Tableau-based decision procedure for linear time temporal logic: implementation, testing, performance analysis and optimisation

Angelo Kyrilov

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science

Johannesburg, 2011

Declaration

I declare that this is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before any degree or examination in any other University.

Angelo Kyrilov
7^{th} day of April 2011

Abstract

This thesis reports on the implementation and experimental analysis of an incremental multi-pass tableau-based procedure à la Wolper for testing satisfiability in the linear time temporal logic LTL, based on a breadth-first search strategy. I describe the implementation and discuss the performance of the tool on several series of pattern formulae, as well as on some random test sets, and compare its performance with an implementation of Schwendimann's one-pass tableaux by Widmann and Goré on several representative series of pattern formulae, including eventualities and safety patterns. The experiments have established that Schwendimann's algorithm consistently, and sometimes dramatically, outperforms the incremental tableaux, despite the fact that the theoretical worst-case upper-bound of Schwendimann's algorithm, 2EXPTIME, is worse than that of Wolper's algorithm, which is EXPTIME. This shows, once again, that theoretically established worst-case complexity results do not always reflect truly the practical efficiency, at least when comparing decision procedures.

Acknowledgements

I would like to thank my supervisor, Prof. Valentin Goranko, for all his help and guidance throughout this project.

I would like to thank my co-supervisor, Dr Dmitry Shkatov, for his invaluable input.

I am greatly indebted to Rajeev Goré and Florian Widmann for providing me with their implementation of Shcwendimann's one-pass tableau as well as many pieces of valuable advice.

I am also extremely grateful to Mayya Tokman for all her support and encouragement, without which this work would not have been possible.

Last but not least, I would like to thank my co-workers and my family for putting up with me during this work and for always supporting me.

To Mervyn Curtis

Contents

List of Figures

Chapter 1

Introduction

As the current increase in computing power continues, systems are becoming tremendously large and complex. This has allowed computer systems to be used for new applications that were not possible before. In some applications, such as heart surgery performed by robots, it is crucial that the system is completely free of bugs and malfunctions. Therefore now, more than ever, there is a need for us to make sure we build systems correctly. Engineers, who build these systems, ask themselves two questions during the development cycle. The first one is "Have we built the product right?". The second one is "Have we built the right product?". These questions correspond to the notions of verification and validation respectively. Validation is concerned with the needs of end users and whether the product is suitable for the intended purpose. Validation is an important activity because products change or evolve during the development cycle. It is important to make sure that as the changes are occurring, the product remains suitable for the purposes it was intended to fulfill. Verification is ensuring that the product works correctly.

1.1 Traditional verification

Verification is an important phase of the development process. It assures the developers and the users that the product meets the specification according to which it was designed. Verification also attempts to prove that the product works correctly and is of certain quality. Traditional methods for verification include testing and simulation.

Testing is a procedure whereby the finished product is run through a series of pre-designed test cases. This aims to show that the product operates correctly under different possible scenarios. One disadvantage of this approach is that it can only begin when the product has been built. This means that bugs will only be detected after completion of the work and developers need to go back, revise the work and then test again. This is an expensive process since changes to the fundamental design may be required in order to remove a bug.

This means that all the phases of the development cycle have to be repeated. The new change, which removed certain bugs, may introduce other bugs so all previous test results are nullified and have to be repeated. Furthermore it is difficult to design test cases that will cover all possible runtime situations. The developers would have to predict all possible scenarios where the product could malfunction and design good test cases. For very large and complex systems, this task is next to impossible. Developers run extensive tests but some bugs remain undetected even after the product has shipped. These bugs are discovered by the end users of the product and are even more expensive to fix compared to bugs discovered during the testing phase.

Simulation refers to the modelling of a real-life system on a computer. This process allows the developers to run the system without having to physically build it. This is more applicable to the design of hardware products. Simulation is used to show the state of the product after completing certain operations. A disadvantage of simulation is that it is an expensive process, which requires a lot of time. Finally, since it models only key characteristics of the system it may not be able to detect all the problems.

Both approaches have many limitations which could lead to shipping products that do not operate according to the specification because of the presence of bugs. One such bug, which is difficult to detect is deadlock in a system. Deadlock is a state where no further actions can be taken. Typically it occurs in concurrent systems that are sharing resources, where several processes could end up waiting for each other indefinitely without proceeding. It is difficult to detect this kind of situation because it may only occur under special circumstances. Unless these exact circumstances are tested explicitly the presence of deadlock may be overlooked. It is also difficult to detect whether a system satisfies any liveness properties. A liveness property promises that something will eventually happen in the future. For example, in a printing system, a reasonable liveness property is to guarantee that if a document is sent to the printer, it will eventually be printed. Lastly fairness conditions, which ensure that every process gets a fair share of resources, are hard to detect. In a general computer system, a fairness condition would be that every process must get allocated to the CPU for a certain time. Early systems did not meet that condition and low priority processes never got the chance to use the CPU because higher priority processes were constantly coming in. Testing whether a system satisfies fairness conditions would require very extensive testing and it would consume a lot of resources. Given the importance of verifying systems correctly, especially large, performance critical systems, it became necessary to develop better methods to do this.

1.2 Formal methods for verification

Formal verification refers to the process of using formal or mathematical methods to prove the correctness of systems. Formal methods have proved to be good alternatives for the traditional methods of verification because they overcome the problems of the traditional approaches discussed above. The increased use of formal methods for verification has come about because of the ever-growing size and complexity of systems being built. A single computer processor is made up of millions of transistors and commercial software applications are made up of millions of lines of code. Even the most extensive testing procedures could leave potential problems undetected because of the immense size of the systems being tested. Testing and debugging makes up a large percentage of the total development time of a system. This percentage grows rapidly as systems get bigger. The need to verify complex systems by formal methods led to the development of proof procedures in the late 1960s. Amongst these were dynamic logics and fixpoint calculi.

1.3 Temporal Logic

Temporal logic stems from philosophical analysis of time and temporality, first introduced as formal logical system by Prior [10]. In a seminal paper [9], Pnuelli proposed the use of temporal logic for specification and verification of properties of reactive and concurrent systems. Since then temporal logic has found many more applications in computer science and artificial intelligence, including: analysis and verification of real-time processes and systems, hardware verification, synchronisation of concurrent processes, temporal databases, etc., see [2].

A major application of temporal logic in computer science is *model checking*. Model checking is testing whether a given program or a transition system satisfies a property specified by a temporal logic formula. This is a method for formal verification because important types of properties, such as *safety, liveness, and fairness conditions*, can be expressed as temporal logic formulae, see [1, 5, 6]. Safety properties ensure that the system will not crash or exhibit unwanted behaviour; liveness properties prevent the possibility of deadlocks, while fairness conditions ensure that each process gets a fair access to shared resources. Depending on the type of systems and properties to specify and verify, two major types of temporal logics have been used: *linear* and *branching time logics*. This research focuses on linear time logic. Checking whether the program satisfies the specification is equivalent to checking whether the structure, which represents the program, is a model for the formula expressing the specification. There are procedures that reduce model checking to validity testing, see [3] for an example of this. In turn, validity testing is analogous to

satisfiability testing, therefore the process of model checking can be reduced to satisfiability testing.

1.4 Related Work

Model checking and satisfiability testing of temporal specifications are computationally demanding tasks that require efficient algorithms. Two of the major techniques for solving these computational tasks are based on *automata* [13, 4] and on *semantic tableaux* [14]. Because of the conceptual elegance and technical power and convenience, the automata-based methods have been favoured by the researchers in the area and most of the satisfiability checking tools that have been implemented are based on automata. However, automata-based methods are not very flexible and are often computationally too expensive. Recent extensive testing of automata-based tools done by Rozier and Vardi [11] shows that few of them really satisfy the efficiency requirements arising from practical applications.

Part of this research was to implement a tableau-based decision procedure. The implementation will be based largely on the work of Wolper in [14]. Tableau methods for temporal logic are based on the methods for propositional logic. The temporal formulae at each state of the tableau are decomposed into formulae that represent the requirements for the current state and requirements for future states. The tableau procedure consists of constructing a directed graph. Each node of the graph is labelled with a set of formulae. At first there is only one node labelled only with the formula we are testing. The construction phase of the tableau then begins. It involves adding new nodes to the graph while decomposing the formula according to the construction rules described in [14]. The construction stage ends when the application of a construction rule does not lead to the creation of a new state. The finiteness of the tableau is guaranteed because we identify nodes. That is when a new node gets generated, we first check whether it already exists in the tableau and if it does not, it gets added. The extended closure of a formula in LTL tells us that there are only a finite number of sub-formulae, which means that we can not go on forever generating new states that do not already appear in the tableau.

The elimination phase of the procedure involves removing, from the already constructed graph, all the nodes that are inconsistent and all the nodes that have no successors. Inconsistent nodes are these which contain contradictory information. Also all states in a LTL structure are required to have successors, so all nodes without successors are removed. Finally we eliminate all nodes that contain unsatisfied eventualities. An eventuality is a formula which "promises" that something will happen in the future. To check if a certain eventuality is satisfied from a state s, we need to look at all the successor states of s and see if that which was 'promised' has been fulfilled. All states that contain

unfulfilled eventualities are removed. Finally if the initial states have not been removed from the tableau after the elimination phase is over, then we declare the formula to be satisfiable. Otherwise the formula is not satisfiable. For a more detailed description of the procedure please see [14].

In [12], Schwendimann presented a one-pass tableau procedure for LTL. The main advantage of this method is that there is no elimination phase, thus the name "one-pass". The checks for unfulfilled eventualities are performed during the construction phase. The procedure generates a tree instead of a graph, which means that only one branch of the tree needs to be kept in memory while the procedure is running. This also opens the door to possible parallelisation of the decision procedure.

Another important goal of this research was to test the performance of the implemented tableau procedure and compare it to other tools available. Rozier and Vardi have presented useful information about the performance of automata based tools in [11]. Their paper also provides LTL formula patterns that can be used for measuring the performance of a tool.

1.5 Outline of document

The rest of this dissertation is organised as follows. Chapter 2 presents background information on transition systems and LTL. It is useful to introduce transition systems and show how they are used to specify properties for programs and how LTL can be used to verify these properties. The syntax and semantics of LTL are also given. Chapter 3 introduces tableaux systems for LTL. The entire procedure is explained, including proofs of termination, soundness and completeness. This is followed by chapter 4, where the implementation is described. Each part of the procedure presented in chapter 3 is converted into Python code, so all the low level details are explained. Chapter 5 shows the different optimisation techniques that were used to make the implementation more efficient. Chapter 6 presents the results of the various tests that were carried out and finally, chapter 7 contains some concluding remarks and plans for future work.

Chapter 2

Theoretical Background

2.1 Introduction

This chapter presents the theoretical background necessary for this research. Before going into the details of the decision procedure that is to be implemented and tested, it is useful to introduce transition systems, which are tools for representing the execution of programs. We then show how certain properties of transition systems can be formulated. These include liveness and safety properties amongst others. Linear Time Temporal Logic (LTL) is then introduced as a tool for writing and verifying such properties on transition systems. The chapter begins with a formal definition of transition systems. This is followed by detailed descriptions of different types of transition systems as well as definitions of important terminology. Next we show how to specify liveness, fairness and safety properties for transition systems. Linear time temporal logic is then introduced. The syntax and the semantics of LTL are described. The chapter ends with some examples.

2.2 Transition Systems

A typical transition system consists of a set of states and a relation that defines transitions between the states. Transition systems are used to model processes. These processes can be sequential or concurrent. The states in a transition system can represent program states or memory registers whereas the transitions represent program instructions and other actions. It is useful to label the transitions because there are different types of actions or processes that can cause a transition from one state to another. Such transition systems are called *labelled* transition systems. They are defined as follows.

Definition 2.1. A labelled transition system is a structure

$$T = (S, A)$$

that consists of:

- a set of states $S \neq \emptyset$;
- a set $A \neq \emptyset$ of transitions (called the *signature* of T)
- a binary transition relation subset of $S \times S$.

Some transition systems have an initial state specified. Such systems are called *rooted* transition systems or *initialised* transition systems. There are also other types of states besides initial states. There are also terminal, accepting, deadlock, safe and unsafe states. These properties of states are described with a suitable description language. One such language is propositional logic. Every state has a set of propositions that are true at that state. This set is called the description of the state. Such a transition system is called an *interpreted* transition system. The formal definition is as follows.

Definition 2.2. An interpreted transition system is a pair $M = (T, L)$, where T is a transition system of some signature A, PROP is a fixed set of atomic propositions, and $L : S \to \mathbf{2}^{\mathrm{PROP}}$ is a state description, which assigns to every state Δ the set of atomic propositions that are true at Δ.

In relation to modal and temporal logics, transition systems are Kripke frames and interpreted transition systems are Kripke models. State descriptions are called truth assignments. In classical modal and temporal logics, sometimes *valuations* are used instead of truth assignments. A valuation $V : \mathrm{PROP} \to \mathbf{2}^S$ assigns to each atomic propositions a set of states where it is true.

Definition 2.3. A path in a transition system is a (finite or infinite) sequence of states, separated by actions that transform each state into its successor.

$$\Delta_0 \to \Delta_1 \to \Delta_2 \dots$$

A path $\pi = \Delta_0 \to \ldots \to \Delta_n$ is said to be *rooted* at Δ_0 and its length is denoted by $|\pi| = n$. A path is said to be *maximal* if it is either infinite or is finite and ends in a state with no successors. Sometimes the terms *run* or *execution* are used as synonyms of *path*.

Definition 2.4. A computation or a trace, in an interpreted transition system (T, L), is a (finite or infinite) sequence of state descriptions separated by actions.

$$L(\Delta_0) \to L(\Delta_1) \to L(\Delta_2) \dots$$

Intuitively a computation is the observable effect of the actions along a path. It is a record of all the successive intermediate results of the computation process. Sometimes the words *computation* and *trace* are used interchangeably. A finite computation is called *terminating*. A terminating computation can always be converted to a non-terminating computation by appending an infinite repetition of an idle state to the end of the computation.

Examples of transition systems include finite automata and Turing machines. These are abstract transition systems. Concrete examples include vending machines, clocks and elevators. An example of a vending machine can be seen in figure 2.1. This is a simplified model whereby the customer has to put in the exact amount of money and there is only one type of drink available. There are two coin denominations, namely 25 and 50 units. A drink costs 100 units. A customer is allowed to insert one of the two coin denominations at a time. The machine moves to different states to reflect the total amount of money inserted so far. When the amount reaches 100 units a drink is dispensed and the total amount is reset to 0.

