6, and 7)

Gasquet, O.; Herzig, A.; Longin, D.; and Sahade, M., 2005. Lotrec: Logical tableaux research engineering companion. In *Automated Reasoning with Analytic Tableaux and Related Methods*, 318–322. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 7)

Girlando, M. and Strassburger, L., 2020. Moin: A nested sequent theorem prover for intuitionistic modal logics (system description). In *Automated Reasoning*, 398–407. Springer International Publishing, Cham. (cited on page 8)

Goré, R., 2019. Introduction to modal and temporal logic. (cited on pages 3, 4, and 21)

Goré, R. and Kikkert, C., 2021. Cegar-tableaux: Improved modal satisfiability via modal clause-learning and sat. In *Automated Reasoning with Analytic Tableaux and Related Methods*, 74–91. Springer International Publishing, Cham. (cited on pages v, 1, 3, 10, 11, 12, 16, 20, 23, 25, and 28)

Goré, R.; Olesen, K.; and Thomson, J., 2014. Implementing tableau calculi using bdds: Bddtab system description. doi:10.1007/978-3-319-08587-6_25. (cited on page 22)

Massacci, F. and Donini, F. M., 2000. Design and results of tancs-2000 non-classical (modal) systems comparison. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, 52–56. Springer. (cited on page 25)

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 pages 22 and 25)

Sorensson, N. and Een, N., 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005, 53 (2005), 1–2. (cited on page 22)

Tsarkov, D. and Horrocks, I., 2006. Fact++ description logic reasoner: System description. In *Automated Reasoning*, 292–297. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 22)

Wu, M. and Goré, R., 2019. Verified decision procedures for modal logics. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. (cited on page 1)