A Model Checking Tool for Modal Labelled Deductive Systems

Lidia Medhin
(www.doc.ic.ac.uk/~lm104/MCT)

Supervised by Dr. Alessandra Russo and Dr. Krysia Broda

June 20, 2008
Abstract

Labelled deductive systems (LDS) were formally introduced as a unifying framework for different logics. In the past LDS has been used when defining proof systems where the labelling algebra carries the semantic properties of the logic syntactically and this can be handled in parallel with the formulae. A hybrid system was proposed for modal logic where by modal formulae are combined with first order relational theory. Starting from Kripke semantics for modal logic we can use LDS to enable us to explicitly refer to worlds and relations. Each formula is combined with a label as $w_i; \phi$ where the label indicates the possible world the formula belongs to. The aim of this project is to use this hybrid system to generate models for propositional modal logic. More specifically, the aim is to write and implement an algorithm that takes a set of labelled modal formulas (not necessarily in the same possible world) together with relations and frame properties expressed in first order logic and generates a model if one exists.
Acknowledgements

I would like to thank my supervisors, Dr. Krysia Broda and Dr. Alessandra Russo, for their continuous support and encouragement during my project. I would also like to thank

- Prof. Ian Hodkinson for introducing me to modal logic and providing me with the lecture material in advance.

- Azalea Raad and Charence Wong for introducing me to javaCUP.
Contents

1 Introduction 4
  1.1 Motivation ................................................. 4
  1.2 Contribution ............................................... 5
  1.3 Report Structure ............................................ 6

2 Background 7
  2.1 Modal Logic .................................................. 7
    2.1.1 History of Modal Logic ................................. 7
    2.1.2 Relational Structures ................................. 7
    2.1.3 Basic Modal Formula ................................. 8
    2.1.4 Models and Frames ................................. 9
    2.1.5 Validity and Satisfiability ......................... 10
    2.1.6 Modal Labelled Deductive Systems .................. 11
    2.1.7 LDS for Modal logic .................................. 11
  2.2 SAT and k-SAT .............................................. 13
    2.2.1 DPLL algorithm ...................................... 13
    2.2.2 KSAT: sat-based decision procedure for normal modal logics 15

3 Algorithms for model verification and model generation 18
  3.1 Stage1: Model Checking algorithm ........................... 18
  3.2 Stage2: Model generation algorithm ....................... 24
  3.3 Stage3: Model generation algorithm with frame conditions .... 34

4 Design, Implementation and Testing 43
  4.1 Data Structures design ...................................... 43
    4.1.1 Formulas as Binary Trees ................................ 43
    4.1.2 Relations as a matrix ................................ 44
    4.1.3 Assignments as Hash map ............................ 45
  4.2 Implementation ............................................. 46
4.2.1 Formulas and parsing ..................................... 46
4.2.2 Algorithm implementation ............................... 46
4.3 Testing ....................................................... 48
  4.3.1 Stage 1: Testing the parser ............................ 48
  4.3.2 Stage 2: Testing the data structures ................. 49
  4.3.3 Stage 3: Testing the model checker ................. 49
  4.3.4 Stage 4: Testing the model generator without relations and frame conditions ............... 50
  4.3.5 Stage 5: Testing the model generator with relations but no frame conditions ............... 51
  4.3.6 Stage 6: Testing the model generator with relations and frame conditions ............... 53

5 Future work, Evaluation and Conclusion ............... 55
  5.1 Future work ............................................. 55
  5.2 Evaluation ............................................. 59
  5.3 Conclusion ............................................. 60
Chapter 1

Introduction

1.1 Motivation

Consider the following two sets of modal formulae.

\[ F_1 = \{ \Box(a \land b), \Diamond(b \lor \neg c) \} \]
\[ F_2 = \{ \neg b \lor c, \Box \Diamond a \rightarrow (b \land \Box c) \} \].

Assume that these two sets formulae are in different possible worlds \( w_1 \) and \( w_2 \) as shown in Figure 1.1. We ask the question ‘Does this have a model?’ (i.e. is it satisfiable?). This raises the question of how we represent the given information before we can begin to check for satisfiability. Also, assume we also want to include the information that \( w_1 \) is related to \( w_2 \) so the diagram is now as given in figure 1.2. Again, we ask the question ‘Does this have a model?’. During my research none of the satisfiability checkers I have come across can be applied for problems of this type.

Labelled deductive systems (LDS) were formally introduced by Gabby [Gab96] as a unifying framework for different logics. In the past LDS has been used when defining proof systems where the labelling algebra carries the semantic properties of the logic syntactically and this can be handled in parallel with the formulae.
In [Rus96] Russo has proposed hybrid systems for modal logic where by modal formulae are combined with first order relational theory. Starting from Kripke semantics for modal logic we can use LDS to enable us to explicitly refer to worlds and relations. Each formula is combined with a label as $w_j; \phi$ where the label indicates the possible world the formula belongs to.

The aim of this project is to use this hybrid system to generate models for propositional modal logic. More specifically, the aim is to write and implement an algorithm that takes a set of labelled modal formulas (not necessarily in the same possible world) together with relations and frame properties expressed in first order logic and generates a model if one exists.

In fact, the answer to the questions above is ‘yes’ and using the algorithm developed we get the possible model below (the actual output from the system is included in the appendix).

This project was inspired by Lotrec which a generic tableau theorem prover for modal logic.

1.2 Contribution

- I have written and implemented an algorithm that takes a model (a set of labelled formulae, relations, and assignments) and verifies the correctness of the model
• I have written and implemented an algorithm that takes a set of labelled formulae, set of (possibly empty) relations expressed as binary predicates and standard frame properties such as transitivity, reflexivity, and seriality and generates a small model if one exists.

• I have investigated the possibilities of how the algorithm can be extended to accommodate non-standard frame properties expressed in first order logic.

This project was also done by a previous MEng student [Cao07] with the main focus being implementing a good GUI. And here I have focused on writing a stable model generation algorithm.

1.3 Report Structure

We start by looking at a brief history of modal logic and some of the developments made in the areas of propositional and propositional modal logic satisfiability checking. This can be found in chapter 2. Chapter 3 details the three stages in the development of the model generator starting with a model checker in stage 1, a model generator with no specific frame properties in stage 2 and finally a model generator for specific classes of frames in stage 3. The data structures used to represent the different components used in the algorithms in chapter 3 are explained in chapter 4 followed by implementation details and testing. This is followed by a discussion of what has been achieved and useful extensions for the future in chapter 5. The last chapter contains an evaluation of the system and some conclusions.
Chapter 2

Background

Satisfiability checking is an area of research that is of great interest. Having a good satisfiability checker could lead us to solve many practical problems by reducing them to the question of whether certain formulae are satisfiable in a given logic system. Many of the efficient Propositional Satisfiability(SAT) solvers that have been developed tend to have very limited application since they only operate on propositional formulas. Satisfiability checking for first and higher order logic has also seen some development. Some of the well known first order satisfiability checker include Alloy and Mace.

2.1 Modal Logic

2.1.1 History of Modal Logic

The study of logic with modal properties dates as far back as the times of Aristotle. Many scholars including William of Ockham and John Duns Scotus, have informally reasoned and analysed ideas about what is necessary and possible. However, symbolic modal logic was introduced by C.I.Lewis around 1918 in his publications survey of symbolic logic. Possible-world semantics for propositional modal calculus were developed by J.C.C. MacKensie and Alfred Tarski in the 1940s; and they were extended to modal predicate calculus by Saul Kripke in the 1959.

2.1.2 Relational Structures

A formal system is defined by a language together with a structure, relationship between the language and the structure and some rules to infer new sentences from the existing ones. The properties of the structure are defined by a set of axioms and
the set of inference rules help to reason about the structure.

**Definition:** A relational structure is:

1. A non-empty set \( W \) and a set \( R \) of relations defined on the set \( W \)
2. If \( A = \{ W, R \} \) is a graph with \( W \) a set of relational structures and \( R \) a set of edges, then \( A \) is a relational structure.
3. Only those structures that satisfy 1 and 2 are relational structures.

**Example 1:** LTSs (Label Transition Systems) are a good example of a relational structures widely used in computer science. In this case the set \( W \) is the set of states a machine can be in and the set \( R \subseteq (W \times W) \) is a binary relation. For example for \( \{a,b\} \subseteq W \), we say \((a,b)\), typically represented \( a \rightarrow b \), is a member of \( R \) if there exists a state transition from state \( a \) to state \( b \).

**Example 2:** A tree is also another typical example of relational structure. The set \( W \) in this case is the set of nodes of the tree which contains the unique node ‘r’ which is the root of the tree and the set \( R \) is a predecessor relation defined as follows:

1. For every node \( w \) (excluding the root node \( r \)) in \( W \), \( R_\times(r,w) \) holds (i.e. there is a path from the root node to each node)
2. For every node \( w \) in \( W \) (excluding \( r \)) there exists a unique \( v \) in \( W \) such that \( R(w,v) \) holds (i.e. except for the root node, each node has a unique parent)
3. \( R \) is acyclic. This means for all \( w \) in \( W \), \( R_\times(w,w) \) does not hold. (i.e. there does not exist a path of length \( \geq 0 \) from any node to itself). This also implies that \( R \) is irreflexive

\[ R_\times(a,b) \iff \exists (b_1,..,b_n) \ n \geq 0 \text{ such that } R(a,b_1) \land R(b_1,b_2) \land ... \land R(b_n,b) \]

### 2.1.3 Basic Modal Formula

Formally a propositional modal formula is defined as follows.

**Definition:**

- Any propositional atom is an (L-) formula
- \( \top \) is a formula
If \( A \) and \( B \) are formulas, then so are \( \neg A \), \( A \land B \) and \( \Box A \).

Nothing else is a formula.

Propositional modal logic is essentially propositional logic with an additional unary connective ‘\( \Box \)’ (pronounced ‘box’). Hence there no quantifiers like \( \forall \) and \( \exists \). The normal propositional abbreviations (e.g. ‘\( \bot \)’ abbreviated by ‘\( \neg \top \)’) apply. In addition \( \Diamond A \) abbreviates \( \neg \Box \neg A \).

The modal operator ‘\( \Box \)’ has many meanings but is often understood to mean ‘is necessary that’ with corresponding meaning for ‘\( \Diamond \)’ ‘is possible that’. So to say that a proposition is possible is equivalent to saying it is not necessarily false. Also, to say that a proposition is necessary is equivalent to saying that its negation is impossible. There are several variations of modal logic (depending on, for example, the meaning of ‘\( \Box \)’ and ‘\( \Diamond \)’) like Propositional Dynamic Logic but we will not consider these.

### 2.1.4 Models and Frames

Truths of statements like \( \Box A \rightarrow \Box A \) is not dependent on the meaning of ‘\( \Box \)’ and is true for any \( A \). These are called **natural truths**. However, truth of formulas like \( \Box A \) depend both on the truth value of \( A \) and the meaning we give to ‘\( \Box \)’. These are called **contingent truths**. Therefore the interpretations of the modality of ‘\( \Box \)’ and ‘\( \Diamond \)’ are dependent both on the truth value of the statement (formulas) they are applied to and the ‘quantification’ over entities called ‘possible worlds’. These are entities of a mathematical structure called a **frame**.

**Definition:** A frame \( F \) (also known as Kripke frame) for a modal language is a pair \((W, R)\) where

- \( W \) is a set of possible worlds
- \( R \) is a binary relation on \( W \)

i.e. a frame is simply a relational structure with a single binary relation \( R \).

**Definition:** A model ‘\( M \)’ (also known as Kripke model) is a pair \((F, V)\) where

- \( F \) is a frame
- \( V \) is a function which maps a propositional letter ‘\( p \)’ of the frame to a subset ‘\( V(p) \)’ of \( W \) where ‘\( p \)’ is true
2.1.5 Validity and Satisfiability

Having defined a model and a frame we now define what it means for a modal formula to be satisfiable and what it means to say it is valid in Kripke semantics. Bear in mind that definition of validity and satisfiability in other semantics could be different.

