LONDON'S GLOBAL UNIVERSITY

Modal Logic Theorem Provers and Validity Rates

Sergio Christian Hernández Gutiérrez*

BSc Computer Science

Professor Robin Hirsch

April 18, 2019

*__Disclaimer:__ This report is submitted as part requirement for the BSc Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Contents

Abstract

The project hereby presented describes an implementation of a frame-based analytical tableau theorem prover for propositional modal logics K, KT, KB, K4, KD and linear modal logic, as well as any sensible combination of them. Additionally, a random modal formula generator has been implemented. Using both programs, the validity rates of propositional modal logics have been studied. This empirical study consisted on finding the percentage of valid formulas with respect to different modal logics.

1 Introduction

Propositional logic allows us to describe the world by using propositions, which have a value of true or false, and logical connectives, which can create semantical dependencies and relations between propositions. Modal logics can incorporate the semantics of necessity and possibility to propositional logic, augmenting its expressivity. We can express ideas about the world in such logics by writing formulas (such as, for example, $(p \vee \Diamond q)$, which can be interpreted as "p or possibly q"). We can also infer properties of such formulas, such as their truth or falsehood if we have knowledge of the truth value of the propositions, or their satisfiability and validity if we do not. A propositional formula is satisfiable if it is true under some valuation of its propositions, and valid if it is true under all possible valuations.

The initial focus of this project was to develop a theorem prover for propositional modal logics capable of determining the validity of propositional modal formulas. Such prover aims to serve as a tool for researchers

and students to conduct investigations on the subject and other kinds of work. After preliminary research was done on the theory of modal logics and existing proving methods for them, a theorem prover for the modal logic K was initially developed. Java was the language of choice; this decision is explained in section 3.1. The prover was implemented with the consideration that future logics could be added to the set of logics it recognises and can prove. During the course of the project, modal logics KT, KB, K4, KD, as well as linear modal logic were added in such a way.

The prover proves one array of formulas at a time, where the last formula given is the one to prove, and the other ones, if any, are considered axioms. The program reads the input, generated by the user in a *.txt file, and parses it to extract the arrays of formulas and a set of frame conditions. Next, it proves each formula by constructing a model based on the idea of logical tableaux and adapted to the requirements of modal logic. Finally, it outputs information about the parsing process and the validity of each formula and writes it to a *.txt file for the user to examine.

Most notably regarding the testing process, the prover's results were validated against another prover designed in the *Politecnico di Milano*, called Molle [1]. A random input generator (described in section 3.4) is able to quickly generate a large number of modal formulas with different parameters, such as their size. A wrapper to use Molle was developed to automatise the testing process, due to its graphical user interface proving to be inefficient for this task. Using the input generator, over a million formulas were inputted in both provers. This process helped to uncover bugs in Molle, by means of examining inconsistent outputs between the two provers using other proving methods. The developers of Molle were notified about these

instances and the bugs were acknowledged. By the results of this testing strategy, we obtained substantial evidence of the correctness of the theorem prover hereby presented.

Using the prover and input generator programs implemented during the project, an empirical study on validity rates was conducted in order to understand how particular properties of a logic affect the validity rate of formulas as their size tends to infinity. A program to perform the study was developed, with which the user can change a set of parameters such as the frame conditions applied to the logic, the connectives used and the maximum allowed number of propositions. The program runs a large batch of formulas, changing the size of such formulas at each iteration and measuring the validity rate after proving them.

This report will first present to the reader background knowledge of propositional modal logic, as well as of proving techniques used to identify valid formulas for modal logics and other types of logic. Next, it will describe the set of requirements postulated at the beginning of the project. It will then explain in detail the implementation of the programs engineered during the course of this project, as well as the testing and evaluation strategies followed to guarantee the correctness, resilience, and efficiency of such programs. A description of the methodology used in the study on validity rates will also be provided, as well as the findings obtained by it. Finally, the conclusions obtained from the project will be presented, as well as possible future works which could be done following the programs, ideas and methodologies described in this report. Additional appendices have also been included, providing datasets, manuals, code fragments of the programs, the project plan and the interim report.

2 Background and Context

There are several textbooks on the topic of modal logics that can provide a thorough overview of the subject. A specialised recommended text is *Modal Logic* [2], by P. Blackburn, M. de Rijke and Y. Venema. A less specialised book which can also provide insights into the application of modal logics, as well as other logics, to Computer Science is *Handbook of Logic in Computer Science: Volume 2* [3], by S. Abramsky, D. M. Gabbay and T. S. E. Maibaum. The book *Logics for Artificial Intelligence* [4], by R. Turner, presents the applications of modal and other logics to the field of Artificial Intelligence. The *Stanford Encyclopedia of Philosophy* has a section on modal logic [5], and it is a complete and accessible online resource on the topic. Many of the definitions and concepts presented in this report (particularly throughout this chapter) are based on the contents of these works.

A formula ϕ in propositional modal logic can be recursively defined as

$$\phi ::= p \mid \neg\phi \mid \Diamond\phi \mid \Box\phi \mid (\phi \circ \psi),$$

where $\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ and p is a proposition, an atomic logical structure with a binary value of true or false. A literal is a proposition or a negated proposition (e.g.: p and $\neg p$ are literals). The formula $\Diamond p$ can be interpreted semantically as "possibly p" and $\Box p$ can be interpreted as "necessarily p".

Modal logic allows for the interpretation of the concepts of necessity and possibility. Such philosophical concepts are represented by the notion that a necessary truth will be true in all possible realities or worlds derived from the current one, and a possible truth will be true in at least some such world. Thus, we can define a set of worlds $\mathcal{W}$ which represents the set of all relevant

realities. We can also define a relation between worlds $\mathcal{R} \subseteq (\mathcal{W} \times \mathcal{W})$, which can be best graphically understood as transitions between worlds, as illustrated in Figure 1. If $(\omega, \upsilon) \in \mathcal{R}$, we can write $\omega \mathcal{R} \upsilon$. With these definitions, we can define a modal frame as $\mathcal{F} = (\mathcal{W}, \mathcal{R})$. In propositional logic, we must establish a valuation $\mathcal{V}$ on the set of propositions $\mathcal{P}$, such that $\mathcal{V} : \mathcal{P} \mapsto \{\top, \bot\}$, where $\top$ represents truth and $\bot$ represents falsehood. However, in propositional modal logic, we can define $\mathcal{V} : \mathcal{L} \mapsto \wp(\mathcal{W})$, where $\mathcal{L}$ is a set of literals, so that a given valuation will define a labelling which determines which propositions of the logic hold true or false in which worlds. Finally, we can define a modal model as $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{V}) = (\mathcal{F}, \mathcal{V})$.

$$\begin{aligned} \mathcal{W} &= \{\omega_1, \omega_2, \omega_3\} \\ \mathcal{R} &= \{(\omega_1, \omega_2), (\omega_1, \omega_3), (\omega_2, \omega_2)\} \\ \mathcal{V} : p &\mapsto \{\omega_1, \omega_3\} \\ q &\mapsto \{\omega_2\} \\ \neg q &\mapsto \{\omega_3\} \end{aligned}$$

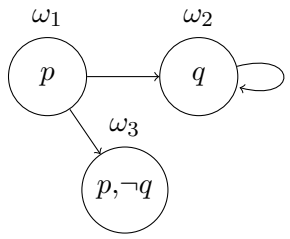

Figure 1: Graphical illustration of an arbitrary modal model.

If a formula ϕ is true at a world ω under a modal model $\mathcal{M}$, we write $\omega \models_{\mathcal{M}} \phi$. The truth values of the different types of propositional modal

formulas at ω under $\mathcal{M}$ are then defined as:

$$\begin{aligned}
\omega \models_{\mathcal{M}} p \quad &\iff \omega \in \mathcal{V}(p) \\
\omega \models_{\mathcal{M}} \neg\phi \quad &\iff \omega \not\models_{\mathcal{M}} \phi \\
\omega \models_{\mathcal{M}} \phi \vee \psi \quad &\iff \omega \models_{\mathcal{M}} \phi \text{ or } \omega \models_{\mathcal{M}} \psi \\
\omega \models_{\mathcal{M}} \Diamond\phi \quad &\iff \exists \upsilon \in \mathcal{W} : \omega \mathcal{R} \upsilon \text{ and } \upsilon \models_{\mathcal{M}} \phi \\
\omega \models_{\mathcal{M}} \Box\phi \quad &\iff \forall \upsilon \in \mathcal{W} : \omega \mathcal{R} \upsilon \text{ and } \upsilon \models_{\mathcal{M}} \phi
\end{aligned} \tag{1}$$

Two formulas ϕ and ψ are equivalent if and only if their truth value is the same under every possible modal model, and we write $\phi \equiv \psi$. We can use the equivalences of $(\phi \wedge \psi) \equiv \neg(\neg\phi \vee \neg\psi)$, $(\phi \rightarrow \psi) \equiv (\neg\phi \vee \psi)$, and $(\phi \leftrightarrow \psi) \equiv ((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$ to also define the truth of a formula under connectives $\{\wedge, \rightarrow, \leftrightarrow\}$. Also note that, by the definitions in (1), $\Diamond\phi \equiv \neg\Box\neg\phi$. We can now observe that the set $\{\neg, \vee, \Diamond\}$ is a minimal complete set of connectives for propositional modal logic. A minimal complete set of connectives is a subset of the set of all possible connectives which is able to express any formula in the logic it uses, without redundancy. For example, the set $\{\Diamond, \vee\}$ is not complete, as such connectives cannot express the formula $\neg p$; and the set $\{\Diamond, \vee, \neg, \rightarrow\}$ is not minimal, as the formula $(p \rightarrow q)$ can be expressed using just $\vee$ and $\neg$ as $(\neg p \vee q)$.

More generally, we can say that ϕ is true under $\mathcal{M}$ if and only if $\forall \omega \in \mathcal{W}$ such that $\mathcal{M}, \omega \models \phi$, and we can write $\mathcal{M} \models \phi$. Then, ϕ is satisfiable under a modal model $\mathcal{M} = (\mathcal{W}, \mathcal{R}, \mathcal{V})$ if and only if $\exists \omega \in \mathcal{W}$ such that $\mathcal{M}, \omega \models \phi$. For a modal frame $\mathcal{F}$, we can say that ϕ is satisfiable in $\mathcal{F}$ if and only if ϕ is satisfiable under some model $\mathcal{M} = (\mathcal{F}, \mathcal{V})$ (for some valuation $\mathcal{V}$); and that ϕ is valid in $\mathcal{F}$ if and only if ϕ is true under all models $\mathcal{M} = (\mathcal{F}, \mathcal{V})$ (for all possible valuations $\mathcal{V}$), and we can write $\mathcal{F} \models \phi$. For a class or set of frames $\mathcal{C}$,

we can say that ϕ is satisfiable in $\mathcal{C}$ if and only if ϕ is satisfiable in some frame of $\mathcal{C}$; similarly for validity, we can say that $\mathcal{C} \models \phi \iff \forall \mathcal{F} \in \mathcal{C} : \mathcal{F} \models \phi$.

Classes of frames in which all frames have $\mathcal{R}$ with the same formal properties define different modal logics, which are a set of valid modal formulas. Such properties are called frame conditions in the context of $\mathcal{R}$ being a modal transition relation. All relevant modal logics for this project are listed in Table 1, with the exception of linear modal logic. Other modal logics can be defined by obtaining the union of different logics, such as for the cases of S4 = KT4 and S5 = KTB4 (which results from an equivalence relation, therefore always producing a complete connected graph when a frame is represented graphically). Also, note that all propositional logic tautologies are additionally axioms of K, and that all modal logics other than K generally refer to their axioms as well as the axioms of K (so that we refer to 4 instead of K4, for example).

Frame Condition	**Modal Logic**	**Axioms**
	K (Kripke)	$(\Box(p \to q) \to (\Box p \to \Box q))$, $(\Diamond p \leftrightarrow \neg\Box\neg p)$
$\forall \omega \in \mathcal{W} : \omega \mathcal{R} \omega$	T (Reflexive)	$(\Box p \to p)$
$\forall \omega, \upsilon \in \mathcal{W} : (\omega \mathcal{R} \upsilon \Rightarrow \upsilon \mathcal{R} \omega)$	B (Symmetric)	$(p \to \Box\Diamond p)$
$\forall \omega, \upsilon, \mu \in \mathcal{W} :$ $((\omega \mathcal{R} \upsilon \wedge \upsilon \mathcal{R} \mu) \Rightarrow \omega \mathcal{R} \mu)$	4 (Transitive)	$(\Box p \to \Box\Box p)$
$\forall \omega \in \mathcal{W}, \exists \upsilon \in \mathcal{W} : \omega \mathcal{R} \upsilon$	D (Serial)	$(\Box p \to \Diamond p)$

Table 1: List of modal logics used in this project.

Let $\mathcal{C}$ be a class of frames and Σ be a set of modal axioms (that is, a set of modal formulas which are true). For some modal formula ϕ, we can write $\Sigma \vdash \phi$ if and only if we can prove ϕ is true using the axioms in Σ in addition to the rules of *modus ponens* $(\phi, (\phi \to \psi) \vdash \psi)$, *subsitution* (if $\models \phi$ then $\models \psi$ where ψ has been constructed by substituting any propositions in

ϕ by any modal formulas) and *generalisation* ($\phi \vdash \Box\phi$). Then, Σ is sound in $\mathcal{C}$ if and only if $\mathcal{C} \models \Sigma$; and Σ is complete in $\mathcal{C}$ if and only if, for any formula $\phi : (\mathcal{C} \models \phi \Rightarrow \Sigma \vdash \phi)$. The axioms presented in Table 1 are complete and sound in their respective classes.

Linear modal logic has the axiom

$$(\Box p \rightarrow \Box\Box p),$$

which is the transitive axiom, as well as the axiom

$$((\Diamond p \wedge \Diamond q) \rightarrow (\Diamond(p \wedge \Diamond q) \vee (\Diamond(\Diamond p \wedge q) \vee \Diamond(p \wedge q)))).$$

Such axioms are complete for linear modal logic. Nonetheless, linear modal logic is also antisymmetric. In a modal frame, if a world ω can be reached following any number of transitions from a world υ, then we can say that ω is in the future of υ. Then, antisymmetry requires that if, for any two worlds ω and υ, ω is in the future of υ, then υ is not in the future of ω, except for when $\omega = \upsilon$. A linear modal frame is also total, such that, for any two worlds ω and υ, either $\omega = \upsilon$, or ω is in the future of υ, or υ is in the future of ω.

ϕ is satisfiable or valid if and only if ϕ is satisfiable or valid on the class of all frames, respectively. If ϕ is valid, we write $\models \phi$. Throughout this report, the concept of validity will be contextualised depending on the frame conditions which apply in such context; for example, when describing an algorithm for the modal logic T (that is, on the class of modal frames which are reflexive) which can determine if a formula is valid, the validity of this formula will be on the class T.

A modal logic theorem prover aims at proving modal logic formulas or, in other words, at labelling modal formulas ϕ as valid or invalid. A rudimentary but unsustainable approach to do this, given a modal logic $\mathcal{C}$, would be to check that ϕ is valid in every frame in $\mathcal{C}$. However, this process may never terminate as the cardinality of $\mathcal{C}$ can be infinite (and such is the case for all of the aforementioned modal logics). A more sensible approach can make use of the relationship between validity and satisfiability: ϕ is valid if and only if $\neg\phi$ is unsatisfiable. Therefore, a theorem prover can work by attempting to build a modal model under which $\neg\phi$ is satisfiable. If such a model cannot be built, then $\neg\phi$ is unsatisfiable, and hence ϕ is valid; otherwise, ϕ is not valid.

The tableau method is a tree-based structure which is able to decide the satisfiability of logic formulas. Given that, for any formula ϕ, ϕ is valid if and only if $\neg\phi$ is unsatisfiable; we can also decide the validity of such formulas by negating them and constructing a tableau. A tableau is built using a set of expansion rules which are applied depending on the type of formula at a leaf of the tableau, a classification which is based on its connectives (the classification of formulas and rules used in this project is specified in section 3.3.2). The negation of the formula to prove is placed as the starting formula of the tableau, and its type is identified and mapped to one of the aforementioned expansion rules. Then, the expansion of the formula is made and, as a result, other formulas may appear deeper in the tableau. We can repeat this expansion process until only unexpandable or already expanded formulas remain in the tableau. Given its tree structure, we can define a closed tableau as one in which all of its branches are closed. A branch in a tableau is closed if a contradiction is found or, in other words, if there exist two formulas ϕ and ψ in the branch such that $\phi \equiv \neg\psi$. If a tableau

with a starting formula $\neg\phi$ is closed, then $\neg\phi$ is unsatisfiable, thus ϕ is valid. Otherwise, ϕ is not valid.