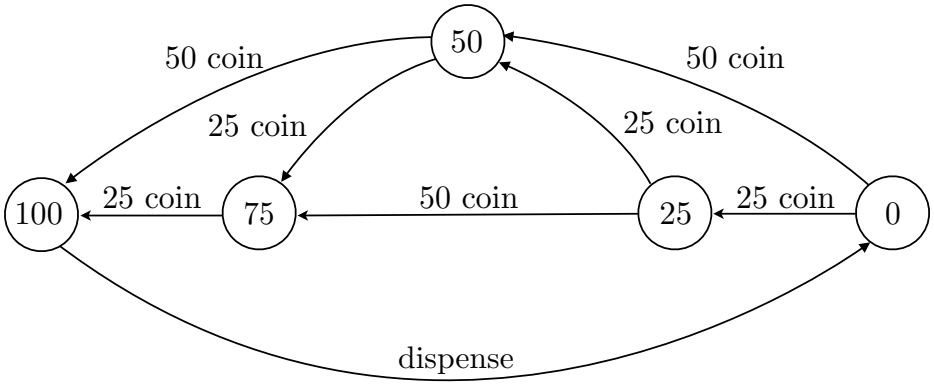

Figure 2.1: A simple vending machine

There are various properties that we would like to express. These include safety properties, liveness properties and fairness conditions. Safety properties promise that nothing bad will ever happen in the system. In the case of a vending machine a safety property can be *a drink is never dispensed unless the customer has deposited enough money.* Safety properties are related to reachability of states. If a formula γ describes a scenario where a drink is dispensed before the correct amount of money has been deposited, and if γ is true at some state Δ then Δ is said to be unsafe. In order for the whole system to be safe, the unsafe states can not be reachable on any computation. Fairness properties guarantee that the system must periodically pass through a stage that ensures

fairness. In the vending machine example a fairness property could be that *there will always be new customers visiting the vending machine*. Another way of stating this is that customers will visit the vending machine *infinitely often*. Liveness properties ensure that progress is made or that something good will eventually happen. With vending machines a liveness property could be that *after the customer has deposited the correct amount of money, a drink will be dispensed*.

To state these and other properties formally, it is necessary to first introduce a suitable formal language. The following section is an introduction to Linear Time Temporal Logic, the language that would allow us to express the properties stated above.

2.3 Linear Time Temporal Logic

This section describes syntax and semantics of Linear Time Temporal Logic. It is based on lecture notes by Goranko, found in [3]. Temporal logics contain modalities with a temporal interpretation. A modality is an object that modifies the relationship between a predicate and a subject. For example, in the sentence "Eventually, we will get there", the term "Eventually" is a temporal modality. Temporal modalities (also known as temporal operators) allow one to reason about the sequencing of states along a path, rather than about states taken individually. Formally, the temporal operators are `X` (next), `F` (sometime in the future), `G` (always in the future) and `U` (until). The usual boolean operators, $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction) and $\Rightarrow$ (implication), are also used.

2.3.1 Syntax of LTL

The operator `X` is a unary operator. It is used to specify properties not for the current state but for the next state of a path. For example if the formula $\texttt{X}\varphi$ is true at state Δ_i, it means that state Δ_{i+1} has the property φ. This is depicted in the Figure 2.2.

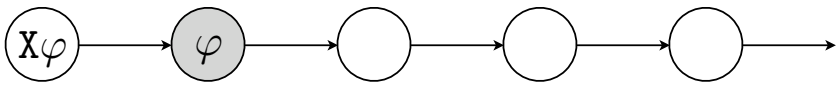

Figure 2.2: The `X` (next time) operator

The operator `F` is a unary operator. It is used to specify properties for some future state, further down the execution path. For example the if the

formula $\mathtt{F}\varphi$ is true at a state Δ, it means that Δ or some successor of Δ has the property φ. The operator does not specify exactly which successor will have that property. It only promises that eventually something will happen. This can be seen visually in Figure 2.3

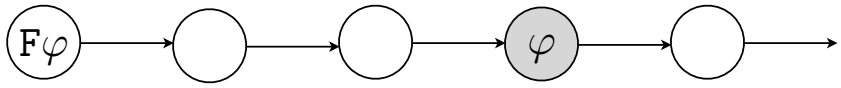

Figure 2.3: The `F` (sometime in the future) operator

The operator `G` is a unary operator. It is used to specify properties for the current state and all its successors. For example if the formula $\mathtt{G}\varphi$ is true at a state Δ, it means that Δ has the property φ and all the successors of Δ also have the property φ. The operator `G` is the *dual* of `F` so the formula $\mathtt{G}\varphi$ can be equivalently rewritten as $\neg\mathtt{F}\,\neg\varphi$. Figure 2.4 shows this operator.

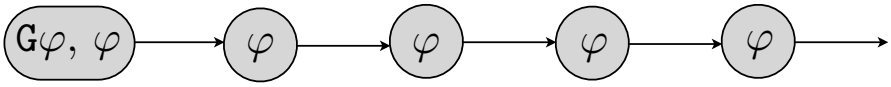

Figure 2.4: The `G` (always in the future) operator

The operator `U` is a binary operator. It is richer and more complicated than the other temporal operators. For example the formula $\varphi\ \mathtt{U}\ \psi$ states that φ will be true until ψ becomes true. That is ψ will be true at some time in the future but until that time φ will be true. Figure 2.5 shows this behaviour visually.

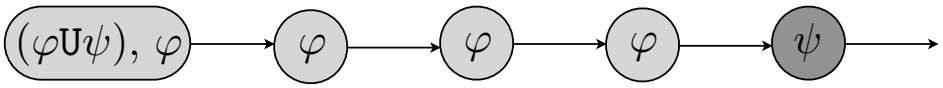

Figure 2.5: The `U` (until) operator

LTL formulae are obtained from the grammar in Figure 2.6, where p is an element of the countably infinite set PROP of propositional variables.

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \mathtt{X}\varphi \mid \mathtt{F}\varphi \mid \mathtt{G}\varphi \mid \varphi \ \mathtt{U} \ \psi$$

Figure 2.6: Grammar for LTL formulae

The set of sub-formulae of a formula φ is denoted by $sub(\varphi)$ and the length of the formula φ is denoted by $|\varphi|$. LTL models are computations, that is executions viewed as ω-sequences. The reason why LTL models are infinite objects is because they allow us to specify limit behaviours, which would not be possible with a finite object. So a model for LTL is an infinite sequence $\sigma : \mathbb{N} \to P(PROP)$. Figure 2.7 shows an example of the first few states of an LTL model.

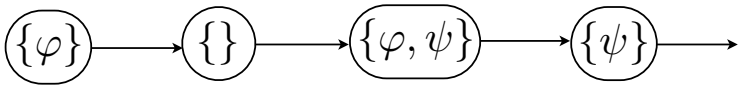

Figure 2.7: An example of an LTL model

2.3.2 Semantics of LTL

Definition 2.5. Given a LTL model $\sigma : \mathbb{N} \to \mathcal{P}(\text{PROP})$, a position $i \in \mathbb{N}$ and a formula φ, the satisfaction relation $\models$ is defined as follows:

- $\sigma, i \models \top$,
- $\sigma, i \not\models \bot$,
- $\sigma, i \models p \Leftrightarrow p \in \sigma(i)$, for every $p \in \text{PROP}$,
- $\sigma, i \models \neg\varphi \Leftrightarrow \sigma, i \not\models \varphi$,
- $\sigma, i \models \varphi \wedge \psi \Leftrightarrow \sigma, i \models \varphi$ and $\sigma, i \models \psi$,
- $\sigma, i \models \varphi \vee \psi \Leftrightarrow \sigma, i \models \varphi$ or $\sigma, i \models \psi$,
- $\sigma, i \models \mathtt{X}\varphi \Leftrightarrow \sigma, i+1 \models \varphi$,
- $\sigma, i \models \mathtt{F}\varphi \Leftrightarrow$ there is $j \geq i$, such that $\sigma, j \models \varphi$,
- $\sigma, i \models \mathtt{G}\varphi \Leftrightarrow$ for all $j \geq i$, we have $\sigma, j \models \varphi$,
- $\sigma, i \models \varphi \ \mathtt{U} \ \psi \Leftrightarrow$ there is $j \geq i$, such that $\sigma, j \models \psi$ and $\sigma, k \models \varphi$ for all $i \leq k < j$.

Given a set of temporal operators $\mathcal{O} \subseteq \{\mathtt{X},\mathtt{F},\mathtt{G},\mathtt{U}\}$, we write LTL($\mathcal{O}$) to denote LTL formulae that are restricted to using only temporal connectives from $\mathcal{O}$.

Definition 2.6. We write $\mathrm{LTL}_n^k(\mathtt{O}_1, \mathtt{O}_2, \ldots)$ to denote the fragment of LTL restricted to formulae such that

- the temporal operators are from $\mathtt{O}_1, \mathtt{O}_2, \ldots$,
- the temporal depth is bounded by k,
- at most n distinct propositional variables occur.

The temporal depth of a formula is the maximal nesting of temporal operators. For example the temporal depth of the formula $\mathtt{GF}\varphi$ is 2. If k takes on the value ω then there is no restriction on the temporal depth. Similarly if n takes on the value of ω then the number of propositional variables is not limited. So for example $\mathrm{LTL}_4^2(\mathtt{G}, \mathtt{F})$ represents the set of LTL formulae that are restricted to use only the G and F operators, with nesting level of 2 and at most 4 distinct propositional variables.

Two LTL formulae φ and ψ are said to be *equivalent* if for all models σ and positions i, we have $\sigma, i \models \varphi$ if and only if $\sigma, i \models \psi$. This equivalence is denoted by $\varphi \equiv \psi$. Two formulae, φ and ψ are *initially equivalent* if for all models σ, we have $\sigma, 0 \models \varphi$ if and only if $\sigma, 0 \models \psi$. For example we have $\mathtt{F}\varphi \equiv (\top\ \mathtt{U}\ \varphi)$ and $\mathtt{G}\varphi \equiv \neg(\top\ \mathtt{U}\ \neg\varphi)$. This example shows that the set of temporal operators is not minimal in terms of expressiveness since the F and G operators can be expressed in terms of the U operator. Having F and G operators however provides handy notations that allow us to write more readable and concise formulae.

Models for a formula φ can be viewed as a language $L(\varphi)$ over the alphabet $\mathcal{P}(\mathrm{PROP}(\varphi))$, where $\mathrm{PROP}(\varphi)$ denotes the set of propositional variables occurring in φ:

$$L(\varphi) = \{\sigma \in (\mathcal{P}(\mathrm{PROP}(\varphi)))^\omega | \sigma \models \varphi\}$$

The formula φ is *satisfiable* if $L(\varphi) \neq \emptyset$. This leads to the satisfiability problem for LTL, denoted by SAT(LTL), being stated as follows:

Input: an LTL formula, φ,

Question: Is there some model σ such that $\sigma \models \varphi$

Similarly a formula φ is *valid* if $L(\neg\varphi) = \emptyset$. Thus we have the validity problem for LTL, denoted by VAL(LTL), stated as follows:

Input: an LTL formula, φ,

Question: Is it the case that for all models σ, we have $\sigma \models \varphi$

2.4 Properties of Transition Systems

Section 2.2 ended off with some examples of properties of a vending machine. This section presents these and other properties stated formally, in LTL.

2.4.1 Safety Properties

In a transition system T, a state Δ is said to be *reachable* if there is a computation from the initial state of T to the state Δ. If a formula, say φ, expresses an undesirable property and φ is true at a state u then u is an unsafe state. The reachability of u is represented by the temporal formula $\mathtt{F}\varphi$. So the total safety of the system is represented by the non-reachability of u, that is the formula $\mathtt{G}\neg\varphi$. The unsafe property, identified in section 2.2, was to dispense a drink before the customer has deposited the correct amount of money. That can be represented with the formula $dispense \wedge money < price$, where *dispense* is a boolean variable, *money* is the amount of money the customer has deposited and *price* is the price of the drink. So the safety property can be expressed as:

$$\mathtt{G}(\neg dispense \vee money = price)$$

Another safety property can be that the machine is never empty. This is expressed with the following formula:

$$\mathtt{G}(has\_drink)$$

A more refined version of that property is that whenever there is a customer then the machine must contain a drink. This is expressed with the following LTL formula:

$$\mathtt{G}(customer \neq none \Rightarrow has\_drink)$$

2.4.2 Fairness Properties

Fairness is a constraint imposed on the transition system that requires the system to pass through a state that satisfies a certain property. In the vending machine example, the fairness constraint formulated in section 2.2 was that there would always be customers at the vending machine, that is customers will visit the machine infinitely often. This property is expressed by the following formula:

$$\mathtt{GF}customer \neq none$$

2.4.3 Liveness Properties

In many systems, where multiple processes operate, certain processes usually send requests that need to be acknowledged by other processes. A liveness property states that every request is eventually acknowledged or responded to. The liveness property identified for a vending machine was that after the customer has paid, he/she will eventually get a drink. This can be expressed by the following formula:

$$\mathtt{G}(request \Rightarrow \mathtt{F}dispense)$$

This chapter introduced LTL as a suitable language for specifying properties, such as liveness, fairness and safety, for transition systems. The syntax and semantics of LTL were presented and important terminology was defined. The chapter ended off with statements of the satisfiability and validity problems for LTL. The tableau-based tool implemented as part of this research solves these two problems. The theoretical description of the tool is given in the next chapter and the description of the actual implementation appears in chapter 4.

Chapter 3

Tableaux for LTL

3.1 Introduction

This chapter presents a procedure for testing LTL formulae for satisfiability. The method is based on the semantic tableau method for propositional logic, introduced in [14]. The tableau takes as input a LTL formula and returns *satisfiable* if the formula is satisfiable, otherwise it returns *not satisfiable*. The input formula may contain all the boolean and temporal connectives. This chapter proceeds to explain decomposition rules for LTL formulae. This is followed by a description of Hintikka structures for LTL. The details of the tableau procedure are then presented. The proofs for soundness and completeness are also given.

3.2 LTL Formulae

Given a formula in LTL it can be classified as *elementary* or *non-elementary*. If an elementary formula is true in some state Δ of a model, it can not be further decomposed to reveal any additional information about Δ. For example if $\mathtt{X}\varphi$ is true at a state Δ, there is no further information about Δ, only about the successor of Δ. On the other hand if the formula $\mathtt{G}\varphi$ is true at Δ then φ is true at Δ and $\mathtt{XG}\varphi$ is true at Δ. Formally, elementary formulae are defined as follows:

Definition 3.1. Let φ be a formula in LTL. Then φ is elementary if it one of the following:

- $p \in \mathtt{AP}$ (atomic proposition);
- $\neg p$ for some $p \in \mathtt{AP}$;
- $\top$;

- $\mathtt{X}\psi$ for some formula ψ.

Otherwise φ is non-elementary.

All non-elementary formulae are further divided into two categories. These are α-formulae and β-formulae. The former are conjunctive formulae while the latter are disjunctive.

An α-formula is true at a state Δ of a model $\mathcal{M}$ if and only if two other formulae, namely the conjuncts α_1 and α_2 are true at Δ. For example $\mathtt{G}\varphi$ is true at Δ if and only if φ is true at Δ and $\mathtt{XG}\varphi$ is true at Δ. Figure 3.1 lists all α-formulae and their conjuncts.

