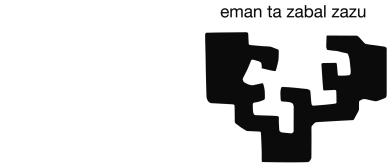
An implementation of PLTL-model-checking using context-based tableaux and SAT-solvers

Luis Pérez Guerrero

Supervisors:
Montserrat Hermo & Paqui Lucio



Contents

List of Figures **5**

List of Tables **6**

1 Abstract **7**

2 Acknowledgement **9**

3 Introduction **11**

4 Background **15**
4.1 Propositional Logic 15
4.2 Temporal Logic 16

5 Propositional Linear Temporal Logic **19**
5.1 Syntax and Semantics of PLTL 19
5.2 Our Language for Symbolic Model Checking 20
5.3 NNF and simplification 21
5.4 Tableau Rules 21

6 The algorithm **24**

7 Examples **28**
7.1 A counterexample showing that one formula is false 28
7.2 A closed tableau showing that one formula is True 30
7.3 More examples 32

8 Implementation **41**
8.1 Modules 41

9 Description of the stages **43**
9.1 Parser 43
9.2 CNF-SAT 44
9.3 Tableau 45

10 Use of the program **47**
10.1 Connectives 47
10.2 Usage 47
10.3 Dependecies 48

11 Conclusion **48**

References **50**

List of Figures

1 A proposition p on a infinite sequence of states. 16
2 Always temporal connective. 16
3 Until temporal connective. 16
4 Eventually temporal connective. 17
5 Next temporal connective. 17
6 Release temporal connective. 17
7 Cyclic sequence of states. 20
8 Negation propagation rules 21
9 How a child feeds 28
10 Opened tableau 29
11 Closed tableau 31
12 Another way to breastfeed babies 32
13 Two finite directed graphs G_1 and G_2. 32
14 Opened Tableau for G_1 33
15 A counterexample. 35
16 Closed Tableau for G_2 36
17 Example of conversion. 41
18 Example of conversion. 41
19 Application scheme. 42
20 Scheme of operation of the parser's module. 43
21 Scheme of operation of the SAT and CNF modules. 44
22 Scheme of operation of the Tableau module. 46

List of Tables

1 Truth table for example 1 15
2 Minimal set of α-rules 21
3 Minimal set of β-rules 22
4 The next-state rule 22
5 Derived α-rules 22
6 Derived β-rules 23

1 Abstract

The software development and the increase of software complexity has shown that an automation of the software verification process can be very helpful. Among several methods of verification, the formal verification methods have emerged as the ones with sound mathematical foundations, allowing reasoning about system properties.

Temporal logics are formal systems for reasoning about time, for that, they use temporal operators. Temporal logics have found extensive application in computer science, because the behavior of both hardware and software systems can be seen as a sequence of events that evolves along time. *Model Checking* is one of the most popular methods for automated verification of concurrent systems. Model checking using formulas to represent the system is called *Symbolic Model Checking*. Properties to be verified are usually expressed in some temporal logic. Moreover, if the property does not hold, the checker returns a counterexample.

For larger systems, model checkers need too many resources. To try to reduce the impact of this problem we propose advances in previous works in symbolic model checking. We blend the concept of propositional satisfiability (SAT) solvers for efficient auxiliary tools with linear temporal logic. We use tableaux method for temporal logics to check whether a system satisfies a property. Modern SAT-Solvers are used for the non-temporal reasoning part of the tableau construction. In this master thesis we have developed in Python an algorithm that uses SAT solvers to solve purely propositional formulas leaving the temporary ones for tableaux methods.

2 Acknowledgement

Thanks to my wife Manuela for cheering me up during difficult times. I think this job is more of her than mine. Thanks also to my children Luis and Carlos who have also encouraged me to continue. I hope to return to them the time I have stolen.

Thank you Paqui Lucio and Montse Hermo, for all your support and understanding of the difficulties I have had in developing this work.

Thanks to all!

3 Introduction

As the complexity of the hardware and software systems grows, the need for some mechanism to check the correct behavior of the system also increases. The process of providing complete manual tests became too cumbersome and questioned whether long and laborious proofs of correction could be relied upon. The development of hardware and software is often much more expensive than expected, especially when defects appear at the end of the process. The sooner a development defect is discovered, the less impact it has on both time scales and costs. Errors discovered in the last stages of the development cycle risk the integrity of a system. This marked a trend towards the automation of some parts of the process, allowing the human to provide guidance to an automatic tool.

The formal methods of software verification are the procedures to verify in a scientific way that there are no ambiguous specifications of the system, articulate implicit assumptions, expose failures in the system requirements and its rigor allows a better understanding of the problem. Formal methods have emerged as points of view that allow verifying the development of systems with a solid mathematical and logical basis, providing significant benefits to improve the quality of programs and allowing reasoning about system properties such as safety, life, reliability or safety. Formal methods must be present in software development, as they have become the basis for applying the verification process.