To illustrate the concepts above, we can use the example of propositional logic tableaux. A propositional logic tableau has two rules: an α-rule places all subformulas of the current formula below it in a linear fashion, without branching; a β-rule places each subformula of the current formula below it but in a different branch. In general, formulas which can be written as a conjunction or a negated disjunction are expanded using the α-rule, and formulas which can be expressed as a disjunction or a negated conjunction are handled by the β-rule. Literals are not expanded. With such rules, we can fully construct a tableau in order to decide the validity of the formula $\neg(p \wedge \neg(p \wedge \neg p))$, as shown in Figure 2. It can be observed that the left branch contains a contradiction (p and $\neg p$), hence it is closed; however, the right branch is open. Therefore, the root formula is satisfiable, which means that $\neg(p \wedge \neg(p \wedge \neg p))$ is not valid.

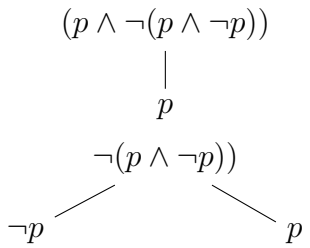

Figure 2: Propositional tableau deciding $\neg(p \wedge \neg(p \wedge \neg p))$.

While there exist several methods to prove modal formulas, tableaux have been described as the most used approach [6]. Early work to adjust analytical tableaux to modal logics was presented by Melvin Fitting in 1972 [7], describing tree-based tableau structures which support first-order and modal operators. A paper by L. Catach in 1991 [8] describes a proving method

adapting tableaux to modal models, which was able to prove different kinds of modal logics including temporal, epistemic and dynamic logics. Catach also describes an approach to perform loop-checking when creating transitive models similar to the one which was designed during the project. However, this paper was found in March of 2019, after my implementation had been designed, implemented and reported. A comprehensive list of theorem provers and other analytical tools for modal logics has been gathered by the *AiML* (*Advancements in Modal Logic*) group [9]. This collection was used to find Molle [1], a theorem prover used in combination with my prover to perform comparison testing.

3 Implementation

This section describes in detail the Java implementation of the propositional modal logic theorem prover presented in this project. First, the syntax and grammar which need to be followed by the user will be given. This specification allows the reader to understand the implementation of the parser, which works recursively on each formula and builds *Formula* Java objects which also hold information about their type. Then, the functioning of the prover will be explained. The prover is based on the tableau method, but adapted to be constructed using a queue of labelled frames, which can be represented as labelled graphs. Nodes in a graph represent worlds, which can be connected by transitions, and are labelled by formulas. The prover follows a set of rules which define the steps to take when expanding each type of formula; these rules are applied following a well-defined and fair schedule. The steps of these rules and other algorithms of the prover can change depending on the frame conditions inputted by the user. Finally, the generation process of random formulas will be explained in detail; this process is used for testing purposes, as well as for the study on validity presented in section 5.

3.1 Choice of Language

The language used to write the theorem prover was Java. Many factors were considered when choosing this language as the main one for the project: previous personal experience with the language, portability and accessibility, scalability, etc. However, the main aspect that was key to this decision was the idea of developing the theorem prover as an Object Oriented program. This idea is based on the great flexibility that modal logic has; there

are many kinds of modal logics which need additional algorithms and rules to be dealt with when proving formulas. Nonetheless, all modal logics rely on the same core concepts. Hence, the ease of extensibility OOP (Object Oriented Programming) presents without losing cohesion and adding unnecessary complexity was an interesting idea to explore. The Python language was also considered due to its fast developing process and OOP support, but its lower reliability at runtime was a factor for its dismissal. Another language which was studied was C, specially due to its flexibility in memory management (which becomes a significant feature when dealing with large formulas) and fast runtime, but it was dismissed because of its limited scalability and lack of modularity.

3.2 Syntax, Grammar and the Parser

3.2.1 Syntax and Grammar

The first step towards building a theorem prover is defining a language by which a user can express an idea in the form of a logic formula. By using the rules and concepts of propositional modal logic, a set of syntactical and grammatical rules that allowed the user to express any propositional modal formula and the theorem prover to unequivocally recognise such formula could be constructed. Some key mechanisms to eliminate ambiguity in the formulas are the usage of parenthesis (as defined in the syntax specification within the user manual, given in Appendix D), as well as the irrelevance of spaces, tabs and newlines. In the input file, the user can firstly introduce the modal logic that the formulas should be interpreted by. Next, the array of formulas are given one by one, separated by semicolons (and each formula

in an array of formulas is separated by a comma). Each array of formulas is interpreted such that the last formula given is the one to prove, and the rest are axioms.

The grammar of the logic language recognised by the prover includes the connectives of $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \Diamond, \Box\}$. Such connectives are semantically defined as explained in section 2. Propositions can be written as any string of characters, although with restrictions on characters used for other purposes throughout the program in order to avoid confusion (for example, a proposition cannot have & as any of its symbols, since this is a reserved symbol used for conjunction; the complete set of restrictions is provided in the user manual in Appendix D). In general, propositions in logic formulas are written as lowercase letters, but the decision of allowing longer strings was made for scalability (so that the set of possible propositions is infinite) and expressive power, given that real-world applications of the program will give each proposition a meaning, which in this way can be expressed as a word (and could not be expressed as well with just a single letter). Other non-ASCII symbols pertaining to modal logic, such as the necessity and possibility symbols ($\Box$ and $\Diamond$), are given an alternative set of symbols to be expressed with (in the example given, [] and $\langle\rangle$ respectively), improving the accessibility of the program. Also, note that there are two symbols which have been included in the set of reserved symbols in the program: T and F (which represent $\top$ and $\bot$, respectively). They were included for convenience, and they will be further used when the implementation of serial frames is discussed in section 3.3.4.

3.2.2 Parsing Process

The input file is parsed sequentially. First, the specified set of frame conditions is parsed, and a $ModalLogic$ object is created with the substring extracted from the input file. This object parses the string, stores the set of frame conditions such string expresses and is able to answer questions such as "is the logic reflexive?" or "is the logic transitive?". A $ModalLogic$ object is also able to flag the existence of incompatible frame conditions or the lack of required ones by throwing appropriate exceptions; for example, linear frames cannot be symmetric but must be transitive.

Next, the parser creates a $FormulaArray$ object for each array of formulas separated by commas and delimited by a semicolon. Each $FormulaArray$ object is a collection of formulas and relies on $Formula$ objects for its functioning. Each $Formula$ object parses the string given to initialise it, and it does so recursively. At each step of the recursion process, it looks for a connective and extracts its subformulas accordingly. Each subformula is used to create a further $Formula$ object which will be stored as a subformula in the original $Formula$ object. Each subformula is also parsed recursively, and the process stops (i.e.: the base case is reached) once a proposition is found. This process clearly terminates given that the given formula is of finite size. During the parsing process, multiple negations will also be reduced to single or no negations (for example, $\neg\neg p \equiv p$). Once the parsing process is finished, each $Formula$ object will be composed of its string, a set of subformulas, and a $FormulaType$ (this is a Java Enum Type which represents the types of modal logic formulas, such as propositions, negated implications, etc.). Other attributes of $Formula$ objects will be introduced in this report when they become relevant for the proving process.

3.3 A Tableau Method for Propositional Modal Logic

3.3.1 A Tableau Based on Labelled Frames

The algorithm used to prove a modal formula was devised using the core ideas of more general analytic tableaux. An outline of its design is given in Algorithm 1. The algorithm places the negation of the formula to prove as the starting formula in the tableau-based structure, as well as any given additional axioms. Then, by expanding all formulas using specific rules (which will be presented later in section 3.3.2), it builds all possible logical instances that the negated formula and the given axioms present, and looks for contradictions in such instances. If all instances are contradictory, then the negated formula must be unsatisfiable, hence the formula to prove is valid; otherwise, the formula to prove is not valid. In a regular tableau, such instances are represented as branches. The algorithm builds modal labelled frames instead of branches. A branching rule in a regular tableau (in other words, a β-rule) is represented in the algorithm as a labelled frame which is cloned and modified according to the rule, where cloning a labelled frame means creating a different labelled frame with the same worlds and transitions. A non-branching rule in a regular tableau (in other words, an α-rule) is represented in the algorithm as new formulas added to a given world of the same modal labelled frame.

This algorithm benefits from the choice of programming paradigm for this project. OOP allows different *Frame* objects to be built and kept track of. These *Frame* objects are stored as a queue in a *Tableau* object, which orchestrates the expansion process until a conclusion is reached on the validity of the formula to prove. In the case of an array of formulas being supplied to

the *Tableau* object, it places all axioms as well as the negation of the formula to prove in an initial *Frame* object (which consists of a single *World* object holding these formulas). Each *Frame* object can have many *World* objects within it, as well as *Transition* objects which define the relation between the set of worlds in the labelled frame. Each *World* object holds *Formula* objects, and is able to recognise contradictions amongst these formulas.

On each step of the process, a *Frame* object is chosen to be expanded once, and it in turn chooses one of its *World* objects to expand one of its formulas. If a formula has no possibilities of being expanded in the future (in general, because it has been already expanded), then it will be ticked; ticked formulas are not considered when searching for a formula to be expanded. After each step is completed, the Tableau checks if the chosen *Frame* object has introduced any contradiction, and if so it removes it from the set of labelled frames (since expansions can introduce contradictions but never remove them, this avoids contradictory frames to be expanded further, which would unnecessarily consume time and memory space). If the chosen *Frame* object has no formulas in any of its worlds that can be expanded (i.e.: unticked formulas), then it notifies the *Tableau* object. If a given labelled frame is not able to be expanded further and it does not have any contradiction within it, then we can conclude that the negated initial negated formula is satisfiable, and thus the formula to prove is not valid, and finish the tableau expansion process. This check allows the program to save time and memory space by not continuing to expand the tableau further if a satisfiable labelled frame (one which does not contain any contradiction) is found. The other case where the tableau expansion process halts is if the set of labelled frames in the tableau is empty, so that no labelled frame could be expanded in a future expansion step. The set of labelled frames being empty means that all

possible labelled frames that were built during the tableau expansion process contained some contradiction. If such is the case, we can declare the initial negated formula unsatisfiable, and thus the formula to prove is valid. Given an input file containing many sets of formulas, one *Tableau* object is built and run for each set, and the results provided by each tableau are collected and expressed in the output file.

Algorithm 1 Tableau

```
1:  function TABLEAU(axioms, φ)
2:      formulas = axioms ∪ ¬φ
3:      frames = { Frame(formulas) }                ▷ frames is a queue
4:      while not frames.empty() do
5:          frame = frames.get()
6:          frame.expandOnce()
7:          if frame.hasContradiction() then
8:              frames.remove()
9:          else
10:             if not frame.isExpandable() then
11:                 return not valid
12:             frames.put(frames.remove())          ▷ Shuffle frames
13:     return valid
```

3.3.2 Expansion Rules

There are four main expansion rules that are used when expanding the tableau. The choice of rule is made depending on the type of formula that is to be expanded at each relevant step of the process. We can therefore map each type of formula, depending on its main set of connectives, to an expansion rule. Let ϕ and ψ be any two propositional modal formulas, and let them be referred to as the expansion subformulas. All possible propositional modal logic formulas can be expressed as one of the formulas present on the right column of Table 2. Also, no propositional modal logic formula

is equivalent to another formula of different type. Then, we can assign one expansion rule to all propositional modal logic formulas, such that the mapping from the set of expansion rules to the set of propositional modal logic formulas is surjective. A group of formulas will be henceforth referred to using the expansion rule to which they conform. For example, the formula $(\phi \wedge \psi)$ can be referred to as an α-formula, since it conforms to the α-rule.

α-formulas	$(\phi \wedge \psi)$ $\neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$ $\neg(\phi \rightarrow \psi) \equiv (\phi \wedge \neg\psi)$ $(\phi \leftrightarrow \psi) \equiv ((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$
β-formulas	$(\phi \vee \psi)$ $\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi)$ $(\phi \rightarrow \psi) \equiv (\neg\phi \vee \psi)$ $\neg(\phi \leftrightarrow \psi) \equiv (\neg(\phi \rightarrow \psi) \vee \neg(\psi \rightarrow \phi))$
δ-formulas	$\Diamond\phi$ $\neg\Box\phi \equiv \Diamond\neg\phi$
γ-formulas	$\Box\phi$ $\neg\Diamond\phi \equiv \Box\neg\phi$

Table 2: Types of propositional modal formulas.

α-rule

Given a formula σ in a world ω, and with two expansion subformulas ϕ and ψ, an α-rule will tick σ as expanded in ω and place ϕ and ψ unticked in ω as well. An α-expansion is illustrated in Figure 3, in which a rectangle represents a labelled frame, a circular node represents a world with formulas within it, a ✓before a formula means the formula has been ticked, and an arrow from one world to another represents a transition. The arrow which is labeled with a type of expansion represents a step of the proving process,

where the left hand side represents the state of the relevant tableau before the given expansion, and the right hand side represents the state of the tableau after such expansion.

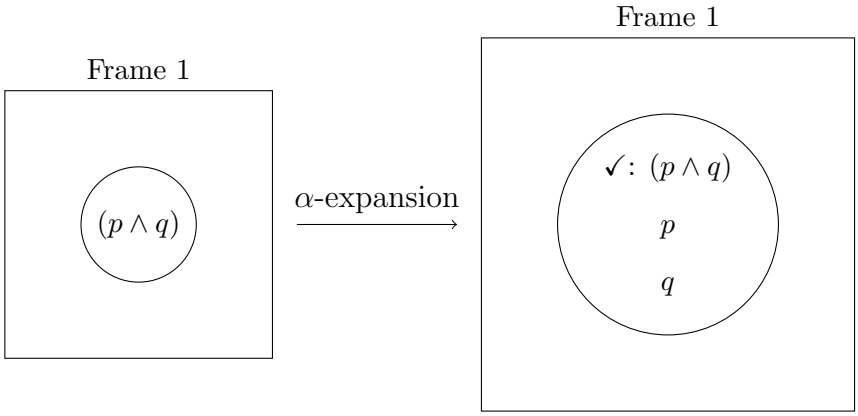

Figure 3: α-expansion.

β-rule

Given a formula σ in a labelled frame τ and a world ω, and with two expansion subformulas ϕ and ψ, a β-rule will tick σ as expanded in ω, then will clone τ to create a labelled frame τ' (let ω' be the cloned world ω in τ'), and finally place ϕ unticked in ω and ψ unticked in ω'. τ' will also be added to the queue of labelled frames in the tableau. A β-expansion is illustrated in Figure 4.

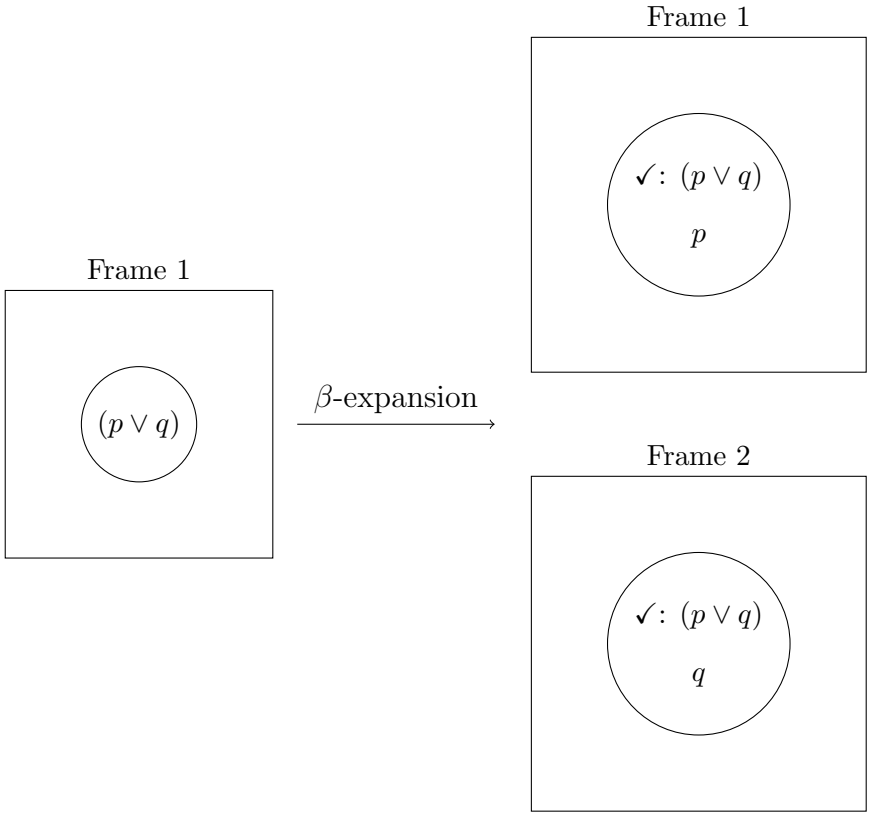

Figure 4: β-expansion.