α	α_1	α_2
$\neg\neg\varphi$	φ	φ
$\varphi \wedge \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$
$\neg(\varphi \Rightarrow \psi)$	φ	$\neg\psi$
$\neg\ \mathtt{X}\ \varphi$	$\mathtt{X}\neg\varphi$	$\mathtt{X}\neg\varphi$
$\mathtt{G}\varphi$	φ	$\mathtt{XG}\varphi$
$\neg\mathtt{F}\varphi$	$\neg\varphi$	$\neg\mathtt{XF}\varphi$

Figure 3.1: α-formulae and their conjuncts

A β-formula is true at a state Δ of a model $\mathcal{M}$ if and only if either β_1 is true at Δ or β_2 is true at Δ. For example the formula $\mathtt{F}\varphi$ is true at Δ if and only if φ is true at Δ or $\mathtt{XF}\varphi$ is true at Δ. This shows that β-formulae are disjunctive and β_1 and β_2 are the disjuncts. Figure 3.2 lists all β-formulae and their disjuncts.

β	β_1	β_2
$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$
$\varphi \vee \psi$	φ	ψ
$\varphi \Rightarrow \psi$	$\neg\varphi$	ψ
$\varphi\ \mathtt{U}\ \psi$	ψ	$\varphi\wedge\mathtt{X}(\varphi\ \mathtt{U}\ \psi)$
$\neg(\varphi\ \mathtt{U}\ \psi)$	$\neg\varphi \wedge \neg\psi$	$\neg\psi \wedge \neg\mathtt{X}(\varphi\ \mathtt{U}\ \psi)$
$\neg\mathtt{G}\varphi$	$\neg\varphi$	$\neg\mathtt{XG}\varphi$
$\mathtt{F}\varphi$	φ	$\mathtt{XF}\varphi$

Figure 3.2: β-formulae and their conjuncts

3.3 Hintikka Structures for LTL

The tableau procedure for LTL is related to Hintikka structures for LTL, which are analogous to Hintikka structures for modal logic.

Definition 3.2. A Hintikka Structure for an LTL formula ϕ is a tuple $\mathcal{H} = (R, S, H, \sigma)$, where $S \neq \emptyset$, R is a serial binary relation on S, H is a labelling of the members of S and $\sigma : \mathbb{N} \longmapsto S$ is a state sequence on S where,

(H0) $\phi \in H(\sigma(n))$ for some $n \in \mathbb{N}$;

(H1) if $\neg\varphi \in H(\sigma(n))$, then $\varphi \notin H(\sigma(n))$;

(H2) if $\alpha \in H(\sigma(n))$, then $\alpha_1 \in H(\sigma(n))$ and $\alpha_2 \in H(\sigma(n))$;

(H3) if $\beta \in H(\sigma(n))$, then $\beta_1 \in H(\sigma(n))$ or $\beta_2 \in H(\sigma(n))$;

(H4) if $\mathtt{X}\varphi \in H(\sigma(n))$, then $\varphi \in H(\sigma(n+1))$;

(H5) if $\varphi \ \mathtt{U} \ \psi \in H(\sigma(n))$, then for some $i \geq n, \psi \in H(\sigma(i))$, and for every $n \leq j < i$, $\varphi \in H(\sigma(j))$;

(H6) if $\mathtt{F}\varphi \in H(\sigma(n))$, then for some $i \geq n, \varphi \in H(\sigma(i))$;

(H7) if $\neg\mathtt{G}\varphi \in H(\sigma(n))$, then for some $i \geq n, \neg\varphi \in H(\sigma(i))$;

Lemma 1. *Every LTL model $\mathcal{M} = (S, R, L, \sigma)$ for ϕ induces a Hintikka structure (S, R, L^+, σ) for ϕ where L^+ is an extended labeling on $\mathcal{M}$. Conversely, every Hintikka structure (S, R, H, σ) for ϕ can be extended to a model satisfying ϕ.*

A proof for lemma 1 can be found in [3]. The tableau procedure is an attempt to construct a structure $\mathcal{T}^\phi$ for ϕ, called tableau, representing a Hintikka structure for ϕ. More precisely the structure is a directed graph where each node is labelled with a set of formulae. The label of each node is a subset of the extended closure of the initial formula. If the construction is successful then the formula ϕ is satisfiable otherwise ϕ is not satisfiable. The procedure has two main phases. These are construction and elimination phases.

3.4 Tableau Procedure

3.4.1 Construction Phase

The construction phase involves building a structure $\mathcal{P}^\phi$, which is called a pretableau. This is a directed graph whose nodes labelled with sets of formulae,

but it is a strict of superset the final tableau structure obtained at the end of the procedure. The pretableau contains two types of nodes. These are *prestates* and *states*. The difference between the two is the saturation level. Prestates may contain formulae that have not yet been decomposed. States on the other hand are fully downward saturated. That is they contain only elementary formulae and decomposed non-elementary formulae. For example a prestate may be contain only the formula $(\varphi \ \mathtt{U} \ \psi)$. If a state contains this formula then it also contains the formula ψ or the formula $\varphi \wedge \mathtt{X}(\varphi \ \mathtt{U} \ \psi)$.

States are obtained from prestates by a process of downward saturation, with the use of temporary protostates. That is given a prestate, labelled with a set of formulae, clone the prestate to obtain a protostate and apply the appropriate α or β rule to each formula and mark it. Continue this process until the label contains only marked or elementary formulae. At this point the set of formulae is downward saturated, which means it is the label of a state. The formal definition of pretableau follows:

Definition 3.3. (Pretableau) A pretableau, $\mathcal{P}^{\phi}$ is a structure (S, P, R^S, R^P), where:

- S is the set of states of $\mathcal{P}^{\phi}$;
- P is the set of prestates of $\mathcal{P}^{\phi}$;
- $R^P \subseteq S \times P$;
- $R^S \subseteq P \times S$;

The α and β rules are defined as follows.

Definition 3.4. (Cloning-rule) Given a prestate Γ, create a protostate $\Gamma' = \Gamma$ and put $\Gamma R^S \Gamma'$.

The role of the cloning rule is to create a replica of a prestate from which we can create one or several states. This allows us to use protostates instead of prestates, for the construction of new states and preserve prestates in their original form.

Definition 3.5. (α-rule) Given a prestate Γ and an α-formula $\varphi \in \Gamma$, create a child Γ' of Γ and add $\varphi^*, \alpha_1(\varphi)$ and $\alpha_2(\varphi)$ to the label of Γ'.

Definition 3.6. (β-rule) Given a prestate Γ and a β-formula $\varphi \in \Gamma$, create two children Γ' and Γ''. Add φ^* to both Γ' and Γ''. Add $\beta_1(\varphi)$ to Γ' and $\beta_2(\varphi)$ to Γ''.

In the two definitions above φ^* refers to the formula φ being marked as "processed". Prestates are are also obtained from states. This is done by going

from a fully saturated set of formulae to a possible description of a successor. This is formalised with the X-rule defined below.

Definition 3.7. (X-rule) Given a state $\Delta \in \mathcal{P}^{\phi}$ create a prestate $\Gamma = \{\varphi | \mathtt{X}\varphi \in \Delta\} \cup \{\top\}$ and create an arc $\Delta R^P \Gamma$. If Γ is already part of the structure at the beginning of this rule then simply put $\Delta R^P \Gamma$.

The construction phase ends when no new nodes can be created by applying the rules above.

3.4.2 Elimination Phase

The elimination phase involves obtaining the final tableau $\mathcal{T}^{\varphi}$ from the pretableau $\mathcal{P}^{\varphi}$. More specifically it involves removing all the prestates because they are no longer needed. This is followed by removing all inconsistent states. These are states that contain both φ and $\neg\varphi$ in their labels. This is contradictory information so these states have to be removed. States with no successors are also removed from the tableau because a model of the formula being tested can not contain states with no successors. Finally all states that contain unfulfilled eventualities are removed from the tableau. All these elimination rules are discussed in detail below.

Definition 3.8. (Eliminate prestates) From pretableau $\mathcal{P}^{\phi}$ obtain a tableau $\mathcal{T}_0^{\phi}$ by eliminating all the prestates in $\mathcal{P}^{\phi}$. For every prestate Γ, where $\Delta R^P \Gamma$ and $\Gamma R^S \Delta'$, put the arc $\Delta R \Delta'$

The elimination of prestates rule states that we first eliminate all the prestates form the pretableau $\mathcal{P}^{\phi}$ in order to obtain the initial tableau $\mathcal{T}_0^{\phi}$. Since the first node in the pretableau, namely the node labelled $\{\phi\}$, is a prestate and is removed from the tableau all its children are marked as *initial nodes*.

The next rule eliminates all the inconsistent states from $\mathcal{T}_0^{\phi}$. The rule is formally defined as follows:

Definition 3.9. (Eliminate inconsistent states) If for some $\Delta \in \mathcal{T}_m^{\phi}$ we have $\varphi \in \Delta$ and $\neg\varphi \in \Delta$, obtain $\mathcal{T}_{m+1}^{\phi}$ by eliminating Δ from $\mathcal{T}_m^{\phi}$

The third rule eliminates all states that have no successors. That is because states with no successors can not be part of the model we are trying to build. The rule is defined as follows:

Definition 3.10. (Eliminate states with no successors) If for some $\Delta \in \mathcal{T}_m^{\phi}$, Δ has no successors, obtain $\mathcal{T}_{m+1}^{\phi}$ by eliminating Δ from $\mathcal{T}_m^{\phi}$.

The last elimination rule deals with formulae called *eventualities*. These are formulae that promise something will eventually happen, that is a formula

will be true in the future. If an eventuality occurs at a state Δ then the procedure needs to check whether or not the eventuality is fulfilled from Δ. It needs to check whether the promise from Δ has been fulfilled in one of the successors of Δ. There are three types of eventualities that the procedure will deal with. They are formulae of the form $\mathtt{F}\varphi$, $\neg\mathtt{G}\varphi$ and $(\varphi\ \mathtt{U}\ \psi)$. The conditions for realisation of all these eventualities are as follows:

Definition 3.11. (Eventuality realisation) Let $\mathcal{T}_m^\phi = (S_m, R_m)$ be a tableau and $\Delta \in S_m$,

- eventuality $(\varphi\ \mathtt{U}\ \psi)$ is realised from Δ in $\mathcal{T}_m^\phi$ if there exists in $\mathcal{T}_m^\phi$ an R_m-path $\pi = \Delta_0, \Delta_1, \ldots, \Delta_i$ such that $\psi \in \Delta_j$ for all $0 \leq j < i$ and $\varphi \in \Delta_i$;
- eventuality $\mathtt{F}\varphi$ is realised from Δ in $\mathcal{T}_m^\phi$ if there exists in $\mathcal{T}_m^\phi$ an R_m-path $\pi = \Delta_0, \Delta_1, \ldots, \Delta_i$ such that $\varphi \in \Delta_i$;
- eventuality $\neg\mathtt{G}\varphi$ is realised from Δ in $\mathcal{T}_m^\phi$ if there exists in $\mathcal{T}_m^\phi$ an R_m-path $\pi = \Delta_0, \Delta_1, \ldots, \Delta_i$ such that $\neg\varphi \in \Delta_i$.

It is important to note that if a state contains multiple eventualities, then all of these eventualities need to be realised along a common path. Say a state contains two eventualities $\mathtt{F}\varphi$ and $\mathtt{F}\psi$. Suppose that $\mathtt{F}\varphi$ gets realised in a state Δ while $\mathtt{F}\psi$ has not yet been realised. The label of Δ would then contain φ and $\mathtt{F}\psi$. That is the eventuality which is not realised, will propagate through and will need to be realised from Δ. If the eventuality does not get realised before the end of the procedure, the state Δ will be removed from the tableau. This removal would cause some successors of Δ to become unreachable from the initial state/s. All such successors would also get removed. Removing Δ may also cause some of the parents of Δ to be left with no successors (if Δ was the only successor). Naturally, these parents get removed from the tableau. This process continues recursively until all states with no successors have been removed. This ensures that if there is a path in the tableau where one eventuality is realised while another one is not then the entire path will be removed from the tableau.

The rule for eliminating eventualities is stated as follows:

Definition 3.12. (Eliminate unrealised eventualities) If eventuality $\theta \in \Delta$ is not realised by Δ in $\mathcal{T}_m^\phi$, obtain $\mathcal{T}_{m+1}^\phi$ by eliminating Δ from $\mathcal{T}_m^\phi$.

The final tableau is obtained when the application of any of the above elimination rules does not result in any states being removed form the tableau. Formally it is defines as follows.

Definition 3.13. Final tableau The final tableau, $\mathcal{T}^\phi$ is a structure (T, R), where:

- $T = S_{l+1} = S_l = \{\Delta \in \mathcal{T}_0^\phi \mid \Delta \in \mathcal{T}_m^\phi, \text{ for all } 0 \leq m \leq l\}$;

- $R = R_0 \cap (T \times T)$

Definition 3.14. (Open tableau) The final tableau, $\mathcal{T}^\phi$ is declared *open* if it contains at least one of the initial states of $\mathcal{T}_0^\phi$.

Definition 3.15. (Closed tableau) The final tableau, $\mathcal{T}^\phi$ is declared *closed* if all the initial states of $\mathcal{T}_0^\phi$ have been removed.

These definitions relate to the satisfiability and validity problems discussed in section 2.3. An open tableau $\mathcal{T}^\phi$ represents a model, or several models, for the formula ϕ. A closed tableau $\mathcal{T}^\phi$ shows that we have failed to build a model for ϕ, because there are no initial states. Therefore an open $\mathcal{T}^\phi$ means that ϕ is satisfiable and a closed $\mathcal{T}^\phi$ means that ϕ is not satisfiable.

3.5 Termination

The tableau procedure terminates because, during the construction phase, only finitely many nodes are generated. The idea of *extended closure* illustrates this. Intuitively, the extended closure of a formula ϕ consists of all the formulae necessary to label states during the downward saturation process.

Definition 3.16. (Closure) Let ϕ be an LTL formula. The closure of ϕ, $\mathsf{cl}(\phi)$ is the least set such that:

- $\mathsf{sub}(\phi) \subseteq \mathsf{cl}(\phi)$,
- if $(\varphi \ \mathtt{U} \ \psi) \in \mathsf{cl}(\phi)$, then $\mathtt{X}(\varphi \ \mathtt{U} \ \psi) \in \mathsf{cl}(\phi)$,
- if $\mathtt{G}\varphi \in \mathsf{cl}(\phi)$, then $\mathtt{XG}\varphi \in \mathsf{cl}(\phi)$,
- if $\mathtt{F}\varphi \in \mathsf{cl}(\phi)$, then $\mathtt{XF}\varphi \in \mathsf{cl}(\phi)$.