Model Checking [4, 15] is one of the most popular methods of automated verification of concurrent systems. Properties to be verified are expressed in some temporal logic. Model Checking is an algorithmic method for determining whether a system – a hard/software design– satisfies a property expressed as a temporal logic formula. Moreover, if the property does not hold, the checker returns a counterexample: a trace/model of the system that does not satisfy the property. This countermodel acts as a 'certificate' of failure and its role is to help the user to identify the source of the problem. For years, no certificate was produced if the system meets the property. Hence, for positive answers the model checker should be trusted. Given the high complexity of the implementation of model checkers, its verification is a great challenge, though recently some model checkers has been formally verified [6]. An alternative is to return a proof evidence for every positive answer [12, 11]. Compared to other formal verification techniques model checking is practically automatic.

Moreover, sometimes model checking is used to generate counterexamples as the desired output (cf. e.g. [7] where counterexamples are mobile robots trajectories). In this framework a proof of a property (that is expected to do not hold) should help to find (and to fix) some mistake in the system specification.

Model checking using formulas to represent the system is called *Symbolic Model Checking* [3, 10]. The term emphasizes that the model of the system is represented symbolically, namely, by a formula. The earliest symbolic model checker SMV [10] applies *Ordered Binary Decision Diagrams* OBDDs [2, 3] as a canonical form for boolean formulas that is very compact because of variable-sharing. Very efficient algorithms has been developed for manipulating OBDDs. This made possible to verify systems with an extremely large number of states –some orders of magnitude larger than could be handled by the explicit-state model checkers (e.g. SPIN [9]).

However, for larger systems, the OBDDs generated during model checking become too large, and the generation of a variable ordering that results in small OBDDs is often time consuming or needs manual intervention. For many examples no space efficient variable ordering exists. Over the last years,

the increase in power of modern SAT/SMT solvers has been so awesome that they have became the key enabling technology for symbolic model checking tools. Propositional satisfiability (SAT) is the problem of determining, for a formula in the propositional calculus, whether there exists a satisfying assignment for its variables. SAT-solvers are nowadays very efficient tools for deciding SAT (see [14] for a survey). In the positive case, they provide the models of the formula. Their several applications fueled the investigation in more efficient and scalable methods.

There exists many tableaux techniques for a rich variety of temporal logics. The most difficult task in tableaux methods for temporal logics is to check out the fulfillment of eventualities, i.e. formulas of the form "eventually φ". Traditional tableaux methods require two phases to perform this test. In the first phase, they construct a graph of states. This graph represents all possible pre-models. In the second phase, for each state s that contains some "eventually φ", a graph-theoretic algorithm should look for a state, reachable from s, that satisfies φ. Moreover, nodes with negative test should be pruned. The two-pass tableaux methods fails to carry out the classical correspondence between tableaux and sequents that associates a sequent-proof to any closed tableau. To avoid the second phase and, hence, to keep the ability of generate proofs from tableaux, in [8], dual systems of tableaux and sequents, for PLTL, was presented. PLTL is the well-known propositional linear temporal logic [13]. Both are proved to be correct and complete. The tableau method TTM in [8] is a tree-style one-pass tableaux system that make use of the so-called context of an eventuality to force its fulfillment. The context of a formula is simply the set of formulas that accompanies it in the node. Therefore, we say that TTM is a *context-based* tableaux method. When TTM checks whether a property φ holds or not, it is always able to issue a certificate: either a countermodel or the complete explanation (formal proof) of why φ is true. In symbolic model checking two-pass tableaux are extensively used to construct different kind of state machines (graphs) for representing systems and properties (cf.e.g. [5]). The aim is to evaluate the context-based approach of TTM for implementing a symbolic model checker amenable for producing certificates as well as counterexamples.

SAT solvers provide a generic combinatorial reasoning and search platform. The underlying representational formalism is propositional logic described in CNF (Conjunctive normal form). Frequently , the full potential of SAT solvers only becomes apparent when one considers their use in applications that are not normally viewed as propositional reasoning tasks. In these applications Satisfiability Solvers and is hidden from the user: the user only deals with the appropriate higher-level representation language of the application domain. Note that the translation to SAT generally leads to a substantial increase in problem representation. However, large SAT encodings are no longer an obstacle for modern SAT solvers. In fact, for many combinatorial search and reasoning tasks, the translation to SAT followed by the use of a modern SAT solver is often more effective than a custom search engine running on the original problem formulation. The explanation for this phenomenon is that SAT solvers have been engineered to such an extent that their performance is difficult to duplicate, even when one tackles the reasoning problem in its original representation.

In this work we try delegating to SAT solvers, the non-temporal reasoning part of the tableau construction will likely lead to a competitive tool. In this task we have created an algorithm which uses an SAT-Solver to manage model satisfiability. This way we can apply an effective algorithm to the temporal logic and reach more complex formulas, thus more complex systems described and checked. The main reason is that context-based tableaux are much more simpler to construct when their input is the sort of formulas that specifies a transition system (along with a temporal property to test). Only a very

restricted subset of temporal formulas can be used to design such systems. Coding them into classical propositional formulas, allows us to leave the temporal reasoning only in the scope of the properties to be checked and not in the systems to be tested. By contrast with proposals that translate the whole model checking problem on one SAT-solver query (e.g. [11]), our proposal combines SAT-solving with tableau-based temporal reasoning. As a consequence, the proof certificate output shall contain the trace of the temporal reasoning, which provides hints that are close to the tested system.