**Definition:** [Hod07] Let $\varphi$ be a formula in a model $M = (W, R, V)$ then the notion of validity at world ‘$w$’ is defined inductively as follows:

- $M, w \models p$ iff $w$ is an element of $V(p)$
- $M, w \models \top$ always true
- $M, w \models \neg \varphi$ iff not $M, w \models \varphi$
- $M, w \models \varphi \land \omega$ iff $M, w \models \varphi$ and $M, w \models \omega$
- $M, w \models \Box \varphi$ iff $M, u \models \varphi$ $\forall u$ element of $W$ with $R(w,u)$

**Definition:** [Hod07] A formula $\varphi$ is

- *valid in a frame* if it is valid in any model based on that frame
- *valid in a model* if it true at all worlds of that model
- And *valid in a class $C$ of frames* if it is valid in every frame $F$ in $C$

**Definition:** A formula $\varphi$ is *satisfiable* if there exists a model $M$ and world ‘$w$’ of $M$ such that $M, w \models \varphi$

**Example:** [Hod07] For any formulas $\omega$ and $\varphi$, $\Box(\omega \rightarrow \varphi) \rightarrow (\Box \omega \rightarrow \Box \varphi)$ is a valid formula

**Proof:** Assume $M, w \models \Box(\omega \rightarrow \varphi)$ for an arbitrary model $M$ and world ‘$w$’ of $M$. We need to show that $M, w \models (\Box \omega \rightarrow \Box \varphi)$. Assume now that $M, w \models \Box \omega$ and let ‘$w_1$’ be any world such that $R(w,w_1)$ then we need to show that $M, w_1 \models \varphi$. From our first assumption it is true that $M, w_1 \models \omega$ and $M, w_1 \models (\omega \rightarrow \varphi)$. Therefore $M, w_1 \models \varphi$ which concludes the proof.

**Example:** For any formula $\varphi$, $\neg (\Box \varphi \rightarrow \Box \Box \varphi)$ is not satisfiable in a transitive frame.
Proof: \( \neg (\Box \varphi \rightarrow \Box \Box \varphi) \equiv \Box \varphi \land \Diamond \Diamond \neg \varphi \). So \( \neg (\Box \varphi \rightarrow \Box \Box \varphi) \) is satisfiable iff \( \Box \varphi \land \Diamond \Diamond \neg \varphi \) is satisfiable. Now, assume it is satisfiable in a transitive frame. So there exists a model \( M \) (based on the transitive frame) and a world ‘w’ of \( M \) such that \( M,w \models \Box \varphi \land \Diamond \Diamond \neg \varphi \). This implies \( M,w \models \Box \varphi \) and \( M,w \models \Diamond \Diamond \neg \varphi \). So \( \exists w',w'' (R(w,w') \land R(w',w'') \land M,w'' \models \neg \varphi) \). Since the frame is transitive we have \( R(w,w'') \) and since \( M,w \models \Box \varphi \) we have \( M,w'' \models \varphi \). Contradiction! Hence \( \neg (\Box \varphi \rightarrow \Box \Box \varphi) \) is not satisfiable in a transitive frame.

Example (reflexivity): [Hod07] For an atom \( p \), \( \Box p \rightarrow p \) is valid in a frame \( F = (W,R) \) if \( R \) is reflexive (i.e. \( R(t,t) \) holds for all \( t \in W \)).

Proof: \( \implies \) Suppose \( R \) is reflexive. We show that \( \Box p \rightarrow p \) is valid in \( F \). So let \( M \) be any model with frame \( F \), and let \( t \) be any world of \( F \). If \( M,t \models \Box p \), then \( M,u \models p \) for all worlds \( u \in W \) with \( R(t,u) \). But by reflexivity, \( R(t,t) \). So \( M,t \models p \). Hence \( M,t \models \Box p \rightarrow p \). As \( M,t \) were arbitrary, \( \Box p \rightarrow p \) is valid in \( F \).

\( \impliedby \), assume that \( \Box p \rightarrow p \) is valid in \( F \). Let \( t \in W \) be arbitrary. We need to show that \( R(t,t) \). Define a model \( M = (F,V) \) on the frame \( F \), by \( V(p) = \{ u \in W : R(t,u) \} \). (So \( p \) is made true at just the worlds that are possible at \( t \)) Then obviously, \( M,t \models \Box p \). But \( \Box p \rightarrow p \) is valid in \( F \). So \( M,t \models p \). That is, \( t \in V(p) \). So by definition of \( V(p) \) we have \( R(t,t) \). And this concludes the proof.

2.1.6 Modal Labelled Deductive Systems

When we think about comparing two types of Logics, the considerations between them are metalevel considerations on the proof theory and semantics. Most logics are based on modus ponens and the quantifier rules are formally the same. Labelled deductive systems (LDSs) were formally proposed by Dov M.Gabbay[Gab96] as a framework for formalising a variety of seemingly different logics. LDS makes the transition from one logic to another natural and along predefined acceptable modes of variation.

2.1.7 LDS for Modal logic

As mentioned above LDS enables us to find a uniform formalization for different logic. Formulas of Modal logic are paired with labels which in essence encode metalevel information about the formula which is not encoded inside the formula. Starting from the Kripke structure for modal logic, we can use LDSs to enable us to explicitly refer to world and their relationships which we are not able to do with most deductive systems. Before we define what a labelling language is, lets first
see what we mean when we say ‘well formed formula’.

**Definition:** Well formed formula (wff)

- any propositional letter is a well formed formula
- if \( \varphi \) is a well formed formula and \# any unary classical or modal operator then \#\( \varphi \) is a well formed formula
- if \( \varphi \) and \( \omega \) are well formed formulas and \# a binary classical operator then \( \varphi \#\omega \) is a well formed formula.

**Definition:** A labelling language \( A \) is a higher order language which consists of the following symbols

- Function symbols \{ \( f1, f2, \ldots \) \}
- Relation symbols \{ \( R1, R2, \ldots \) \}
- The classical connectives, equality(=), and other possible higher order functions on predicates
- Element variable \( x_1, x_2, \) and individual constants \( c_1, c_2, \ldots \)
- Relation and Function variables of all arities
- Higher order quantifier symbols ‘\( \forall \)’ and ‘\( \exists \)’ for individual relation and function variables of all arities

As proposed in [Rus96] for modal logic we can think of a simplified representation of labelling language \( L_l \) as a first order language composed of

- a countable set of constant symbols \( w_1, w_2, \ldots \)
- a countable set of variables \( x, y, z, \ldots \)
- a binary relation symbol \( R \)
- logical connectives \( \neg, \land, \lor, \rightarrow, \equiv \)
- quantifier symbols ‘\( \exists \)’ and ‘\( \forall \)’

**Definition:** [Rus96]: A propositional modal language \( L_m \) is composed of

- propositional letters \( p, q, r, \ldots \)
- Logical connectives $\neg, \land, \lor, \rightarrow$
- modal operators $\Diamond, \Box$

**Definition:** [Rus96] Modal labelled deductive language (MLDL) is the ordered pair $(L_l, L_m)$ where $L_l$ is a labelling language and $L_m$ is a propositional modal language.

### 2.2 SAT and k-SAT

SAT, also known as the Boolean Satisfiability Problem, is one of the basic problems of complexity. Given a Boolean formula $\phi$, the question of satisfiability is the question of whether we can assign values to the variables of the formula so as to make the whole formula evaluate to true or determine that no such assignment exists (i.e. the formula is unsatisfiable). When we restrict the length of clauses to ‘k’ (i.e. k literals per clause) we get the so called k-SAT problem.

NP (Non-deterministic Polynomial time) problems are decision problems that can be solved in polynomial time by non-deterministic Turing machine. A decision problem is NP-Complete if it is NP and every other NP problem is reducible to it. Cook’s Theorem states that any problem in class NP can be reduced to an instance of SAT in polynomial time (i.e. the Boolean Satisfiability problem is NP-Complete).

Boolean Satisfiability checking is an area that has seen a lot of interest from theoretical computer science. It has also got great potential in helping to solve a lot of practical problems. About 40 years ago a basic resolution based search algorithm was proposed by Davis, Putnam, Logemann and Loveland, commonly known as DPLL algorithm. Since then the area has seen much development with many state-of-the-art SAT solvers being developed which are able to handle many complex SAT problems with hundreds and even thousands of variables.

#### 2.2.1 DPLL algorithm

The DPLL is a decision algorithm first proposed in 1960 and later improved to deal with its ‘memory explosion’ problem. The formula, whose satisfiability is being checked, is presented in CNF \(^1\) (conjunctive normal form) for the efficiency of the search algorithm. Since for any problem there exists an equivalent problem

---

\(^1\)A formula is said to be conjunctive normal form if it is a conjunction of clauses, where a clause is a disjunction of literals.
in conjunctive normal form which has same satisfiability as the original one this do
not limit the algorithm in any way. Also, implemented versions that take problems
that are not in CNF do exist.
For a formula in CNF to be satisfiable we require that each clause be satisfiable. So
if any of the clauses has all its literals assigned \textit{false} then (since false \& any\textit{thing} = false) this assignment and any assignment that contains this assignment clearly
does not satisfy the formula. Given below is the basic DPLL algorithm taken from
[LZ02]. A \textit{conflicting clause} is one that has all its literals assigned \textit{false} causing
the whole formula to evaluate to \textit{false}.

DPLL (formula , assignment ) {
1. necessary = deduction (formula, assignment);
2. new.asgnmnt = union( necessary, assignment);
3. if ( is.satisfied(formula,new.asgnmnt))
4. return SATISFIABLE;
5. else if (is.conflicting(formula, new.asgnmnt))
6. return CONFLICT;
7. var = choose_free_variable(formula, new.asgnmnt);
8. asgn1 = union(new.asgnmnt, assign(var, 1));
9. if (DPLL (formula, asgn1) ==SATISFIABLE)
10. return SATISFIABLE;
else {
11. asgn2 = union (new.asgnmnt, assign(var, 0));
12. return DPLL(formula, asgn2);
}
}

This simple version of DPLL is a recursive algorithm that takes in a formula and
a set of variable assignments (initially empty). The function deduction(x,y) returns
with a set of the possible variable assignments that can be deduced from the current
variable assignments. If the formula is satisfied (evaluates to true) for the current
set assignments then the algorithm terminates returning \textit{SATISFIABLE} and if it is
unsatisfiable (evaluates to false) it terminates returning \textit{CONFLICT}. Otherwise it
picks an unassigned variable from the formula and branches for the two possible
Boolean values of that variable. This is a very basic form of the DPLL and many
specialized implementations for SAT solver do exist. Given below are examples of
the application of the algorithm.

\textbf{Example 1:} Consider the formula $\phi = (A \rightarrow B) \land (B \rightarrow C) \land \neg(A \rightarrow (\neg C \land B))$
Solution:
\( \varphi \) is equivalent to the CNF formula \((\neg A \lor B) \land (\neg B \lor C) \land A \land (\neg C \lor B))\)

DPLL is called with the CNF formula above and the initially empty set assignments:

1. necessary = \{ A = 1 \}; /* It is necessary that \( A \) is true*/
2. new_assgnmnt = \{ A = 1 \};
7. var = B; /*next free variable in the formula*/
8. asgn1= \{A=1, B=1\}; /*assign \( B \) true*/
1. necessary = \{ C = 1 \}; /*\( \neg B \lor C \) has to be true so it is necessary that \( C \) is true*/
2. new_assgnmnt = \{A=1, B=1, C=1\};
10. SATISFIABLE

Example 2: Consider \( \varphi = (A \lor B) \land \neg A \land \neg B \)

Solution:
\( \varphi \) is already in CNF so DPLL is called with \( \varphi \) and an empty set assignments

1. necessary = \O; 
2. new_assgnmnt = \O;
7. var = A; /*first free variable in the formula*/
8. asgn1= \{A=1\}; /*assign true to \( A \)*/
1. necessary = \O;
2. new_assgnmnt = \{A=1\};
6. CONFLICT;
11. asgn2= \{A=0\}; /*backtrack and assign false to \( A \)*/
1. necessary = \{B=1\}; /*\( A \lor B \) has to hold so it is necessary that \( B \) is true if \( A \) is false*/
2. new_assgnmnt = \{A=0, B=1\};
6. CONFLICT;

Both ‘assign \( A \) true’ and ‘assign \( A \) false’ cases fail hence \( \varphi \) is not satisfiable.

2.2.2 KSAT: sat-based decision procedure for normal modal logics

KSAT [SV98] is a decision procedure for modal logic K built on top of a decision procedure for propositional satisfiability (SAT). It takes a modal propositional wff \( \varphi \) as input and returns a truth value asserting whether \( \varphi \) is K-satisfiable or not. It does this by calling a recursive function \( KSAT_W \) (where \( W \) stands for a ‘wff’) which is a non-CNF variant of DPLL. Initially \( KSAT_W \) is called with the formula \( \varphi \) and an empty set (i.e. \( KSAT_W \) takes the modal formula as atomic formulas and checks for propositional satisfiability). If an initial assignment \( \mu \) is found, instead of just returning true it invokes another function \( KSAT_A(\mu) \). \( KSAT_A \) then checks the K-satisfiability of the truth assignments. The basic KSAT algorithm for K(1) (i.e. K with 1 modality) taken from [SV98] is given below.

```
function KSAT (\( \varphi \))
```
1. return KSATw(φ, Ø);

function KSATw(φ, μ)
2. if φ=T /*base*/
3. then return KSAT(μ);
4. if φ=F /*backtrack*/
5. then return False;
6. if {a unit clause (l) occurs in φ} /*unit*/
7. then return KSATw(assign(l, φ), μ U {l});
8. l :=choose literal (φ); /*split*/
9. return KSATw(assign(l, φ), μ U {l}) or
10. KSATw(assign(¬l, φ), μ U {¬l});

function KSATw(ϕ₁,...,ϕₙ, μ₁,...,μₙ, A₁,...,Aₙ)
11. for each conjunct '¬□βₖ' do
12. ϕ' := ∨(αᵢ ∧ ¬βⱼ); /* ¬□/□ elimination*/
13. if not KSAT(ϕ')
14. then return False;
15. return True;

Fig. 1. The basic version of KSAT algorithm

If we start with an implicit world w₁, the function KSATₗ plays the same role as
the DPLL function mentioned above and generates a list of assignments μ₁,...,μₙ
where μᵢ = {□α₁,...,□αₙ, ¬□β₁,...,¬□βₘ, A₁,...,¬Aₙ}. The main difference is when KSATₗ finds an assignment μₖ, instead of just returning true it checks if this is indeed a satisfying assignment by invoking KSATₗ as mentioned
above.