Definition 3.17. (Extended closure) Let ϕ be an LTL formula. The extended closure of ϕ, $\mathsf{ecl}(\phi)$ is the least set such that:

- $\mathsf{cl}(\phi) \subseteq \mathsf{ecl}(\phi)$,
- if $\varphi \in \mathsf{cl}(\phi)$, then $\neg\varphi \in \mathsf{ecl}(\phi)$,
- if $\mathtt{X}\varphi \in \mathsf{cl}(\phi)$, then $\mathtt{X}\neg\varphi \in \mathsf{ecl}(\phi)$,
- $\top \in \mathsf{ecl}(\phi)$.

The extended closure of any formula ϕ is a finite set. This is because an LTL formula has finitely many sub-formulae. The tableau procedure only deals with subsets of the extended closure, which ensures the termination of the procedure.

3.6 Soundness and Completeness

In this section, we prove that the final tableau for ϕ, $\mathcal{T}^\phi$, is open iff ϕ is satisfiable. The proofs are taken from [3].

Theorem 2 (Soundness)**.** *If $\mathcal{T}^\phi$ is closed, then ϕ is unsatisfiable.*

Proof. We prove that if $\Delta \in S_0$ is eliminated from some $\mathcal{T}^\phi_m$ $(0 \leq m \leq l)$, then Δ is unsatisfiable.

As the elimination process proceeds in stages, we will prove by induction on the number n of stages that, if $\Delta \in S_0$ is satisfiable, then Δ will not be eliminated at stage n.

The base case is trivial. If $n = 0$, then no eliminations have taken place yet, hence all satisfiable states are still in place.

As the inductive hypothesis, assume that if $\Delta' \in S_0$ is satisfiable, it has not been eliminated during the previous n stages of the elimination process, and thus $\Delta' \in S_n$. Consider stage $n+1$ and any satisfiable $\Delta \in S_0$. By inductive hypothesis, $\Delta \in S_n$. We will now show that no elimination rule allows us to eliminate Δ from $\mathcal{T}^\phi_n$; thus, Δ will be in $\mathcal{T}^\phi_{n+1}$.

If Δ is satisfiable, then clearly it can not contain both φ and $\neg\varphi$; therefore, it can not be eliminated from $\mathcal{T}^\phi_n$ due to the rule for eliminating inconsistent states.

If Δ is satisfiable, then it is easy to see that the prestate Γ that was created from Δ in $\mathcal{P}^\phi$ by ($\mathtt{X}$-rule) is satisfiable, too. Trivially, the protostate Γ' that was obtained from Γ by (Cloning rule) is also satisfiable. Now, for any set Θ of LTL formulae, if Θ is satisfiable and $\alpha \in \Theta$, then $\Theta \cup \{\alpha_1, \alpha_2\}$ is satisfiable; likewise, if Θ is satisfiable and $\beta \in \Theta$, then either $\Theta \cup \{\beta_1\}$ or $\Theta \cup \{\beta_2\}$ is satisfiable. Therefore, at least one state Δ' obtained from Γ' by downward saturation is satisfiable. By inductive hypothesis, $\Delta' \in S_n$ and hence $\Delta R_n \Delta'$; therefore, Δ can not be eliminated from $\mathcal{T}^\phi_n$ due to the rule for eliminating states with no successors.

Lastly, we show that Δ can not be removed from $\mathcal{T}^\phi_n$ due to unfulfilled eventualities. The argument for eventualities of type $\varphi \,\mathtt{U}\, \psi$ is presented here. The arguments for the other types of eventualities are similar.

Suppose $\varphi \mathtt{U} \psi \in \Delta$ and Δ is satisfiable. Thus, there exists a model $\mathcal{M} = (S, R, L, \sigma)$ such that, for some $m \in \mathbb{N}$, $\mathcal{M}, \sigma(m) \Vdash \Delta$. In particular, $\mathcal{M}, \sigma(m) \Vdash \varphi \mathcal{U} \psi$. Then, for some $i \geq m$, $\mathcal{M}, \sigma(i) \Vdash \psi$ and for all $m \leq j < i$, $\mathcal{M}, \sigma(j) \Vdash \varphi$. We can assume without a loss of generality that i is the smallest number with this property (and thus, $\mathcal{M}, \sigma(j) \nVdash \psi$ for all $m \leq j < i$).

If $\psi \in \Delta$, then Δ constitutes the path realising $\varphi \,\mathtt{U}\, \psi$ from Δ in $\mathcal{T}^\phi_n$; therefore, Δ can not be eliminated from $\mathcal{T}^\phi_n$ due to unfulfilled eventualities. (This is the case when $i = m$.)

Otherwise, due to (β-rule), $\varphi \in \Delta$ and $\mathtt{X}(\varphi \mathtt{U} \psi) \in \Delta$. Consider the prestate Γ created from Δ by ($\mathtt{X}$-rule). Clearly, $\mathcal{M}, \sigma(m+1) \Vdash \Gamma$ and so $\mathcal{M}, \sigma(m+1) \Vdash$

Γ', where Γ' is a protostate created from Γ by (Cloning rule). Then, it is easy to show that for some state Δ' created from Γ' we have $\mathcal{M}, \sigma(m+1) \Vdash \Delta'$. Indeed, let $B = \{\beta^1, \ldots, \beta^l\}$ be the set of all β-formulae featuring in the process of creating states from Γ' (not all of these β's have to be members of Γ'; for example, if $(\varphi \vee \psi) \vee \chi \in \Gamma'$, then $\varphi \vee \psi \in B$, even though $\varphi \vee \psi \notin \Gamma'$). Now, consider the set $B_{\sigma(m+1)} = \{\, \beta^i{}_1 \mid \mathcal{M}, \sigma(m+1) \Vdash \beta^i{}_1 \text{ and } \beta^i \in B) \,\} \cup \{\, \beta^i{}_2 \mid \mathcal{M}, \sigma(m+1) \Vdash \beta^i{}_2 \text{ and } \beta^i \in B) \,\}$. It is easy to see that for some Δ' created from Γ' we have $B_{\sigma(m+1)} \subseteq \Delta'$ (this is the state Δ' such that every time we had to decide for $\beta \in B \cap \{\, \chi \mid \mathcal{M}, \sigma(m+1) \Vdash \chi \,\}$ whether to add β_1 or β_2 to "a state to be", we added the alternative contained in $\{\, \chi \mid \mathcal{M}, \sigma(m+1) \Vdash \chi \,\}$). Then, it is easy to check that $\mathcal{M}, \sigma(m+1) \Vdash \Delta'$. Thus, Δ' is satisfiable; hence, by inductive hypothesis, $\Delta' \in S_n$.

Now, as $\varphi \mathcal{U} \psi \in \Delta'$, the argument can be repeated for Δ'. Repeating the argument $i - m$ times, we get a sequence $\Delta = \Delta_0, \ldots, \Delta_{i-m}$ such that (1) for every $0 < k \leq i-m$, $\Delta_{k-1} R_0 \Delta_k$, (2) for every $0 \leq k \leq i-m$, $\mathcal{M}, \sigma(k+m) \Vdash \Delta_k$, (3) $\psi \in \Delta_{i-m}$, and (4) for every $0 \leq j < i - m$, $\varphi \in \Delta_j$. Due to inductive hypothesis, it follows from (2) that $\Delta_0, \ldots, \Delta_{i-m} \in S_n$. Therefore, the path $\Delta_0, \ldots, \Delta_{i-m}$ realises $\varphi \mathtt{U} \psi$ from Δ in $\mathcal{T}_n^\phi$; hence, Δ can not be eliminated from $\mathcal{T}_n^\phi$ due to unfulfilled eventualities.

Corollary 3. *If $\mathcal{T}^{\neg\phi}$ is closed, then ϕ is valid.*

Theorem 4 (Completeness)**.** *If $\mathcal{T}^\phi$ is open, then ϕ is satisfiable.*

Proof. Suppose that $\mathcal{T}^\phi = (T, R)$ is open. We will construct a Hintikka structure $\mathcal{H}$ for ϕ, which is enough to prove that ϕ is satisfiable.

The set of states of $\mathcal{H}$ is going to be T, and its successor relation is going to be R. Next, we show how to define a state sequence σ on T.

Arrange all the members of T in a sequence $\Delta_0, \Delta_1, \ldots, \Delta_i$ such that Δ_0 is an initial state of $\mathcal{T}_0^\phi$. (As $\mathcal{T}^\phi$ is open, at least one such state will be in T).

First, for each Δ_n $(0 \leq n \leq i)$, we will construct a finite R-path σ_n through T such that all eventualities $\theta \in \Delta_n$ are realised from Δ_n by σ_n. Let $\theta_1, \ldots, \theta_k$ be all the eventualities in Δ_n. As $\Delta_n \in T$, there exists an R-path through T, $\Delta_n = \Delta'_0, \ldots, \Delta'_j$, that realises θ_1 from Δ_n. It then follows from the construction rules for $\mathcal{P}^\phi$ that, if θ_2 is not realised by (a sub-path of) $\Delta'_0, \ldots, \Delta'_j$, then $\theta_2 \in \Delta'_j$. As $\Delta'_j \in T$, there exists a finite R-path through T, $\Delta'_j = \Delta''_0, \ldots, \Delta''_m$, that realises θ_2 from Δ'_j. If we join the two paths together, we obtain a finite path $\Delta_n = \Delta'_0, \ldots, \Delta'_j, \Delta''_1, \ldots, \Delta''_m$ that realises both θ_1 and θ_2 from Δ_n. As there are only k eventualities in Δ_n, k repetitions of this procedure will give us a finite path σ_n that realises all eventualities in Δ_n from Δ_n.

Next, take the paths $\sigma_0, \sigma_1 \ldots, \sigma_i$ so defined. Inductively define an infinite state sequence σ as follows. First, take σ_0 and make it the initial segment of σ, σ'_0. Now, inductively assume that σ'_j has been defined. Take the last state of σ'_j, which is Δ_n for some $0 \leq n \leq i$, and append σ_n to σ'_j to form the path σ'_{j+1}.

Repeat this procedure ad infinitum, thus producing an infinite state sequence σ.

Finally, define a structure $\mathcal{H} = (T, R, H, \sigma)$ by putting $H(\Delta) = \Delta$. It is easy to show that $\mathcal{H}$ is a Hintikka structure for ϕ.

(H0) As $\sigma(0) = \Delta_0$, where Δ_0 is an initial state (and hence $\phi \in \Delta_0$), and $H(\Delta_0) = \Delta_0$, we have $\phi \in H(\sigma(0))$.

(H1) If we had $\neg\varphi, \varphi \in \Delta$ for some state Δ, then Δ would have been eliminated due to the rule for inconsistent states, so for all $\Delta \in T$, (H1) holds.

(H2) Follows from the fact that all $\Delta \in T$ are downward saturated.

(H3) Follows from the fact that all $\Delta \in T$ are downward saturated.

(H4) If $\mathtt{X}\varphi \in H(\sigma(n))$, then by construction of σ, we have $(\sigma(n), \sigma(n+1)) \in R$; therefore, as $H(\sigma(i)) = \sigma(i)$ for all $i \in \mathbb{N}$, we have $\varphi \in H(\sigma(n+1))$.

(H5) Suppose that $\varphi \mathtt{U} \psi \in H(\sigma(n))$. By construction of σ, $\sigma(n)$ belongs to some σ_i out of which σ was built. Suppose that the length of σ_i is j. It follows from the construction rules for $\mathcal{P}^\phi$ that if $\varphi \mathtt{U} \psi$ is not realised from $H(\sigma(n))$ by σ_i, then $\varphi \mathtt{U} \psi \in H(\sigma_i(j))$. But $\sigma_i(j)$ is the initial state of some other finite path σ_k that, by construction, realises all eventualities in $H(\sigma_k(0))$ form $\sigma_k(0) = \sigma_i(j)$. Then, the path $\sigma_i^n \cdot \sigma_k$ realises $\varphi \mathtt{U}\psi$ from $H(\sigma(n))$; hence, (H5) holds.

The arguments (H6) and (H7) are similar to (H5).

Corollary 5. *If formula ϕ is valid, then $\mathcal{T}^{\neg\phi}$ is closed.*

Chapter 4

Description of Implementation

4.1 Introduction

This chapter presents the implementation details of the tableau procedure described in chapter 3. The implementation is available online at `http://msit.wits.ac.za/ltltableau`. The Python programming language was chosen for this project. The main reasons for this are the cross-platform compatibility and the simple and elegant syntax of the language. The chapter contains a description of the data structures used followed by pseudo code of the main algorithm. Each module used in the program is presented in detail.

4.2 Syntax

The algorithm takes as input a string containing the formula the user wishes to test and returns the string `'satisfiable'` if the formula is satisfiable, otherwise returns `'not satisfiable'`. The implementation supports all the usual boolean and temporal connectives, listed in Figure 4.1.

Connective	**Meaning**
`A`	and
`O`	or
`I`	implies
`N`	not
`U`	until
`F`	sometime in the future
`G`	always in the future
`X`	next time

Figure 4.1: Logical connectives

The syntax of a formula string is defined inductively. The definition is given below.
If `p` is a formula then `Np, Xp, Fp` and `Gp` are formulae
If `p` is a formula and `q` is a formula then `(p A q), (p O q), (p I q)` and `(p U q)` are formulae

4.3 Data Structures

The tableau is a directed graph, made up of states and prestates. The generic term *node* will be used to refer to either states or prestates when it is not important to distinguish between them. This graph is implemented as a list of nodes, where each node contains adjacency lists. Nodes also contain other useful information such as their type and the formulae that are true at the state. Each node of the tableau is represented by a Python dictionary. A dictionary is a list type where the elements are not indexed by an integer but by a key. Each key has a value associated with it. The node dictionary has the following fields:

`id`: A unique integer identifier for the node.

`parents`: A list of integers containing the ids of the parents of the node.

`children`: A list of integers containing the ids of the children of the node.

`type`: A string that specifies what type the node is. Possible values are `pre`, `proto` and `state`

`initial`: A boolean flag that indicates whether the node is a child of the initial node, ie a child of the root node

`formulae`: A list of strings that contains all the formulae that belong to the given node.

`rank`: An integer used for checking eventualities

`succRank`: An integer defined as the minimum of all children's ranks

An example of a typical node in the tableau is shown in figure 4.2.