In the development of this model checker we have used PySAT wich is a Python (2.7, 3.4+) toolkit, which aims at providing a simple and unified interface to a number of state-of-art Boolean satisfiability (SAT) solvers like CaDiCaL, Minicard, Minisat, Glucose among others. All these SATs can be used in this implementation with slight code modifications.

This master thesis is organized as follows. In Section 4 we describe the basics of propositional and temporal logic. In Section 5 we describe the syntax and semantics of PLTL and its relationship with Symbolic Model Checking. Also, we recall the set of rules for the tableau method TTM. The reformulation of TTM to be used as a symbolic model checker is presented in Section 6. This section contains an algorithmic description of our proposal, which basically consists of using a SAT-solver to help finding models of a temporal formula. In Section 7, we include two examples to show how our algorithm works. Finally, we draw the conclusions and routes for the future work.

4 Background

4.1 Propositional Logic

The propositional logic is the branch of the logic that joins atomic propositions into more complex propositional formulas (PL-formulas) using logical connectives. In this master thesis we are using pure propositional formulas (PPL-formulas from here on) representing propositional formulas composed by propositions and boolean combinations of propositions. These PPL-formulas can represent statements and these can be part of more complex statements by themselves. To represent PPL-formulas we use greek low letters for instance φ or ψ. A PPL-formula is constructed this way:

- Atomic propositions: an unbounded set of symbols that are denoted by lower case letters (p, q, r). They are also known as boolean variables.
- Classical connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\implies$ (implies), $\iff$ (equivalence),...

The propositional logic works as follows: each atomic proposition has a **Truth Value** that is assigned true or false (never both). The Truth Value of each statement can be obtained through certain rules collected in **Truth tables**. These tables are made by mapping the interpretation of each variable that builds the formula. At the end all the variables are evaluated and the Truth Value is assigned to the formula. The mapping is done this way:

$$M : Prop \rightarrow Bool$$

Where *Prop* is the set of atomic propositions.

Example 1 *Given the phrase "If I study and I pass the exam", we could split the statement in 2 atomic prepositions:*

$$p = \textit{If I study}$$
$$q = \textit{I pass the exam}$$

In the example above the related PPL-formula would be the following one:

$$p \implies q$$

The truth table for that formula is:

p	q	$p \implies q$
$True$	$True$	$True$
$True$	$False$	$False$
$False$	$True$	$True$
$False$	$False$	$True$

Table 1: Truth table for example 1

A model in Propositional Logic is any interpretation M. When the Truth Table of φ by M is True, it is said that φ holds in M. The satisfaction relation $\models$ describes when a formula holds in a model. In propositional logic, $M \models \varphi$ is read as "M is a model of φ". The formula φ can be:

- *Satisfiable*: φ has at least one model M

- *Valid*: every M is model of φ
- *Unsatisfiable*: φ has no model M

Two PPL-formulas φ and ψ are equivalent iff they have exactly the same models. One of the most common mechanisms used to obtain the validity, satisfiability and unsatisfiability of the formula is to use a semantic tableaux: the formula is decomposed into its sub-formulas according to certain rules called the alpha- and betha-rules. These results in a tree-line tableau where each branch is terminated by a leaf of complementary pair of formulas (a closed branch) or by a leaf containing a set of not contradictory literals (open branch). Each open branch should be a model matching the given formula. Then if the formula is valid all branches in the tableau are open. If the formula is unsatisfiable then all branches are closed. Otherwise, the formula is satisfiable.

4.2 Temporal Logic

Temporal logics are formal systems for reasoning about time and adding temporal information to the logic. Temporal logic has found extensive application in computer science, because the behavior of both hardware and software is a function of time.

One of the most recurrent types of temporal logic is the Propositional Linear Temporal Logic (PLTL), which contains logical operators for reasoning in a infinite linear sequence of states. This sequence wants to imitate the flow of the time. PLTL-formulas are constructed exactly the same as PPL-formulas using temporal connectives in addition to classical connectives.

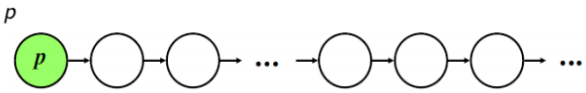


Figure 1: A proposition p on a infinite sequence of states.

- Always ($\Box$): on a given formula $\Box\varphi$, φ happens every state of the sequence of time.

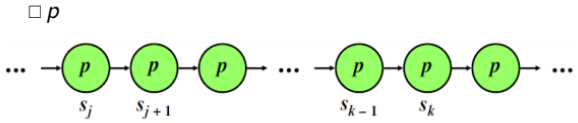


Figure 2: Always temporal connective.

- Until (U): on a given formula $\varphi \mathrm{U} \psi$, φ will be true in all states until ψ happens.

Figure 3: Until temporal connective.

- Eventually ($\diamondsuit$): on a given formula $\diamondsuit\varphi$, eventually φ will happen on a state sooner or later. Its a special case of Until connective (TUφ).

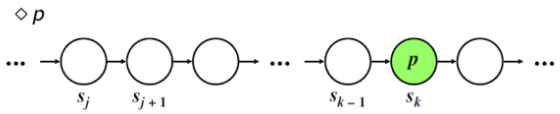


Figure 4: Eventually temporal connective.