δ-rule

Given a formula σ in a labelled frame τ and a world ω, and with an expansion subformula ϕ, a δ-rule will attempt to find an existing world υ in τ such that, given that a transition from ω to υ was created, the resulting labelled frame conforms to the prevalent frame conditions and where ϕ nor any other expansion of ω would introduce a contradiction in υ. In order to choose υ as the world to expand the δ-formula to, the algorithm requires υ to already contain a given set of formulas Σ, ticked or unticked. In order to understand why this technique can prevent future contradictions, the expansion schedule

must be introduced, so a more detailed explanation is given in section 3.3.3. For the modal logic K, Σ contains ϕ, as well as the expansion subformulas of any γ-formulas in ω. By including the expansion subformulas of γ-formulas in Σ, the algorithm is effectively expanding the γ-formulas; this can be seen as an optimisation of the expansion process, which prevents having to expand many γ-formulas individually in the future. Note that γ-expansions may still need to occur individually in some instances, and will be carried out by the prover following the schedule presented in section 3.3.3. Other frame conditions can expand the definition of Σ by adding more formulas to it; this will be further explained for each case in section 3.3.4. If υ is found, then ϕ will be placed in υ. If no such world υ is found, then ϕ will be placed in a new world and a transition from ω to such new world will be created. Finally, σ will be ticked in ω. This choosing algorithm is specially affected by different frame conditions, which change its behaviour. This will be discussed in detail in section 3.3.4. The three main cases of δ-expansions are illustrated in Figures 5, 6 and 7: where a looping transition from and to ω is made, a transition between ω and some other existing world υ is made, and a new world is created, respectively. Note that, in Figure 6, the labelled frame presented previous to the expansion is in fact not a possible labelled frame resulting from the prover's algorithm; such example serves only as a simpler illustration to more complex real cases.

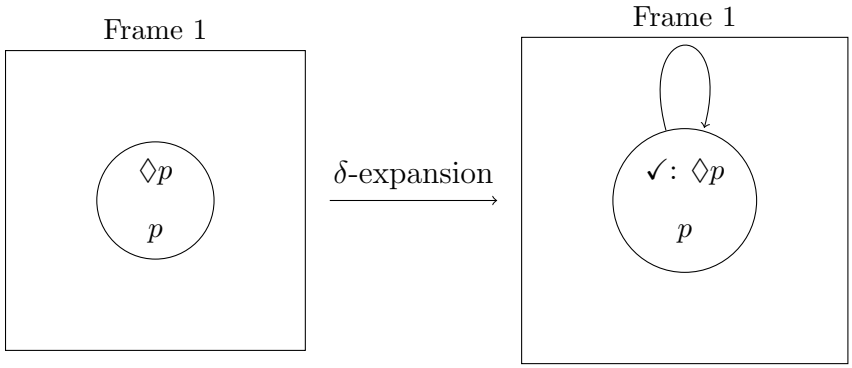

Figure 5: δ-expansion making a looping transition.

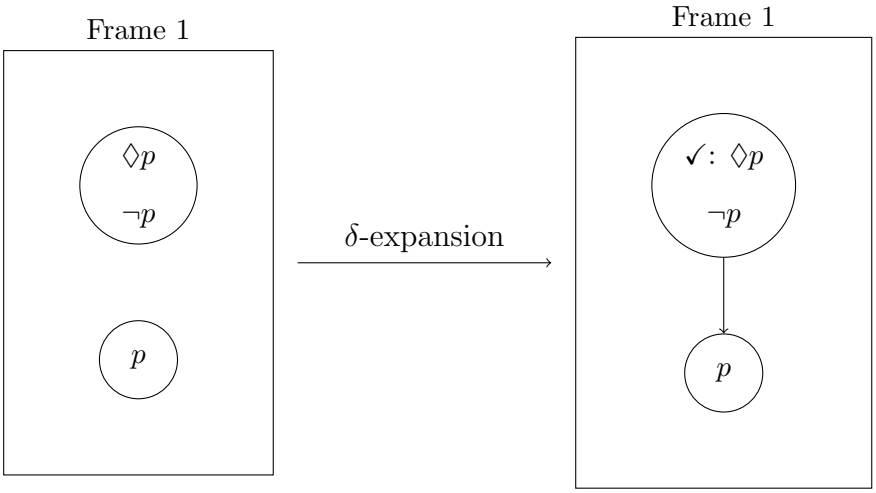

Figure 6: δ-expansion making a transition to an existing world.

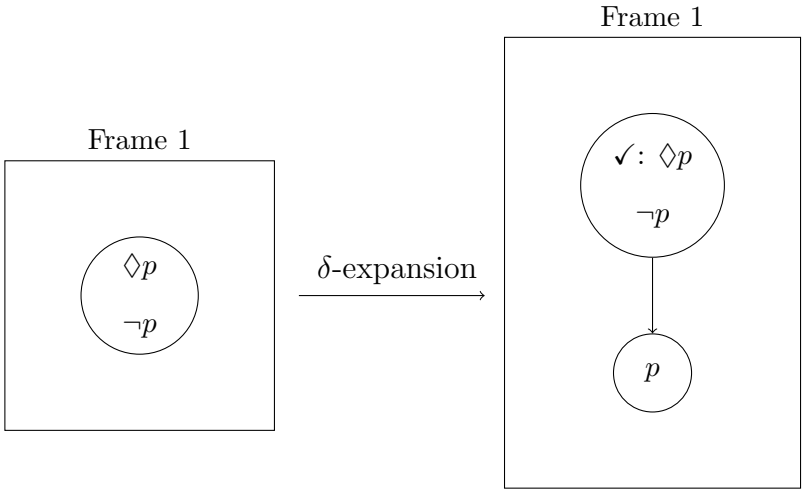

Figure 7: δ-expansion creating a new world.

γ-rule

Given a formula σ in a labelled frame τ and a world ω, and with an expansion subformula ϕ, a γ-rule will find all worlds υ in τ such that there exists a transition from ω to υ, and will place ϕ in all such worlds υ. σ cannot be ticked immediately after being expanded, because a future introduction of a new world in τ to which ω transitions would mean that σ could be expanded again. The process used to tick γ-formulas is based on the assumption that if the only unticked formulas remaining in the labelled frame are γ-formulas and all such γ-formulas are currently fully expanded (for a γ-formula σ in a world ω, if all transitions ω has are to some other worlds υ and μ, and σ has been already expanded to υ and μ, then σ is currently fully expanded), then there is no possibility of introducing new worlds or transitions in the labelled frame, and hence we can tick all γ-formulas in τ. If a γ-formula σ in a world ω within a labelled frame τ is found to be not fully expanded during the proving process, then all γ-formulas in τ will be unticked, as further expan-

sions of the remaining formulas in τ may introduce new transitions between worlds, opening the possibility that currently fully expanded γ-formulas may be expandable in the future. A γ-expansion is illustrated in Figure 8.

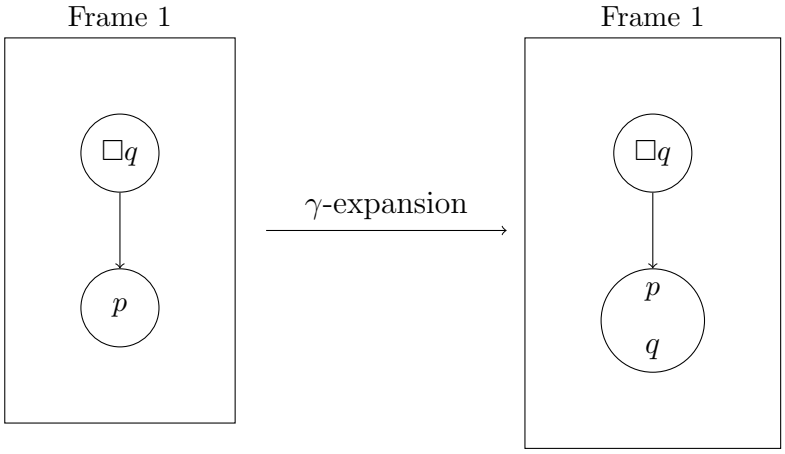

Figure 8: γ-expansion.

3.3.3 Fair Scheduling

In order to make one expansion step of the tableau, the program needs to choose a labelled frame, a world within such frame and a formula within such world to expand. In this section, it will be shown how this choosing process works, as well as why it is designed in such way. The first step in expanding the tableau once is choosing a labelled frame to expand. *Frame* objects are stored in a queue within *Tableau* objects. Using this data structure, we can create a fair schedule for choosing labelled frames. We always choose the *Frame* object at the front of the queue and, upon using it to make one expansion step of the tableau, we retrieve it from the queue and place it at the back of it. When we add a new labelled frame to the queue (for example, via

a β-rule expansion), we also place it at the back of it. Fair scheduling gives all labelled frames in the tableau equal opportunities of being expanded. This is especially important when considering potentially infinite frames, which could be introduced in future implementations of first-order modal logics and other modal logics. If there exists an infinite frame (meaning it can be expanded infinitely) generated by the program when attempting to prove a formula σ, and σ is satisfiable by a finite frame, by using a fair schedule we can always build this finite frame and conclude the satisfiability of the starting formula. However, if we use an unfair schedule, such as always expanding the same frame until it is fully expanded and only then expanding the next frame, then it is possible that an infinite frame was chosen and the conclusion of satisfiability of the starting formula was never reached.

The process of choosing a world to expand in a labelled frame is similar to that of choosing a frame to expand in a tableau. $World$ objects are stored in a queue within the $Frame$ object they belong to. The front of the queue is always chosen to make one expansion step, and once such step is finished, the front $World$ object is retrieved from the queue and placed at the end of it. If a new world is added to a labelled frame, the corresponding $World$ object is placed at the end of the queue within the relevant $Frame$ object. This also creates a fair schedule when choosing worlds.

$Formula$ objects are also stored in queues within $World$ objects. However, the process of choosing a formula during the expansion process is more complex than that of choosing a frame or a world. In order to choose a formula within a world, all $Formula$ objects within the chosen $World$ object are iterated through and chosen based on a hierarchy. This hierarchy defines which formulas should be expanded first depending on the expansion rules

that apply to them. First, all α formulas are expanded, then β-formulas, next δ-formulas and finally γ-formulas. Illustratively, this means that before expanding any β-formulas in some world, all α-formulas currently existing in such world must be ticked.

Expanding α-formulas before β-formulas potentially reduces exponentially the number of expansion steps (therefore reducing time and memory space used). This is because each β-formula doubles the amount of labelled frames available to expand. Hence, if there are x α-formulas and y β-formulas in the frame (ignoring the subformulas generated on each expansion, thus only considering expansions of the initial formulas without loss of generality), then expanding α-formulas first means that $x + y$ expansion steps are made, while expanding β-formulas first means that $2^y x + y$ steps are made.

δ-formulas are expanded after α- and β-formulas. This is because the δ-rule needs to identify future contradictions arising from expanding the pertinent δ-formula to a given world. In order to do this, the δ-rule algorithm requires the world it has chosen to already contain a given set of formulas, ticked or unticked (the definition of this set depends on the frame conditions which apply), such that if all of the formulas in such set were already present and fully expanded in the world (because of them being literals; or otherwise by α- or β-expansions), then any possible contradiction which could be introduced by the currently processed δ-expansion would have already existed and been found in such world. Therefore, we want to make sure that a suitable world is not overlooked by the δ-rule algorithm because all of its α- and β-formulas have not yet been expanded, possibly introducing in the future the required set of formulas the algorithm needed to choose a given world. Also, we want to make sure that checking if such a set of formulas is already

in a given world guarantees that no new contradiction is introduced by the δ-expansion, and for that we need to expand all α- and β-formulas available first so that no new contradiction is introduced by a future such expansion.

γ-formulas are expanded after δ-formulas, due to this schedule producing a lower number of expansions on average. The reason for this optimisation is that, if a world ω contains a set of δ-formulas Σ and a γ-formula σ, and given that each δ-expansion can potentially introduce a new world to which ω can transition, then expanding γ-formulas before δ-formulas means that it may be needed to perform one γ-expansion after each δ-expansion, such that $|\Sigma|$ γ-expansions of σ could be needed. However, if the algorithm prioritises the δ-rule over the γ-rule, then all necessary transitions introduced by a δ-expansion involving ω will be created before σ is expanded, and so σ will only need to be expanded once. When the algorithm used to handle transitivity as a frame condition is introduced later on section 3.3.4, it will also be shown that the number of γ-expansions can be further optimised by expanding γ-formulas during δ-expansions.

The schedule described in this section chooses only formulas which are not currently ticked for the program to expand them at each step of the process. *Formula* objects which have been expanded are ticked and never deleted from the *World* objects they belong to. This is done in order to allow the δ-rule to be able to compare the formulas present within different worlds, regardless of whether they have been already expanded or not, so that it is simpler to implement the algorithm designed to find prospective contradictions arising from δ-expansions described in section 3.3.2. If at some step of the process no formula is chosen by the schedule for some labelled frame τ, meaning all formulas in τ are ticked, and there is no contradiction

found at this step (nor at any other step, given that if a contradiction is found in some labelled frame, then no further expansion steps of such frame are made), then the program can conclude that τ satisfies the negated formula, hence the formula to prove is not valid. Note that we can safely tick all γ-formulas in some labelled frame τ at some step t given that none of them are currently expandable at t, since in order to arrive at such situation all α-, β- and δ-formulas in τ must have been ticked following the specification of the schedule, hence no possible future expansions of τ resulting in new transitions between worlds are possible.

3.3.4 Frame Conditions

Reflexive Frames

T is the symbol used to enforce reflexivity when proving a formula, and it can be added to the modal logic specified at the beginning of the input file. A reflexive modal labelled frame τ requires $\omega \mathcal{R} \omega, \forall \omega \in \tau$, where the relation $\mathcal{R}$ is the transition relation between worlds. In other words, all worlds in a reflexive modal labelled frame must be accessible from themselves via a looping transition. The algorithm used to enforce reflexivity adds a transition from ω to ω whenever such a world ω is created in a any frame. The creation of worlds in a labelled frame can be a result of the initial creation of the frame by initialising a *Frame* object, in which an initial world is created; or by a δ-expansion, which can introduce a new world to the relevant frame.

Symmetric Frames

The symbol B can be added to the modal logic specified at the beginning of the input file to enforce symmetry when proving the formulas provided. A symmetric modal labelled frame τ requires, for any two worlds $\omega, \upsilon \in \tau : \omega \mathcal{R} \upsilon \Rightarrow \upsilon \mathcal{R} \omega$. In other words, symmetry in a modal labelled frame enforces a symmetric transition relation between worlds in the frame. The algorithm used to enforce symmetry creates an inverse transition for any transition created in a given labelled frame; for example, if at some point in the proving process, a transition $\omega \mathcal{R} \upsilon$ is created, the symmetry algorithm will also create the transition $\upsilon \mathcal{R} \omega$. Similarly to creating a new world, creating a new transition can only happen when a δ-expansion takes place.

Transitive Frames

To enforce transitivity in the prover, one can add the symbol 4 at the beginning of the input file. A transitive modal labelled frame τ requires, for all worlds $\omega, \upsilon, \mu \in \tau : (\omega \mathcal{R} \upsilon \wedge \upsilon \mathcal{R} \mu) \Rightarrow \upsilon \mathcal{R} \mu$. The implications of this requirement affect more than just transition creation. Let, for example, the starting formula of the prover be $(\Diamond \mathrm{p} \wedge \Box \Diamond \mathrm{p})$. Enforcing transitivity while using the simplest algorithm for δ-expansions (and the first one implemented during this project), where a δ-formula $\Diamond \sigma$ in a world ω always results in a new world υ with σ and a transition $\omega \mathcal{R} \upsilon$, derives in the process shown in Figure 8 being carried out by the prover, where each labeled step is one expansion of the tableau and the first step (an α-expansion) has been omitted.