Example: Let φ = (¬□A ∧ C) ∨ (□A ∧ ¬B ∧ ¬□(B ∨ C))
Then KSATₗ gives μ₁ = {¬□A, C} and μ₂ = {□A, ¬□B, ¬□(B ∨ C)}
For any conjunct of the form ¬□βₖ occurring in the truth assignment μ, it checks
whether the formula ϕ' = ∨(αᵢ ∧ ¬βⱼ) holds. If this returns true, the formula is
satisfiable hence KSAT returns true. However, if it returns false then the next
assignment μₖ₊₁ is checked and this continues until we have found a suitable
assignment or exhausted the list. In the latter case the formula is unsatisfiable and
KSAT returns false. This is demonstrated by the example below.

Example: Let φ = (□(A ∨ B ∨ ¬C) ∨ (□A ∨ □¬D)) ∨ ¬□(B ∨ D ∨ ¬C) ∨ (□¬A ∨ D))

KSATₗ
6,7. μ = {¬□(B ∨ D ∨ ¬C)} /*unit clause*/
φ = (□(A ∨ B ∨ ¬C) ∨ (□A ∨ □¬D)) ∨ □¬(A ∨ D)
6,7. μ = {¬□(B ∨ D ∨ ¬C), □¬A ∨ D)} /*unit clause*/
φ = (□(A ∨ B ∨ ¬C) ∨ (□A ∨ □¬D))
8,9. μ = {¬□(B ∨ D ∨ ¬C), □¬A ∨ D), □(A ∨ B ∨ ¬C)} /*assign true to □(A ∨ B ∨ ¬C)*/
φ = T

KSATₗ
11,12. ϕ' = (¬A ∨ D) ∧ (A ∨ B ∨ ¬C) ∧ ¬(B ∨ D ∨ ¬C)
Hence the initial formula $\varphi$ is satisfiable.
Chapter 3

Algorithms for model verification and model generation

In this chapter I describe how I developed an algorithm for model generation in three stages. The first stage was model verification i.e. given a model asserting whether it is a correct model or not. This is shown in section 3.1 below. This was then followed by a model generation algorithm described in section 3.2. This takes a set of formulas and relations and attempts to build a model for them. If terminates returning true if successful and false otherwise. The third stage to update the algorithm in stage two so that it can build models in specific frames such as reflexive, transitive and serial frames. So at this stage in addition to a list of formulas and a list of relations the algorithm takes a list standard frame conditions to be satisfied by the model returned. Again it terminates returning true if such a model exists and false otherwise.

Note: In each case we will be considering labelled formulae whose connectors are ‘NOT’, ‘AND’ and ‘OR’ and the modal operators ‘box’ and ‘diamond’. Formulas which are not of this form have an equivalent formula of the required form so this does not limit the algorithms.

3.1 Stage1: Model Checking algorithm

This section focuses on an algorithm for verifying the correctness of a given model. This algorithm will be referred to as ‘checkSAT’ from now on. This is the first step I have taken in the development of a model generator. Given a set of labelled formulae, set of relations and assignments checkSAT, which is a recursive algorithm, returns a truth value asserting whether the given model satisfies the set of formulas. So the three input lists are:
• **formulas**: list of labelled formulas \( \{w_i: \phi_1, w_j: \phi_2, \ldots\} \)

• **relations**: list of relation \( \{R(w_i, w_j), R(w_k, w_l), \ldots\} \)

• **assignments**: list of assignments \( \{w_i: \psi_j = T, w_i: \psi_k = T, w_j: \psi_k = F, \ldots\} \), where \( \psi_i, \psi_j, \ldots \) are propositional atoms. \( (w_i: \psi_j = T) \in \text{assignments} \) means the atom \( \psi_j \) is true in the world \( w_i \), etc. The list assignments is assumed to be ‘complete’ and consistent. What I mean by this is that given any atom \( \psi \) and world \( w_i \), exactly one of \( (w_i: \psi = T) \) and \( (w_i: \psi = F) \) is in \( \text{assignments} \). A step by step description the basic idea behind the algorithm is given below followed by an example. This is then followed by the algorithm.

Note: For convenience formulas of the form \( w_i: \square(\varphi) \) where \( \varphi \) is a formula are refereed to as ‘box formula’ and all other formulas are refereed to as ‘non-box formulas’.

**Idea of algorithm:**

The algorithm goes through the elements of **formulas** until it has either exhausted the list or has found a formula that is not satisfied. It starts with the first formula in the list and geos to one of the following cases:

If the formula is a box formula then it expands it. For example if the formula is \( w_1: \square(\phi_1 \land \phi_2) \) and \( R(1, 2) \) and \( R(1, 3) \) are in the list **relations** then the formula is expanded by adding \( w_2: (\phi_1 \land \phi_2) \) and \( w_3: (\phi_1 \land \phi_2) \) to the list.

If the formula is of the form \( w_i: (\phi_1 \land \phi_j) \) then the formulas \( w_i: \phi_1 \) and \( w_i: \phi_2 \) are added to **formulas** as these need to be satisfied for \( w_i: (\phi_1 \land \phi_j) \) to be satisfied.

If the formula is of the form \( w_i: (\phi_1 \lor \phi_j) \) then we branch in to two cases where one case checks if the left part of the disjunction is satisfied and the other case checks if the right part of the disjunction is satisfied. (for example if the list **formulas** = \( \{w_1: A \lor B, w_2: \square A, w_2: \Diamond C\} \) then it checks if \( \{w_1: A, w_2: \square A, w_2: \Diamond C\} \) is satisfiable or \( \{w_1: B, w_2: \square A, w_2: \Diamond C\} \) is satisfied).

If the formula is a propositional atom then it checks the list **assignments** to see if this labelled atom has been assigned the value \text{true}. If it has then algorithm moves to the next formula in the list. Otherwise the algorithm terminates returning FALSE.

If the formula is a negated atom then it checks the list **assignments** to see if this labelled atom has been assigned the value \text{false}. If it has then the algorithm moves to
the next formula in the list. Otherwise the algorithm terminates returning FALSE.

If it is a diamond formula then it checks to see if it is satisfied by considering all
the worlds that the world containing the diamond formula is related to. If it is then
it moves to the next formula in the list. Otherwise it terminates returning FALSE.

(For example if the list \texttt{formulas} = \{\texttt{w}_1 : \Diamond A, \texttt{w}_1 : A \land B, \texttt{w}_2 : \neg B\} and the list \texttt{relations} = \{R(1, 1), R(1, 2)\} then it checks if either \{\texttt{w}_1 : A, \texttt{w}_1 : A \land B, \texttt{w}_2 : \neg B\}
is satisfied or \{\texttt{w}_2 : A, \texttt{w}_1 : A \land B, \texttt{w}_2 : \neg B\} is satisfied.)

If all formulas in the list \texttt{formulas} are satisfied then the algorithm terminates re-
turning TRUE.

Example 3.1 :
Let \texttt{formulas} = \{ \texttt{w}_1 : \Box (a), \texttt{w}_1 : \Diamond (
eg c), \texttt{w}_1 : \neg b \}
\texttt{relations} = \{ R(\texttt{w}_1, \texttt{w}_1), R(\texttt{w}_1, \texttt{w}_2) \}
\texttt{assignments} = \{ \texttt{w}_1 : a = T, \texttt{w}_1 : b = F, \texttt{w}_1 : c = T, \texttt{w}_2 : c = F, \texttt{w}_2 : a = T, \texttt{w}_2 : b = T \}

Then it first expands the box formula \texttt{w}_1 : \Box (a) by adding \texttt{w}_1 : (a) and \texttt{w}_2 : (a) to the
list of formulas. So the list is now \texttt{formulas} = \{ \texttt{w}_1 : \Box (a), \texttt{w}_1 : \Diamond (\neg c), \texttt{w}_1 : \neg b, \texttt{w}_1 : (a), \texttt{w}_2 : (a) \}. Then checkSAT picks the next formula \texttt{w}_1 : \Diamond (\neg c). For this to be satisfies
there needs to be a world related to \texttt{w}_1 and (\neg c) needs to hold in that world. It first
tries \texttt{w}_1 but atom \texttt{c} has been assigned the value \texttt{true} in \texttt{w}_1 so this fails . Then it
tries \texttt{w}_2 which is the next world \texttt{w}_1 is related to. This succeeds as atom \texttt{c} has been
assigned the value \texttt{false} in \texttt{w}_2. It then moves to the next formula in the list which
is \texttt{w}_1 : \neg b which also holds since atom \texttt{b} has been assigned the value false in \texttt{w}_1.
Similarly it checks that \texttt{w}_1 : a and \texttt{w}_2 : a also hold. Since it has exhausted the list
it terminates returning TRUE. Given below is the model diagram.

![Model Diagram]
The algorithm: checkSAT

1. function checkSAT(formulas, relations, assignments)
2. if formulas = Ø return TRUE / * base case */
3. else let $w_i:\phi_j$ be the first non-box formula in formulas
4. case0: $\phi_i$ is an atom
5. if $(w_i:\phi_j = T) \in assignments$
6. then formulas' = formulas / {$(w_i:\phi_j)$}
7. return checkSAT(formulas', relations, assignments)
8. return FALSE
9. case1: $\phi_j = \varphi \land \varphi'$ where $\varphi$ and $\varphi'$ are formulas
10. return checkSAT(formulas', relations, assignments)
11. where formulas' = formulas $\cup \{w_i:\varphi, w_i:\varphi'\}/\{w_i:\phi_j\}$
12. case2: $\phi_j = \varphi \lor \varphi'$ where $\varphi$ and $\varphi'$ are formulas
13. formulas' = formulas $\cup \{w_i:\varphi\}/\{w_i:\phi_j\}$
14. formulas'' = formulas $\cup \{w_i:\varphi'\}/\{w_i:\phi_j\}$
15. return checkSAT(formulas'', relations, assignments) ||
16. neg-sat = checkSAT(formulas', relations, assignments)
17. if neg-sat then return FALSE
18. return TRUE
19. case3: $\phi_j = \neg \varphi$ where $\varphi$ is a formula
20. formulas' = formulas $\cup \{w_i:\varphi\}/\{w_i:\phi_j\}$
21. neg-sat = checkSAT(formulas', relations, assignments)
22. if neg-sat then return FALSE
23. return TRUE
24. case4: $\phi_j = \Box \varphi$
25. formulas' = formulas $\setminus \{w_i:\phi_j\}$
26. for every $R(w_i,w_j) \in relations$
27. formulas' = formulas' $\cup \{w_j:\varphi\}$
28. end
29. return checkSAT(formulas', relations, assignments)
30. case5: $\phi_j = \Diamond \varphi$
31. try for every $R(w_i,w_j) \in relations$
32. formulas' = formulas $\cup \{w_j:\varphi\}/\{w_i:\phi_j\}$
33. sat = checkSAT(formulas', relations, assignments)
34. if sat then return TRUE;
35. return FALSE;
36. end

Worked Examples

Example 1:

Let

formulas = \{ $w_1:(a \land b), w_1:-(b \lor c)$ \}
relations = \{ $R(w_1,w_2), R(w_1,w_1)$ \}
assignments = \{ $w_1.a = T, w_1.b = T, w_2.a = F, w_2.b = T$ \}

3. $w_1:(a \land b)$ //first formula in the list formulas*/
9. $w_1:(a \land b) \equiv w_1:(a) \land w_1:(b)$  //case1*/
Example 1 explanation:

checkSAT starts by picking the first labelled formula in the list, namely \(w_1:(a \land b)\). It then removes this formula from the list \(formulas\) and adds \(w_1:(a)\) and \(w_1:(b)\) to the list. It then picks \(w_1:(a)\) and since this is atomic it checks if this is in assignments which is true in this case. So it removes it from the list. In then picks the next formula in the list which is \(w_1:(b)\). Similarly it sees that this is also in the list assignments and removes it from the list \(formulas\).