This node has a unique id 27 and it is a state in the tableau. The formulae that are true at that state are listed. The cloned flag being true indicates that the nextRule has been applied to this state and a prestate was obtained from it. This state is a child of 14 and 27 and is the parent of 25 and 27. The fact that the state is a parent and a child of itself shows that there is a loop edge there. The rank being infinity shows that we are in the process of checking whether an eventuality is realised. All the children of this state also have infinite rank.

```
{
  'id': 27,
  'type': 'state',
  'formulae': ['(N (p3 U p2))', '((N p2) A (N (X (p3 U p2))))'],
  'initial': False,
  'parents': [14, 27],
  'children': [25, 27],
  'rank': infinity,
  'succRank': infinity
}
```

Figure 4.2: A typical node in the tableau

4.4 Tableau Algorithm

The tableau algorithm, shown in figure 4.3, begins by constructing a pretableau from the input formula. This corresponds to the construction phase. The rest of the algorithm (lines 3-6) is the elimination phase. The first step is to remove all the prestates. Then all the inconsistent states are removed. These are states that contain contradictory information. The next step is to remove all states that have unfulfilled eventualities. These are states that contain formulae which promise that something must happen in the future but it never does. The process of removing inconsistent states and states with unfulfilled eventualities may leave some states with no successors. Such states are not allowed to be part of the tableau and so they have to be removed as well. This is done by the method on line 6 of the algorithm.

```
1 Algorithm:Test(formula)
2 tableau = constructPretableau(formula);
3 tableau = removePreStates(tableau);
4 tableau = removeInconsistent(tableau);
5 while not done do
6     tableau = removeEventualities(tableau);
7     tableau = removeNonSuccessors(tableau);
8 end
9 return isOpen(tableau);
```

Figure 4.3: Algorithm to test a formula for satisfiability

4.4.1 Construction Phase

The construction phase of the tableau consists of creating a single prestate labelled with the input formula and then repeatedly applying the construction rules until no more nodes can be added to the pretableau. The code for the construction procedure is given in figure 4.4.

```
1  Algorithm:constructPretableau(formula)
2  global last = 0;
3  node.id = last;
4  node.parents = ∅;
5  node.children = ∅;
6  node.type = pre;
7  node.formulae = formula;
8  node.cloned = False;
9  node.rank = ∞;
10 node.succRank = ∞;
11 node.initial = False;
12 insert(tableau, node);
13 while True do
14     current = copy(tableau);
15     clone(tableau);
16     if current = tableau then
17         break
18     end
19     alphaBetaRules(tableau);
20     tableau = removeProtoStates(tableau);
21     nextRule(tableau);
22 end
23 return tableau
```

Figure 4.4: Algorithm to construct the pretableau

The construction algorithm starts off by initialising all the global variables. The variable `last` is used to keep count of the number of nodes in the tableau. Each time a new node is added, the value stored in `last` is used as the id for the node. The value of `last` is then incremented by 1. The next step is to create the initial node of the tableau. This is the prestate labelled with the input formula. The *while* loop that follows is where all the construction rules are applied. The loop terminates when the application of construction rules does not add any new nodes to the tableau. The first construction procedure is called `alphaBetaRules` and it creates states from prestates. The code for this procedure is given in figure 4.5.

```
1  Algorithm:alphaBetaRules(tableau)
2  for i in tableau do
3      if i.type = 'proto' then
4          for j in i.formulae do
5              if j = 'T' then
6                  node.id = last+1;
7                  node.parents = {i};
8                  node.children = ∅;
9                  node.formulae = {'T'};
10                 insert(tableau, node);
11             end
12             if not marked(j) and not elementary(j) then
13                 mark(j);
14                 if classify(j) = 'alpha' then
15                     handleAlpha(tableau, i, j)
16                 else
17                     handleBeta(tableau, i, j)
18                 end
19             end
20         end
21     end
22     if i.children = ∅ then
23         i.type = 'state';
24     end
25 end
```

Figure 4.5: Algorithm to apply the α and β rules

The *alphaBetaRules* algorithm iterates through all the proto-states in the tableau using the for loop in line 2. The current node considered by the algorithm is always indexed by i. As the algorithm iterates through all the proto-states, it tries to expand the formulae they contain. If a proto-state contains a formula j that is not marked and not elementary, meaning that j can be expanded but has not been processed yet, the formula j is marked. The procedure then checks whether j is an α-formula or a β-formula, with the help of the *classify* procedure, and calls the appropriate helper method. When the *handleAlpha* procedure is called, it expands the α-formula j into α_1 and α_2, according to the rules defined in figure 3.1 and creates a successor of the node i. The process of creating a successor involves copying the original node i, which already contains the marked formula j, and appending to its label the newly created formulae α_1 and α_2. If the *handleBeta* method is called then the formula j is a β-formula and is expanded according to the rules defined

in figure 3.2. The method then creates two successors of i, namely i' and i''. Finally it appends β_1 to i' and β_2 to i'', thereby creating branching in the tableau. If the algorithm goes to a state i and the application of α or β rules does not result in the creation of successors of i this means that i is fully expanded and it is declared to be a state. The second construction rule, called *nextRule*, creates prestates from states. The code for this algorithm is given in figure 4.6.

```
1  Algorithm:nextRule(tableau)
2  for i in tableau do
3      if i.type = 'state' and i.cloned = False then
4          i.cloned = True;
5          if consistent(i.formulae) then
6              next = checkNext(i.formulae);
7              if next = ∅ then
8                  formulae = {T};
9              else
10                 formulae = next;
11             end
12             check = preExist(tableau, formulae);
13             if not check[0] then
14                 new.id = last + 1;
15                 new.parents = {i};
16                 new.formulae = formulae;
17                 new.type = pre;
18                 new.children = ∅;
19                 insert(tableau, new);
20                 last = last + 1;
21             else
22                 check[1].parents = check[1].parents + {i};
23                 i.children = {check[1]};
24             end
25         end
26     end
27 end
```

Figure 4.6: Algorithm to apply the "Next time Rule"

The `nextRule` method goes through all the newly created states in the tableau. These states are recognised by the fact that they have not yet been cloned. The current state under consideration, as usual, is indexed by i. The first step of the algorithm is to set the *cloned* flag to true. In future runs of the method, this flag will show that i has already been processed. The algorithm

then checks whether i is consistent or not. If not then the algorithm ignores i and no successors get generated for it. If the formulae of i are consistent then the method checks whether any of the formulae have the next-time operator, `X`, as a main connective. All such formulae are stored in a list called *next*, which represents the label of the successor prestate that needs to be created. If *next* is empty, that is none of the formulae of i had `X` as their main connective, then the label of the successor prestate is simply $\{\top\}$. Lastly, after the label of the new prestate has been computed, the method checks to see if there exists another prestate with the same label. If such a prestate exists then the procedure simply draws an edge between i and the existing prestate. If no prestate with the same label exists then a new prestate is created as a child of i.

The *alphaBetaRules* and *nextRule* procedures are applied to the tableau repeatedly until such time that no new states get generated. This condition is recognised with the help of the `clone` function. The code of this function is given in figure 4.7.

```
1 Algorithm:clone(tableau)
2 for i in tableau do
3     if i.type = pre and not i.cloned and i.children = ∅ then
4         i.cloned = True;
5         insert(tableau, id = last + 1, parent = i.id, formulae =
          i.formulae, children = ∅);
6         last = last + 1;
7     end
8 end
```

Figure 4.7: Algorithm to clone the tableau

The clone function iterates through the tableau, looking for newly created prestates. These prestates would have been created during the execution of the *nextRule* method. Each new prestate is cloned before the downward saturation process begins. The cloning procedure consists of simply making a copy of the prestate. The new node created is called a protostate, which is an intermediate node, used during the saturation process. The reason we create protostates instead of simply saturating the prestates is because we would like to keep prestates in their original form. If we do not, then we would not be able to identify that a prestate already already exists in the tableau when the *nextRule* procedure runs. Failing to recognise existing states would make the tableau infinite because more prestates will be added every time the *nextRule* method is executed. Creating protostates from prestates, that is cloning, allows us to expand the prestates fully by keeping intermediate information in the protostates. When the formulae have been fully expanded the protostates are

removed from the tableau, leaving only the states as successors of the original prestate that was colned.

When the *clone* procedure does not add any protostates to the tableau then the while loop terminates and the construction phase comes to an end. At that point the tableau contains both states and prestates. We note that every consistent state in the tableau has exactly one successor prestate and every prestate has at least one successor state. States that are not consistent, that is they contain contradictory information, do not have successors. That is because the *nextRule* method checks whether a state is consistent before it adds a successor prestate. The next section describes the elimination process of the tableau procedure.

4.4.2 Elimination Phase

The elimination process consists of removing all the prestates from the tableau as well as all the states that contain contradictory information or unsatisfied eventualities. After the elimination process is completed the structure that is left is the final tableau. The elimination method is broken up into several sub-methods. These are `removePrestates`, `removeInconsistent`, `removeEventualities` and `removeNonSuccessors`. Each one of these methods is described in detail below.

```
1 Algorithm:removePreStates(tableau)
2 for i in prestates do
3     tableau = remove(tableau, i)
4 end
5 return tableau;
```

Figure 4.8: Algorithm to remove all prestates from a tableau

The method in figure 4.8 simply finds all the prestates in the tableau and removes them one by one. The removal is done by the `remove` method which is discussed below. It is important to note the difference between removing prestates and states. In the case of prestates we remove the node from the tableau and we connect all of its children to its parents. The parents "adopt" all the children of the removed node. The code that accomplishes that task is given in figure 4.9. The algorithm takes in the tableau data structure and n, the node to be removed. Firstly the algorithm disconnects n from its parents. This is done at line 3 of the code. Then the children of n are added to the children lists of the parents of n. That is, grandparents are adopting their grandchildren. Now every grandparent node is fully updated because its children list includes its grandchildren. The algorithm then proceeds to repair the children nodes, whose parent lists are not complete. The parent list of

children of n should contain all the original parents, except for n, as well as all the grandparents, i.e. the parents of n. Lines 12 - 17 of the code accomplish this. The type of n is changed to "Removed" and the algorithm terminates.

```
1  Algorithm:remove(tableau, n)
2  for i in n.parents do
3      i.children = i.children - {id};
4      for j in n.children do
5          if j not in i.children then
6              add j to i.children
7          end
8      end
9  end
10 grandparents = n.parents;
11 children = n.children;
12 for i in children do
13     parents = i.parents;
14     parents = parents - {id};
15     newset = parents ∪ grandparents;
16     i.parents = newset;
17 end
18 n.type = Removed;
19 return tableau;
```

Figure 4.9: Algorithm to remove node and connect its parents to its children

The removal of states works differently. If we remove a state we cannot connect the children of the removed state to its parent. If we did that, we would be changing the meaning of the formula. Therefore when we remove a state from the tableau, we also have to remove all its successors which are not reachable from an initial state. The code for this is shown later in 4.10.

Firstly pointers to all the successors of the state being removed are stored in a variable. These successors, that is all the states that are reachable from s, are added to the list called `successors`. There is another list called `candidates` that contains all the states that will possibly be removed. Before we do that however, we need to check whether any of the candidates for removal can actually be left in the tableau. If a candidate has a common ancestor with s then it must not be removed. Figure 4.11 shows an example of removing a state. The state to be removed is labelled with 2. The successors of state 2 are 4,5,6,7 and 8. These states form the set of candidates for removal. The algorithm however detects that state 5 and state 8 are still reachable from an initial state, namely state 1, after state 2 has been removed. States 4, 6 and

```
1  Algorithm:removeState(tableau, id)
2  successors = findAllSuccessors(tableau, id);
3  candidates = successors;
4  bad = {id} ∪ successors;
5  for i in successors do
6      for j in i.parents do
7          if j not in bad then
8              candidates = exclude(candidates, i);
9          end
10     end
11 end
12 for i in {id} ∪ candidates do
13     tableau = simpleRemove(tableau, i);
14     states = exclude(states, i);
15 end
16 return tableau;
```

Figure 4.10: Algorithm to remove states from the tableau

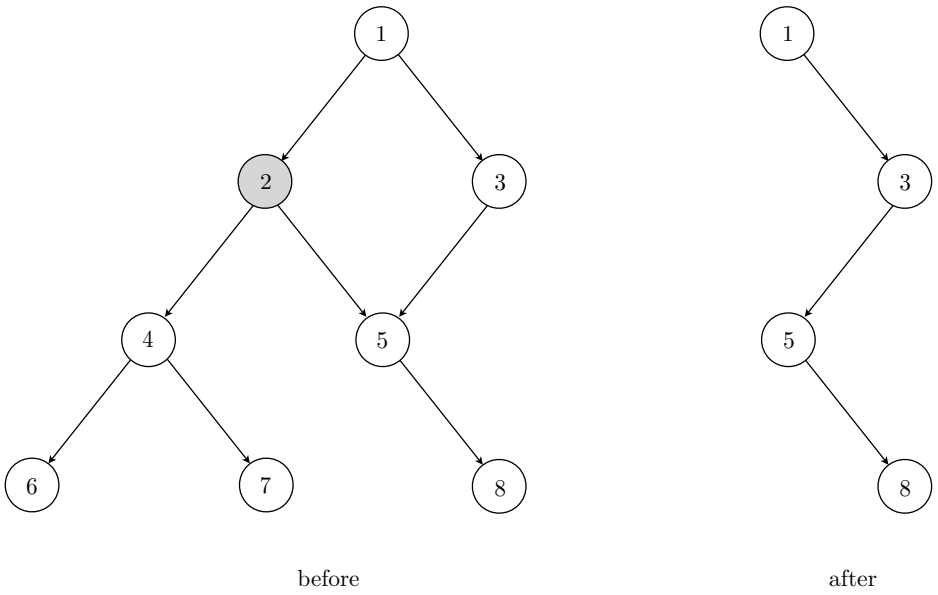

Figure 4.11: Removing a state from the tableau

7 are no longer reachable, therefore they are removed from the tableau along with state 2.

The next step of the elimination process it to remove all inconsistent states. These are the states that contain contradictory formulae in their labels. They are also easily recognised by the fact that they have no successors. That is because they were recognised as inconsistent during the construction phase and no successor prestates were added. That means that the inconsistent states can be removed by a simple remove procedure, which simply changes the type of the node to "Removed". The `simpleRemove` method, shown in figure 4.12, simply excludes the node being removed from the "children" lists of its parents and from the "parents" lists of its children. In other words the code above breaks the link between a grandparent and its grandchildren. Only inconsistent states are removed using this algorithm. All other states that need to be removed from the tableau are removed by the `removeState` algorithm, shown in figure 4.10.

```
1  Algorithm:simpleRemove(tableau, index)
2  parents = tableau[index].parents;
3  for i in parents do
4      i.children = exclude(i.children, index);
5  end
6  children = tableau[index].children;
7  for i in children do
8      i.parents = exclude(i.parents, index)
9  end
10 tableau[index] = {'id' = index, type = Removed};
11 return tableau;
```

Figure 4.12: Algorithm to remove a node from a graph

The final stage of the elimination process is to remove states that contain unfulfilled eventualities. An eventuality is a formula that "promises" something must happen in the future. If such a formula appears at a state s then we need to check all the successors of s, and see whether the eventuality has been realised. There are three types of eventualities that the tableau checks for. The first one is of the form $\mathtt{F}\varphi$. This promises that sometime in the future φ will be true. If this formula appears at a state s, the tableau goes through all the successors of s. If one of these successors contains the formula φ then the eventuality is realised and s is not removed from the tableau. If however none of the successors of s contain φ then the eventuality is not realised and s is removed from the tableau. The other type of eventuality is of the form $\neg\mathtt{G}\varphi$. This formula says it is not the case that φ is true always in the future. This means that if this formula occurs at a state s, then some successor of

s must contain the formula $\neg\varphi$. If such a successor is found then s is left in the tableau, otherwise it is removed. The last type of eventuality is of the form $(\varphi \ \mathtt{U} \ \psi)$. This formula promises that φ will be true until ψ becomes true sometime in the future. So if a formula of this form occurs at a state s then, in order for the eventuality to be fulfilled, there must be a successor of s, say s' which contains the formula ψ. Furthermore all the states along the path $s - s'$ must contain the formula φ.