- Next ($\circ$): on a given formula $\circ\varphi$, φ will happen on the next state.

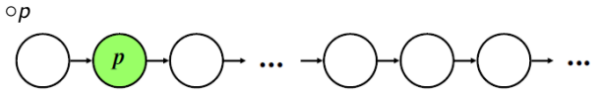


Figure 5: Next temporal connective.

- Release ($\Re$): on a given formula $\varphi\Re\psi$, ψ must be true from starting where φ becomes true for first time.

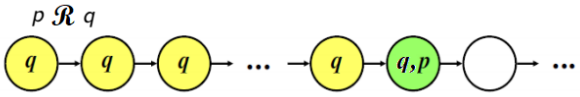


Figure 6: Release temporal connective.

With these new connectives we can introduce time in the propositions and build more complex propositional formulas.

Example 2 *Following the previous example of the phrase "If I study I pass the exam" and the atomic propositions p ("I study") and q ("I pass the exam"), we can add the temporal connectives to make a wide variety of PLTL-formulas like the following ones:*

- *"I always study":* $\Box\, p$
- *"I study , so eventually I pass the exam":* $p \implies \diamondsuit\, q$
- *"On next state I will study I will pass the exam":* $\circ\, (p \wedge q)$
- *"I will study until I pass the exam":* $p \,\mathrm{U}\, q$

As we can see in the examples below, PLTL-formulas and PPL-formulas are similar, just temporal connectives are added. We explain temporal logic more deeply on next chapter.

5 Propositional Linear Temporal Logic

5.1 Syntax and Semantics of PLTL

In this subsection, in a similar way to [8], we define the syntax of PLTL-formulas and their models. A PLTL-formula is built using the constant proposition *false*, propositional variables (denoted by lowercase letters like p and q) from a set Prop, the classical connectives $\neg$ and $\wedge$, and the temporal connectives $\circ$ and U. A lowercase Greek letter like φ or ψ denotes a PLTL-formula. Those of the form p and $\neg p$, where $p \in$ Prop, are called literals. In the rest of this paper, we employ formula instead of PLTL-formula. Formulas of the form $\varphi \mathrm{U} \psi$ and $\diamondsuit\varphi$ are called eventualities. Those of the form $\circ\varphi$ are next-formulas. The usual ASCII syntax for temporal operators are X for $\circ$, G for $\Box$, F for $\diamondsuit$, and U for U.

Definition 1 *A PLTL-structure M is a pair (S_M, V_M) where S_M is a denumerable sequence of states $s_0, s_1, s_2, \ldots$ and $V_M : S_M \rightarrow 2^{Prop}$ maps each state $s_i \in S_M$ into a subset of Prop.*

Intuitively, V_M specifies which propositional variables are necessarily true in each state.

Definition 2 *The truth of a formula φ in the state $s_j \in \mathbb{N}$ of a PLTL-structure M, which is denoted by $\langle M, s_j \rangle \models \varphi$, is inductively defined as follows:*

$\langle M, s_j \rangle \not\models$ *false*

$\langle M, s_j \rangle \models p$ *if, and only if,* $p \in V_M(s_j)$ *for any* $p \in$ *Prop*

$\langle M, s_j \rangle \models \neg\varphi$ *if, and only if,* $\langle M, s_j \rangle \not\models \varphi$

$\langle M, s_j \rangle \models \varphi \wedge \psi$ *if, and only if,* $\langle M, s_j \rangle \models \varphi$ *and* $\langle M, s_j \rangle \models \psi$

$\langle M, s_j \rangle \models \circ\varphi$ *if, and only if,* $\langle M, s_{j+1} \rangle \models \varphi$

$\langle M, s_j \rangle \models \varphi \mathrm{U} \psi$ *if, and only if,* $\langle M, s_k \rangle \models \psi$ *for some* $k \geq j$ *and* $\langle M, i \rangle \models \varphi$ *for every* $j \leq i < k$

As usual, other connectives can be defined in terms of the previous ones: $true \equiv \neg false$, $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, $\diamondsuit\varphi \equiv true \mathrm{U} \varphi$, $\Box\varphi \equiv \neg\diamondsuit\neg\varphi$. The extension of the above formal semantics to other connectives yields:

$\langle M, s_j \rangle \models true$

$\langle M, s_j \rangle \models \varphi \vee \psi$ if, and only if, $\langle M, s_j \rangle \models \varphi$ or $\langle M, s_j \rangle \models \psi$

$\langle M, s_j \rangle \models \diamondsuit\varphi$ if, and only if, $\langle M, s_k \rangle \models \varphi$ for some $k \geq j$

$\langle M, s_j \rangle \models \Box\varphi$ if, and only if, $\langle M, s_k \rangle \models \varphi$ for every $k \geq j$

Previous concepts can be extended to sets in the usual way: $\langle M, s_j \rangle \models \Phi$ if, and only if $\langle M, s_j \rangle \models \varphi$ for all $\varphi \in \Phi$. M is a model of Φ, in symbols $M \models \Phi$ when $\langle M, s_0 \rangle \models \Phi$. A satisfiable set of formulas has at least one model, otherwise it is unsatisfiable.