It then picks the next formula in the list: \(w_1:-(b \lor c)\) which is the only formula remaining in \(formulas\). This is a negated formula so it goes to case 3. It forms a new list by removing this negative formula and adding its complement \(w_1:(b \lor c)\). checkSAT is then called with this new list. If this returns true then checkSAT asserts that the model does not satisfy the original set of formulas. Since \(w_1:(b)\) is in the list assignments the new list is satisfiable by the model. Hence FALSE is returned since the original list is not satisfiable.

Example 2:

let
formulas = \{ \text{w}_1:□(a), \text{w}_1:◊(c) \}
relations = \{R(\text{w}_1, \text{w}_1), R(\text{w}_1, \text{w}_2)\}
assignments = \{\text{w}_1.a=T, \text{w}_1.b=T, \text{w}_1.c=F, \text{w}_2.a=T, \text{w}_2.b=T, \text{w}_2.c=T\}

3. \text{w}_1:□(a) \quad \text{/* first formula in the list formulas */}
21, 22. formulas' = \{\text{w}_1.a, \text{w}_1.b, \text{w}_1:◊(c)\} \quad \text{/* case 4: since R(\text{w}_1, \text{w}_1), R(\text{w}_1, \text{w}_2) */}
25. checkSAT(formulas', relations, assignments)
3. \text{w}_1:(a) \quad \text{/* next formula in the list formulas */}
5. \text{w}_1:(a) \in \text{assignments}
6. formulas' = \{\text{w}_1:b, \text{w}_1:◊(c)\} \quad \text{/* the list of formulas is updated */}
7. checkSAT(formulas', relations, assignments)
3. \text{w}_1:(b) \quad \text{/* next formula in the list formulas */}
5. \text{w}_1:(b) \in \text{assignments}
6. formulas' = \{\text{w}_1:◊(c)\} \quad \text{/* the list of formulas is updated */}
7. checkSAT(formulas', relations, assignments)
3. \text{w}_1:◊(c) \quad \text{/* next formula in the list formulas */}
27. formulas' = \{\text{w}_1:c\} \quad \text{/* case 5: R(\text{w}_1, \text{w}_1) first it tries this */}
29. sat = checkSAT(formulas', relations, assignments)
3. \text{w}_1:c \quad \text{/* case 5: R(\text{w}_1, \text{w}_2) second time it backtracks and tries this */}
29. sat = \text{FALSE}
28. formulas' = \{\text{w}_2:c\} \quad \text{/* case 5: R(\text{w}_1, \text{w}_2) by base case since formulas' = Ø */}
30. return \text{TRUE}

Example 2 explanation:

Again checkSAT starts by looking at the first labelled formula in the list formulas: \text{w}_1:□(a). This is a box formula and the worlds related to \text{w}_1 are \text{w}_1 itself and \text{w}_2. so it removes \text{w}_1:□(a) from the list and replaces it with \text{w}_1:(a) and \text{w}_2:(a). Similar to the previous example it checks \text{w}_1:(a) and \text{w}_2:(a) are in the list assignment. It then picks the next formula in formulas: \text{w}_1:◊(c). For this case (case 5 in the algorithm) it goes through the list of worlds that \text{w}_1 is related to and tries to find one the satisfies \text{w}_1:(c). The first one it tries is \text{w}_1 itself since R(\text{w}_1, \text{w}_1) \in relations. But this fails because (\text{w}_1:c=T) \notin \text{assignments}. It then tries \text{w}_2 which is the next world \text{w}_1 is related to. This trial succeeds so the checkSAT terminates returning TRUE.
3.2 Stage2: Model generation algorithm

In this section an algorithm is given that takes a set of formulas and relations and builds a model if one exist. This algorithm will be referred to as HSAT from now on. Formulas of the form $\square \varphi$ where $\varphi$ is a formula are referred to as a box-formulas and and the ones that are not of this form are referred to as non-box formulas. HSAT takes as input a list of labelled formulas $F$, and a list of relations $R$ and terminates returning TRUE if it finds a model and FALSE otherwise. It has two interleaving parts (sub-functions) MSAT and PSAT. The first one implements satisfiability of box formulas and the second one implements satisfiability non-box formulas. The basic idea of how it attempts to generate a model is given below followed by the algorithm itself and some example.

Idea of algorithm HSAT:

First expand all the box formulas in the list $F$. For example if the formula $w_1: \square (\phi_1 \land \phi_2)$ is in $F$ and $R(1, 2)$ is in the list $R$ then the formula is expanded by adding $w_2: (\phi_1 \land \phi_2)$ to the list. If at this stage there are no non-box formulas in the list then the algorithm terminates returning TRUE. Otherwise, it picks any of the non-box formulas removes it from the list and goes to one of the following cases depending on what type of formula it is.

- If the formula is of the form $w_i: (\phi_i \land \phi_j)$ then it replaces it with the formulas $w_i: (\phi_1)$ and $w_i: (\phi_2)$ as these need to be satisfied for $w_i: (\phi_i \land \phi_j)$ to be satisfied. If in this process a box-formula is introduced then this is expanded as described above for the current set of relations.

- If the formula is of the form $w_i: (\phi_i \lor \phi_j)$ then it branch into two cases where one case tries to satisfy the left part of the disjunction and the other case tries to satisfy the right part of the disjunction. Again if a new box formula is introduce int he process then it is expanded for the current set of relations.

- If the formula is an atom then it check the list assignments to see if this labelled atom has already been assigned a truth value. If it has and this value is true then it does nothing and moves to the next formula in this list. If it has and this value is false then the algorithm terminates returning FALSE. Otherwise it assigns the value true to the labelled atom and adds this to the list of assignments.

- If the formula is a negated atom then it checks the list assignments to see if this labelled atom has already been assigned a truth value. If it has and
this value is false it does nothing and moves to the next formula in the list. If it has been assigned true then the algorithm terminates returning FALSE. Otherwise it assigns the value false to the atom and add this to the list of assignments.

- If it is a diamond formula then it first checks to see if it is already satisfied by considering all the worlds that the world containing the diamond formula is related to. If it is then it does nothing and moves to the non-box formula in the list. Otherwise it tries to satisfy it by adding the necessary formula to one of the world that the world containing the diamond formula is related to. (For example if the formula is $w_i; \phi_j$ and $w_j$ is related to $w_k$ then it tries adding $\phi_j$ to $w_k$). If all of the trials fail then it creates a new world and adds a relation between the world containing the diamond formula and the new world it created. If the diamond formula is still not satisfiable then the algorithm terminates returning FALSE.

The above process is repeated until no non-box formulas are left in the list. For negated formulas the algorithm converts them into an equivalent formula whose main connector is not negation deals with them in one of the ways discussed above (the case for a negated atom is already discussed above) The data flow diagram below further illustrates the process of model generation.
The function HSAT given below has two steps:

Step 1: split the list of formulas into two list for box and non-box. In algorithms these are referred to as 'BList' and 'nonBList' respectively.

Example: If \( F = \{ w_1: (a \land \Box b), w_1: \neg (b), w_2: \Diamond (a \lor d), w_2: \Diamond (\neg c \lor \Box (a)) \} \)
then \( BList = \{ w_2: \Box (a \lor d) \} \) and \( nonBList = \{ w_1: (a \land \Box b), w_1: \neg (b), w_2: \Diamond (a) \} \)

Step 2: Initialize the list of assignments \( A \) as an empty list and call the function MSAT with \( F, R, A \) as its parameters.

Given below are the utility functions used by the functions MSAT and PSAT followed by HSAT.

function split(nonBList, BList) {
    for any \( w_i: \phi_j \) in nonBList do;
        if \( \phi_j = \Box \phi \) where \( \phi \) is a formula
            then nonBList = nonBList - \{ \phi_j \};
            BList = BList \cup \{ \phi_j \};
    }

The function Split takes two list. One containing a possibly mixture of box and non-box formulas, 'nonBList', and one containing box formulas only, 'BList'. It updates the two lists by moving all box formulas from the 'nonBList' to the 'BList'.

Example: let nonBList = \{ w_1: (a \land \Box b), w_2: \Diamond (b) \} and BList = \{ w_2: \Box (a) \}
then after calling Split(nonBList, BList)
nonBList = \{ w_1: (a \land \Box b), w_2: \Diamond (b) \} and BList = \{ w_2: \Box (a), w_2: \Diamond (a) \}

function expand(nonBList, BList, R) {
    for any \( w_i: \Box \phi_j \) in nonBList do
        for any \( R(w_i, w_j) \in R \) do
            nonBoxList = nonBoxList \cup \{ w_j: \phi_j \};
    }

The function Expand takes a list of box formulas, 'BList', and a list of non box formulas 'nonBList' and a list relations \( R \). For any box formula in 'BList', \( w_i: \Box \phi \) and for every \( R(w_i, w_j) \) in \( R \) it adds the formula \( w_j: \phi \) to 'nonBList' if it doesn’t already contain it.

Example: let nonBList = \{ w_1: (a \land \Box b), w_2: \Diamond (b) \} and BList = \{ w_2: \Box (a) \}
and \( R = \{ R(w_1, w_1), R(w_1, w_2), R(w_2, w_1) \} \) then after calling Expand(nonBList, BList, R)
nonBList = \{ w_1: (a \land \Box b), w_2: \Diamond (b), w_1: \Box (a), w_2: \Box (a), w_1: (\neg b) \}
BList = \{ w_1: \Box (a), w_2: \Box (b) \}

function rewriteNot(w_i: \neg \phi_j) {
    if \( \phi_j = \varphi \land \varphi' \) where \( \varphi \) and \( \varphi' \) are formulas
        return \( w_i: (\neg \varphi \lor \neg \varphi') \);

26
if $\phi_j = \varphi \lor \varphi'$ where $\varphi$ and $\varphi'$ are formulas
return $w_i: (\neg \varphi \land \neg \varphi')$;

if $\phi_j = \neg \varphi$ where $\varphi$ is a formula
return $w_i: (\varphi)$;

if $\phi_j = \lozenge \varphi$ where $\varphi$ is a formula
return $w_i: (\square \neg \varphi)$;

The function rewriteNot takes a negated labelled non-atomic formula and gives an equivalent formula whose main connector is not negation.

Example: let $f = w_1:\neg (a \land \square b)$ and $g = w_1:\neg \square b$ then rewriteNot($f$) = $w_1:\neg a \lor \neg \square b$ and rewriteNot($g$) = $\lozenge \neg b$

function HSAT($F$, $R$) {
1. $BList = \emptyset$;
2. $nonBList = F$;
3. Split($nonBList$, $BList$);
4. $A = \emptyset$;
5. return MSAT($nonBList$, $BList$, $R$, $A$);
}

function MSAT($nonBList$, $BList$, $R$, $A$) {
1. Until $BList$ and $nonBList$ stop changing
2. do {
3. split($nonBList$, $BList$);
4. expand($nonBList$, $BList$, $R$);
    } 
3. return PSAT($nonBList$, $BList$, $R$, $A$);
}

function PSAT($nonBList$, $BList$, $R$, $A$) {
1. if $nonBList = \emptyset$ then return TRUE; \* base case \*
2. let $w_i: \phi_j$ be the first formula in $nonBList$;
3. $nonBList = nonBList - \{w_i: \phi_j\}$; \* remove the fist formula from nonBList *\n4. case0: $\phi_j$ is an atom
5. if ($w_i: \phi_j = T$) $\in A$
6. then return PSAT ($nonBList$, $BList$, $R$, $A$)
7. if ($w_i: \phi_j = F$) $\in A$
8. then return FALSE;
9. else $A = A \cup \{w_i: \phi_j = T\}$
10. return PSAT ($nonBList$, $BList$, $R$, $A$);
11. case1: $\phi_j = \varphi \land \varphi'$ where $\varphi$ and $\varphi'$ are formulas
12. $nonBList = nonBList - \{w_i: \phi_j\}$;
13. return MSAT($nonBList$, $BList$, $R$, $A$);
14. case2: $\phi_j = \varphi \lor \varphi'$ where $\varphi$ and $\varphi'$ are formulas \*branch*]
15. $nonBList' = nonBList \cup \{w_i: \varphi\}$;
16. $nonBList'' = nonBList \cup \{w_i: \varphi'\}$;
17. return MSAT($nonBList'$, $BList$, $R$, $A$) || MSAT($nonBList''$, $BList$, $R$, $A$);
18. case3: $\phi_j = \neg \varphi$ where $\varphi$ is an atom
19. if ($w_i: \phi_j = F$) $\in A$
20. then return PSAT ($BList$, $nonBList$, $R$, $A$);
if \((w_i:\phi_j = T) \in A\) then return false;

else \(A = A \cup \{w_i:\phi_j = F\}\);

return PSAT (BList, nonBList, R, A);

case 4: \(\phi_j = \neg \varphi\) where \(\varphi\) is a non atom formula 

\(f = \text{rewriteNOT}(w_i: \neg \varphi)\);

nonBList = nonBList \cup \{f\};

return MSAT(nonBList, BList, R, A);

case 5: \(\phi_j = \Diamond \varphi\) where \(\varphi\) is a formula /* backtrack*/

if \(\exists j: (R(w_i,w_j) \in R) \text{ and } w_j: \varphi \notin F\)

return PSAT(nonBList, BList, R, A);

sat = FALSE;

while (sat = FALSE) { /* try to satisfy the diamond formula */
    try for j such that \(R(w_i,w_j) \in R\) and \(w_j: \varphi \notin F\)
    nonBList' = nonBList \cup \{w_j: \varphi\};
    sat = MSAT(nonBList', BList, R, A);
}

while (sat = FALSE) { /* try to satisfy the diamond formula */
    try for any j such that \(R(w_i,w_j) \notin A\)
    nonBList' = nonBList \cup \{w_j: \varphi\};
    \(R' = R \cup \{R(w_i,w_j)\}\)
    sat = MSAT(nonBList', BList, R, A);
}

return sat;

}

The function MSAT expands the box formulas as much as possible for the current set of relations by calling the functions 'split' and 'expand' repeatedly. It then calls PSAT. PSAT then goes through the list of formulas and tries to satisfy all the non-box formulas. If at any point it introduces a new relation or a new box formula then it call MSAT again.