Each type of eventuality discussed above is checked for separately. The code for checking whether future eventualities are satisfied is given in figure 4.13. Checking the other two types of eventualities is done in a similar manner.

```
1  Algorithm:handleFutureEventualities(tableau)
2  candidates = [];
3  for i in tableau do
4      futures = findFutures(i.formulae);
5      for j in futures do
6          if not fulfillFutures(tableau, i, j) then
7              candidates = candidates ∪ i
8          end
9      end
10 end
11 for i in candidates do
12     tableau = removeState(tableau, i)
13 end
14 return tableau
```

Figure 4.13: Algorithm to remove unfulfilled "future" eventualities

The algorithm goes through each state in the tableau and checks whether the current state i contains any future eventualities. This check is performed by the algorithm in figure 4.14. If there are any eventualities in i the algorithm checks whether they have been fulfilled. This check is done by the algorithm in figure 4.15. If there are any eventualities, contained in i, that have not been fulfilled then i is removed from the tableau. Since this algorithm goes to every node of the tableau, any states that contain unfulfilled future eventualities will be removed. There are more efficient ways of checking whether eventualities are fulfilled. These are discussed in chapter 5.

The algorithm, given in figure 4.14, for finding future eventualities in a list of formulae works by iterating through all the formulae in a list, indexing the current formula under consideration by i. It calls the *split* function, which takes as input a formula splits it about its main connective. If it finds that the main connective is `F`, then the formula is added to a list, which is returned at the end.

```
1  Algorithm:findFutures(formulae)
2  result = ∅;
3  for i in formulae do
4      if main connective of i = F then
5          if i not in result then
6              add i to result;
7          end
8      end
9  end
10 return result;
```

Figure 4.14: Algorithm to find future eventualities in a list of formulae

```
1  Algorithm:fulfillFutures(tableau, node, eventuality)
2  successors = findAllSuccessors(tableau, node);
3  formula = split(eventuality)[0];
4  for i in successors ∪ {node} do
5      if formula in i.formulae then
6          return True;
7      end
8  end
9  return False;
```

Figure 4.15: Algorithm to check if a "future" eventuality is fulfilled

The algorithm, shown in figure 4.15, that checks whether an eventuality is fulfilled from a certain state takes as input a tableau, a state and an eventuality. If finds all the successors of the input node in the tableau and splits the eventuality about its main connective. If the eventuality $\mathtt{F}\varphi$ is used as input to the *split* function the result would be a set $\{\varphi, \mathtt{G}\}$. The algorithm takes the first element of that set and checks if it occurs in any of the successors of the node.

The process of removing states that contain unsatisfied eventualities may have left certain states without any successors. It is necessary to remove all such states before we can check whether the tableau is open or closed. The code that accomplishes that is given in figure 4.16.

The code repeats the check for states with no successors until every state has at least one successor. This, however, can cause some state to contain unfulfilled eventualities. That could happen if a state s contains an eventuality fulfilled in state s'. Suppose that s' has no successors and is removed from the tableau. Now the eventuality in s is not fulfilled. Therefore it is necessary to repeat the `fulfillFutures` procedure and the `removeNonSuccessors`

```
1  Algorithm:removeNonSuccessors(tableau)
2  done = False;
3  while not done do
4      prev = copy(tableau) ;
5      for i in states do
6          if i.children = ∅ then
7              remove i from tableau;
8              if prev = tableau then
9                  done = True;
10             end
11         end
12     end
13 end
14 return tableau
```

Figure 4.16: Algorithm to remove states without successors

procedure until the tableau stabilises.

After this happens, we can finally check whether the tableau is open or closed. The code that does this check is given in figure 4.17.

```
1  Algorithm:isOpen(tableau)
2  for i in tableau do
3      if i.initial then
4          return True;
5      end
6  end
7  return False;
```

Figure 4.17: Algorithm to check whether a tableau is open or closed

The algorithm takes in a tableau as input and returns $True$ if the tableau is open and $False$ if it is not open. The code simply goes through all the states in the tableau and if the current state i has been marked as initial, that is i is a child of the original prestate, the algorithm returns $True$. If it iterates through all the states and has not yet returned $True$ then none of the states were *initial* so $False$ is returned.

Chapter 5

Optimisations

This chapter shows all the different optimisations that were applied to the tableau procedure. They are divided into implementation optimisations and algorithmic optimisations.

5.1 Implementation Techniques

5.1.1 Indexing of Nodes

The tableau is made up of a list of dictionaries. Many operations during the procedure require us to locate a certain node and perform the operation on it. An example of such an operation is the removal of nodes. Before we can remove node i from the tableau, we have to find where node i is located in the tableau. The initial version of the implementation use a linear search function to locate nodes. This meant that every time a node had to be located, so that its attributes could be updated, a linear search procedure was called. One way to reduce the running time is to sort the list of nodes by their unique identifiers and use binary search instead of linear search thereby reducing the time of locating a node from linear to logarithmic.

Indexing however enables us to locate nodes in constant time. We only need to make sure that one property is always satisfied. That property is that the identifier of the node matches its position in the list. It is easy to ensure the list satisfies this property during construction. We only have to check what is the position of the last node inserted and increment it by 1. We then set the identifier property of the next node to have the same value.

It is obvious that as nodes get removed the property will no longer hold. For example if we have 3 nodes numbered 0, 1 and 2 then the identifiers match the positions. However if node 1 is removed from the graph then node 0 will be in position 0 but node 2 will be in position 1. In order to solve this problem we do not remove the node from the list completely but simply delete all the information from the dictionary, except for the identifier field. We also change

the type of the node to `removed`. This guarantees that the identifier of a node will always match its position in the list.

5.1.2 Data Structures

Sometimes we are required to perform operations on all the nodes of certain type. For example the elimination stage of the procedure begins by eliminating all the prestates. For this reason it is useful to have auxiliary data structures that keep pointers to nodes. Three such data structures are used, namely `prestates`, `states` and `newstates`. Intuitively these are lists that contain pointers to the different types of nodes and are updated dynamically as nodes are created or removed.

The `prestates` list is useful when we remove the prestates at the beginning of the elimination phase. Without this list we would have to go through every node of the tableau and check whether it is a prestate or not and if it is then remove it. Since there are usually more states than prestates in a tableau, not having to go through all the nodes is a worthwhile saving. The next auxiliary data structure is a list of all the states. It is necessary because most of the operations in the elimination phase, such as checking eventualities, require us to go through all the states. If we iterated through the list of nodes there would be many empty dictionaries there. These represent all the nodes that have already been removed. This means that as we iterate through the nodes in the list, we need to check whether the current node we are on is empty and ignore it if is. Again this is extra, unnecessary work which we avoid with the help of the `states` list. The last list we have is called `newstates`. It contains pointers to all the states that were created during the last run.

5.2 Improvements to Procedure

5.2.1 Ranking Eventualities

A naive approach to checking eventualities from a state s is to follow every path from s and check whether any of the successor states visited satisfy the eventuality. This approach is computationally expensive as it requires all the paths to be computed every time a state is checked for eventuality. Instead the we can use ranking algorithms to check whether eventualities have been satisfied. The ranking algorithms for the different types of eventualities are similar to each other. The algorithm for checking eventualities of the form Fp is presented.

All nodes of the tableau should have `rank` and `succRank` fields. They represent the rank of the node and the rank of the child respectively. If a node has more than one child, then the minimum rank of all the children is stored

in `succRank`. When a state is added to the tableau both of these fields are initialised to ∞.

The algorithm for checking eventualities of the form $\mathtt{F}\varphi$ works as follows.

1. Find all states that contain φ and set their rank to 0
2. For every state set succRank to the minimum of the ranks of all children
3. For every state with infinite rank and finite succRank, set rank to succRank + 1
4. Repeat step 2 and 3 until no more states can be assigned finite ranks
5. Remove all states that contain φ and have an infinite rank

The code that accomplishes this is broken up into several modules. The first of these is a module that finds future eventualities in the entire tableau. The code for this procedure is shown in figure 5.1

```
1  Algorithm:findFutureEventualities(tableau)
2  result = ∅;
3  for i in states do
4      new = findFutures(i.formulas);
5      for j in new do
6          if j not in result then
7              add j to results;
8          end
9      end
10 end
11 return result;
```

Figure 5.1: Algorithm to find all future eventualities in the tableau

The algorithm goes through all the states of the tableau, indexing the current state by i, and finds all the formulas of i that are future eventualities. These eventualities are appended to a list and at the end of the loop the list is returned. It is a complete list of future eventualities found in the tableau. For all eventualities $\mathtt{F}\varphi$ in the list, assign a rank 0 to every state that contains φ. The code for this is shown in figure 5.2.

We have now introduced states with finite ranks. This means that the `succRank` property of other states needs to be updated. That is because the `succRank` property reflects the minimum rank of all the children of a particular state. Therefore it is necessary to call the function in figure 5.3, which is used to rank successors.

```
1  Algorithm:assignZero(tableau, eventuality)
2  for i in states do
3      if eventuality in i.formulas then
4          i.rank = 0;
5      end
6  end
7  return tableau;
```

Figure 5.2: Algorithm to assign zero rank to states that fulfil eventualities

```
1  Algorithm:rankSuccessors(tableau)
2  ranks = ∅;
3  for i in states do
4      for j in i.children do
5          add j.rank to ranks;
6      end
7      if |ranks| > 0 then
8          i.succRank = min{ranks};
9      end
10     ranks = ∅;
11 end
12 return tableau
```

Figure 5.3: Algorithm to rank successors of states

The code for ranking successors visits every state in the tableau. At each state i, it finds the ranks of all the children of i. The minimum of these ranks is then assigned to the `succRank` field of i. The next step of the algorithm is to try and assign a finite rank to states in the tableau. The code in figure 5.4 accomplishes that.

Assigning a finite rank to states corresponds to step 4 of the description of the algorithm. The code iterates through all the states of the tableau. Every state with an infinite rank and a finite succRank is assigned a rank one greater than its succRank. Again this introduces more states with a finite rank. Their parents' `succRank` property may need to be updated so the `rankSuccessors` method is called. This process continues until there are no states with an infinite rank and a finite succRank in the tableau. Finally a method is called to remove all the states that contain the eventuality and have an infinite rank. The code is given in figure 5.5.

The code goes through the states of the tableau. Every state, that has an infinite rank and contains the eventuality being tested, is added to a list. Every element of that list is then removed from the tableau. This completes

```
1  Algorithm:assignFinite(tableau)
2  prev = copy(tableau);
3  done = False;
4  while not done do
5      for i in states do
6          if i.rank = ∞ and i.succRank ≠ ∞ then
7              i.rank = i.succRank + 1;
8              tableau = rankSuccessors(tableau);
9          end
10     end
11     if prev = tableau then
12         done = True;
13     else
14         prev = copy(tableau);
15     end
16 end
17 return tableau
```

Figure 5.4: Algorithm to assign finite rank to states

```
1  Algorithm:eliminateInfinite(tableau, eventuality)
2  candidates = ∅;
3  for i in states do
4      if eventuality in i.formulas and i.rank = ∞ then
5          add i to candidates;
6      end
7  end
8  for j in candidates do
9      remove j from tableau;
10 end
11 return tableau
```

Figure 5.5: Algorithm to remove states with infinite rank

the process of checking for future eventualities. The ranking algorithms offer great performance gains, as can be seen in chapter 6.

5.3 Pre-processing Formulas

Many formulas that were tested, especially the randomly generated formulas, contained redundant information that caused the procedure to generate many more states than necessary. An example of such a formula is $a = \varphi \wedge \varphi \wedge \ldots \wedge \varphi$. Assuming the length of a is n, the procedure would apply an α-rule n times. This is unnecessary because a simplifies to $a = \varphi$. If a formula $b = \varphi \vee \varphi \vee \ldots \vee \varphi$, with length n occurs is a prestate of the tableau, then the procedure would introduce n unnecessary branches in the tableau. It is useful to have an algorithm that would detect such redundancies and simplify the formulas accordingly. Such an algorithm is shown in figure 5.6.

```
1  Algorithm:preProcess(formula)
2  p = postfix(formula);
3  stack = ∅;
4  for i in p do
5      if not isConnective(i) then
6          stack.append(i);
7      else
8          if binary(i) then
9              v1 = stack.pop();
10             v2 = stack.pop();
11             if v1 ≠ v2 then
12                 f = '(' + v2 + ' ' + i + ' ' + v1 + ')';
13             else
14                 f = v1;
15             end
16         else
17             v = stack.pop();
18             f = '(' + i + ' ' + v + ')';
19             stack.append(f);
20         end
21     end
22 end
23 return stack[0]
```

Figure 5.6: Algorithm to pre-process formulas

The pre-processing algorithm first converts the formula to postfix notation. This is a notation whereby the operands are listed before the operator. For

example the formula (φ `U` `X`ψ) would be $\varphi\psi$`XU`. As the algorithm iterates through the postfix string, it pushes all variables onto a stack. If the current character of the postfix string is a connective then the algorithm pops off the appropriate number of operators, builds up an expression from the operands and the operator and pushes that expression back onto the stack. In the case of a binary operator the algorithm first checks whether the two operands are the same as each other. If they are then the expression is simply one of the operands. At the end of the postfix string the stack will contain exactly one expression, which is the simplified formula.

Chapter 6

Testing and Analysis

6.1 Introduction

This chapter presents the results of the experiments performed. These include correctness testing, recording runtimes of Wolper's and Schwendimann's procedures on series of patterns as well as on randomly generated formulae. This is followed by a short summary of the performance of automata-based tools, found in [11]. The chapter end off with a brief empirical analysis of the implementation of Wolper's algorithm.

6.2 Correctness Testing

Many bugs with the implementation were found during its development. Therefore it is necessary to do theoretical analysis of the code as well as experiments to verify the correctness. A large set of random formulas was generated for this experiment. These formulas were tested on the implemented program as well as on other available tools. The implementation of Schwendimann's tableau was chosen for the comparison. Both tools were returned consistent results. At this point we only consider the final outcome of the two tools. A comparison of running times is done later in the chapter. The results of this experiment were pleasing as the two tools returned the same results for every formula that was tested.

Another way to test the correctness was to generate formulas that we know are satisfiable and formulas we know are not satisfiable. We would then test these formulas on the tableau and see if it returns the correct result. To generate these formulas, the generator developed by Marco Montalli was used. The tableau was tested with a large set of satisfiable formulas and it found all of them satisfiable. The same consistent result was found with the unsatisfiable formulas where every one of them was declared unsatisfiable.