Example 3 *Let ϕ be the formula $\Box(\diamondsuit p \wedge \diamondsuit q)$. Let M be the PLTL-structure defined as $V_M(s_0) = \{\neg p\}, V_M(s_1) = \{\neg q\}, V_M(s_2) = \{p\}, V_M(s_3) = \{q\}, V_M(s_{2k-1}) = V_M(s_2)$ and $V_M(s_{2k}) = V_M(s_3)$ for every $k \geq 3$. It is easy to see that, according to Definition 2, $M \models \{\Box(\diamondsuit p \wedge \diamondsuit q)\}$.*

We are interested in cyclic PLTL-structures to construct models for satisfiable sets of formulas. They are defined in terms of paths over cycling sequences. Any infinite sequence $s_0, s_1, \cdots, s_k, \cdots$ involves an implicit successor relation, namely R, such that $(s_i, s_{i+1}) \in R$ for all $i \in \mathbb{N}$. A finite sequence $S = s_0, s_1, \cdots, s_k$ is said to be cyclic if, and only if, its successor relation extends the implicit R with a pair (s_k, s_j) for some $0 \leq j \leq k$ (see Fig 7). Then, $s_j, \cdots, s_k$ is called the loop of S, s_j is called the cycling element of S, and the path over S is the infinite sequence

$$path(S) = s_0, s_1, \cdots, s_{j-1} \centerdot \langle s_j, s_{j+1}, \cdots, s_k \rangle^{\omega}$$

where $\_ \centerdot \_$ is the infix operator of concatenation of sequences and U^{ω} denotes the infinite sequence that results by concatenation of the sequence U infinitely many times. Naturally, for any non-cyclic finite sequence S we consider that $path(S) = S$.

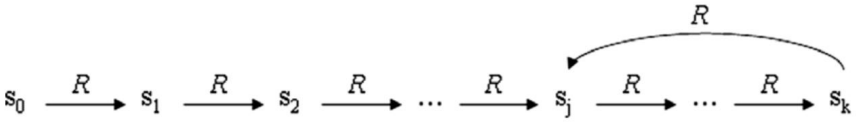


Figure 7: Cyclic sequence of states.

A PLTL-structure M is cyclic if its (infinite) sequence of states $V_M(s_1), V_M(s_2), \cdots V_M(s_k), \cdots$ is a path over a cyclic sequence of states. The PLTL-structure in Example 3 is cyclic because the path over its sequence of states is $V_M(s_1), V_M(s_2) \centerdot \langle V_M(s_3), V_M(s_4) \rangle^{\omega}$.

5.2 Our Language for Symbolic Model Checking

In Symbolic Model Checking, since the system to be tested is represented by a formula, and nothing forces the user to give an specification with a unique model, the formula to be handle often is satisfied by a collection of 'logical' models.

$$a \wedge \Box(a \rightarrow \circ b)$$

This formula has more that one model in temporal logic. From the logical point of view, symbolic model checking really decides whether a property (temporal formula) φ is satisfied in all the models of the given specification (a set of premises) Φ, that is $\Phi \models \varphi$. Hence, the underlying metalogical concept is logical consequence that corresponds with derivability (since PLTL is a complete logic). The crucial fact is that the allowed formulas for specifying the system are a syntactical restriction of general temporal formulas. In other words, model checking is a particular case of temporal deduction, i.e. the deduction of a (general) temporal formula from a set of (non-general) temporal formulas as premises.

In this paper, the system to be checked (by PLTL-formulas as properties) is specified by a (finite) set of formulas of the following three forms:

INIT-formulas	Boolean Combinations of Atoms
INVAR-formulas	$\Box\varphi$ where φ is an INIT-formula
TRANS-formulas	$\Box((\bigwedge_{i=1}^{n} \ell_i) \rightarrow \bigvee_{j=1}^{m}(\bigwedge_{k=1}^{m_j} \circ \ell_k))$ where each ℓ_i is a literal

We call permanent formulas to the formulas in INVAR-formulas plus the formulas TRANS.

5.3 NNF and simplification

In this implementation we convert the formulas to NNF (Negative normal form) in most situations. This conversion allows us to reduce the number of rules used in tableaux method. Formulas like $\neg(\varphi \wedge \psi)$ do not appear in our implementation because they have previously been converted to $\neg\varphi \vee \neg\psi$ which an equivalent form in NNF.

Similarly, NNF facilitate the comparison of formulas: if two formulas expressed differently are converted to their NNF often become the same mathematical expression. Using the same example above the formulas $\neg(\varphi \wedge \psi)$ and $\neg\varphi \vee \neg\psi$ are equivalent but not equal so they could not be compared with expressions of programmatic equality. Passing them to NNF greatly helps simplify the comparison algorithms.

All the formulas of LTL can be transformed into negation normal form, where

- all negations appear only in front of the atomic propositions,
- only other logical operators true, false, $\wedge$, and $\vee$can appear, and
- only the temporal operators $\circ$, U, and R can appear.

Using the next equivalences for negation propagation, it is possible to derive the normal form.