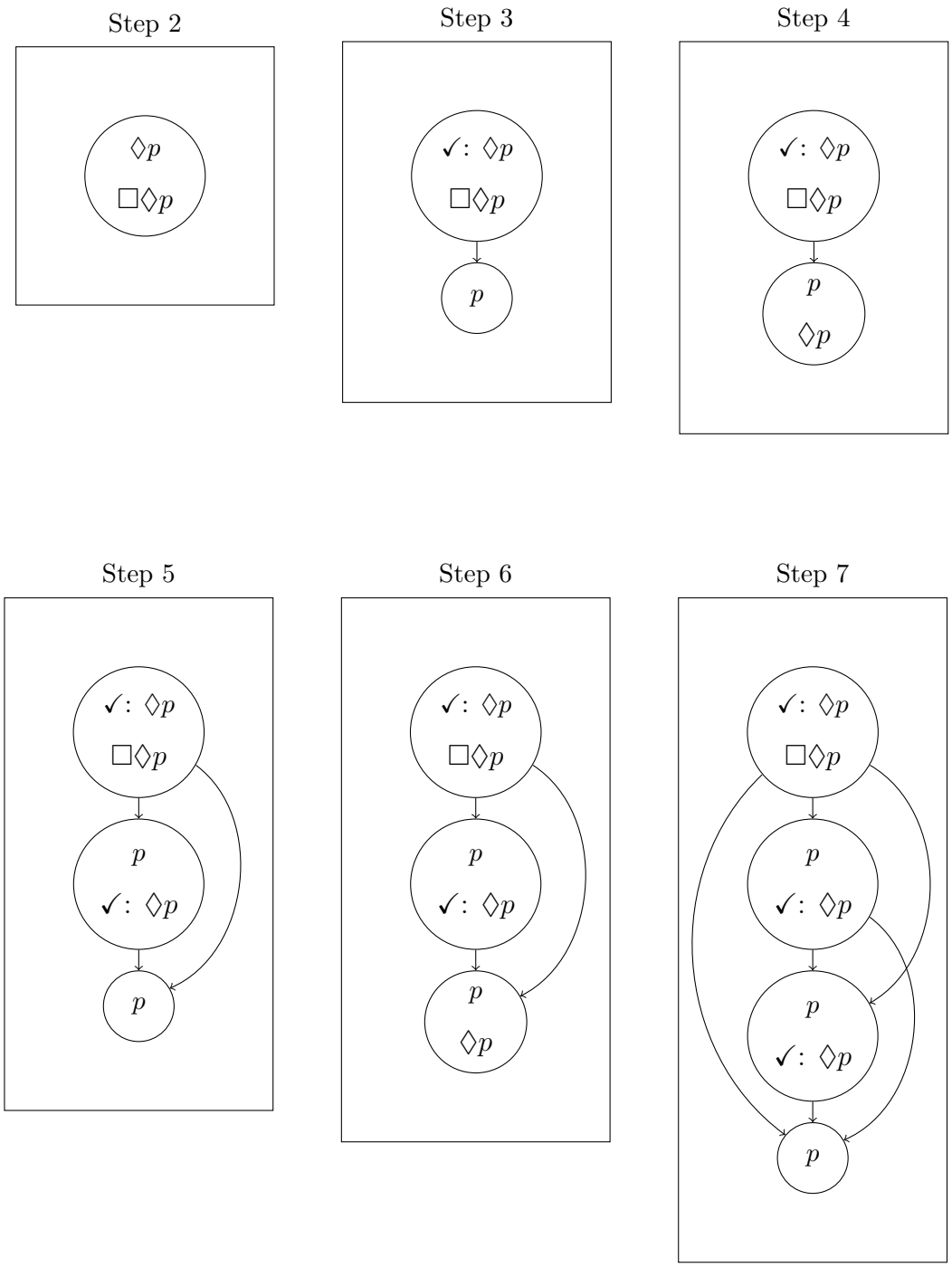

Figure 9: Creation of an infinite frame satisfying $(\Diamond p \wedge \Box\Diamond p)$.

At this point, one can infer that the process will not terminate. However,

the starting formula is indeed satisfiable by, for example, the finite labelled frame illustrated in Figure 10.

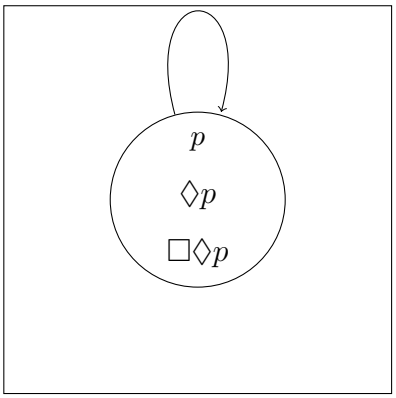

Figure 10: Finite frame satisfying $(\Diamond p \wedge \Box\Diamond p)$.

In order to find such labelled frame, a more general algorithm is needed. This is the reason for the development of the algorithm described in section 3.3.2. As mentioned in such section, the algorithm chooses a set of formulas Σ which any world to be chosen for the δ-formula expansion must contain, and Σ depends on the frame conditions applied. Let $\Diamond\sigma$ in some world ω be the δ-formula to be expanded. Then, in a transitive labelled frame, Σ contains σ, as well as all γ-formulas in ω and their expansion subformulas. Also including γ-formulas in Σ enforces transitivity by making use of an equivalent definition of it, where in a transitive labelled frame τ, it is the case that $\omega, \upsilon, \mu \in \tau : (\omega\mathcal{R}\upsilon \wedge \upsilon\mathcal{R}\mu) \Rightarrow \upsilon\mathcal{R}\mu$ is equivalent to $\omega, \upsilon \in \tau : (\Box\sigma \in \omega \wedge \omega\mathcal{R}\upsilon) \Rightarrow \Box\sigma \in \upsilon$, where the transitions generated due to transitivity are implicit.

Serial Frames

The symbol D can be added to the modal logic string at the beginning of an input file to enforce seriality in labelled frames. A serial modal labelled frame τ requires that $\forall\omega\exists\upsilon \in \tau : \omega\mathcal{R}\upsilon$. In other words, all worlds in τ must transition to another world in τ. In order to implement seriality, two new concepts were added to the prover: truth ($\top$) and falsehood ($\bot$); they are represented within the syntax available to the user as "T" and "F", respectively. $\top$ is equivalent to any valid formula, such as $(p \vee \neg p)$, and $\bot$ is equivalent to any unsatisfiable formula, such as $(p \wedge \neg p)$. Then, we can design the following equivalent definition of seriality: $\forall\omega \in \tau : \Diamond\top \in \omega$. Using this definition, we can define the algorithm to enforce seriality as the one which includes the formula $\Diamond\top$ in all worlds.

Linear Frames

In order to enforce linearity, one can include the symbol L in the modal logic string at the beginning of the input file. Let $\omega\mathcal{R}_n\upsilon$ mean that the world υ is reachable from the world ω after n transitions (note that, if transitive transitions were explicit, then $\forall n : \mathcal{R}_n = \mathcal{R}$ in transitive frames). If $\exists n \in \mathbb{N}$ such that $\omega\mathcal{R}_n\upsilon$, we say that υ is in the future of ω. Then, a linear frame is transitive and anti-symmetric, such that if a world ω is in the future of a world υ, it cannot also be the case that υ is in the future of ω, unless $\omega = \upsilon$. A linear frame is also total, such that, for any two worlds ω and υ, either ω is in the future υ, υ is in the future of ω, or $\omega = \upsilon$. Since linear frames are transitive and given that transitive transitions are implicit in the prover, a linear frame therefore can be represented as points in a directed line segment, where the points are the worlds in the frame and the directed line segment

can be constructed as the union of all the transitions between such worlds.

The algorithm which handles linearity changes the way δ-formulas are expanded. The procedure for a δ-expansion in a linear labelled frame is defined in Algorithm 2. Since linear labelled frames are also transitive, the set of formulas Σ used in such a δ-expansion is the same as the one defined in section 3.3.4, which is what the function $getTransitiveSigma(formula)$ returns. The function $getFutureWorldContaining(\Sigma, \omega)$ returns a world υ, such that $\exists n : (\omega \mathcal{R}_n \upsilon \wedge \Sigma \in \upsilon)$ (if no such world is found, then *null* is returned). The function $getPosition(\Sigma, \omega)$ returns a pair of worlds (υ, μ) such that υ and μ are in the future of ω and any γ-expansion from υ to ω or from ω to μ would not introduce a contradiction; if no such pair of worlds is found, the last world in the line segment representation of the frame is returned as υ and μ will be *null*.

Algorithm 2 Linear δ-expansion

```
1:  procedure LINEARDELTAEXPANSION(ω, φ)
2:      Σ = ω.getTransitiveSigma(φ)
3:      if ¬∃υ ∈ W : ∃n(ωR_n υ ∧ Σ ⊆ υ) then
4:          υ = World(Σ)
5:          findPosition(ω, υ)
6:          transitions.add({μ, υ})
7:          if χ ≠ null then
8:              transitions.add({υ, χ})
9:              transitions.remove({μ, χ})
10: function FINDPOSITION(ω, υ)
11:     Σ_υ = {σ, □σ | □σ ∈ υ}
12:     for (μ, χ) ∈ W² s.t. μRχ ∧ ∃n : ωR_n μ do
13:         if ({σ, □σ | □σ ∈ μ} ⊆ υ) ∧ (Σ_υ ⊆ χ) then
14:             return (μ, χ)
15:     μ = μ ∈ W s.t. ωR_n μ and n is maximised
16:     if μ == null then                          ▷ No such μ exists
17:         return (ω, null)
18:     return (μ, null)
```

3.4 Random Formula and Input Generators

3.4.1 Random Formula Generation Process

In order to test the theorem prover, two things were needed: a vast database of formulas and an external labelling of this database into valid and invalid formulas. In this section, the focus will be put on how to obtain such a database. One way of obtaining modal logic formulas is to find a public third-party database. However, these are not common (in fact, none were found), would need a parsing program due to the syntax and grammar likely not being the same as the one used in the theorem prover presented in this report, and would offer low flexibility in posing constraints and requirements to the formulas (e.g.: number of propositions used and size of the formulas). Hence, it was decided to create a random formula generator.

The process of creating a random formula is recursive, creating subformulas within each formula until propositions are inserted (i.e.: the base case). It is illustrated in Algorithm 3. The first step in such process is randomly choosing its type. This step is achieved by a switch statement. Each type is equally likely to be chosen. In order to meet the size requirement of each formula, a proposition type is not chosen unless the required size of the formula is equal to one. An integer within a range is randomly generated and, depending on its value, a type of formula is chosen. This formula type is processed by the random generator, which creates a string with the type information and recursively appends any substring of subformulas. For example, if the type is a disjunction, the string "(" + $subformula_1$ + "$\vee$" + $subformula_2$ + ")" will be created (where the + operator represents string concatenation); let it be the parent formula. The first and second subformulas in such example

will be recursively obtained by a call to the *generate*() method of the same *FormulaGenerator* object. This recursive call will have as parameters the same number of propositions but a size which will be calculated depending on the chosen type of the parent formula. In the disjunction example given above, a random integer will be generate with a range between 1 and the size of the parent formula minus 1. The size of the first subformula will be this integer, and the size of the second subformula will be the size of the parent formula minus such integer. If the type of a formula is negation, possibility or necessity, it will only have one subformula with a size equal to its own minus 1. Since, for any formula, the size of its subformulas is always less than its own size, this process will always reach the base case of size 1.

Functions *possibility* and *necessity* are similar to the implementation of *negation* shown in Algorithm 5; and functions *conjunction*, *implication* and *bicondition* are similar to the implementation of *disjunction* shown in Algorithm 6. Note that the *random* function used within *generate* is a linear congruential generator provided by java.util.Random[1]. The variables *size* and *propositions* represent the size of the formula and the maximum number of propositions, respectively; these parameters are initially specified by the user. Also, note that the *ascii* function used in Algorithm 4 returns the ASCII string representation of its integer argument; 97 is the ASCII value of the first alphabet lowercase letter "a".

[1] Java Util.Random documentation, last accessed on the 17th of March of 2019: `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`

Algorithm 3 Generate

function GENERATE(n, m)
  if $n == 1$ **then**
    return p_i where $i < m$ with prob. $\frac{1}{m}$
  if $n < 5$ **then**
    $u = \neg, \Diamond, \Box$ with with prob. $\frac{1}{3}$
    return (u + generate($n - 1$, m))
  if $n \geq 5$ **then**
    $c = \neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box$ with with prob. $\frac{1}{7}$
    if $c == \neg$ or $\Diamond$ or $\Box$ **then**
      return (c + generate($n - 1$, m))
    else
      return (‘(’ + generate(i, m) + c + generate ($n - i - 3$, m) + ‘)’) where $1 \leq i \leq n - 4$ with with prob. $\frac{1}{n-4}$

Algorithm 4 Proposition

function PROPOSITION($maxPropositions$)
  formula = ascii(random($maxPropositions$) + 97)
  return $formula$

Algorithm 5 Negation

function NEGATION($size$, $maxPropositions$)
  $subformula$ = generate($size - 1$, $maxPropositions$)
  $formula$ = “$\sim$” + $subformula$
  return $formula$

Algorithm 6 Disjunction

function DISJUNCTION($size$, $propositions$)
  $sublength = size - 3$ ▷ Each parenthesis and connective is 1 symbol
  $length1$ = random($sublength - 1$) + 1
  $length2 = sublength - length1$
  $subformula1$ = generate($length1$, $propositions$)
  $subformula2$ = generate($length2$, $propositions$)
  $formula$ = “(” + $subformula1$ + “|” + $subformula2$ + “)”
  return $formula$

3.4.2 Random Input Generator, its parameters and uses

Using the random formula generator described above, an input generator has been designed to provide such randomly generated formulas as input in various formats for various uses. The input generator can output a set of formulas as an aggregate Java object, write the formatted formulas to an input file for the prover and translate formulas in such format to the Molle format. Using this functionality, the random input generator has been used for standard testing, for testing against Molle (the modal logic theorem prover developed at the *Politecnico di Milano*) as described in detail in section 4.2, as well as to enable the validity study presented in section 5.

The random formula generator can be easily altered in order to produce formulas using only a subset of the available connectives. Any valid combination of frame conditions (as defined in section 3.3.4) can be chosen as a modal logic to be used when proving the formulas. The maximum number of propositions considered when randomly producing formulas can also be chosen. The minimum value for this parameter is 1 and, currently, the maximum value is 26 (the number of lowercase letters in the alphabet), as for the purposes of this project a greater flexibility was not needed. This feature can be easily expanded in the future to offer no upper limit, as each integer can be mapped to sequence of ASCII symbols to be representatives of a proposition, since the parser of the theorem prover does not pose an upper limit on the number of symbols used to express a single proposition. Another key parameter is the size of each formula to be generated. This parameter is the main variable used in the study of validity presented in section 5. The last parameter currently available is the number of formulas to be generated.

4 Testing

The testing strategy followed during the course of this project consists on three key approaches: functional testing on the different algorithms and key parts of the program, system testing using a comparison process with a publicly available modal prover, and performance testing for different sizes of formulas. This strategy demonstrates the correctness and resilience of the program, as well as its capabilities and limitations.

4.1 Functional Testing

The Random Formula and Input Generators have been tested on producing the right number of formulas, of the correct size, with the correct number of propositions and which adhere to the syntax and grammar rules specified. The parser was tested with examples of formulas with valid and invalid syntax, with various examples of mistakes and a great variety of logical structures within them. The set of algorithms described throughout section 3 used for proving formulas in the modal logic K were tested by a testing suite containing checks for correctness, created using a set of formulas known to be valid or invalid and which covered all of the key procedures of such algorithms. Each of the frame conditions' additional algorithms were also tested for correctness in this way, as well as for termination in the case of linear and transitive frames using a set of formulas known to cause all the key potential non-terminating instances. All such tests were powered by the JUnit 5 testing framework in Java[2].

[2]JUnit 5 documentation, last accessed on the 17$^{\text{th}}$ of March of 2019: `https://junit.org/junit5/`

4.2 Comparison Testing: Molle

In order to test the correctness of the theorem prover as a complete system (which was considered once the program was able to prove modal logic K formulas and passed all functional tests), two main resources were needed: a large dataset of modal formulas using the same syntax and grammar as the hereby presented prover, and a labelling of validity for each formula in the dataset. The first of such was obtained using the random formula and input generators described in section 3.4. The second would need an external validity classifier which could provide an accurate labelling of such formulas. Such classifier was chosen to be Molle [1], a modal logic theorem prover developed by Michele Mazzucchi and Andrea Mocci at the *Politecnico di Milano*, under the supervision of Prof. Marco Colombetti and Prof. Alessandra Cherubini. This prover was found within the collection of modal logic tools gathered by *AiML* (*Advancements in Modal Logic*) [9].