The recursive algorithm, HSAT, does two types of backtracking

1. When it comes across disjunctive formulas \(w_i:(\phi \lor \phi')\): In this case it first tries to satisfy the first part of the formula i.e \(w_i:(\phi)\). If it is successful it returns true and terminates. Otherwise it backtracks and tries to satisfy the second part. If this is successful then it returns TRUE and terminates. Otherwise it terminates returning FALSE.(case 2 of function PSAT)

2. When it comes across diamond formulas \(w_i:(\Diamond \phi)\): This is done in four steps(case 5 of the function PSAT given above)
step 1: It first checks if this formula is already satisfied (i.e. \( \exists w_j ( R(w_i, w_j) \land w_j; \phi ) \)). If it is then the algorithm terminates returning FALSE. Otherwise it goes to step 2.

step 2: It picks the first world that \( w_i \) is related to, say \( w_j \) and it adds \( \phi \) to \( w_j \). If this is successful then it returns TRUE and terminates. Otherwise it backtracks and tries the next world that \( w_i \) is related to this. In this way it tries all the worlds that \( w_i \) is related to. If none of them are successful then goes to step 3.

step 3: It tries to introduce a new relation with one of the existing worlds backtracking each time it is unsuccessful. If none of the new relations leads to the algorithm returning true then it goes to the final step, step 4.

step 4: It introduces a new world \( w_k \), adds a relation between \( w_i \) and \( w_k \) and adds \( \phi \) to the new world. If this is successful then the algorithm terminates returning TRUE. Otherwise it terminates returning FALSE.

These four steps are covered by case 5 of the function \( \text{PASAT} \) above. The first case is covered by the if case, step 2 is covered by the first while loop and steps 3 and 4 are covered by the second while loop.

**Example 1:**

Let \( F = \{ w_1:(a \land b), w_1:\neg(a \land c), w_1:\square(a) \} \)
\( R = \{ R(w_1, w_1) \} \)

**\( \text{HSAT} (F, R) \)**
1. \( \text{BList} = \emptyset \)
2. \( \text{nonBList} = \{ w_1:(a \land b), w_1:\neg(a \land c), w_1:\square(a) \} \)
3. \( \text{nonBList} = \{ w_1:(a \land b), w_1:\neg(a \land c) \} \)
   \( \text{BList} = \{ w_1:\square(a) \} \)
   \( \text{*the list is split in to box and non-box list*} \)
4. \( A = \emptyset \)
5. return \( \text{MSAT (BList, nonBList, R, A)} \)

**\( \text{MSAT} \)**
1. \( \text{nonBList} = \{ w_1:(a \land b), w_1:\neg(a \land c), w_1:(a) \} \)
   \( \text{BList} = \{ w_1:\square(a) \} \)
3. return \( \text{PSAT (nonBList, BList, R, A)} \)

**\( \text{PSAT} \)**
3. \( w_1:(a \land b) \)
   \( \text{*first formula in nonBList*} \)
4. \( \text{nonBList} = \{ w_1:\neg(a \land c), w_1:(a) \} \)
12. \( w_1:(a \land b) \equiv w_2:(a) \land w_1:(b) \)
   \( \text{*case 1*} \)
13. \( \text{nonBList} = \{ w_1:(b), w_1:(a), w_1:\neg(a \land c) \} \)
14. return \( \text{MSAT (BList, nonBList, R, A)} \)
no box formula introduced so MSAT does nothing

1,2. nonBList = \{ w_1:(b), w_1:(a), w_1:(\neg (a \land c)) \}
BList = \{ w_1:\Box(a) \}
3. return PSAT(nonBList, BList, R, A)

PSAT
3. w_1:(b) \hspace{1cm} \text{*first formula in nonBList*}
4. nonBList = \{ w_1:(a), w_1:(\neg (a \land c)) \}
10. A = \{ w_1:(b)= T \} \hspace{1cm} \text{*case 0: b assigned true at w_1*}
11. return PSAT(nonBList, BList, R, A)

PSAT
3. w_1:(a) \hspace{1cm} \text{*first formula in nonBList*}
4. nonBList = \{ w_1:(a) \}
10. A = \{ w_1:(a)= T, w_1:(b)= T \} \hspace{1cm} \text{case 0*}
11. return PSAT(nonBList, BList, R, A)

10. A = \{ w_1:(b)= T \} \hspace{1cm} \text{*case 0*}
11. return PSAT(nonBList, BList, R, A)

25. (w_1:(a)= T) \in A \hspace{1cm} \text{case 0*

24. A = \{ w_1:(b)= T, w_1:(c)=F \}
25. return PSAT(nonBList", BList, R, A)

MSAT
1,2. nonBList" = \{ w_1:(\neg c) \}
BList = \{ w_1:(\neg c) \}
3. return PSAT(nonBList", BList, R, A)

PSAT
3. w_1:(\neg (a \land c)) \hspace{1cm} \text{first formula in nonBList*}
4. nonBList = \{ \}
27. f = w_1:(\neg (a) \lor \neg (c)) \hspace{1cm} \text{case 4*}
28. nonBList = \{ w_1:(\neg a \lor \neg c) \}
3. w_1:(\neg a \lor \neg c) \hspace{1cm} \text{first formula in nonBList*}
4. nonBList = \{ w_1:(\neg a) \}
16. nonBList' = \{ w_1:(\neg a) \} \hspace{1cm} \text{branch*}
17. nonBList" = \{ w_1:(\neg c) \}
18. return MSAT(nonBList", BList, R, A) || \text{splits in to the 2 cases*}
MSAT(nonBList", BList, R, A)

Branch 1
MSAT
1,2. nonBList' = \{ w_1:(\neg a) \}
BList = \{ w_1:(\neg a) \}
3. return PSAT(nonBList, BList, R, A)

PSAT
4. w_1:(\neg a) \hspace{1cm} \text{first formula in nonBList*}
5. nonBList' = \{ \}
25. (w_1:(a) = T) \in A \hspace{1cm} \text{case 0*}
26. return FALSE

Branch 2
MSAT
1,2. nonBList" = \{ w_1:(\neg c) \}
BList = \{ w_1:(\neg c) \}
3. return PSAT(nonBList, BList, R, A)

PSAT
4. w_1:(\neg a) \hspace{1cm} \text{first formula in nonBList*}
5. nonBList" = \{ \}
20. (w_1:(c)=F) \notin A \hspace{1cm} \text{case 0*}
22. (w_1:(c)=T) \notin A
24. A = \{ w_1:(b)= T, w_1:(c)=F \}
25. return PSAT(nonBList", BList, R, A)

PSAT
1. nonBList = Ø
2. return TRUE  
   \*base case\*
18 return (FALSE || TRUE) ≡ TRUE

Given below is the model for this example:

A second example is given below with less detailed but hopefully more easily understandable solution to the one given above. Notice that at each stage of the algorithm all we do is update the four list: list of non-box formulas, list of box formulas, list of relations and list of assignments. So for the example below I have given the state of each list as the function is applied.

**Example 2:** (This example builds on example 1 above)

Let $F = \{ w_1 : (a \land \neg b), w_1 : \neg (a \land c), w_1 : a, w_1 : (a \land \lozenge b) \}$

$R = \{ R(w_1, w_1), R(w_1, w_2) \}$

$A = \emptyset$

**HSAT 1-4**
nonBList = \{ $w_1 : (a \land \neg b), w_1 : \neg (a \land c), w_1 : (a \land \lozenge b) $ \}
BList = \{ $w_1 : \Box (a)$ \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \emptyset

**MSAT 1,2**  \*Expand the box formulas*
nonBList = \{ $w_1 : (a \land \neg b), w_1 : \neg (a \land c), w_1 : (a \land \lozenge b), w_1 : a, w_2 : a $ \}
BList = \{ $w_1 : \Box (a)$ \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \emptyset

**PSAT 3,4,12-14**  \*case 1*
nonBList = \{ $w_1 : \neg (a \land c), w_1 : (a \land \lozenge b), w_1 : a, w_2 : a $ \}
BList = \{ $w_1 : \Box (a)$ \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \emptyset

**PSAT 3,4,19-24**  \*case 3*
nonBList = \{ $w_1 : (a \land \neg c), w_1 : (a \land \lozenge b), w_1 : a, w_2 : a $ \}
BList = \{ $w_1 : \Box (a)$ \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ $w_1 : b = F$ \}

**PSAT 3,4,26-29**  \*re-write the negated formula*
nonBList = \{ $w_1 : (\neg a \lor \neg c), w_1 : (a \land \lozenge b), w_1 : a, w_2 : a $ \}
BList = \{ $w_1 : \Box (a)$ \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}

31
A = \{ w_1 : b = F \}

It branches on $w_1 : (\neg a \lor \neg c)$, BRANCH 1 and BRANCH 2 as given below:

PSAT 3,4,15-18
nonBList' = \{ w_1 : (\neg a), w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
nonBList" = \{ w_1 : (\neg c), w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F \}

BRANCH 1
nonBList' = \{ w_1 : (\neg a), w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
nonBList" = \{ w_1 : (\neg a), w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F \}

PSAT 3,4,19-24
nonBList' = \{ w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
nonBList" = \{ w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F, w_1 : a = F \} \* assign false to a at w_1*

PSAT 12-14
nonBList' = \{ w_1 : a, w_1 : ♦ b, w_1 : a, w_1 : a \}
nonBList" = \{ w_1 : a, w_1 : ♦ b, w_1 : a, w_1 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F, w_1 : a = F \} \* assign false to a at w_1*

return FALSE

BRANCH 2
nonBList' = \{ w_1 : (\neg c), w_1 : (a \land ♦ b), w_1 : a, w_2 : a \}
nonBList" = \{ w_1 : (\neg c), w_1 : (a \land ♦ b), w_1 : a, w_2 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F \}

PSAT 19-25
nonBList' = \{ w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
nonBList" = \{ w_1 : (a \land ♦ b), w_1 : a, w_1 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F, w_1 : c = F \} \* assign false to c at w_1*

PSAT 3,4,12-14
nonBList' = \{ w_1 : ♦ b, w_1 : a, w_2 : a \}
nonBList" = \{ w_1 : ♦ b, w_1 : a, w_2 : a \}
BList =\{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F, w_1 : c = F \}

PSAT 34-37
nonBList' = \{ w_2 : b, w_1 : a, w_2 : a \}
nonBList" = \{ w_1 : □ (a) \}
R = \{ R(w_1, w_1), R(w_1, w_2) \}
A = \{ w_1 : b = F, w_1 : c = F \}
A = \{w_1: b = F, w_1: c = F, w_1: a = T\}

PSAT 3, 4, 5-11
nonBList' = \{w_1: a, w_2: a\}
BList = \{w_1: □(a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: b = F, w_1: c = F, w_1: a = T, w_2: b = T\}

PSAT 3, 4, 5-11
nonBList' = \{w_2: a\}
BList = \{w_1: □(a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: b = F, w_1: c = F, w_1: a = T, w_2: b = T\}

PSAT 5-11
nonBList' = Ø
BList = \{w_1: □(a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: b = F, w_1: c = F, w_1: a = T, w_2: b = T, w_2: a = T\}

PSAT 1-2
return TRUE  \\
\text{\textasciitilde BRANCH 2 succeeds \textasciitilde}

Hence the model generated for this example is as shown below where
A = \{w_1: b = F, w_1: c = F, w_1: a = T, w_2: b = T, w_2: a = T\}
3.3 Stage3: Model generation algorithm with frame conditions

In this section we consider how to build models that satisfy some of the standard frame conditions listed below. We revise the algorithm from the previous section so that, in addition to a set of formulas and relations, it also takes a list of frame conditions \( C \) that we would like the model to satisfy. This new revised version will be refereed to as HSAT’.