Negation propagation			
X *is self-dual*	***F*** *and* ***G*** *are dual*	***U*** *and* ***R*** *are dual*	***W*** *and* ***M*** *are dual*
¬**X** φ ≡ **X** ¬φ	¬**F** φ ≡ **G** ¬φ	¬ (φ **U** ψ) ≡ (¬φ **R** ¬ψ)	¬ (φ **W** ψ) ≡ (¬φ **M** ¬ψ)
	¬**G** φ ≡ **F** ¬φ	¬ (φ **R** ψ) ≡ (¬φ **U** ¬ψ)	¬ (φ **M** ψ) ≡ (¬φ **W** ¬ψ)

Figure 8: Negation propagation rules

5.4 Tableau Rules

Our proposal is based on the temporal tableaux method, TTM ([8]). Its main feature is to be one-pass in the sense of not requiring a second round to check, through an auxiliary graph of states, whether all the eventualities are satisfied or not.

We use the following α-, β-, γ-rules in the construction of semantic tableaux for, respectively, α-formulas (conjunctions), β-formulas (disjunctions) and next-state formulas. α_1 denotes the (set of) conjuncts of a α-formula, β_1, β_2 denote the (sets of) disjuncts of a β-formula and γ_1 denotes the application of the operator next-state to a set of formulas. First the minimal set of α-rules is introduced.

	α	α_1
$(\wedge)$	$\varphi \wedge \psi$	φ, ψ

Table 2: Minimal set of α-rules

The minimal set of β-rules is given in Table 3 where $\neg\Sigma = \bigvee_{\sigma\in\Sigma} \neg\sigma$. All but the $(\mathrm{U})^+$ rule are the usual ones in temporal tableaux construction (see for instance [1]). The $(\mathrm{U})^+$ rule is crucial in TTM because is the only rule that uses the so-called *context* Σ to force the eventuality $\varphi \mathrm{U} \psi$ to be fulfilled as soon as possible. That is, when $(\mathrm{U})^+$ is applied to a node, labeled by a set $\Sigma, \varphi \mathrm{U} \psi$, it forces some formula in the context Σ to change in future states (while ψ is not satisfied). The $(\mathrm{U})^+$ rule only applies to a unique eventuality, and a mark is used to distinguish it. Each application of the $(\mathrm{U})^+$ rule to $\Sigma, \varphi \mathrm{U} \psi$ introduces the so-called *next-step variant* $(\varphi \wedge \neg\Sigma)\mathrm{U}\psi$ which keeps the mark. Each node of the tableau must have at most one marked eventuality. When a node of the tableau does not contain any marked eventuality, then one of them is randomly marked. In TTM, each branch in the tableau either closed or is an open branch that represents a PLTL-structure, which is a model of the input formula (for more information see [8]). The use of the $(\mathrm{U})^+$ rule avoids the construction of the auxiliary graph of stages (in two-pass tableau methods) to determine whether all eventualities are satisfied or not.

	β	β_1	β_2
$(\vee)$	$(\varphi \vee \psi)$	φ	ψ
$(\Re)$	$(\varphi \Re \psi)$	φ, ψ	$\psi, \circ(\varphi \Re \psi)$
(U)	$\varphi \mathrm{U} \psi$	ψ	$\varphi, \circ(\varphi \mathrm{U} \psi)$
$(\mathrm{U})^+$	$\Sigma, \varphi \mathrm{U} \psi$	Σ, ψ	$\Sigma, \varphi, \circ((\neg\Sigma \wedge \varphi)\mathrm{U}\psi)$

Table 3: Minimal set of β-rules

The next-rule $(\circ)$, in Table 4, is applied to jump to a new state. Given a set of formulas Φ, we denote by $\circ\Phi$ the set $\{\circ\varphi \mid \varphi \in \Phi\}$. Henceforth, the rule $(\circ)$ gets the set Σ_2 from a set composed by a set of the form $\circ\Sigma_2$ and a set of literals Σ_1.

	γ	γ_1	
$(\circ)$	$\Sigma_1, \circ(\Sigma_2)$	Σ_2	where Σ_1 is a set of literals

Table 4: The next-state rule

For simplicity of the presentation we will describe the algorithm for the minimal set of operators, however for clarity of the examples we use the usual derived temporal operators, in particular $\Diamond$ and $\Box$. In a practical implementation, and in our examples, the rules that are derived from its abbreviations are also considered. In Tables 5 and 6 we provide the usually derived α- and β-rules. It is worthy noting that the rules $(\Diamond)^+$ and $(\neg\Box)^+)$ are derived from the $(\mathrm{U})^+$ rule by respectively using that $\Diamond\varphi \equiv (\mathit{true}\mathrm{U}\varphi)$ and $\neg\Box\varphi \equiv \neg(\mathit{true}\mathrm{U}(\neg\varphi))$.

	α	α_1
$(\Box)$	$\Box\varphi$	$\varphi, \circ\Box\varphi$

Table 5: Derived α-rules

	β	β_1	β_2
$(\Diamond)$	$\Diamond\varphi$	φ	$\circ\Diamond\varphi$
$(\Diamond)^+$	$\Sigma, \Diamond\varphi$	Σ, φ	$\Sigma, \circ(\neg\Sigma \mathrm{U} \varphi)$

Table 6: Derived β-rules

6 The algorithm

The algorithm builds a tree-like tableau creating nodes and branches. Each branch terminates either by a leaf that contains a contradiction, and it is a closed branch; or by a loop and it is an open branch. Having an open branch means that there is a model that makes the set of initial formulas satisfiable. On the other hand, if all branches are closed, it means that the set of formulas in the root node is unsatisfiable. Tableaux rules apply for temporal formulas while the SAT-solver deals with classical propositional formulas.