Molle is a modal logic theorem prover with a graphical user interface, which enables the user to input formulas one at a time, to check the validity of the formula, and to obtain a graphical model satisfying the formula in the case it is valid. Due to the tight limitation in the number of formulas the user can enter per unit time, a wrapper program was built, which would make use of Molle's source code [10] (which was not modified and is licensed under the Academic Free License version 1.2[3]) to prove the formulas without a graphical input interface, hence automatising the input process. In order to do this, a careful study of the source code was made. The Molle adapter consists on an object class named $MolleAdapter$ which is able to use Molle's internal object classes and methods in order to provide a method named $proveFormula()$,

[3]Molle's license, last accessed on the 14$^{\text{th}}$ of March of 2019: `http://molle.sourceforge.net/license.html`

which is able to take a formula string formatted to adhere to Molle's syntax and grammar specification and return the validity of such formula using Molle's algorithms. Note that the syntax and grammar Molle uses is not the same as the one specified for the prover presented in this report. Hence, a conversion method was included in the input generator, which translates the generated formulas to Molle's format. A testing suite was created for *MolleAdapter* in order to check correctness following the same strategy as with the testing suits designed for the various frame conditions, as detailed in section 4.1.

A cross validation testing method was created to compare the output of the *MolleAdapter* and my prover for over one million formulas over many batches, each consisting of 10 to 100 thousand formulas. Random input formulas were generated and translated to the Molle format using the input generator, then proved by both programs and, if a discrepancy was found between both results, such event would be notified to the tester and the pertinent formula would be recorded in a file containing all discrepancies. Approximately 1 out of every 2,500 formulas produced inconsistent results between the two provers. Ten of these formulas were proven using manual methods (such as hand-made tableaux). The validities of all such formulas were concluded to be correctly outputted by my prover, and incorrectly so by Molle. The developers of Molle were notified about this issue by an email dated on the 29$^{\text{th}}$ of November of 2018. An answer by Andrea Mocci and Prof. Marco Colombetti confirmed our conclusions by means of a logic proof, and a bug within Molle was recognised to exist, but not found nor corrected. This testing strategy allowed to obtain strong evidence of the correctness of my theorem prover.

4.3 Performance Testing

A test suite was created to assess the performance of the prover program by measuring the time taken to prove formulas of different sizes. The tests fixed parameters such as maximum number of propositions per formula and the set of modal frame conditions. For each given size, 1,000 formulas were generated by the input generator and proved, and the time taken to prove each formula was recorded. This process was done for two different set of connectives: $\{\neg, \vee, \Diamond\}$ and $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box\}$. The results of such tests are provided in Appendix C. Also note that a timeout of 10 seconds was applied to each formula, such that any proving process which exceeded that figure would be aborted; the abortion rate of formulas for each size is also indicated in the data.

The memory usage of the program has been monitored using VisualVM[4]. This software allowed to examine in real time which Java objects were consuming more memory resources, and how this varied depending on the size and other parameters of the formulas. It was found that $Formula$, $World$ and $Transition$ objects were the most numerous, while the first one was consuming approximately 20% of available heap memory. VisualVM was also used to find code which was preventing memory used by a proving process for a single formula from being freed before starting a new proving process for a different formula.

[4]VisualVM documentation, last accessed on the 14$^{\text{th}}$ of March of 2019: `https://visualvm.github.io/`

5 Study on validity rates

Given a thoroughly tested modal logic theorem prover which could determine the validity of thousands of modal logic formulas in a matter of minutes, as well as a modal logic formula generator which could produce a fair subset of a given logic and feed it as input to the prover, it was possible to perform a study on the validity rates of different modal logics, for different formula sizes, set of connectives used in the logic, and maximum number of propositions. The dependent variable as defined in the study is the validity rate of a large sample of formulas, while the main independent variable is the size of the formulas measured by their number of symbols, as it was thought this could have interesting philosophical consequences. Another key variable is the set of connectives defined in the logic.

A logic formula can be thought of as a statement which conveys information about the universe as the designer of the formula understands it. The validity of such formula can then be understood as a universal and objective conclusion of its truth under all circumstances and states of the universe. It seems to be true that the number of symbols we are allowed to use in order to express a perception via logic formulas (or, in other words, the size of the formula), affects the validity rate of all possible formulas of exactly such size. However, it is not clear how and to which extent different syntactical and semantical parameters, such as the axioms and connectives of the logic, as well as the maximum allowed number of propositions, affect such validity rate as the size of formulas tends to infinity.

It sounds sensible to say that all logics in which all the aforementioned parameters are fixed with the exception of their set of connectives, and in

which their set of connectives is complete (so that all such logics have the same expressive power), will all have the same validity rate as such size tends to infinity. The rationale behind this idea is that, even though a particular statement about the universe as expressed in a logic formula may take more symbols to convey in its minimal form (in other words, using the least amount of symbols possible) depending on the set of connectives used, as the size of the formulas tends to infinity, this issue is of no concern. The empirical results presented in this section show that reality is seemingly more complex.

A program to carry out the study was created, which makes use of the prover and input generator. Such program allows the user to change the set of aforementioned parameters, as well as other parameters such as the size of the sample (how many formulas to prove for each size) and the jump size (the increment which is applied to the size variable at each iteration). In each iteration, the program generates as many formulas as the size of the sample specifies, which comply with all parameters, as well as with the current size being considered. Then, it proves all formulas, stores the results in an array (results are given as a *Result* Java object which, in addition to storing the validity of the formulas, contains the abortion rate for a given timeout limit and the time taken to prove each formula) and calculates the validity rate of such results. Finally, the size of the formula is incremented by the jump size. A timeout limit was used because of the considerable amount of time it took to run each iteration for high formula and sample sizes, as run on a 2015 Mackbook Pro with a 2.7 GHz Intel Core i5 and a 8 GB 1867 MHz DDR3 RAM memory. A timeout limit of x seconds means that the prover will abort each process of proving a formula after x seconds, and such formula will not be considered in the results.

Data for each of the analyses conducted (meaning each run of the program with different parameters) is presented in tables in Appendix B. The analyses were stopped at a given formula size for one of two reasons: rather the validity rate seemed to be converging or the abortion rate was too high. A high abortion rate makes the validity rate results inconclusive, as assumptions cannot be made in relation to the validity of the aborted formulas. An analysis for modal logic K, using the set of connectives $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box\}$, a maximum number of propositions of 2 and a sample of 10,000 formulas per iteration with unit jumps, yielded the validity rates illustrated in Figure 11 (the complete data is also provided in Appendix B).

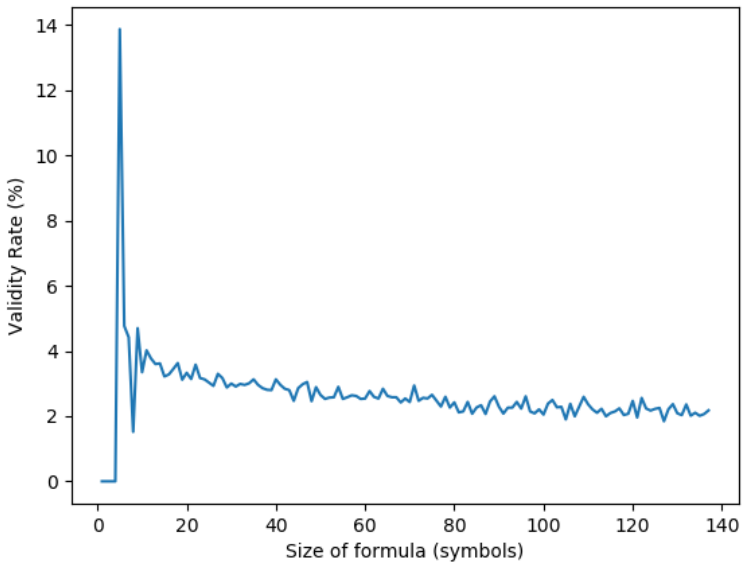

Figure 11: Validity rates for parameters:
- Connectives = $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

As it can be observed in Figure 11, it seems that the validity rates are

either tending to 0 or converging to some other value of less than 3%. If we run another analysis using the same parameters, with the exception of the set of connectives, for which we will use $\{\neg, \vee, \Diamond\}$, we obtain the validity rates illustrated in Figure 12.

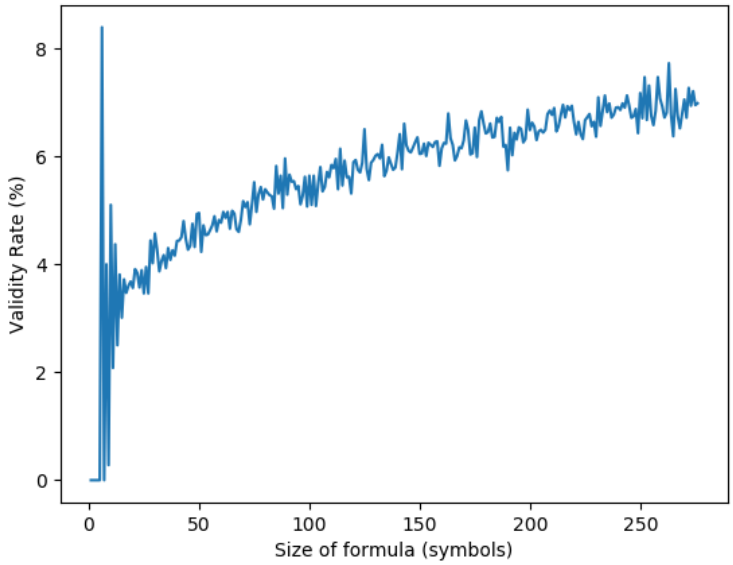

Figure 12: Validity rates for parameters:
- Connectives = $\{\neg, \vee, \Diamond\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

By observing Figure 12, it seems that the validity rates show initial convergence behaviour to a value of more than 6%. If we assume these convergence figures, as suggested by the data collected, then different complete set of connectives for the same logic can produce different validity rates as the size of formulas tends to infinity. Analyses have also been conducted for propositional logics with no modal connectives, such as for the set of connectives $\{\neg, \vee\}$ and the rest of parameters being those previously described.

The results are shown in Figure 13.

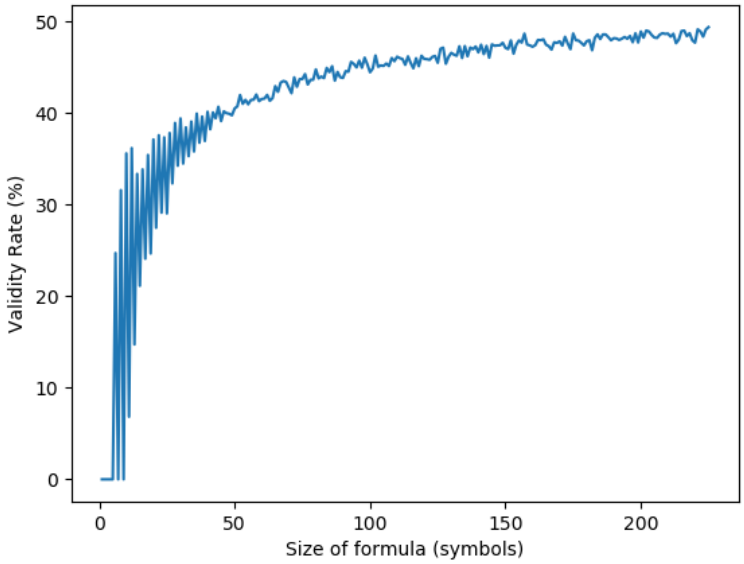

Figure 13: Validity rates for parameters:
- Connectives = $\{\neg, \vee\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

By observing Figures 12 and 13, it seems that the validity rates of such logics do not tend to neither 0% nor 100%. However, there exists a body of theoretical research, including a paper written by J. Halpern and B. Kapron published in 1994 [11], which shows that the zero-one law applies to the validity rates of propositional modal logic. The zero-one law states that the probability of an event should either tend to zero or to one. Therefore, the results obtained in this study are surprising. It is plausible that the inferred tendencies are not correct. It can be the case that what seems to be a convergence in the data is in fact an inflexion point, for example. It can also be the case that the data reflects very slow growth or decline, and

not convergence. The prover, input generator and validity programs can be used in the future to further investigate the relation between such theoretical works and these empirical results.

6 Conclusions and Future Work

During this project, a reliable, resilient and scalable theorem prover for propositional modal logics has been designed, implemented and thoroughly tested. Additionally, programs to generate modal formulas as well as to study their validity rates have been created. These programs can be used now to enhance research and educational activities. The study on validity rates has yielded interesting results with respect to the current set of knowledge and theories on the topic. Overall, this report presents the reader with a complete work on how to prove, analyse and understand the concept of validity in modal logics.

In the future, and given that the implementation was designed with scalability in mind, the theorem prover can be expanded in order to be able to recognise and decide other propositional modal logics such as temporal logic, as well as first-order modal logics. It may also be possible to further optimise the implementation in terms of memory and time usage, which can be useful when running analyses on validity rates and for other purposes. The results obtained in the study of validity rates can be further analysed and coupled with theoretical work in order to learn more about this matter. Additionally, new analyses can be run using the programs by simply changing the parameters available to the user. I look forward to future collaborations on these and other themes emanating from the work presented in this report.

References

[1] M. Mazzucchi, A. Mocci, M. Colombetti, and A. Cherubini, "Molle modal logic prover." Last accessed on the 17^{th} of March of 2019. Available at: `http://molle.sourceforge.net/index.html`.

[2] P. Blackburn, M. de Rijke, and Y. Venema, *Modal Logic*. Cambridge University Press, 2001.

[3] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, *Handbook of Logic in Computer Science: Volume 2*. Clarendon Press, 1992.

[4] R. Turner, *Logics for Artificial Intelligence*. Ellis Horwood Limited, 1984.

[5] J. Garson, "Modal logic (stanford encyclopedia of philosophy)," Sep 2018. Last accessed on the 14^{th} of March of 2019. Available at: `https://plato.stanford.edu/entries/logic-modal/`.

[6] R. Girle, *Modal Logics and Philosophy*. Acumen, 2000.

[7] M. Fitting, "Tableau methods of proof for modal logics," *Notre Dame Journal of Formal Logic*, vol. XIII, no. 2, pp. 237–247, 1972.

[8] L. Catach, "Tableaux: A general theorem prover for modal logics," *Journal of Automated Reasoning*, no. 7, pp. 489–510, 1991.

[9] R. Schmidt, "Advancements in modal logic (aiml): tools," April 2018. Last accessed on the 17^{th} of March of 2019. Available at: `http://www.cs.man.ac.uk/~schmidt/tools/`.

[10] M. Mazzucchi, A. Mocci, M. Colombetti, and A. Cherubini, "Molle: source code." Last accessed on the 14^{th} of March of 2019. Available at: `https://sourceforge.net/projects/molle/files/`.

[11] J. Y. Halpern and B. Kapron, "Zero-one laws for modal logic," *Annals of Pure and Applied Logic*, vol. 69, no. 2–3, pp. 157–193, 1994.