The frame can be:

- **Transitive** (4): \( \forall u, v, w (R(u, v) \land R(v, w) \rightarrow R(u, w)) \)
- **Reflexive** (T): \( \forall w (R(w, w)) \)
- **Euclidean** (5): \( \forall u, v, w (R(u, v) \land R(u, w) \rightarrow R(v, w)) \)
- **Symmetric** (B): \( \forall u, v (R(u, v) \rightarrow R(v, u)) \)
- **Serial** (D): \( \forall w \exists u (R(w, u)) \)

or any combination of the above.

The algorithm described in this section is very similar to the one in the previous section. The key difference is that every time we introduce a new relation we also introduce all the relations necessary for the frame condition to hold.

Example: let the set of relations \( R = (R(w_1, w_2), R(w_1, w_3)) \) and assume the frame is transitive. During the model construction if the new relation \( R(w_2, w_3) \) is introduced then we also need to introduce \( R(w_1, w_3) \) for the transitivity condition to hold.

We will see that HSAT’ deals with the first four frame conditions in a similar way. However the last condition is different from the others because it is an ‘existence’ condition. It says given any world \( w \) there exists a world \( u \) that is related to it. For example, for symmetry if we have \( R(u, v) \) then we have to have \( R(v, u) \) and similarly for transitive, reflexive and euclidean frames. However, for serial frames we have a choice of worlds that a world \( w \) can be related to. So it would make sense to deal with this case differently.

For serial frame the idea is that we introduce a special formula ‘\( \diamond \xi \)’ to each existing world where ‘\( \xi \)’ is a special character that does not appear as part of any of
the formulas whose model we are interested in finding. This makes sure that in the model each world is related to at least one other world.

Example: Let list of formulas $F = \{ w_1: a \land b, w_2: c \}$
list of relations $R = \{ R(1,1) \}$
list of frame conditions $C = \{ D \}$
Then since the frame is a serial frame we add the $\Diamond \xi$ to $w_1$ and $w_2$ to ensure seriality. So $F$ now equals $\{ w_1: a \land b, w_2: c, w_1: \Diamond \xi, w_2: \Diamond \xi \}$

**Idea of algorithm HSAT’:**

As already mentioned HSAT’ is very similar to HSAT. The only differences are

- As a first step of the model construction add the relations necessary to satisfy the frame conditions
  Example: Let the initial set of relations $R = \{ R(w_1, w_2), R(w_1, w_3) \}$ and let the set of frame conditions $C = \{ 4, 5 \}$. Then the after adding the new relations to satisfy this conditions $R = \{ R(w_1, w_2), R(w_1, w_3), R(w_2, w_3), R(w_3, w_2), R(w_2, w_2), R(w_3, w_3) \}$
- If the frame is serial (i.e. $D \in C$) then add the formula $\Diamond \xi$ to all the existing worlds
- Whenever a new relation is added by the algorithm also add all the necessary relations to satisfy the frame conditions.
- If the frame is serial, whenever a new world is added then add the formula $\Diamond \xi$ to the new world

Notice that a new relation or a new world is only introduced when the algorithm deals with diamond formula which is case 5 in HSAT.

The algorithm HSAT’ is given below. The utility functions ‘split’ ‘expand’ and ‘rewriteNot’ are as given for the algorithm HSAT in the previous section. And one additional utility function ‘update’ is given which updates the list of relations based on the list of frame condition.

[Again $F$ is the list labelled formulas, $R$ is the list of relations and $C$ is the list of frame conditions]
function update(F,R,C) {
  if(4 ∈ C) for all pairs R(u,v) and R(v,w) in R add R(u,w) to R
  if(T ∈ C) for all worlds w add R(w,w) to R
  if(5 ∈ C) for all pairs R(u,v) and R(v,w) in R add R(u,w) and R(w,u) to R
  if(B ∈ C) for all R(u,v) in R add R(v,u) to R
  repeat the above four until no more relations can be added
}

function HSAT′(F, R, C) {
  1. BList = Ø;
  2. nonBList = F;
  3. Split(nonBList, BList);
  4. A = Ø;
  5. if(D ∈ C) for all worlds w add w:♦ξ to nonBList
  6. update(F, R, C)
  7. return MSAT′(nonBList, BList, R, A, C);
}

function MSAT′(nonBList, BList, R, A, C) {
  1. Until BList and nonBList stop changing
  2. do {
      split(nonBList, BList);
      expand(nonBList, BList, R, C);
    } while not stop
  3. return PSAT′(nonBList, BList, R, A, C);
}

function PSAT′(nonBList, BList, R, A, C) {
  1. if nonBList = Ø then return TRUE; /* base case */
  2. let w_i:φ_j be the first formula in nonBList;
  3. nonBList = nonBList - {w_i:φ_j};
  4. case0: φ_j is an atom
  5. if (w_i:φ_j = T) ∈ A
  6. then return PSAT′(nonBList, BList, R, A, C)
  7. if (w_i:φ_j = F) ∈ A
  8. then return FALSE;
  9. else A = A ∪ {w_i:φ_j}
  10. return MSAT′(nonBList, BList, R, A, C);
  11. case1: φ_j = φ ∧ φ’ where φ and φ’ are formulas
  12. nonBList = nonBList ∪ {w_i:φ, w_i:φ’};
  13. return MSAT′(nonBList, BList, R, A, C);
  14. case2: φ_j = φ ∨ φ’ where φ and φ’ are formulas /* branch */
  15. nonBList” = nonBList ∪ {w_i:φ};
  16. nonBList” = nonBList ∪ {w_i:φ’};
  17. return MSAT′(nonBList”, BList, R, A, C) || MSAT′(nonBList”, BList, R, A);
  18. case3: φ_j = ¬φ where φ is an atom
  19. if (w_i:φ_j = F) ∈ A
}
21. then return $\text{PSAT}'(\text{BList}, \text{nonBList}, R, A)$;
22. if $(w_i; \phi_j = T) \in A$
23. then return false;
24. else $A = A \cup \{w_i; \phi_j = F\}$;
25. return $\text{PSAT}'(\text{BList}, \text{nonBList}, R, A, C)$;
26. case 4: $\phi_j = \neg \varphi$ where $\varphi$ is a non-atom formula
27. $\xi = \text{rewriteNOT}(w_i; \neg \varphi)$;
28. $\text{nonBList} = \text{nonBList} \cup \{\xi\}$;
29. return $\text{MSAT}'(\text{nonBList, BList, R, A, C})$;

30. case 5: $\phi_j = \Box \varphi$ where $\varphi$ is a formula /* backtrack */
31. if $\exists j: R(w_i, w_j) \in R$ and $w_j; \varphi \in F$
32. return $\text{PSAT}'(\text{nonBList, BList, R, A, C})$;
33. sat = FALSE;
34. while (sat = FALSE){ /* try to satisfy the diamond formula */
35. try for $j$ such that $R(w_i, w_j) \in R$ without adding a new relation*/
36. nonBList' = nonBList $\cup \{w_j; \varphi\}$;
37. sat = MSAT'(nonBList', BList, R, A, C);
}
38. while (sat = FALSE){ /* try to satisfy the diamond formula */
39. try for any $j$ such that $R(w_i, w_j) \notin A$
40. nonBList' = nonBList $\cup \{w_j; \varphi\}$
41. $R' = R \cup \{R(w_i, w_j)\}$
42. if new world $w_j$ added and $D \in C$ /*if frame is serial add $\Box \xi$ to $w_j$*/
43. then add $w_j; \Box \xi$ to nonBList
44. update($R'$)
45. sat = MSAT'(nonBList', BList, $R'$, A, C);
}
46. return sat;

The main difference between the new algorithm and the previous one can be found on lines 5 and 6 of function $\text{HSAT}'$ and lines 42 and 43 of function $\text{PSAT}'$.

Example 1:
Let $F = \{w_1:(a \land b), w_1:\Box(\neg a), w_1:\Box(a)\}$
R = $\{R(w_1, w_2)\}$
C = $\{T, 5\}$

$\text{HSAT'}$ 1-4
nonBList = $\{w_1:(a \land b), w_1:\Box(\neg a)\}$ /* the list is split in to box and non-box list */
BLlist = $\{w_1:\Box(a)\}$
R = $\{R(w_1, w_2), R(w_1, w_1), R(w_2, w_1), R(w_2, w_2)\}$
A = $\emptyset$

$\text{MSAT'}$ 1-3 /* expand box formulas */
nonBList = $\{w_1:(a \land b), w_1:\Box(\neg a), w_1:a, w_2:a\}$
BList = $\{w_1:\Box(a)\}$
R = $\{R(w_1, w_2), R(w_1, w_1), R(w_2, w_1), R(w_2, w_2)\}$
A = $\emptyset$
C = $\{T, 5\}$
nonBList = { w₁:b, w₁:¬¬a, w₁:a, w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T }
C = { T, 5 }

nonBList = { w₁:¬¬a, w₁:a, w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T }
C = { T, 5 }

nonBList = { w₁:a, w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T, w₁:a = F }
C = { T, 5 }

nonBList = { w₁:a, w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T, w₂:a = F }
C = { T, 5 }

nonBList = { w₁:a, w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T, w₁:a = F, w₂:a = T }
C = { T, 5 }

nonBList = { w₂:a }  
BList = { w₁:□(a) }
R = { R(w₁,w₂), R(w₁,w₁), R(w₂,w₁), R(w₂,w₂) }
A = { w₁:b = T, w₂:a = F, w₁:a = T }
C = { T, 5 }

return FALSE  

return FALSE  

return FALSE  

return FALSE  

return FALSE
PSAT′ 3,4,30,38-42

backtrack again and try introducing a new
world w3 related to w1 and adding ¬a to it.

nonBList = {w3:a, w3:(¬a), w1:a, w2:a}
BList = {w1:□(a)}
R = { R(w1,w2), R(w1,w1), R(w2,w1), R(w2,w2), R(w1,w3), R(w2,w3), R(w3,w1), R(w3,w2), R(w3,w3)}
A = {w1:b = T}
C = {T, 5}

PSAT′ 3,4,19,22,23
return FALSE.

Hence the conclusion is that the initial set of formulas F = { w1:(a∧b), w1:◊(¬a), w1:□(a) } are not satisfiable for R(w1,w2) and a reflexive and euclidean frame.

Example 2:
Let F = {w1:(□(a) ∧ ◊(◊(¬a)))}
R = {R(w1,w2)}
C = {4}

HSAT′ 1-5
BList = Ø
nonBList = {w1:(□(a) ∧ ◊(◊(¬a)))}
A = Ø
C = {4}

MSAT′ 1-3
BList = Ø
nonBList = {w1:(□(a) ∧ ◊(◊(¬a)))}
R = Ø
A = Ø
C = {4}

PSAT′ 3,4,12,13
BList = Ø
nonBList = {w1:□(a), w1:◊(◊(¬a)))
R = Ø
A = Ø
C = {4}

MSAT′ 1-3
BList = {w1:□(a)}
nonBList = {w1:◊(◊(¬a)))
R = Ø
A = Ø
C = {4}
\*try to satisfy $w_1: \Box (\neg a)$ by adding $\Diamond (\neg a)$ to $w_1$ and adding the relations $R(1,1)\*\$

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: \Diamond (\neg a)\}
R = \{R(w_1, w_1)\}
A = \emptyset
C = \{4\}
```

\*expand the box formula for the new relation added\*

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: a, w_1: \Diamond (\neg a)\}
R = \{R(w_1, w_1)\}
A = \emptyset
C = \{4\}
```

\*try to satisfy $w_1: \Box (\neg a)$ by adding $\neg a$ to $w_1$*

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: (\neg a)\}
R = \{R(w_1, w_1)\}
A = \{w_1: a = T\}
C = \{4\}
```

\*backtrack and try to satisfy $w_1: \Box (\neg a)$ by adding $\neg a$ to $w_2$*

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: (\neg a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: a = T, w_2: a = T\}
C = \{4\}
```

\*expand box formula for the new relation added\*

```
BList = \{w_1: \square (a)\}
nonBList = \{w_2: a, w_2: (\neg a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: a = T, w_2: a = T\}
C = \{4\}
```

PSAT’ 3,4,19,22,23
return FALSE

PSAT’ 3,4,30,38-41
\*try to satisfy $w_1: \Box (\neg a)$ by adding $\Diamond (\neg a)$ to $w_1$ and adding the relations $R(1,1)\*\$

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: (\neg a)\}
R = \{R(w_1, w_1)\}
A = \emptyset
C = \{4\}
```

PSAT’ 3,4,5-10
\*expand box formula for the new relation added\*