Algorithm 1 Tableau($S, P, markedF$)

if `False` $\in S$ or a complementary pair of formulas $\{\psi, \neg\psi\} \subseteq S$ **then**
  return ($\emptyset$, $false$);
else if S is a set of pure propositional formulas and next-formulas **then**
  Let S be a set of the form $\{l_1, ..., l_n, \circ l_1, ..., \circ l_m, \circ\psi_1, ..., \circ\psi_k\}$
    ▷ where $l_1, ..., l_n$ are pure propositional formulas and $\psi_1, ..., \psi_k$ are temporal formulas.
  $input\_SAT := \{l_1, ..., l_n, ol_1, ..., ol_m\} \cup P$
  return `apply-SAT(`$input\_SAT$`, S, P, markedF)`;
else
  Let ψ be the first formula in S which is neither a literal nor a next formula;
  if an α-rule can be applied to ψ **then**
    return `apply-`α`-rule(`ψ`, S, P, markedF)`;
  `else` ▷ a β-`rule can be applied to` ψ.
    return `apply-`β`-rule(`ψ`, S, P,`$markedF$`)`;
  end if;
end if;

The main method Tableau (Algorithm 1), taking as input a system (expressed according to the syntax presented in Subsection 5.2) and the negation of a temporal property, answers whether the property is deduced from the system. Namely, the input is as follows:

- a set S containing the INIT-formulas and the negation of the (temporal) property to be checked,
- a set P containing both INVAR-formulas and TRANS-formulas without the $\Box$ operator, and where each next-formula of the form $\circ l$ is renamed by a fresh letter l'. P is called the set of permanent formulas, and
- a marked eventuality $markedF$. This is the formula which the $(\mathrm{U})^+$ rule is applied to. $markedF$ is null whenever there is not any marked eventuality in the current node.

The initial call is `tableau(`S,P, $null$`)`.

The method Tableau returns a pair (M, $open$) which is either ($\emptyset$, $false$) when $S \cup P$ is unsatisfiable or (M, $true$) when $S \cup P$ is satisfiable and M is a PLTL-structure that is model of $S \cup P$. In the latter case, the open branch that represents the model has a loop. Hence, the PLTL-structure returned by the algorithm always is cyclic.

The method apply-α-rule (Algorithm 2) receives the same parameters as Tableau and a formula that is neither literal nor next formula. The corresponding alpha rule is applied to this formula (according to

figures 2 and 5) and generates a new S' set. Using mutual recursion, Tableau is called with a new node in which the S is replaced by S': `tableau(`S'`,` P`,` *markedF*`)`.

Algorithm 2 Apply-α-rule(ψ, S, P, *markedF*)

switch ψ **:**
  case $\Box\phi$ **:**
    $S' =$ **apply-rule**$((\Box), \psi, S)$
  case $\phi \wedge \phi'$ **:**
    $S' =$ **apply-rule**$((\wedge), \psi, S)$
return `tableau(`S'`,` P`,` *markedF*`)`**;**

Algorithm 3 Apply-β-rule(ψ, S, P, *markedF*)

if $\psi = (\phi \vee \phi')$ **then**
  $(M, open)$**:=** `tableau(apply-part1-rule(`$(\vee)$`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  if $\neg open$ **then return** `tableau(apply-part2-rule(`$(\vee)$`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  end if;
else if $\psi = (\phi \,\mathrm{R}\, \phi')$ **then**
  $(M, open)$**:=** `tableau(apply-part1-rule(`(R)`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  if $\neg open$ **then return** `tableau(apply-part2-rule(`(R)`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  end if;
else if $\psi = \phi \,\mathrm{U}\, \phi'$ **and** $markedF \neq null$ **and** ψ **is not the marked eventuality then**
  $markedF := null$**;**
  $(M, open)$**:=** `tableau(apply-part1-rule(`(U)`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  if $\neg open$ **then return** `tableau(apply-part2-rule(`(U)`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  end if;
else ▷ $\psi = \phi \,\mathrm{U}\, \phi'$ **and** $markedF = null$
  $markedF := \psi$**;**
  $(M, open)$**:=** `tableau(apply-part1-rule(`$(\mathrm{U})^+$`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  if $\neg open$ **then return** `tableau(apply-part2-rule(`$(\mathrm{U})^+$`,` ψ`,` S`),` P`,` *markedF*`)`**;**
  end if;

end if;
return $(M, open)$**;**

In the development of the algorithm the rule $\Diamond\varphi$ has been omitted because it does not add complexity to the explanation since it is equivalent to $(true\mathrm{U}\varphi)$

The method apply-β-rules (Algorithm 3) works in the same way as apply-α-rules with the uniqueness that β rules return two new sets of formulas instead of one (according to Tables 3 and 6). The algorithm uses a depth-first strategy, calling the method Tableau with the first set (`apply-part1-rule(`*rule*`,` ψ`,` S`)` and if this tableau closes, then calling the method Tableau with (`apply-part2-rule(`*rule*`,` ψ`,` S`)`). In both cases, the set of permanent formulas P and the current value of *markedF* are passed to the new call to the algorithm tableau.