Appendix A Discrepancies with Molle

The following formulas have been found to produce inconsistent results in my theorem prover and Molle:

$$((\Box(p \wedge q) \wedge q) \vee (\Box\Diamond(p \wedge p) \vee \neg\Box\Diamond(q \wedge p)))$$

$$((\Box(q \vee p) \vee \Box(q \rightarrow p)) \rightarrow (\Diamond\Box\neg(p \leftrightarrow p) \rightarrow \Diamond\Box q))$$

$$(((p \vee p) \leftrightarrow p) \leftrightarrow (\Diamond\Box(q \leftrightarrow p) \rightarrow \neg\neg\Diamond\Box(p \rightarrow q)))$$

$$(\neg((q \rightarrow q) \wedge \Diamond\Box(p \wedge p)) \vee (q \vee \neg\Box\Diamond\neg(q \vee p)))$$

$$((q \vee p) \rightarrow (\Diamond\Diamond\Box(p \rightarrow q) \vee \Box\Box(\Box(q \leftrightarrow p) \rightarrow q)))$$

$$\Box((\neg\Diamond\neg\Diamond(q \leftrightarrow q) \vee (p \wedge p)) \vee \Diamond\Box p)$$

$$\Box((\Diamond\Box\Box((p \wedge p) \wedge q) \wedge (q \vee q)) \rightarrow \Diamond\Box\Box(q \rightarrow p))$$

$$(\Box\Diamond(p \rightarrow p) \vee (p \rightarrow \Diamond\neg(\neg\Diamond p \leftrightarrow \Diamond(p \wedge q))))$$

$$((\Box\Diamond\neg(p \rightarrow q) \rightarrow \Box\Diamond\neg q) \vee \neg(q \rightarrow \Box\neg q))$$

Appendix B Validity Rates Dataset

Table 3: - Connectives = $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

Size	**Validity Rate (%)**	**Abortion Rate (%)**
1	0.000	0.000
2	0.000	0.000
3	0.000	0.000
4	0.000	0.000
5	13.880	0.000
6	4.770	0.000
7	4.420	0.000
8	1.520	0.000
9	4.700	0.000
10	3.350	0.000
11	4.020	0.000
12	3.770	0.000
13	3.600	0.000
14	3.620	0.000
15	3.220	0.000
16	3.280	0.000
17	3.450	0.000
18	3.630	0.000
19	3.120	0.000
20	3.330	0.000
21	3.140	0.000
22	3.580	0.000
23	3.170	0.000
24	3.130	0.000
25	3.030	0.000
26	2.930	0.000
27	3.300	0.000
28	3.170	0.000
29	2.880	0.000
30	3.000	0.000

31	2.910	0.000
32	2.990	0.000
33	2.960	0.000
34	3.010	0.000
35	3.130	0.000
36	2.960	0.000
37	2.860	0.000
38	2.810	0.000
39	2.800	0.000
40	3.130	0.000
41	2.960	0.000
42	2.840	0.000
43	2.800	0.000
44	2.470	0.010
45	2.860	0.000
46	2.980	0.000
47	3.050	0.000
48	2.460	0.010
49	2.890	0.000
50	2.650	0.000
51	2.531	0.030
52	2.570	0.010
53	2.580	0.010
54	2.900	0.000
55	2.530	0.010
56	2.580	0.010
57	2.641	0.020
58	2.620	0.000
59	2.530	0.000
60	2.541	0.050
61	2.770	0.010
62	2.591	0.020
63	2.541	0.020
64	2.841	0.020
65	2.621	0.040
66	2.581	0.020
67	2.580	0.010
68	2.421	0.040

69	2.543	0.100
70	2.434	0.150
71	2.941	0.030
72	2.472	0.080
73	2.563	0.110
74	2.543	0.130
75	2.655	0.170
76	2.473	0.130
77	2.296	0.240
78	2.595	0.210
79	2.264	0.180
80	2.423	0.130
81	2.117	0.320
82	2.145	0.240
83	2.437	0.280
84	2.075	0.250
85	2.267	0.290
86	2.336	0.270
87	2.067	0.330
88	2.448	0.340
89	2.610	0.390
90	2.291	0.460
91	2.080	0.460
92	2.261	0.500
93	2.261	0.500
94	2.437	0.680
95	2.233	0.590
96	2.609	0.740
97	2.144	0.660
98	2.086	0.790
99	2.209	0.880
100	2.049	0.910
101	2.385	1.040
102	2.497	1.090
103	2.272	0.990
104	2.287	1.170
105	1.901	1.130
106	2.378	1.170

107	1.996	1.300
108	2.290	1.300
109	2.593	1.280
110	2.363	1.410
111	2.206	1.620
112	2.104	1.620
113	2.227	1.640
114	1.996	1.820
115	2.097	1.780
116	2.144	2.040
117	2.239	2.180
118	2.036	2.270
119	2.074	2.140
120	2.469	2.000
121	1.959	2.480
122	2.559	2.310
123	2.236	2.490
124	2.172	2.400
125	2.224	2.430
126	2.252	3.200
127	1.845	2.990
128	2.212	3.250
129	2.369	2.920
130	2.087	2.750
131	2.034	3.140
132	2.360	3.400
133	2.015	3.240
134	2.105	3.110
135	2.013	3.650
136	2.065	3.640
137	2.178	4.040

Table 4: - Connectives = $\{\neg, \vee, \Diamond\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

Size	**Validity Rate (%)**	**Abortion Rate (%)**
1	0.000	0.000
2	0.000	0.000
3	0.000	0.000
4	0.000	0.000
5	0.000	0.000
6	8.390	0.000
7	0.000	0.000
8	4.000	0.000
9	0.280	0.000
10	5.100	0.000
11	2.080	0.000
12	4.370	0.000
13	2.500	0.000
14	3.810	0.000
15	3.010	0.000
16	3.720	0.000
17	3.470	0.000
18	3.600	0.000
19	3.680	0.000
20	3.560	0.000
21	3.910	0.000
22	3.840	0.000
23	3.570	0.000
24	3.890	0.000
25	3.460	0.000
26	3.950	0.000
27	3.460	0.000
28	4.440	0.000
29	4.020	0.000
30	4.570	0.000
31	4.270	0.000
32	3.870	0.000

33	4.060	0.000
34	4.170	0.000
35	3.930	0.000
36	4.300	0.000
37	4.080	0.000
38	4.260	0.000
39	4.160	0.000
40	4.430	0.000
41	4.440	0.000
42	4.500	0.000
43	4.800	0.000
44	4.470	0.000
45	4.270	0.000
46	4.340	0.000
47	4.750	0.000
48	4.320	0.000
49	4.930	0.000
50	4.950	0.000
51	4.230	0.000
52	4.720	0.000
53	4.540	0.000
54	4.550	0.000
55	4.630	0.000
56	4.720	0.000
57	4.890	0.000
58	4.610	0.000
59	4.820	0.000
60	4.770	0.000
61	4.970	0.000
62	4.860	0.000
63	4.970	0.000
64	4.660	0.000
65	4.990	0.000
66	4.940	0.000
67	4.650	0.000
68	4.600	0.000
69	4.830	0.000
70	5.170	0.000

71	5.060	0.000
72	5.150	0.000
73	4.740	0.000
74	5.110	0.000
75	5.520	0.000
76	4.970	0.000
77	5.300	0.000
78	5.430	0.000
79	5.200	0.000
80	5.390	0.000
81	5.330	0.000
82	5.280	0.000
83	5.260	0.000
84	5.030	0.000
85	5.820	0.000
86	5.310	0.000
87	5.640	0.000
88	5.040	0.000
89	5.960	0.000
90	5.290	0.000
91	5.660	0.000
92	5.530	0.000
93	5.540	0.000
94	5.390	0.000
95	5.450	0.000
96	5.110	0.000
97	5.270	0.000
98	5.620	0.000
99	5.070	0.000
100	5.641	0.010
101	5.100	0.000
102	5.640	0.000
103	5.080	0.000
104	5.480	0.000
105	5.800	0.000
106	5.350	0.000
107	5.430	0.000
108	5.710	0.000

109	5.611	0.010
110	5.840	0.000
111	5.780	0.000
112	5.951	0.010
113	5.390	0.000
114	6.140	0.000
115	5.460	0.000
116	5.920	0.000
117	5.610	0.000
118	5.621	0.010
119	5.311	0.010
120	5.891	0.010
121	5.931	0.010
122	5.751	0.010
123	5.700	0.000
124	5.880	0.000
125	6.501	0.010
126	5.770	0.000
127	5.560	0.000
128	5.881	0.020
129	5.930	0.000
130	6.010	0.000
131	6.041	0.010
132	5.961	0.020
133	6.213	0.050
134	5.632	0.030
135	5.731	0.020
136	5.981	0.010
137	5.861	0.010
138	5.750	0.000
139	5.791	0.010
140	6.071	0.010
141	6.411	0.010
142	5.761	0.010
143	6.604	0.060
144	6.204	0.060
145	6.102	0.040
146	6.072	0.040

147	6.162	0.040
148	6.263	0.050
149	6.351	0.020
150	6.044	0.060
151	6.051	0.020
152	6.234	0.070
153	6.005	0.080
154	6.251	0.020
155	6.213	0.050
156	6.174	0.060
157	6.264	0.060
158	6.277	0.110
159	5.824	0.070
160	6.144	0.070
161	6.247	0.120
162	6.234	0.060
163	6.793	0.050
164	6.336	0.090
165	6.214	0.070
166	5.924	0.060
167	6.004	0.070
168	6.167	0.110
169	6.149	0.140
170	6.295	0.080
171	6.664	0.060
172	6.488	0.120
173	6.035	0.090
174	6.056	0.100
175	6.531	0.170
176	5.987	0.110
177	6.672	0.180
178	6.834	0.200
179	6.609	0.140
180	6.424	0.220
181	6.448	0.120
182	6.608	0.120
183	6.348	0.130
184	6.361	0.180

185	6.713	0.200
186	6.637	0.260
187	6.730	0.150
188	6.179	0.140
189	6.211	0.180
190	5.741	0.190
191	6.530	0.160
192	6.021	0.180
193	6.435	0.230
194	6.312	0.190
195	6.536	0.250
196	6.508	0.270
197	6.257	0.270
198	6.324	0.220
199	6.864	0.200
200	6.485	0.230
201	6.623	0.190
202	6.551	0.320
203	6.301	0.340
204	6.461	0.330
205	6.491	0.320
206	6.439	0.290
207	6.493	0.200
208	6.783	0.190
209	6.847	0.400
210	6.766	0.240
211	6.895	0.220
212	6.461	0.320
213	6.556	0.390
214	6.766	0.390
215	6.954	0.340
216	6.719	0.290
217	6.929	0.420
218	6.857	0.390
219	6.936	0.380
220	6.631	0.460
221	6.406	0.410
222	6.639	0.430

223	6.413	0.520
224	6.319	0.460
225	6.669	0.590
226	6.724	0.510
227	6.784	0.500
228	6.556	0.400
229	6.639	0.590
230	6.363	0.680
231	7.092	0.590
232	6.563	0.650
233	6.839	0.710
234	7.126	0.510
235	6.818	0.560
236	6.979	0.840
237	6.716	0.690
238	6.770	0.740
239	6.902	0.610
240	6.909	0.710
241	6.858	0.560
242	6.981	0.590
243	6.902	0.750
244	7.128	0.530
245	6.936	0.670
246	6.713	0.790
247	6.741	0.750
248	6.877	0.830
249	6.429	0.760
250	7.172	0.720
251	6.705	0.820
252	7.466	0.880
253	6.673	0.650
254	7.310	0.690
255	6.748	0.860
256	6.577	0.720
257	6.842	0.910
258	7.467	0.900
259	7.069	0.840
260	6.926	0.810

261	6.718	0.870
262	6.809	1.010
263	7.724	0.830
264	6.772	0.910
265	6.372	0.810
266	7.247	0.790
267	6.751	1.050
268	6.519	0.910
269	6.783	0.930
270	7.052	1.020
271	6.714	1.100
272	7.267	1.060
273	6.936	0.950
274	7.207	0.930
275	6.949	0.990
276	6.982	1.030

Table 5: - Connectives = $\{\neg, \vee\}$
- Frame Conditions = K
- Sample (formulas) = 10,000
- Max. Number of Propositions = 2

Size	**Validity Rate (%)**	**Abortion Rate (%)**
1	0.000	0.000
2	0.000	0.000
3	0.000	0.000
4	0.000	0.000
5	0.000	0.000
6	24.730	0.000
7	0.000	0.000
8	31.600	0.000
9	0.000	0.000
10	35.620	0.000
11	6.840	0.000
12	36.200	0.000
13	14.740	0.000
14	33.380	0.000
15	21.150	0.000
16	33.870	0.000
17	24.110	0.000
18	35.450	0.000
19	24.680	0.000
20	37.140	0.000
21	27.490	0.000
22	37.610	0.000
23	29.140	0.000
24	37.370	0.000
25	29.060	0.000
26	37.850	0.000
27	32.340	0.000
28	38.950	0.000
29	34.270	0.000
30	39.440	0.000
31	34.510	0.000
32	38.470	0.000

33	35.320	0.000
34	39.100	0.000
35	35.830	0.000
36	39.970	0.000
37	36.780	0.000
38	39.650	0.000
39	36.980	0.000
40	40.170	0.000
41	38.260	0.000
42	40.100	0.000
43	39.460	0.000
44	40.720	0.000
45	39.150	0.000
46	40.210	0.000
47	40.020	0.000
48	39.950	0.000
49	39.780	0.000
50	40.530	0.000
51	40.760	0.000
52	42.000	0.000
53	41.040	0.000
54	41.470	0.000
55	40.990	0.000
56	41.460	0.000
57	41.490	0.000
58	42.050	0.000
59	41.330	0.000
60	41.560	0.000
61	41.610	0.000
62	42.010	0.000
63	41.370	0.000
64	41.660	0.000
65	42.970	0.000
66	42.350	0.000
67	43.360	0.000
68	43.530	0.000
69	43.440	0.000
70	42.830	0.000

71	42.200	0.000
72	43.930	0.000
73	42.900	0.000
74	43.750	0.000
75	43.740	0.000
76	44.290	0.000
77	43.150	0.000
78	43.660	0.000
79	43.670	0.000
80	44.800	0.000
81	43.820	0.000
82	44.020	0.000
83	43.850	0.000
84	44.960	0.000
85	44.500	0.000
86	45.150	0.000
87	43.570	0.000
88	44.520	0.000
89	43.940	0.000
90	43.870	0.000
91	44.610	0.000
92	44.550	0.000
93	45.620	0.000
94	45.400	0.000
95	45.020	0.000
96	45.760	0.000
97	44.980	0.000
98	46.080	0.000
99	45.360	0.000
100	44.490	0.000
101	44.890	0.000
102	46.320	0.000
103	45.090	0.000
104	45.240	0.000
105	45.180	0.000
106	45.440	0.000
107	45.170	0.000
108	46.040	0.000

109	45.700	0.000
110	46.190	0.000
111	46.030	0.000
112	45.930	0.000
113	45.300	0.000
114	46.220	0.000
115	45.510	0.000
116	44.920	0.000
117	46.020	0.000
118	45.150	0.000
119	46.290	0.000
120	45.950	0.000
121	45.920	0.000
122	45.850	0.000
123	46.170	0.000
124	46.270	0.000
125	45.520	0.000
126	47.050	0.000
127	47.180	0.000
128	45.420	0.000
129	46.170	0.000
130	46.610	0.000
131	46.410	0.000
132	46.290	0.000
133	47.330	0.000
134	46.040	0.000
135	47.350	0.000
136	46.230	0.000
137	47.190	0.000
138	47.010	0.000
139	47.300	0.000
140	46.660	0.000
141	47.500	0.000
142	46.470	0.000
143	47.370	0.000
144	46.080	0.000
145	47.540	0.000
146	47.380	0.000

147	47.420	0.000
148	47.430	0.000
149	47.710	0.000
150	47.160	0.000
151	47.020	0.000
152	47.970	0.000
153	46.520	0.000
154	47.490	0.000
155	47.970	0.000
156	47.730	0.000
157	48.710	0.000
158	47.520	0.000
159	47.420	0.000
160	47.280	0.000
161	47.440	0.000
162	48.020	0.000
163	47.990	0.000
164	48.090	0.000
165	47.500	0.000
166	47.340	0.000
167	46.960	0.000
168	47.760	0.000
169	47.700	0.000
170	47.870	0.000
171	47.400	0.000
172	48.360	0.000
173	47.635	0.010
174	47.010	0.000
175	48.720	0.000
176	47.990	0.000
177	47.950	0.000
178	47.700	0.000
179	47.420	0.000
180	47.880	0.000
181	47.990	0.000
182	46.890	0.000
183	48.250	0.000
184	48.630	0.000

185	48.120	0.000
186	48.620	0.000
187	48.620	0.000
188	48.320	0.000
189	47.980	0.000
190	48.210	0.000
191	48.180	0.000
192	48.020	0.000
193	48.160	0.000
194	48.340	0.000
195	48.150	0.000
196	48.450	0.000
197	47.770	0.000
198	48.740	0.000
199	47.740	0.000
200	48.940	0.000
201	48.270	0.000
202	49.050	0.000
203	48.940	0.000
204	48.560	0.000
205	48.280	0.000
206	48.240	0.000
207	48.600	0.000
208	48.750	0.000
209	48.680	0.000
210	48.710	0.000
211	48.380	0.000
212	48.690	0.000
213	47.670	0.000
214	48.050	0.000
215	48.990	0.000
216	49.040	0.000
217	48.440	0.000
218	48.750	0.000
219	47.990	0.000
220	47.720	0.000
221	49.180	0.000
222	48.940	0.000

223	48.380	0.000
224	49.160	0.000
225	49.420	0.000

Appendix C Performance Testing Results

Table 6: - Connectives = $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \Diamond, \Box\}$
- Frame Conditions = K
- Sample (formulas) = 1,000
- Max. Number of Propositions = 2

Size	**Average Time (%)**	**Abortion Rate (%)**
1	12.833	0.000
10	17.386	0.000
20	16.625	0.000
30	14.842	0.000
40	16.116	0.000
50	19.101	0.000
60	28.129	0.000
70	34.964	0.000
80	44.686	0.200
90	72.812	0.100
100	116.736	0.300
200	461.003	6.200
300	649.473	14.800

Table 7: - Connectives = $\{\neg, \vee, \Diamond\}$
- Frame Conditions = K
- Sample (formulas) = 1,000
- Max. Number of Propositions = 2

Size	**Average Time (%)**	**Abortion Rate (%)**
1	12.448	0.000
10	14.216	0.000
20	14.405	0.000
30	13.960	0.000
40	13.573	0.000
50	16.112	0.000
60	12.894	0.000
70	14.783	0.000
80	12.720	0.000
90	14.464	0.000
100	13.517	0.000
200	26.670	0.200
300	67.978	0.900
400	62.712	1.300
500	113.956	2.500
600	141.755	3.700
700	189.664	2.900
800	157.832	4.800
900	182.873	5.200
1000	190.521	5.000
2000	233.788	7.500
3000	308.996	8.500
4000	317.685	10.800
5000	387.753	11.200
6000	417.837	11.100
7000	409.020	11.300
8000	357.438	12.700
9000	388.564	11.200
10000	384.928	13.600
20000	489.681	16.800

Appendix D Manual

SYNTAX

The syntax you must use when writing your formulas is the following (format: "what you want to write" = "how you should write it"):

$$\neg\phi = \sim \phi$$

$$\Box\phi = []\phi$$

$$\Diamond\phi = <>\phi$$

$$(\phi \wedge \phi) = (\phi \ \& \ \phi)$$

$$(\phi \vee \phi) = (\phi \mid \phi)$$

$$(\phi \rightarrow \phi) = (\phi \ \text{-> } \phi)$$

$$(\phi \leftrightarrow \phi) = (\phi \ \text{<-> } \phi)$$

You must also note that formulas in an array are separated by commas (","), the last formula being the one to prove. Arrays of formulas are separated by semicolons (";"). The last formula in a given array will be considered as the formula intended to be proven, and all previous formulas of the theorem will be considered as axioms. Also, the use of spaces, new tabs, new lines, etc. is irrelevant (but surely helpful to the reader).