```
BList = \{w_1: \square (a)\}
nonBList = \{w_1: (\neg a)\}
R = \{R(w_1, w_1), R(w_1, w_2)\}
A = \{w_1: a = T, w_2: a = T\}
C = \{4\}
```

PSAT’ 3,4,19,22,23
return FALSE
PSAT' 3,4.

\*backtrack and try to satisfy \( w_1 : \Box (\Diamond (\neg a)) \) by adding a new world \( w_2 \), adding \( \Diamond (\neg a) \) to \( w_2 \) and adding the relations \( R_{1,2} \).*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_2 : (\neg a) \} \)
\( R = \{ R(w_1, w_2) \} \)
\( A = \emptyset \)
\( C = \{ 4 \} \)

MSAT' 1-3

\*expand box formula for the new relation added*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_2 : (\neg a) \} \)
\( R = \{ R(w_1, w_2) \} \)
\( A = \emptyset \)
\( C = \{ 4 \} \)

PSAT' 3,4,5-10

\*try to satisfy \( w_2 : (\neg a) \) by adding \( \neg a \) to \( w_2 \) and adding the relations \( R_{w_2, w_1} \).*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_2 : (\neg a) \} \)
\( R = \{ R(w_1, w_2), R(w_2, w_1), R(w_1, w_1) \} \)
\( A = \{ w_2 : a = T \} \)
\( C = \{ 4 \} \)

return FALSE

MSAT' 1-3

\*expand box formula for new relation added*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_1 : \Box (\neg a) \} \)
\( R = \{ R(w_1, w_1), R(w_1, w_2), R(w_1, w_1) \} \)
\( A = \{ w_2 : a = T \} \)
\( C = \{ 4 \} \)

PSAT' 3,4,19,22,23

\*try to satisfy \( w_2 : (\neg a) \) by adding \( \neg a \) to \( w_1 \) and adding the relations \( R_{w_2, w_1} \).*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_1 : (\neg a) \} \)
\( R = \{ R(w_1, w_1), R(w_2, w_1), R(w_2, w_2) \} \)
\( A = \{ w_2 : a = T \} \)
\( C = \{ 4 \} \)

PSAT' 3,4,30,34-37

\*list of relations also updated for transitive frame*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_2 : (\neg a) \} \)
\( R = \{ R(w_1, w_2), R(w_2, w_1), R(w_1, w_1) \} \)
\( A = \{ w_2 : a = T \} \)
\( C = \{ 4 \} \)

MSAT' 1-3

\*list of relations also updated for transitive frame*

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_1 : (\neg a) \} \)
\( R = \{ R(w_1, w_1), R(w_2, w_1), R(w_1, w_1) \} \)
\( A = \{ w_2 : a = T \} \)
\( C = \{ 4 \} \)

PSAT' 3,4,5-10

\( \text{BList} = \{ w_1 : \Box (a) \} \)
\( \text{nonBList} = \{ w_1 : (\neg a) \} \)
\( R = \{ R(w_1, w_2), R(w_2, w_1), R(w_1, w_1) \} \)
\( A = \{ w_2 : a = T, w_1 : a = T \} \)
C = {4}

PSAT' 3,4,19,22,23
return FALSE

PSAT 3,4,30,34-37
\* try to satisfy w₂:◊(¬a) by adding a new world w₃, adding ¬a to w₃ and and adding the relations R(w₂,w₃)\*
BList = {w₁:□(a)}
nonBList = {w₃:¬(¬a)}
R = {R(w₁, w₂), R(w₂, w₃), R(w₁, w₃)} \*list of relations also updated for transitive frame\*
A = {w₂:a = T}
C = {4}

MSAT' 1-3
\*expand box formula for new relation added\*
BList = {w₁:□(a)}
nonBList = {w₃:a, w₃:¬(¬a)}
R = {R(w₁, w₂), R(w₂, w₃), R(w₁, w₃)}
A = {w₂:a = T}
C = {4}

PSAT' 3,4,5-10
BList = {w₁:□(a)}
nonBList = {w₃:¬(¬a)}
R = {R(w₁, w₂), R(w₂, w₃), R(w₁, w₃)}
A = {w₂:a = T, w₃:a = T}
C = {4}

PSAT' 3,4,19,22,23
return FALSE

Hence the original list of formulas F = {w₁:□(a) ∧ ◊(◊(¬a))} is not satisfiable for a transitive frame. Notice that the axiom □a → □□a characterizes the class of transitive frames and the list of formulas F contains a negation of this axiom so as expected this list has no model with transitive frame.
Chapter 4

Design, Implementation and Testing

4.1 Data Structures design

Before implementing the algorithms given the previous chapter I investigated the different options of data structures that could be used to represent different components like formulas and assignments. Because of the recursive nature of the algorithms data structures that allow for efficient access and modification are key to the overall efficiency of the implemented algorithms. Given below are the data structures used in the actual implementation.

4.1.1 Formulas as Binary Trees

As seen in section 2.3 the different algorithms require all sub-formulas of non-atomic formulas. Representing the formulas as binary trees would allow the algorithms to traverse down the branches and get the sub-formulas efficiently. This data structure together with some of the access methods is given below.

```c
formulaTree{
    formulaType; \* this is one of \{OR, AND, BOX, NOT, DIAMOND, ATOM\}\n    Label; \* this is the label of the formula which world it holds\n    formulaTree leftFormulaTree; \* left branch of formula tree\n    formulaTree rightFormulaTree; \* right branch of formula tree\n    setRightFormula(formulaTree); \* sets the left branch of the formula tree\n}```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setLeftFormula(formulaTree)</td>
<td>sets the left branch of the formula tree*</td>
</tr>
<tr>
<td>getLeftTree()</td>
<td>returns the left sub tree*</td>
</tr>
<tr>
<td>getRightSubTree()</td>
<td>returns the right sub tree*</td>
</tr>
<tr>
<td>getFormulaType()</td>
<td>returns the formula type*</td>
</tr>
</tbody>
</table>

Example:
Let formula \( F = (\neg a \lor \neg \Box b \land \Diamond c) \)
The formula tree for this formula is as shown below. Notice that the left branch is empty for unary operators such as \( \neg \), \( \Box \) etc

4.1.2 Relations as a matrix

Relations are represented by an \( n \times n \) matrix where \( n \) is the maximum no of worlds possible. Each entry \((i,j)\) in the matrix will be ‘1’ if world \( i \) is related to world \( j \) and a ‘0’ otherwise. This matrix structure allows for efficient implementation of frame conditions such as transitivity, reflexivity etc by using algorithms similar to
the Warshall algorithm. For example if at any stage in the algorithm the relation matrix is as shown in fig1 and it is known that the frame is a transitive frame, we can compute the transitive closure of the matrix by applying Warshall’s algorithm a few times (until the matrix stops changing) and this generates the matrix given in fig2.

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

fig1

\[
\begin{pmatrix}
0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

fig2

4.1.3 Assignments as Hash map

Each atom is associated with an assignment which, at any point in the algorithm, contains which worlds the atom is true at and which ones it is false at. Throughout the algorithm the assignment of an atom is accessed for a lookup, insertion and deletion several times. Using a hash map data structure which supports these operations efficiently leads to the overall efficient implementation of the algorithm. One option for a unique search key is to use the atoms themselves as a string which is what was used in this case. So that hash map maps the key which is the atom string to the assignment.

```
formulaMap{
    insert(atom, assignment); \*inserts this new entry into the hash table* \\
    remove(atom); \*removes the entry with this key from the hash table* \\
    getValue(atom); \*returns the assignment of the atom with this key* 
}
```
4.2 Implementation

Apart from flexibility and portability, I chose Java as the implementation language because it provides all the data structures mentioned in the previous section together with all the access methods needed. The implementation was done in four stages. The first stage was writing a parser to parse input formulas and relations. This was then followed by implementation of the model checking algorithms checkSAT (3.1). The third stage was implementation of the model building algorithms HSAT without the frame conditions. The fourth and last stage was implementation the frame conditions. Each stage was also followed by testing.

4.2.1 Formulas and parsing

The first implementation challenge was parsing input formulas and relations. For this I used a system called JavaCUP (Java Based Constructor of Useful Parsers). JavaCUP is written in Java and given simple specifications, it generates LALR parsers. It uses specifications including embedded Java code, and produces parsers which are implemented in Java. Having written the specification and got the parser generated by JavaCUP I was able to begin implementing the algorithms in chapter 3. Given below is the list of symbols I used to represent the logical connectives.

- conjunction(∧): \( \rightarrow \hat{\cdot} \)
- disjunction(∨): \( \rightarrow | \)
- box (□): \( \rightarrow + \)
- diamond(♦): \( \rightarrow < \)
- implication(→): \( \rightarrow > \)
- negation(¬): \( \rightarrow \hat{\cdot} \)

Worlds (labels) are represented using positive integers (0,1,2,...) where 0 represents the first world.

4.2.2 Algorithm implementation

Even though the aim of the project is to write and implement a model generation algorithm, getting a model verification algorithms as a first step gave me a lot of insight into how I could go about writing the model builder. After the model checker,
I implemented the model generator.

The diagram below attempts to give a very general overview of the data flow in the system. Only the main classes have been represented and some classes have been grouped together for convenience and clarity. Parser refers to all classes to do with parsing the input including the scanner, lexical analyzer etc. Formula refers to the formula class which implements the binary tree structure with all the access methods. Relation refers to the relation class which has a two dimensional array for storing the relations and all the necessary access methods including adding relations, removing relations, cloning a relation object etc. It also stores the frame properties and has methods for computing the matrix closures for different frame conditions.

The class that generates the models is modelGenerator. It has instances of Relation and formulaMap, and three lists of formulas.

- **boxList** which has all the box formulas
- **nonBoxList** which has all the non-box formulas the model has yet to satisfy
- **satFormulas** which has the formulas the model has satisfied so far

The reason for having the third list is that if during the model building we come across formulas that we have already seen and satisfied before we don't want to
have to check if they are satisfiable again. For a reasonably large set of formulas this could make a significant difference it the time it takes to build a model.

**Example:**

Let the list of formulas we are building a model for contain formula s

\[ w_1: \Box(a \land \Diamond\Diamond(b \land c) \land \neg(b \lor c)) \text{ and } w_2: (a \land \Diamond\Diamond(b \land c) \land \neg(b \lor c)) \]

Now assume that the model builder has established the second formula is satisfied and has removed it from the list. If the a new relation \( R(w_1, w_2) \) is added at some future point the box formula is expanded for this new relation giving \( w_2: (a \land \Diamond\Diamond(b \land c) \land \neg(b \lor c)) \). This means the model builder has to check this formula is satisfiable again. However if it had kept a list of all the formulas that it has already established are satisfiable then it can avoid having to check this formula again.

Another thing the modelGenerator does for efficiency is to check if the list of formulas contains a formula and its negation (\( w_i: \varphi \) and \( w_i: \neg \varphi \) where \( \varphi \) is a formula). If this is the case it concludes that a model does not exist without having to do anything further.

### 4.3 Testing

I have tested the system thoroughly at different stages of development as detailed below. Since I have implemented it in a very modular way I was able to test small part independently before going on to testing the whole system.

#### 4.3.1 Stage 1: Testing the parser

The parser takes a string representing a formula, parses it and builds a formula tree. For testing I wrote a method called `printFormula` that takes a formula tree and prints it out by traversing down the branches. So the formula tree constructed by the parser is passed to `printFormula` and print formula then prints out the formula as a string which I can compare with the formula passed to the parser. I considered different cases including:

- The formula \( \Box A \land B \lor C \) should be read as \( (\Box A \land B) \lor C \) and not \( \Box A \land (B \lor C) \)
- The formula \( A \rightarrow B \rightarrow C \) should be read as \( (A \rightarrow B) \rightarrow C \) and not \( A \rightarrow (B \rightarrow C) \)
- \( \Box \Diamond \neg A \) should be read as \( \Box(\Diamond(\neg A)) \)
- etc
4.3.2 Stage 2: Testing the data structures

I have tested each data structure separately.

For formulas which have the structure of a binary tree test cases included getting sub-formulas and re-labelling formulas (which we need to do when we expand box and diamond formulas) and cloning formulas (which we need to do frequently since the algorithms need to backtrack at different stages).

In addition to a two dimensional array which holds the actual relation in the model so far the relation class also stores the frame properties. Test cases for this class included adding and removing relations (which the algorithm needs to do when satisfying diamond formulas), cloning relation objects and computing the closure of the relation matrix for different frame properties.

*Example:* Let the relations matrix $R$ initially be as given below. Assume the frame belongs to the class of transitive and symmetric frames.

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

Now, we need to compute the closure of the relation matrix to reflect the frame properties giving us the new relation matrix below.

\[
\begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

For assignments which are implemented as hash maps test cases included adding a new assignment (for example assigning the value *true* to the atom $a$ at world 1), checking assignments (for example checking whether the atom $b$ has been assigned *true*, *false* or neither at world 2). For instance, we should only be able to give an assignment to an atom at a world if one does not already exist.