6.3 Pattern Series

The first set of tests conducted were on pattern series. These patterns used are the same that Rozier and Vardi used for their tests of automata based tools. This section presents the running times of the tableau tool on the different patterns. The running time of Schwendiman's one-pass tableau are included for comparison purposes.

The first pattern is the so called E formula pattern, which is a conjunction of eventualities. It is of the form $\mathtt{F}p_1 \wedge \mathtt{F}p_2 \wedge \ldots \wedge \mathtt{F}p_n$. The pattern was tested on input sizes varying from $n = 1$ to $n = 10$. The running times can be seen on the graph in figure 6.1.

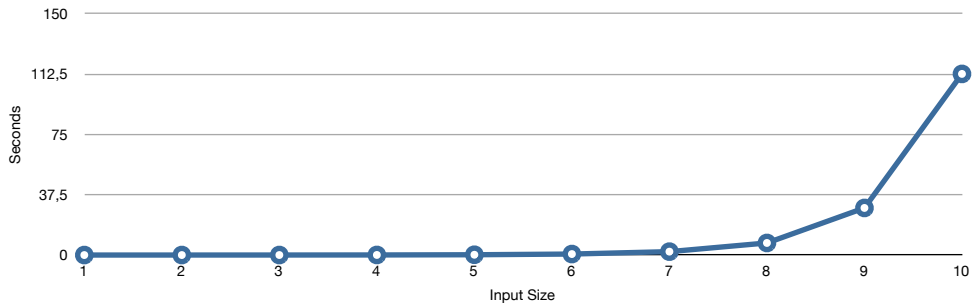

Figure 6.1: Running times of algorithm on E formulas

The algorithm is not able to verify E formulas of more than ten conjuncts within a reasonable time. As the input grows beyond $n = 7$ we begin to see a sharp increase in running time. This increase is caused by the procedure that checks whether eventualities are realised. For example when $n = 10$ the program generates over 120 000 nodes in the tableau and 10 eventualities have to be checked. We expect the one-pass tableau to perform much better.

Figure 6.2 shows the running time of Schwendiman's one-pass tableau on the same E formulas. The one-pass algorithm grows linearly because there is no procedure for checking eventualities.

The next pattern tested was the S formula pattern. It has the form of $\mathtt{G}p_1 \wedge \mathtt{G}p_2 \wedge \ldots \wedge \mathtt{G}pn$. There are no eventualities in this formula pattern so we should expect much better results compared to the E formulas. Indeed, the graph in figure 6.3 confirms this expectation.

The running times seen in figure 6.3 are dominated by the procedure of removing prestates. This is a costly procedure because it iterates through all the nodes of the tableau and for every node which is a prestate it iterates through all its parents and all its children. In a highly connected graph the running time sharply increases. Figure 6.4 shows the running times of Schwendiman's

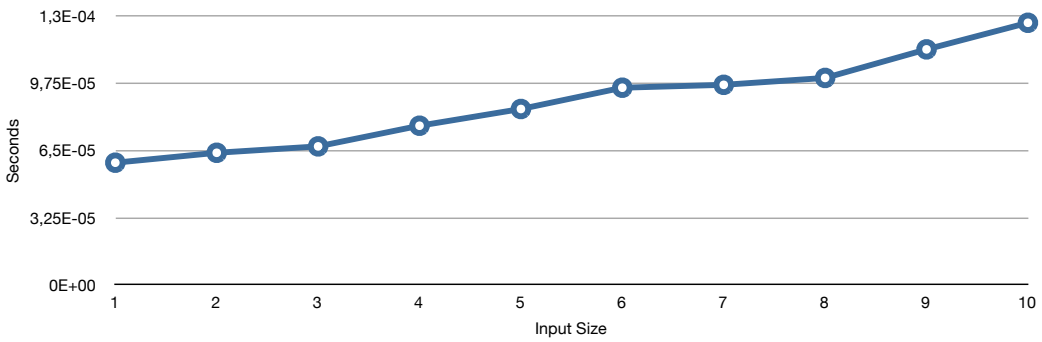

Figure 6.2: Running times of one-pass algorithm on E formulas

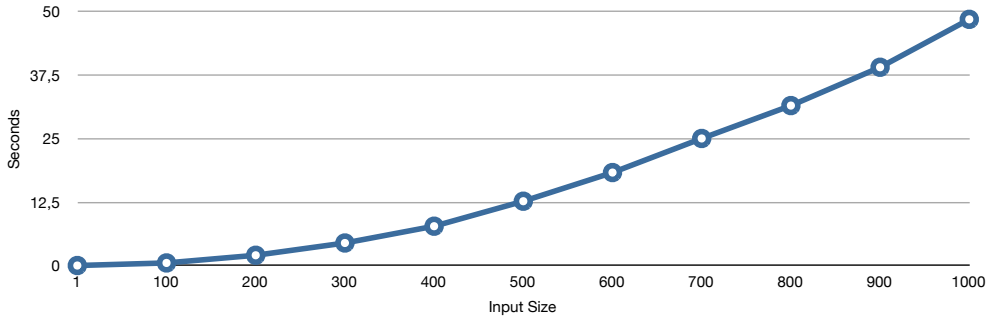

Figure 6.3: Running time of algorithm on S formulas

one-pass tableau on S formulas.

In this case both graphs have the same shape. This is because the elimination phase of Wolper's algorithm is very short, since no eventualities are found. The one-pass algorithm has no elimination phase either. The difference is only a constant factor. One reason for this difference is that different programming languages were used to implement the two algorithms.

The next set of patterns are nested *until* operators. The first of them, $U1$ is of the form $(((p_1 \texttt{ U } p_2) \texttt{ U } p_3) \texttt{ U } \ldots p_n)$. The running times of Wolper's algorithm are shown in figure 6.5.

Again the algorithm only manages to verify formulas with a very low n value. That is because for a formula of size n there are n eventualities to be checked. Also for $n = 7$ the program generates over 68 000 nodes. That is why it is not possible for the algorithm to perform in reasonable time. The one-pass algorithm however is expected to run in linear time here and indeed the graph in figure 6.6 shows that.

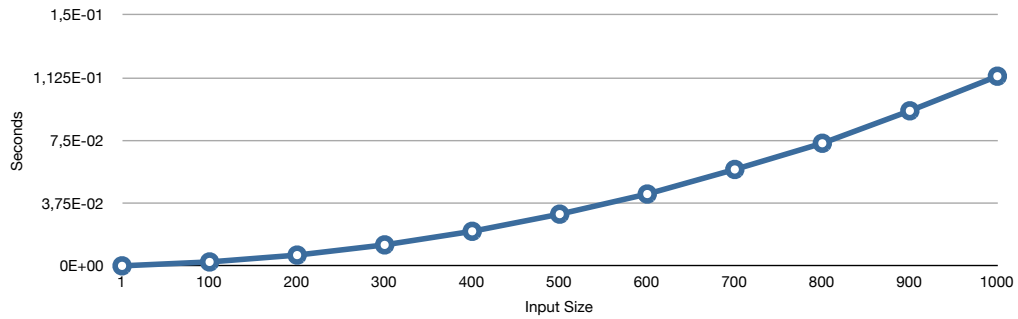

Figure 6.4: Running time of one-pass algorithm on S formulas

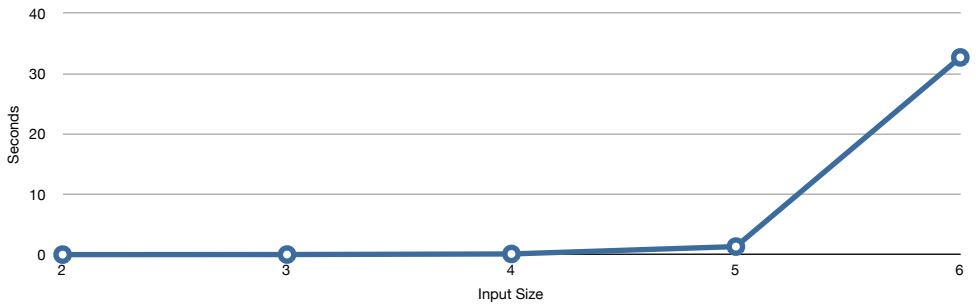

Figure 6.5: Running time of algorithm on $U1$ formulas

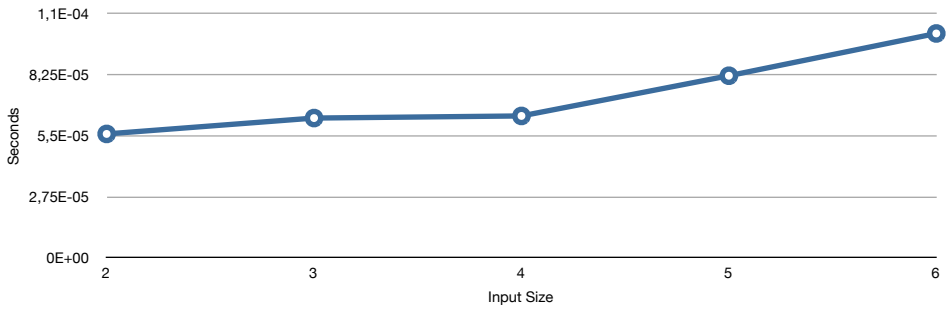

Figure 6.6: Running time of one-pass algorithm on $U1$ formulas

The $U2$ formula pattern has the form of $(p_1 \texttt{ U } (p_2 \texttt{ U } (p_3 \texttt{ U} \ldots p_n))$. Formulas of this pattern also contain n eventualities but the program generates very few states compared to the $U1$ pattern. That is why the algorithm manages to verify much larger input formulas. The running times of the algorithm are shown in figure 6.7.

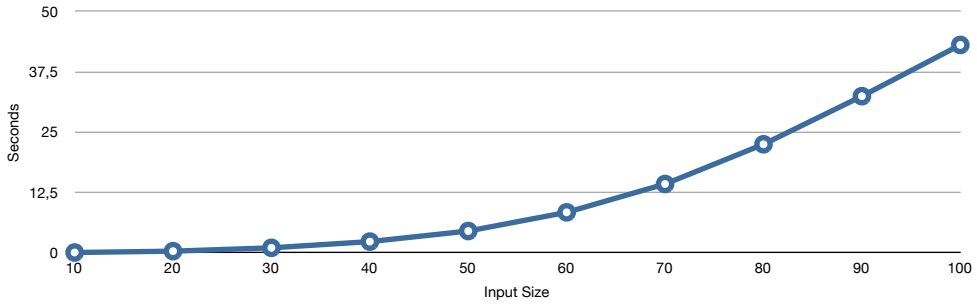

Figure 6.7: Running time of the algorithm on $U2$ formulas

The one-pass algorithm performs better but as can be seen from the graph in figure 6.8, its running time curve grows at a similar rate to Wolper's algorithm.

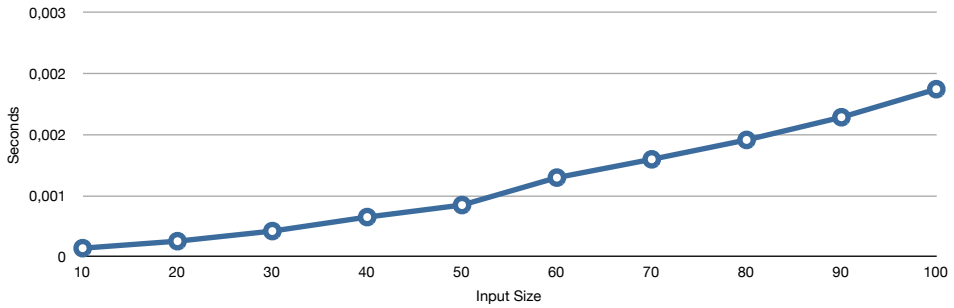

Figure 6.8: Running time of one-pass algorithm on $U2$ formulas

The last pattern set are the so called C patterns. They are made up of elements of the form $\texttt{GF}p_i$. The pattern $C1$ is a disjunction of these elements and $C2$ is a conjunction. The running times of the algorithm on $C1$ formulas are shown in figure 6.9.

The algorithm manages to verify reasonably sized formulas of the $C1$ pattern because very few nodes get generated. The small number of states allows

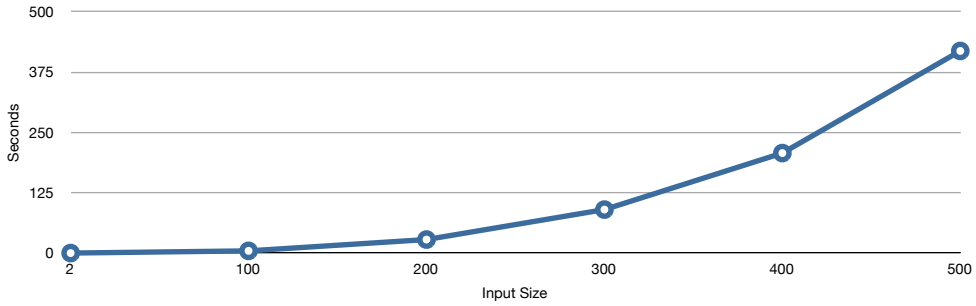

Figure 6.9: Running time of algorithm on $C1$ formulas

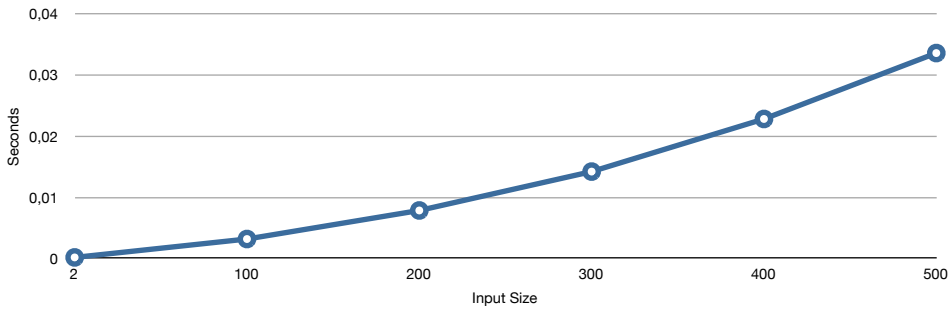

Figure 6.10: Running time of one-pass algorithm on $C1$ formulas

the procedure to check all n eventualities in a reasonable time. Figure 6.10 shows the performance of the one-pass algorithm on the same set of formulas.

The $C2$ pattern is a conjunction of the form $\mathtt{GF}p_1 \wedge \mathtt{GF}p_2 \wedge \ldots \wedge \mathtt{GF}p_n$. The running time of the algorithm on $C2$ formulas is shown in figure 6.11.