GRAMMAR

This theorem prover uses standard modal logic grammar. It is recursively defined as the following:

$prop = \{a : a \text{ is a string}\}$ $\{b : b$ is "T" or "F" or a string such that any of its substrings are "$\sim$", "[]", "<>", "&", "|", "->", "<->", ";", ":", ",", ")" or "("$\}$

$$fmla = \text{"T"} \mid \text{"F"} \mid prop \mid \neg fmla \mid (fmla \wedge fmla) \mid (fmla \vee fmla) \mid$$
$$(fmla \rightarrow fmla) \mid (fmla \leftrightarrow fmla) \mid \Box fmla \mid \Diamond fmla$$

Please, make sure you are making use of parentheses correctly, as specified above. The "T" and "F" literals are used to express truth and falsehood. They are not essential, but useful. For example: "T" is equivalent to "($p \mid \sim p$)"; and "F" is equivalent to "(p & $\sim p$)".

FRAME CONDITIONS

At the beginning of the input file, before any theorem has been defined, you may include a set of frame conditions which will be applied to the theorems in such input file. If no such set is specified, only the frame condition K will be applied. The frame conditions currently supported by this theorem prover are the following:

K = Kripke

T = Reflexive

B = Symmetric

4 = Transitive

D = Serial

L = Linear

In order to specify your frame conditions, you may include the following construct (a string of your frame conditions surrounded by colons):

$$:frame\_conditions:$$

$frame\_conditions$ is a string whose only characters can be 'K', 'T', 'B', '4', 'D' and 'L', in any order but without repetition of any of them. Note that some frame conditions are incompatible. In this version of the program, only L is incompatible with B (as linear frames are antisymmetric). Also, some frame conditions imply other frame conditions. For example, linearity implies transitivity.

Create a file called "input.txt" and locate it within the folder "input". In it, write your arrays of formulas as specified above. Next, run the Java program *Prove*. Finally, find your results in the file "output.txt" in the folder named "output". If a formula is deemed as "not recognised", it is not following the syntax and/or grammar specified above. If you are confident your formula is correct, please report any bugs.

——— FORMULA AND INPUT GENERATORS ———

The formula and input generators can be used to create a large number of formulas with specific properties. Currently, the size of the formulas, as well as the maximum number of propositions can be given as parameters to the *generate* function in the *FormulaGenerator* class. Additionally, in the *InputGenerator* class, the user can specify the number of formulas to be generated and the set of frame conditions which will be added at the top of the generated input text file. These programs are provided as source code since they are meant to be used coupled with other logic tools.

The *InputGenerator* class provides different methods depending on the functionality needed by the user. The $generateFormulas(n, s, p)$ method returns an array of n formulas of size s, each with a maximum of p propositions. The $generateInputFile(n, s, p, conditions)$ works similarly, but it allows the user to give a *conditions* string to specify the frame conditions, and instead of returning the formulas, it will generate an input text file ready to be proven by the theorem prover. A similar method is $generateInput(n, s, p, conditions)$, which instead of creating an input file, it returns what would be its contents as a string; this method can be coupled with $translateInputToMolle(input)$, which can translate such input string to the syntax of Molle (these methods were used for comparison testing, as described in section 4.2).

Appendix E Project Plan

Aims

Throughout this project, I foremost aim to achieve a deep understanding of Modal Logic, in particular Basic Propositional Modal Logic (*BPML*). As the core part of this project, I will build a theorem prover which is able to prove valid *BPML K* formulas and theorems. This theorem prover will take the user's input formulas or theorems, as well as other specifications, and will output the validity of each input element.

The main extensions of the project are for this theorem prover to be able to prove other *BPML's*, such as *BPML T* (reflexive), *BPML B* (symmetric), *BPML 4* (transitive), as well as any combination of these. The user would be able to specify in the input the type of *BPML* (called *frame conditions*) he/she wishes to apply to the input formulas or theorems.

Another possible extension of the project consists on researching other Modal Logics, such as Temporal Logic, Description Logic or Feature Logic. In this extension, I would aim at understanding the underlying theory which links these Modal Logics with *BPML*, as well as researching a Multi-Modal Logic Theorem Prover which could incorporate one or more non-*BPML* logics to its set of supported logics.

One more possible extension of the project is to build a user-friendly interface for the theorem prover, such as a desktop application. Another more complex possibility is to create a visualisation system to represent diagrammatically the proof provided by the theorem prover for a given formula or theorem. This is possible due to the common interpretation of Modal Frames

and Models as labelled graphs.

My overall aim for this project is, realistically, to achieve the core part of the project, and to do at least one of the aforementioned -or otherwise- extensions. The choice of extensions will depend on the future implementation of the core theorem prover, as well as on further research done on such extensions, specifically on their time requirements and possible outcomes. Given the specified time constraint for this project, I believe it is not possible to explore the subject as much as I would desire, but it is possible to complete a substantial and meaningful study of the field.

Objectives

My objectives, based on my aims, are the following:

- To study the relevant literature on Basic Propositional Modal Logic, to understand the key concepts of the subject, and to identify interesting ideas within the field that could be explored in my work.
- To research the existing theorem provers for Modal Logic and their algorithmic approaches, as well as their successes and imitations.
- To build a theorem prover for Basic Propositional Modal Logic using existing tableau methods or other methods devised by myself. I aim at writing readable, extendable and robust code, making use of appropriate software design patterns and coding styles, as well as using or building suitable data structures to represent Modal Logic constructions.
- To test the theorem prover using robust and thorough testing methods.

- To research different types of Modal Logic and how they relate to Basic Propositional Modal Logic.
- To study the user-experience of the theorem prover and any derived programs.
- To justify and document my decisions and actions throughout the project, explaining them with complete, precise and logical arguments and data.

Expected Outcomes

I expect to achieve a deep understanding of Basic Propositional Modal Logic. I also expect to achieve a basic understanding of other Modal Logics, as well as of the underlying theory which links them. The main tangible outcome of this project will be a Modal Logic theorem prover, which will be well designed and implemented, as well as thoroughly tested. A manual and specification of the program will also be provided. All achievements, testing, findings, research and understanding will be presented, explained, and justified in a final report.

Work Plan

I will describe below, in chronological order, the set of actions I aim at taking in order to develop this project. As explained in the *Aims* section, the time constraints of this project will realistically not allow for the completion of all actions presented below, specially those related to what have been previously classified as extensions. Please, note that I understand documentation to be

a continuous process spanning the whole project time-line. Steps which are marked as "*Completed*" have been, at the date of this document, already achieved.

Completed To read relevant literature on Basic Propositional Modal Logic to achieve an adequate understanding of the subject for the task of building a theorem prover for such logic.

Completed To research existing methodologies and algorithms to prove Propositional Modal Logic formulas. Also, to find, if possible, other Modal Logic theorem provers and study them.

Completed To research and decide the set of technologies and systems which will be used to build the core theorem prover.

Completed To build a parser for the input formulas, theorems and specification. This parser should be designed taking into account the future structure and data requirements of the theorem prover.

Completed To build the theorem prover for *BPML K*. This theorem prover should be designed and built having in mind the desired ability to extend to other *BPML's*, and even other Modal Logics.

Completed To test the theorem prover to ensure its completeness and soundness, as well as its performance by doing real-time analysis.

Completed To extend the theorem prover to incorporate *BPML T*, as well as to create a set of tests to ensure the correct functioning of the system under this logic.

Completed To extend the theorem prover to incorporate *BPML B*, as well as to create a set of tests to ensure the correct functioning of the system under this logic.

Completed To extend the theorem prover to incorporate *BPML 4*, as well as to create a set of tests to ensure the correct functioning of the system under this logic.

Mid Nov - Mid Dec To evaluate the theorem prover (in terms of computational complexity or strengths and limitations, for example) and consider possible optimisations of the program. If any optimisations are found, also to try to implement them iteratively.

Mid Dec - End Dec To identify other possible *BPML's* for which the theorem prover could be extended. If any such logics are identified, to incorporate them to the theorem prover iteratively. If achieved, also to create a set of tests to ensure the correct functioning of the system under these logics.

End Dec - Mid Jan To research relevant literature on other Modal Logics to achieve a greater understanding of them, and to understand their relation to BPML.

Mid Jan - End Jan To prepare the Interim Report to be submitted.

Mid Jan - Mid Feb To research and implement iteratively ways of enhancing user experience when using the theorem prover, as well as additional visualisation features which could aid the user in understanding and representing the results offered by the program.

Mid Feb - Mid Apr To prepare the final report to be submitted (even though the writing of the final report will begin at the beginning of the project) and

to develop any other interests within the field which may have arisen throughout the development of the project.

Appendix F Interim Report

Achievements

The core activities of the project consisted on designing and producing a theorem prover for the Modal Logic K. This was achieved during the months of October and November of 2019. Extensive testing of such functionality was subsequently performed, including a side-by-side comparison with another public prover for the Modal Logic K designed in the Politecnico di Milano called Molle, in order to test performance and correctness. Such testing revealed inconsistencies between the output of both programs, which was later found to be an incorrect behaviour of Molle. The developers who worked on the project were notified and acknowledged the issue. A random formula generator was also created for testing purposes.

After having a solid foundation on top of which to build further functionality, compatibility with different types of Modal Logics was targeted as the next goal to achieve (in particular, the frame constraints of B, D, T, S4, as well as the linear frame constraint). Implementing the frame constraints of B, T and S4 was achieved before the start of 2019, while D and the linear constraint were implemented early in the aforementioned year. This functionality was built on top of the core of the program, and the individual implementations of such frame constraints are all compatible with each other; in other words, the user can define a Modal Logic with a set of these frame constraints, not just one of them.

After having designed and implemented a random formula generator, the possibility of doing an study on validity was considered. This study consisted on measuring how certain variables of modal formulas, such as the number of

symbols or the maximum number of propositions used, affect the validity rate of such formulas (i.e.: the percentage of formulas that are valid). In order to achieve this, the random formula generator was improved to support a large level of modularity, such that the size, number of propositions and the set of connectives used in the formulas produced could be easily altered. A number of performance issues slowed down this process due to the need to perform computations on a large number of long formulas to get statistically adequate results. These performance issues were solved and the study is now ongoing.

Next Steps

From this day onwards, the plan consists on finalising the testing process, producing the required documentation for the final report, as well as continuing with the study on validity. In terms of testing, only the last frame constraints to be implemented need further testing, which I estimate to be completed one or two weeks from the date of this report. The study on validity has no specific final goal, and is being seen as a collection of data and an exploration of it. The current activities involving this study are mainly collecting validity rates data for different values of the variables.