4.3.3 Stage 3: Testing the model checker

The model checker implements the algorithm checkSAT which is given in the previous chapter. It takes a set of formulae, set of relations and assignments and checks the correctness of the model. This was the easiest part to test since it was
quite easy to see whether the model we are checking is correct or not. In addition I added several print statements at different stages in the methods making it easy to follow as it checks the satisfiability of each formula.

4.3.4 Stage 4: Testing the model generator without relations and frame conditions

At this stage the model generator was only given a set of labelled formulas (i.e. no relations or frame conditions). When it builds the models it outputs all the failed attempts as well as the final model (if one exists). I tested for cases where all the formulas are in one world and cases where formulas are in more than one world (more test results can be found in the appendix).

Example:
List of formulas $F = \{0:(\Diamond \Diamond p \land \neg p), 0: \neg \Diamond p\}$, List of relations $R = \emptyset$
List of frame conditions $C = \emptyset$

The algorithms first tries to add the relation $R(0,0)$ this fails because world ‘0’ now contains the formula $\Diamond p$ and its negation $\neg \Diamond p$. So it backtracks and adds the relation $R(0,1)$. So world ‘1’ now contains $\Diamond p$ and $\neg p$. All the formulas at world ‘0’ are now satisfied. And at world ‘1’ $\Diamond p$ is the only formula that is not yet satisfied. It tries to satisfy this by adding the relation $R(1,0)$ but this fails because world 1 already contains $\neg p$. Similarly for $R(1,1)$. So it adds a new world, world ‘3’. This time it succeeds. So the final model is as given by the diagram below followed by the output from the system in figure 4.6.1.
4.3.5 Stage 5: Testing the model generator with relations but no frame conditions

At this stage the model generator was given a set or relations as well as formulas. Again it outputs all the failed attempts as well as the final model. I took example of formulas that would have a model if no initial set of relations were given. I then
added relations that would stop them from being satisfiable.

Example:
List of formulas \( F = \{0; \bigBox(\bigDiamond p \land \bigBox \neg p)\} \)
List of relations \( R = \{ R(0,1) \} \)
List of frame conditions \( C = \emptyset \)
Diagrammatically the input is as given below:

All the formulas at world ‘0’ are satisfied. It tries to satisfy the formula at world ‘1’ by adding the relation \( R(1,0) \). But this means we have \( p \) and \( \neg p \) at world ‘0’. So this branch fails. It then backtracks and tries to adding \( R(1,1) \) which also fails. It then tries to introduce a new world, ‘2’ and adding the relation \( R(1,2) \). But this also fails since it gives \( p \) and \( \neg p \) both at world ‘2’. So it concludes that no model exists. The output from the system showing this process is given in figure 4.6.2 below.
4.3.6 Stage 6: Testing the model generator with relations and frame conditions

At this stage the model generator was given a list of formulas, relations and frame conditions. As part of the example formulas I took the negation of the frame condition axioms which are clearly not satisfiable. For example if we take the negation

\begin{verbatim}
failed attempt
0: p
0: !p
0: !p | !(!p)
1: cp ^ !(!p)
1: cp
1: !p

Relations
R(O,3)
R(1,0)


failed attempt
0: !p ^ !(!p)
1: p
1: !p
1: !p | !(!p)
1: cp
1: !(!p)

Relations
R(O,3)
R(1,0)


failed attempt
0: !p ^ !(!p)
1: cp ^ !(!p)
1: cp
1: !(!p)
2: p
2: !p

Relations
R(O,3)
R(1,0)

Not satisfiable
\end{verbatim}

fig 4.6.2
of the transitivity axiom \( \neg (\Box A \rightarrow \Box \Box A) \) this doesn’t have a model in the class of transitive frames. Part of the test result for this example is given by fig 4.6.3 below.

fig 4.6.3

More test results can be found in the appendix.
Chapter 5

Future work, Evaluation and Conclusion

5.1 Future work

Instead of limiting the user to five frame conditions, allowing them to enter any frame condition (expressed in first order logic) and being able to generate models that satisfy these frame conditions. Having spoken with my supervisors we decided that instead of focusing on making a good GUI it would be better to focus on investigating the different possibilities of how one might implement this. More specifically I investigated negated, conjunctive and disjunctive type frame condition and given below are my findings. I believe we could deal with a big group of frame conditions by using a combination of the approaches given here.

1. Negated frame conditions

One possible way of dealing with negative frame conditions (e.g. irreflexivity - \( \forall x (\neg R(x, x)) \)) and integrating it with the model generation algorithms is as follows:

1. As a first stage in the model generation check that the initial matrix satisfies the negative frame conditions. If it does then the algorithm return false and terminate

2. Based on the relations already present update the relation matrix so that its stated reflects what new relations are not allowed (this can be done for example by setting those entries in the matrix as (-1))
3. Whenever the algorithm needs to introduce a new relation it can only do so if the relation is ‘allowed’ based on the state of the matrix

4. Every time a new relation is introduced post-process the relation matrix so that it carries the information of what further relations are and are not allowed before going to the next step in the algorithm

Example:
Let the frame condition be

$$\forall x, y, z (R(x, y) \land R(y, z) \land x \neq y \land y \neq z \rightarrow \neg R(x, z))$$

and let the initial relation matrix be

$$\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}$$

The matrix above passes the pre-check done by step 1 (i.e. does not cause the algorithm to terminate) and then step 2 gives an the updated matrix below

$$\begin{pmatrix}
0 & 1 & -1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}$$

This means that the algorithm wont be able to introduce relations $R(0, 2)$ since this positions in the matrix is set to -1. Now, if the first relation introduced by the algorithm is $R(2, 3)$ resulting with the matrix

$$\begin{pmatrix}
0 & 1 & -1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix}$$

then we go to step 4 and the post-processed matrix will look like
and this step is repeated every time a new relation is introduced.

2. Disjunctive type frame conditions

This is the second possibility that I investigated: how one would deal with frame conditions of the form \( \forall x, y, z \ldots (R(x, y) \land R(y, z) \land \ldots \rightarrow R(y, x) \lor R(x, y) \lor \ldots) \).

One possible way is whenever the left hand side of the frame condition holds for a set of relations we could try making one of the disjuncts on the right hand side true so that the overall formula will evaluate to true. If we try to make the first one true and we fail to find a model we could try the second one and so on. This approach to dealing with could possibly work but, as illustrated by the example below, it might not very efficient.

Example:
Let the frame condition be
\( \forall x, y, z (R(x, y) \land R(y, z) \land x \neq y \land y \neq z \rightarrow R(x, z) \lor R(z, x)) \)
and let the initial relation matrix be
\[
\begin{pmatrix}
0 & 1 & -1 & 0 \\
0 & 0 & 1 & -1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

Now, for the above frame condition to hold we have to have either \( R(0, 2) \) or \( R(2, 0) \) leading to the following two matrices respectively.

\[
\begin{pmatrix}
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]  
(1)
This means we could try building a model for the first one and if that fails then we can try the second matrix. This branching will have to be repeated each time we introduce a new relation. However, the introduction of one new relation could lead to more than two branches (i.e. possible matrices that satisfy the frame condition). For example if we initially choose the first branch (1) and the new relation $R(2,3)$ is introduced then from the frame condition we have $(R(1,3) \vee R(3,1))$. Now consider the case where we make the first one true (i.e. we add the relation $R(1,3)$). Then since we have $R(0,1)$ we now have to have $R(0,3) \vee R(3,0)$ and so on. So we have at least three possible matrices to consider. As we add more and more relations we can see that the number of possibilities grows significantly.

3. Conjunctive type frame conditions
These are frame conditions of the form $\forall x, y, z... (R(x,y) \land R(y,z) \land ... \rightarrow R(y,x) \land R(x,z) \land ... )$. We could deal with these in the same way we deal with the Euclidean frame property which is:

- Make sure the initial relation matrix satisfies the frame conditions by add all the relations necessary.
- Every time a new relation is added by the algorithm also add all other relations implied by the frame conditions.

Example:
Let the frame conditions be
$\forall x, y, z (R(x,y) \land R(y,z) \land x \neq y \land y \neq z \rightarrow R(x,z) \land R(z,x))$
and let the initial relation matrix be as shown below:

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
Then to satisfy the frame condition we add the new relations $R(1,3)$ and $R(3,1)$ giving us the new relations matrix below

$$
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0
\end{pmatrix}
$$

I believe we could deal with many frame conditions by using a combination of the three approaches given in the sections above.

5.2 Evaluation

The idea of a possible world is implicitly present in modal satisfiability checkers. For example when we look at the recursive algorithm KSAT (given in section 2.2.2) the function $\text{KSAT}_W$ takes a formula $\varphi$ and returns assignment $\mu_1, ..., \mu_n$ that propositionally satisfy $\varphi$ (example also given in 2.2.2). $\text{KSAT}_A$ then takes one of the assignments $\mu_i = \{\square \alpha_1, ..., \square \alpha_N, \neg \square \beta_1, ..., \neg \square \beta_M, A_1, ..., \neg A_s\}$ and checks for modal satisfiability. It does this by checking whether the formula $\varphi^j = \alpha_i \land \neg \beta_j$ is satisfiable for every $\neg \square \beta_j$ in $\mu_i$. So in essence what it is doing is checking whether there exists a world accessible to the world containing $\varphi$ where $\neg \beta_j$ is satisfied and so on.

The algorithm proposed here helps to make this type of reasoning more intuitive and allows for flexibility by giving the user the options to start with a sort of half built model (initial set of formulae in different possible worlds, with some world already being accessible to others) and specify some frame properties. The implementation was done in a modular way to allow for easy implementation of the extensions proposed in the previous section. However, I believe the tool could have benefited a lot from a GUI.

Test results showed that the tool works quite well for generating small models but could be even more efficient if some optimization techniques are implemented. For instance when it branches on a disjunction it could make a wiser decision as to which branch it is going to try first. This could lead to less computation and might also lead to a smaller model being generated. Also, it could do some additional checks to detect propositional inconsistency in the set of formulae at a very early stage. For example, at the moment it checks if both a formula and its negation are present. If they are then it terminates returning false. However, it won't be able to detect that $w_1: \neg(\Diamond (b \rightarrow \square c) \land a \land b)$ and $w_1: (\Diamond (b \rightarrow \square c) \land b \land a)$ would lead...
to inconsistency on the initial check and it might have to do a lot of computation before it realizes that no model exists for these. I believe small modifications like this could improve efficiency significantly.

5.3 Conclusion

A stable model generation algorithm for propositional modal formulae has been developed and implemented. The incremental approach taken in the development of the algorithms proved to be very helpful. In addition to making the final algorithm stable and robust, it meant that implementation and testing could also be done incrementally. Initially one of the approaches considered for this project was translating all the modal formulae into first order logic and using a first order logic satisfiability checker. However, the hybrid representation used where formulae are expressed in modal logic and frame properties and relation are expressed in first order logic proved to be much simpler and much more intuitive. It also showed that LDS, which was mainly used in theorem proving in the past, can also be used for model generation successfully.
Bibliography


Appendix

For cases where a model was found I have only given the model. However, if a model does not exist then some of the attempts made to build a model are given.

Test case 1:
List of formulas $F = \{0: □(p \land \Diamond \neg p) \land p\}$
List of relations $R = \{R(0,1)\}$
List of frame conditions $C = \emptyset$
Test case 2:
List of formulas $F = \{0 \square (p \land \Diamond \neg p) \land p\}$
List of relations $R = \{R(0,1)\}$
List of frame conditions $C = \emptyset$

(same set of formulas and relations as test case 1 but now the frame is a transitive frame).
Test case 3:
List of formulas $F = \{0: \Box (a \land b), 0: \Diamond (b \lor \neg c), 1: \neg b \lor c, 1: \Box \Diamond a \rightarrow (b \land \Box c)\}$. 
List of relations $R = \{R(0,1)\}$ 
List of frame conditions $C = \emptyset$
**Test case 4:**

List of formulas $F = \{0: \neg (\square p \rightarrow p)\}$

List of relations $R = \emptyset$

List of frame conditions $C = T$
Test case 5:
List of formulas $F = \{ 0: \Diamond \Diamond p \land \neg p, 0: \neg \Diamond p, 1: \neg p \}$
List of relations $R = \emptyset$
List of frame conditions $C = \emptyset$
Test case 6:
List of formulas $F = \{ 0: \Box \Diamond \neg p , 0: \Box p \}$
List of relations $R = \emptyset$
List of frame conditions $C = \{ 4 \}$
Test case 7:
List of formulas $F = \{ \Diamond p \land \Box (\neg p \land q) \}$
List of relations $R = \{ R(0,1), R(0,2) \}$
List of frame conditions $C = \emptyset$