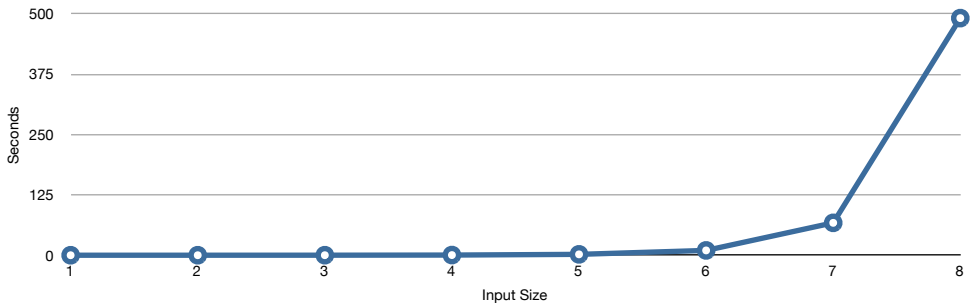

Figure 6.11: Running time of algorithm on $C2$ formulas

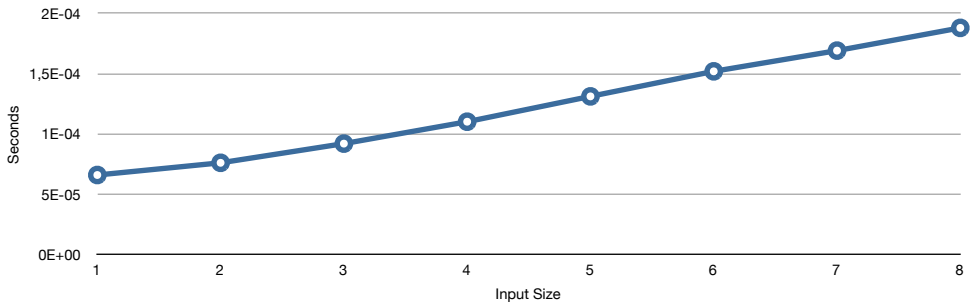

Figure 6.12: Running time of one-pass algorithm on $C2$ formulas

The running time increases sharply after $n = 7$ because the program begins to generate many nodes on the tableau. The need to check whether eventualities are fulfilled together with the high number of states results in a poor performance. The one-pass tableau, where there is no elimination procedure, is expected to perform significantly better. The results are shown in figure 6.12. It is clear that the one-pass tableau runs in linear time for this formula pattern. That is because there is no need to check for eventualities.

It is strange to see that the performance on the $C1$ pattern is significantly better than the performance on the $c2$ pattern because $C1$ is a disjunction and $C2$ is a conjunction. Usually we would expect more states to be generated

on disjunctive formulas because of the branching in the tableau that they introduce. However this was not the case with the C patterns.

Other pattern series used to compare Wolper's tableau to Schwendimann's tableau were generated by M. Montalli [8]. These patterns use two parameters n, d, shown on the abscissa of the graphs on Fig. 6.13 and 6.14, where the running times for both tools are plotted. The first parameter is the number of propositional variables and the second is the depth of nesting of specific temporal patterns. For instance, in one of the series the formula $\mathtt{F}(a_1 \wedge \mathtt{XF}(a_1 \wedge \mathtt{XF}a_1))$ has parameters $(1, 2)$, meaning that it contains 1 variable and the pattern $\mathtt{XF}$ has a nesting depth 2.

For a description of the other patterns and further details on them, see [7].

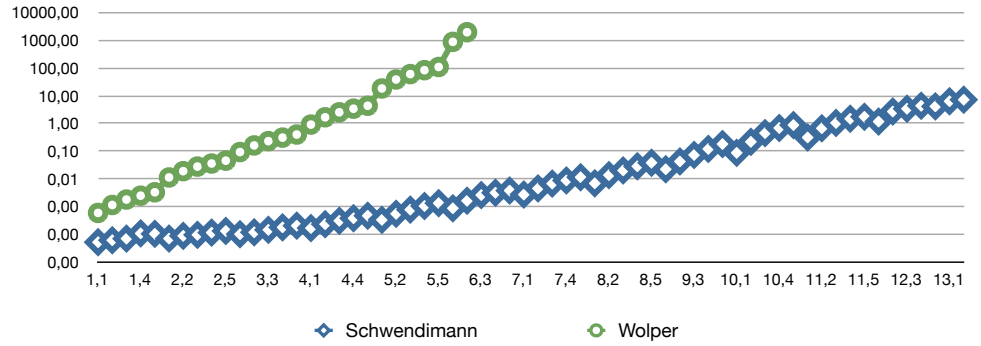

Figure 6.13: Running time of Montali's satisfiable formulae

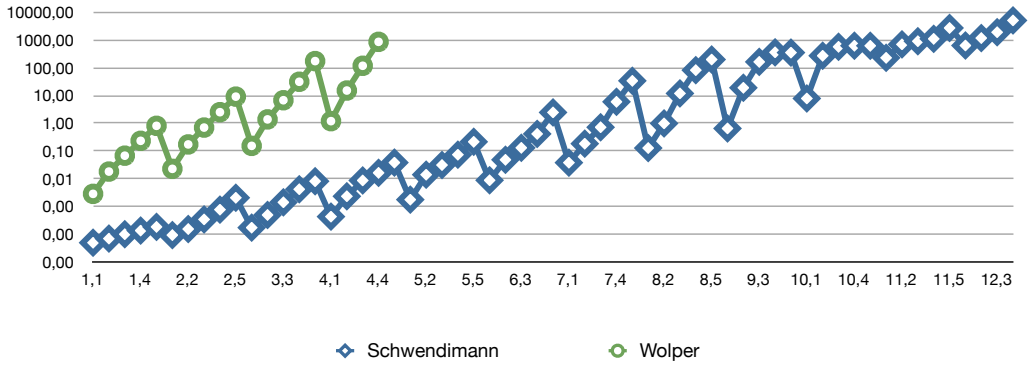

Figure 6.14: Running time of Montali's unsatisfiable formulae

6.4 Random Formulas

A random formula generator was used to generate random test formulae of different sizes. The parameters used were: n – the number of propositional variables, and d – the nesting depth for operators. Wolper's tableau manages to verify formulae with low nesting depth in a very reasonable time. When the depth is increased to 5 and beyond, the algorithm begins to struggle. As we can see from the graph in figure 6.15, which shows random formulae of two variables and nesting depth of 5, certain formulae go well beyond the 0.5 second mark. These are all the spikes in the graph, some of which reach times of over 100 seconds. For random formulae of more than 6 propositional variables and nesting depth over 5, Wolper's tableau has running times of over 1000 seconds while Schwendimann's procedure is consistently fast.

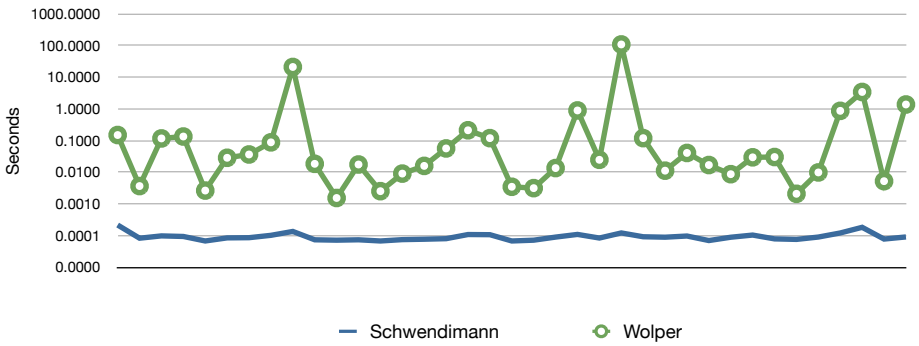

Figure 6.15: Running time of random formulae

6.5 Empirical Algorithm Analysis

As described in chapter 4 the tableau procedure is divided into a construction phase and elimination phase. This section provides some empirical analysis of running times of specific sub-procedures of the algorithm.

The construction phase of the algorithm creates a pretableau, that is there are both states and prestates. The construction procedure creates the initial pretableau, which is made up of one prestate containing only the input formula. The procedure then applies to the tableau the *alphaBetaRules* sub-procedure and the *nextTimeRule* sub-procedure until such time that the application of these two sub-procedures does not yield new sates. When that happens the construction procedure terminates. It is guaranteed to terminate in a finite number of steps because of the finiteness of the extended closure of LTL formulas. The running time of the *alphaBetaRules* sub-procedure in the worst

case is exponential to the size of the tableau because it iterates through all the nodes currently in the tableau but at every step, it dynamically adds nodes to the tableau. It stops adding nodes when all the protostates have been fully expanded into states and stores these newly created states in a list. The running time of the *nextRule* sub-procedure is much lower because it only iterates through the newly created states. For each new state it creates a successor prestate and checks whether that prestate already exists in the tableau, which takes time linear to the number of prestates. If the prestate exists then the procedure loops back to it otherwise a new prestate is creates. If future versions the search can be reduced to constant time if hash-tables are used.

In practice the construction phase usually completes in a reasonable time even for inputs that cause very large graphs to be generated. Experiments have shown that the construction phase comfortably generates hundreds of thousands of states in a reasonable time. Very seldom does the construction stage fail to complete in a reasonable time. The elimination stage is what causes the procedure to run for unreasonably long periods of time.

The elimination procedure begins by removing all the prestates from the pretableau returned at the end of the construction phase. A loop iterates through all the prestates and runs the *remove* procedure on each prestate. The *remove* procedure has a bad worst case complexity because it iterates through all the parents of the node being removed as well as through all its children. In highly connected graphs that have hundreds of thousands of nodes it is likely that a single prestate could have hundreds or even thousands of parents and a similar number of children. For example the prestate containing only $\top$ as its formula may have thousands of parents because it will be the successor prestate of all the states that do not contain formulas starting with the `X` operator. For some large graphs the algorithm fails to proceed beyond the removal of prestates phase.

The next stage of the elimination procedure is to remove inconsistent states. This is done in linear time and is therefore always completed quickly. Finding eventualities in the tableau is also done in linear time so that process is also completed in a very reasonable time. It returns a list of all the eventualities in the tableau. Checking whether these eventualities are fulfilled however is not a cheap task.

The naive approach for checking eventualities which was initially used always ran in exponential time. The ranking method for checking eventualities offers a great improvement. Firstly the ranking method assigns a rank of zero to all the states that fulfil the eventuality currently in question. That is done in linear time so the performance is good. The next step is to through the states and try to assign finite rank to them. A lot of unnecessary work is done here so in the worst case the time taken is exponential to the number of states. On average however the time taken is reasonable. Lastly the method removes all states that contain unfulfilled eventualities. The *removeState* procedure is

costly so this results in an exponential worst case performance but reasonable on average. This entire process is repeated for every eventuality found. In some cases certain eventualities are initially fulfilled but later on as states are removed from the tableau because they have no successors, the eventualities may no longer be fulfilled so the whole check needs to be repeated. Again in the worst case the running time will be exponential but on average it is reasonable.

Therefore the running time of the algorithm is usually dominated by the elimination phase. More specifically the procedures for removing prestates and states are quite costly as well as the procedure for checking eventualities. The elimination phase completes in a reasonable time when the number of states generated during the construction phase is relatively small but for a large number of states the running time of the elimination phase increases sharply. This is even more so when a large number of eventualities need to be checked and removed.

6.6 Comparisons with Automata-based Tools

In their paper, Rozier and Vardi tested both explicit and symbolic automata-based tools for LTL satisfiability checking. The explicit tools were SPIN and SPOT and the symbolic tools were CandenceSMV, NuSMV and VIS. The explicit tools construct the states explicitly and then perform a search whereas the symbolic tools represent the models using binary decision diagrams. The implementation of Wolper's tableau compares well with the explicit tools, but is not as efficient as the symbolic ones. On the other hand, Schwendimann's tableaux have proved to be much more efficient on some formulae patterns.

Chapter 7

Conclusions and Future Work

The purpose of doing the experimental analysis reported in this thesis was twofold: to verify the correctness of the implementation, and to test the performance. The results of the performance testing can be used to determine the suitability of this tool for industrial use, at least for specific formulae patterns.

The correctness was successfully verified with practical certainty, as the last version of the implementation of Wolper's tableau returned correct answers for all the formulae that were tested on it. Also the individual sub-procedures of the tableau were tested independently to ensure their correctness.

As for performance, for formula patterns with no eventualities to be checked, the running times of Wolper's tableau and Schwendimann's tableaux grow at the same rate, typically the growth of the running time of Schwendimann's tableau having much lower constant factors. However, for the formula patterns described above that cause generation of many nodes and there are many eventualities, the running time of Wolper's tableau grows exponentially on the input size, whereas Schwendimann's remains linear.

A suitable future work would be to design a "hybrid" tableau procedure for checking satisfiability. That is a procedure that combines ideas from Wolper's and Schwendimann's methods. A possible way to do this is to organise the procedure to do depth-first search instead of breadth-first search. Wolper's algorithm makes use of breadth-first search, which therefore means that the entire tableau has to be generated before the result of the test for satisfiability can be found. This is a highly unnecessary cost because it suffices for one branch of the tableau to produce a model for a formula ϕ. This would allow us to declare ϕ satisfiable immediately, without having to generate the rest of the tableau structure. Indeed, depth-first search shows a lot of promise and it would be useful for it to be implemented.

An obvious improvement to the current implementation would be to port the code to a different programming language such as C. It is believed that C could reduce the running times of programs by a factor of up to 100 when compared to Python. This would allow for a more accurate comparison be-

tween Wolper's tableau and Schwendimann's tableau. The running times of both tools grew at a similar rate for certain formula patterns that were tested, particularly the patterns that do not contain eventualities, Wolper's method had much higher constant factors. This is largely attributed to the fact that Python is slower than OCaml, which Schwendimann's procedure was implemented in.

Bibliography

[1] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[2] E.A. Emerson. Temporal and modal logics. In *Handbook of Theoretical Computer Science*, pages 995–1072. 1990.

[3] Valentin Goranko. Course notes, FIRST Autumn School on Modal Logic. Copenhagen, Denmark, 2009.

[4] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[5] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Vol. 1: Specifications*. Springer, New York, 1992.

[6] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Vol. 2: Safety*. Springer, New York, 1995.

[7] M. Montali, P. Torroni, M. Alberti, F. Chesani, E. Lamma, and P. Mello. Abductive logic programming as an effective technology for the static verification of declarative business processes. *J. of Algorithms in Cognition, Informatics and Logic*, page to appear, 2009.

[8] Marco Montali. personal communication. 2009.

[9] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. Foundations of Computer Science*, pages 46–57, 1977.

[10] A.N. Prior. *Past, Present and Future*. Oxford University Press, 1967.

[11] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[12] Stefan Schwendimann. A new one-pass tableau calculus for pltl. In *Proceedings of TABLEAUX'98*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 277–291. Springer-Verlag, 1998.

[13] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics of Concurrency: Structure versus Automata*, pages 238–266. Lecture Notes in Computer Science, Vol. 1043. Springer, Berlin, 1996.

[14] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28:119–136, 1985.