Appendix G Code Listing

This code listing does not contain all the code developed throughout this project due to space constraints. The parts listed have been selected by their relevance in this work, and these are the *Tableau*, *Frame* and *World* Java classes.

```
public class Tableau {

  private final FormulaArray formulaArray;
  private final ModalLogic logic;
  private LinkedList<Frame> frames;
  private int frameIdCount;
  private Prover prover;
  private long startTime;

  public Tableau(Prover prover, FormulaArray formulaArray,
      ModalLogic logic) {
    this.prover = prover;
    this.logic = logic;
    this.formulaArray = formulaArray;
    this.frames = new LinkedList<>();
    this.frameIdCount = 1;
    if (prover.isProtected()) {
      this.startTime = System.currentTimeMillis();
    }
  }

  public Boolean run() {
    frames.add(new Frame(formulaArray.getFormulas(), this,
        frameIdCount, logic));
    frameIdCount++;
```

```
    while (!frames.isEmpty()) {
      if (prover.isProtected()) {
        long now = System.currentTimeMillis();
        if (now - startTime > prover.getTimeout()) {
          return null;
        }
      }
      if (!frames.isEmpty()) {
        Frame frame = frames.getFirst();
        frame.scheduleNextExpansion();
        if (frame.hasContradiction()) {
          frames.removeFirst();
        } else {
          if (!frame.isExpandable()) {
            return Boolean.FALSE;    // Not valid
          }
          frames.add(frames.removeFirst());
        }
      }
    }
    return Boolean.TRUE;  // Valid
  }

  public void addFrame(Frame frame) {
    frames.add(frame);
    frameIdCount++;
  }

  public int getNewFrameId() {
    return frameIdCount;
  }
}
```

```
public class Frame {

  private final int id;
  private final ModalLogic logic;
  private final Tableau tableau;
  private LinkedList<World> worlds;
  private int currentWorldId;
  private HashSet<Transition> transitions;
  private int worldIdCount;
  private boolean isExpandable;

  public Frame(LinkedList<Formula> initialFormulas,
        Tableau tableau, int id, ModalLogic logic) {
    this.id = id;
    this.logic = logic;
    this.isExpandable = true;
    this.tableau = tableau;
    this.worlds = new LinkedList<>();
    this.currentWorldId = 0;
    worldIdCount = 1;
    World initialWorld = new World(initialFormulas, worldIdCount);
    if (logic.isSerial()) {
      Formula fmla1 = new Formula("T");
      Formula fmla2 = new Formula("<>T");
      fmla1.parse();
      fmla2.parse();
      initialWorld.addFormula(fmla1);
      initialWorld.addFormula(fmla2);
    }
    this.worlds.add(initialWorld);
    this.transitions = new HashSet<>();
    if (logic.isReflexive()) {
      this.addTransition(new Transition(1, 1));
```

```
  }
}

public Frame(Frame originalFrame, Tableau tableau,
      int id, ModalLogic specification) {
  this.id = id;
  this.logic = specification;
  this.isExpandable = true;
  this.tableau = tableau;
  this.worldIdCount = originalFrame.getWorldIdCount();
  this.worlds = originalFrame.cloneWorlds();
  this.currentWorldId = originalFrame.currentWorldId;
  this.transitions = originalFrame.cloneTransitions();
}

public void scheduleNextExpansion() {

  if (hasContradiction()) {
    isExpandable = false;
    return;
  }

  World chosenWorld = null;
  Formula chosenFormula = null;
  int i, j, k, noFormulas;
  int noWorlds = worlds.size();
  HashSet<FormulaType> types = new HashSet<>();

  outerLoop:
  for (i=1; i<=5; i++) {
    types.clear();
    switch (i) {
      case 1:
```

```
        types.add(FormulaType.NOTTRUE);
        types.add(FormulaType.NOTFALSE);
        break;
      case 2:
        types.add(FormulaType.AND);
        types.add(FormulaType.NOTOR);
        types.add(FormulaType.BICONDITION);
        types.add(FormulaType.NOTCONDITION);
        break;
      case 3:
        types.add(FormulaType.OR);
        types.add(FormulaType.NOTAND);
        types.add(FormulaType.CONDITION);
        types.add(FormulaType.NOTBICONDITION);
        break;
      case 4:
        types.add(FormulaType.POSSIBLY);
        types.add(FormulaType.NOTNECESSARILY);
        break;
      case 5:
        types.add(FormulaType.NECESSARILY);
        types.add(FormulaType.NOTPOSSIBLY);
        break;
    }
    for (j=0; j<noWorlds; j++) {
      World world = worlds.getFirst();
      LinkedList<Formula> formulas = world.getFormulas();
      noFormulas = formulas.size();
      for (k=0; k<noFormulas; k++) {
        Formula formula = formulas.getFirst();
        if (types.contains(formula.getType())) {
          if (!formula.isTicked()) {
            currentWorldId = world.getId();
```

```
                chosenWorld = world;
                chosenFormula = formula;
                break outerLoop;
              }
            }
            formulas.add(formulas.removeFirst());
          }
          worlds.add(worlds.removeFirst());
        }
      }

      if (chosenFormula == null) {
        isExpandable = false;
        return;
      }
      expand(chosenFormula, chosenWorld);
    }

    private void expand(Formula formula, World world) {
      switch (formula.getType()) {

        case NOTTRUE:
        case NOTFALSE:
          expandNegatedBoolean(formula, world);
          break;

        case AND:
        case NOTOR:
        case NOTCONDITION:
        case BICONDITION:
          expandAlpha(formula, world);
          break;
```

```
      case OR:
      case NOTAND:
      case CONDITION:
      case NOTBICONDITION:
        expandBeta(formula, world);
        break;

      case POSSIBLY:
      case NOTNECESSARILY:
        expandDelta(formula, world);
        break;

      case NECESSARILY:
      case NOTPOSSIBLY:
        expandGamma(formula, world);
        break;
    }
  }

  private void expandNegatedBoolean(Formula formula, World world) {
    world.addFormula(formula.getSubformulas().get(0));
    formula.tick();
  }

  private void expandAlpha(Formula formula, World world) {
    world.addFormula(formula.getSubformulas().get(0));
    world.addFormula(formula.getSubformulas().get(1));
    formula.tick();
  }

  private void expandBeta(Formula formula, World world) {
    Formula subformula1 = formula.getSubformulas().get(0);
    Formula subformula2 = formula.getSubformulas().get(1);
```

```
    formula.tick();
    Frame disjunctiveFrame = new Frame(this,
            tableau, tableau.getNewFrameId(), logic);
    world.addFormula(subformula1);
    disjunctiveFrame.addFormula(subformula2);
    tableau.addFrame(disjunctiveFrame);
  }

  private void expandDelta(Formula formula, World world) {
    formula.tick();
    HashSet<Formula> expandingFormulas;
    if (logic.isTransitive()) {
      expandingFormulas =
              world.getTransitiveDeltaExpansionFormulas(formula);
    } else {
      expandingFormulas =
              world.getKripkeDeltaExpansionFormulas(formula);
    }

    World existingWorld;
    if (logic.isSymmetric()) {
      existingWorld =
              worldSymmetricallyCompatible(expandingFormulas, world);
    } else if (logic.isLinear()) {
      existingWorld =
              worldLinearlyCompatible(expandingFormulas, world);
    } else {
      existingWorld = worldContainingFormulas(expandingFormulas);
    }
    if (existingWorld == null) {
      LinkedList<Formula> formulas =
              new LinkedList<>(expandingFormulas);
      worldIdCount++;
```

```
World newWorld = new World(formulas, worldIdCount);
if (logic.isSerial()) {
  Formula fmla1 = new Formula("T");
  Formula fmla2 = new Formula("<>T");
  fmla1.parse();
  fmla2.parse();
  newWorld.addFormula(fmla1);
  newWorld.addFormula(fmla2);
}
worlds.add(newWorld);
world.allCurrentGammaFormulasExpandedTo(newWorld.getId());
if (logic.isLinear()) {
  HashSet<Integer> visitedWorlds = new HashSet<>();
  visitedWorlds.add(world.getId());
  HashSet<World> futureWorlds =
        futureWorlds(world, visitedWorlds);
  HashSet<Formula> necessaryFormulas;
  HashSet<Formula> newWorldGammaFormulas;
  World leftWorld = null;
  World rightWorld = null;
  positionSearch:
  for (World fromWorld : futureWorlds) {
    necessaryFormulas =
          fromWorld.getGammaExpansionFormulas(logic);
    if (!newWorld.containsFormulas(necessaryFormulas)) {
      continue positionSearch;
    }
    World toWorld = linearlyAdjacentWorld(fromWorld);
    newWorldGammaFormulas =
          newWorld.getGammaExpansionFormulas(logic);
    if (toWorld == null
          || !toWorld.containsFormulas(newWorldGammaFormulas)) {
      continue positionSearch;
```

```
      }
      leftWorld = fromWorld;
      rightWorld = toWorld;
    }
    if (!(leftWorld == null)) {
      removeTransition(leftWorld.getId(), rightWorld.getId());
      transitions.add(new Transition(leftWorld.getId(),
            newWorld.getId()));
      transitions.add(new Transition(newWorld.getId(),
            rightWorld.getId()));
    } else {
      World lastWorld = null;
      lastWorldSearch:
      for (World futureWorld : futureWorlds) {
        for (Transition transition : transitions) {
          if (transition.from() == futureWorld.getId()
                && transition.to() != world.getId()) {
            continue lastWorldSearch;
          }
        }
        lastWorld = futureWorld;
        break lastWorldSearch;
      }
      transitions.add(new Transition(lastWorld.getId(),
            newWorld.getId()));
    }
  } else {
    addTransition(new Transition(world.getId(),
          newWorld.getId()));
  }
  if (logic.isSymmetric()) {
    this.addTransition(new Transition(newWorld.getId(),
          world.getId()));
```

```
    }
    if (logic.isReflexive()) {
      this.addTransition(new Transition(newWorld.getId(),
          newWorld.getId()));
    }
  } else {
    if (!logic.isTransitive()) {
      addTransition(new Transition(world.getId(),
          existingWorld.getId()));
    }
  }
}

private void expandGamma(Formula formula, World world) {
  HashSet<World> gammaWorlds;
    if (logic.isTransitive()) {
      HashSet<Integer> visitedWorlds = new HashSet<>();
      gammaWorlds = reachableWorlds(world, visitedWorlds);
    } else {
      gammaWorlds = getGammaWorldsFor(world, formula);
    }
    for (World gammaWorld : gammaWorlds) {
      gammaWorld.addFormula(formula.getSubformulas().get(0));
    }
    if (!expandable()) {
      formula.tick();
    } else {
      untickAllGammaFormulas();
    }
}

private void removeTransition(int from, int to) {
  for (Transition transition : transitions) {
```

```
      if (transition.from() == from && transition.to() == to) {
        transitions.remove(transition);
      }
    }
  }

  private World linearlyAdjacentWorld(World world) {
    for (Transition transition : transitions) {
      if (transition.from() == world.getId()
          && transition.to() != world.getId()) {
        return getWorldWithId(transition.to());
      }
    }
    return null;
  }

  private World worldSymmetricallyCompatible(
      HashSet<Formula> formulas, World currentWorld) {
    worldsLoop:
    for (World newWorld : worlds) {
      for (Formula formula : formulas) {
        if (!newWorld.containsFormula(formula)) {
          continue worldsLoop;
        }
      }
      HashSet<Formula> newWorldFormulas =
          newWorld.getGammaExpansionFormulas(logic);
      for (Formula newWorldFormula : newWorldFormulas) {
        if (!currentWorld.containsFormula(newWorldFormula)) {
          continue worldsLoop;
        }
      }
      return newWorld;
```

```
    }
    return null;
  }

  private HashSet<World> reachableWorlds(World currentWorld,
      HashSet<Integer> visitedWorlds) {
    HashSet<World> reachableWorlds = new HashSet<>();
    for (World world : worlds) {
      for (Transition transition : transitions) {
        if (transition.from() == currentWorld.getId()
            && transition.to() == world.getId()) {
          if (!visitedWorlds.contains(currentWorld.getId())) {
            visitedWorlds.add(currentWorld.getId());
            reachableWorlds.add(world);
            reachableWorlds.addAll(reachableWorlds(world, visitedWorlds));
          }
        }
      }
    }
    return reachableWorlds;
  }

  private HashSet<World> futureWorlds(World currentWorld,
      HashSet<Integer> visitedWorlds) {
    HashSet<World> futureWorlds = new HashSet<>();
    futureWorlds.add(currentWorld);
    reachableWorlds(currentWorld, visitedWorlds);
    return futureWorlds;
  }

  private boolean worldsAreLinearlyCompatible(
      HashSet<Formula> formulas, World toWorld) {
    for (Formula formula : formulas) {
```

```
      if (!toWorld.containsFormula(formula)) {
        return false;
      }
    }
    return true;
  }

  private World worldLinearlyCompatible(
      HashSet<Formula> formulas, World currentWorld) {
    HashSet<Integer> visitedWorlds = new HashSet<>();
    visitedWorlds.add(currentWorld.getId());
    HashSet<World> futureWorlds =
        futureWorlds(currentWorld, visitedWorlds);
    for (World world : futureWorlds) {
      if (worldsAreLinearlyCompatible(formulas, world)) {
        return world;
      }
    }
    return null;
  }

  private World worldContainingFormulas(HashSet<Formula> formulas) {
    worldsLoop:
    for (World world : worlds) {
      for (Formula formula : formulas) {
        if (!world.containsFormula(formula)) {
          continue worldsLoop;
        }
      }
      return world;
    }
    return null;
  }
```

```
private void untickAllGammaFormulas() {
  for (World world : worlds) {
    for (Formula formula : world.getFormulas()) {
      if (formula.getType() == FormulaType.NECESSARILY
          || formula.getType() == FormulaType.NOTPOSSIBLY) {
        formula.untick();
      }
    }
  }
}

private boolean expandable() {
  for (World world : worlds) {
    for (Formula formula : world.getFormulas()) {
      FormulaType type = formula.getType();
      if (type != FormulaType.TRUE && type != FormulaType.FALSE
          && type != FormulaType.NOTPROPOSITION
          && type != FormulaType.PROPOSITION
          && type != FormulaType.NECESSARILY
          && type != FormulaType.NOTPOSSIBLY) {
        if (!formula.isTicked()) {
          return true;
        }
      }
    }
  }
  return false;
}

private HashSet<World> getGammaWorldsFor(World world,
      Formula formula) {
  HashSet<World> gammaWorlds = new HashSet<>();
```

```
    for (Transition transition : transitions) {
      if (transition.from() == world.getId() &&
            !formula.getWorldsExpandedTo().contains(transition.to())) {
        gammaWorlds.add(getWorldWithId(transition.to()));
        formula.addWorldExpandedTo(transition.to());
      }
    }
    return gammaWorlds;
  }

  public boolean hasContradiction() {
    for (World world : worlds) {
      if (world.hasContradiction()) {
        return true;
      }
    }
    return false;
  }

  private LinkedList<World> cloneWorlds() {
    LinkedList<World> clonedWorlds = new LinkedList<>();
    for (World world : worlds) {
      clonedWorlds.add(world.clone());
    }
    return clonedWorlds;
  }

  private HashSet<Transition> cloneTransitions() {
    HashSet<Transition> clones = new HashSet<>();
    for (Transition transition : transitions) {
      Transition clone =
            new Transition(transition.from(), transition.to());
      clones.add(clone);
```

```
    }
    return clones;
  }

  private void addFormula(Formula formula) {
    getWorldWithId(currentWorldId).addFormula(formula);
  }

  private void addTransition(Transition transition) {
    if (transitionExists(transition)) {
      return;
    }
    transitions.add(transition);
  }

  private boolean transitionExists(Transition newTransition) {
    for (Transition transition : transitions) {
      if (transition.from() == newTransition.from()
          && transition.to() == newTransition.to()) {
        return true;
      }
    }
    return false;
  }

  private World getWorldWithId(int id) {
    for (World world : worlds) {
      if (world.getId() == id) {
        return world;
      }
    }
    return null;
  }
```

```
  public String getTransitionsFrom(int id) {
    ArrayList<Integer> accessibleWorlds = new ArrayList<>();
    for (Transition transition : transitions) {
      if (transition.from() == id) {
        accessibleWorlds.add(transition.to());
      }
    }
    return accessibleWorlds.toString();
  }

  public int getWorldIdCount() {
    return worldIdCount;
  }

  public int getId() {
    return id;
  }

  public boolean isExpandable() {
    return isExpandable;
  }
}

public class World {

  private final int id;
  private LinkedList<Formula> formulas;

  public World(LinkedList<Formula> formulas, int id) {
    this.formulas = new LinkedList<>();
    LinkedList<Formula> clonedFormulas = cloneFormulas(formulas);
    for (Formula formula : clonedFormulas) {
```

```
      addFormula(formula);
    }
    this.id = id;
  }

  public boolean hasContradiction() {

    HashSet<String> propositions = new HashSet<>();
    HashSet<String> negatedPropositions = new HashSet<>();
    for (Formula formula : formulas) {
      if (formula.getType() == FormulaType.FALSE) {
        return true;
      }
      if (formula.getType() == FormulaType.PROPOSITION) {
        propositions.add(formula.getString());
      } else if (formula.getType() == FormulaType.NOTPROPOSITION) {
        negatedPropositions.add(formula.getString().substring(1));
      }
    }
    propositions.retainAll(negatedPropositions);
    if (!propositions.isEmpty()) {
      return true;
    }
    return false;
  }

  public void allCurrentGammaFormulasExpandedTo(int worldId) {
    for (Formula formula : formulas) {
      if (formula.getType() == FormulaType.NECESSARILY
          || formula.getType() == FormulaType.NOTPOSSIBLY) {
        formula.addWorldExpandedTo(worldId);
      }
    }
```

```
}

@Override
public World clone() {
   World clone = new World(formulas, id);
  return clone;
}

private LinkedList<Formula> cloneFormulas(
      LinkedList<Formula> formulas) {
  LinkedList<Formula> clonedFormulas = new LinkedList<>();
  for (Formula formula : formulas) {
    clonedFormulas.add(formula.clone());
  }
  return clonedFormulas;
}

public boolean containsFormulas(HashSet<Formula> formulas) {
  for (Formula formula : formulas){
    if (!containsFormula(formula)) {
      return false;
    }
  }
  return true;
}

public boolean containsFormula(Formula newFormula) {
  for (Formula formula : formulas){
    if(newFormula.getString().equals(formula.getString())) {
      return true;
    }
  }
  return false;
```

```
}

public void addFormula(Formula newFormula) {
  if (containsFormula(newFormula)) {
    return;
  }
  formulas.add(newFormula.clone());
}

public void eliminateFormula(Formula deadFormula) {
  int formulasCount = 0;
  int formulasSize = formulas.size();
  while(formulasCount != formulasSize) {
    Formula formula = formulas.getFirst();
    formulasCount++;
    if(formula.getString().equals(deadFormula.getString())) {
      formulas.removeFirst();
    } else {
      formulas.add(formulas.removeFirst());
    }
  }
}

public HashSet<Formula> getTransitiveDeltaExpansionFormulas(
      Formula deltaFormula) {
  HashSet<Formula> transitiveFormulas = new HashSet<>();
  transitiveFormulas.add(deltaFormula.getSubformulas().get(0));
  for (Formula formula : formulas) {
    if (formula.getType() == FormulaType.NECESSARILY
          || formula.getType() == FormulaType.NOTPOSSIBLY) {
      transitiveFormulas.add(formula);
      transitiveFormulas.add(formula.getSubformulas().get(0));
    }
```

```
    }

    return transitiveFormulas;
  }

  public HashSet<Formula> getKripkeDeltaExpansionFormulas(
      Formula deltaFormula) {
    HashSet<Formula> kripkeFormulas = new HashSet<>();
    kripkeFormulas.add(deltaFormula.getSubformulas().get(0));
    for (Formula formula : formulas) {
      if (formula.getType() == FormulaType.NECESSARILY
          || formula.getType() == FormulaType.NOTPOSSIBLY) {
        kripkeFormulas.add(formula.getSubformulas().get(0));
      }
    }

    return kripkeFormulas;
  }

  public HashSet<Formula> getGammaExpansionFormulas(ModalLogic logic) {
    HashSet<Formula> gammaFormulas = new HashSet<>();
    for (Formula formula : formulas) {
      if (formula.getType() == FormulaType.NECESSARILY
          || formula.getType() == FormulaType.NOTPOSSIBLY) {
        if (logic.isTransitive()) {
          gammaFormulas.add(formula);
        }
        gammaFormulas.add(formula.getSubformulas().get(0));
      }
    }

    return gammaFormulas;
  }
```

```
  public LinkedList<Formula> getFormulas() {
    return formulas;
  }

  public int getId() {
    return id;
  }
}
```