We would like to remark that rules like $(\wedge)$ and $(\vee)$ are only applied to boolean combination of temporal formulas (i.e. formulas that involves at least one temporal operator). PPL-formulas are not

decomposed by tableau-rules, they are kept unchanged to be sent to the SAT-solver.

Finally, the apply_SAT (Algorithm 4) receives a set of pure propositional formulas and the set of permanent formulas. Since SAT-solvers do not work with temporal formulas the propositional formulas and next of literals $\{\alpha_1,...,\alpha_n,\circ l_1,...,\circ l_m\}$ are labeled as follow: $\{\alpha_1,...,\alpha_n,l'_1,...,l'_m\}$ (just as it has been done in the set P of permanent formulas). When passed to the SAT-solver, different interpretations can be obtained that satisfy the set of formulas. For each of such interpretation, rule $(\circ)$ is applied, generating a new node. Tableau is called again with it. This process is repeated until an open branch is found or until the SAT-solver returns no more interpretations. In the first case the model is returned and in the second case $(\emptyset, false)$.

Algorithm 4 Apply-SAT($input\_SAT$, S, P, $markedF$)

Let I be the interpretation returned by `SAT(`$input\_SAT$`)`;
if $I = \emptyset$ **then**
  return $(\emptyset, false)$;
else
  $next\_step$:= apply-rule$((\circ),-,S)$;
  $(M, open)$:= `tableau(`$next\_step$`,` P`,` $markedF$`)`;
  if $open$ **then return** $(M, true)$;
  else
    return `apply-SAT(`$input\_SAT \cup$ `neg(`M`),` S`,` P`,` $markedF$`)`;
  end if;
end if;
if there exists S'**, and ancestor of** S**, such that** $S \subseteq S'$ **and all eventualities in the path has been marked at least once then**
  return $(M,$ $true)$**;** ▷ **where** M **is the model constructed from the branch where** S' **and** S **are.**
end if;

When the algorithm calls to the SAT-solver, asking for a new interpretation, it enriches the input formula with the negation of the previous model (written as a formula $\mathtt{neg}(M)$). In practice, we use an API for incremental use of the SAT-solver, henceforth we really only pass $\mathtt{neg}(M)$ to the SAT-solver.

When Tableau ends, it returns an answer. It returns $(\emptyset, false)$ when all branches of the tableau have been closed and in consequence, $S \cup P$ is not satisfied. In other words, the temporal property is deduced from the system specification. The tableau returns $(M,true)$ when $S \cup P$ is satisfiable and M is a model of it. That is, M is a counterexample showing that the property is not deduced from the system specification.

7 Examples

This section shows how the algorithm works through two similar examples. To test the algorithm we need to specify a finite directed graph: that is, the conditions that are met in each state of the graph and the rules to transit through them. Finally we must ask the system for any condition expressed in temporal logic. In one example the program finds a model compatible with the system specifications and in another does not.

7.1 A counterexample showing that one formula is false

In the first example we represent through graphs the way children are fed in the western world: no comment is necessary since the graph is eloquent enough. However, if it is important to note that it is not necessary to indicate the reasons why it is passed from one node to another, it is sufficient to indicate that it is possible. For example, a child goes from being happy to crying for being hungry, but the reasons do not detail in the system specification.

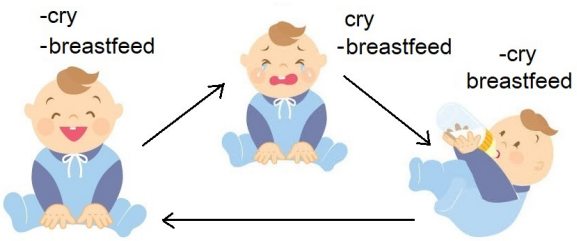


Figure 9: How a child feeds

We can specify the system using two variables: cry: "c" and breastfeed "b". Note that these formulas corresponds to a set of TRANS-formulas.

$$\begin{aligned} \mathrm{TS}_1 = \{\ & \Box((\neg c \wedge \neg b) \rightarrow (\circ(c \wedge \neg b))), \\ & \Box((c \wedge \neg b) \rightarrow (\circ(\neg c \wedge b))), \\ & \Box((\neg c \wedge b) \rightarrow (\circ(\neg c \wedge \neg b))) \ \} \end{aligned}$$

To make the system above more realistic we could add the INVAR-formula that says it is impossible to cry and breastfeed at the same time $\Box(\neg(c \wedge b))$.

The above four TRANS/INVAR formulas are used as before to construct the set P_1 of permanent formulas: The $\Box$ operator is omitted, and $\circ a$, $\circ b$, $\circ c$, and $\circ d$ are renamed by a', b', c', and d' respectively. Finally, we can check if our system is correct by asking the system for premises that we believe it must comply with. For example, if a child cries, does he always end up eating? Or is it possible to eat without crying? A more basic question to the system would be can the child be left without eating? The temporal formula expressing this question is as follows.

$$\Box\neg b \tag{1}$$

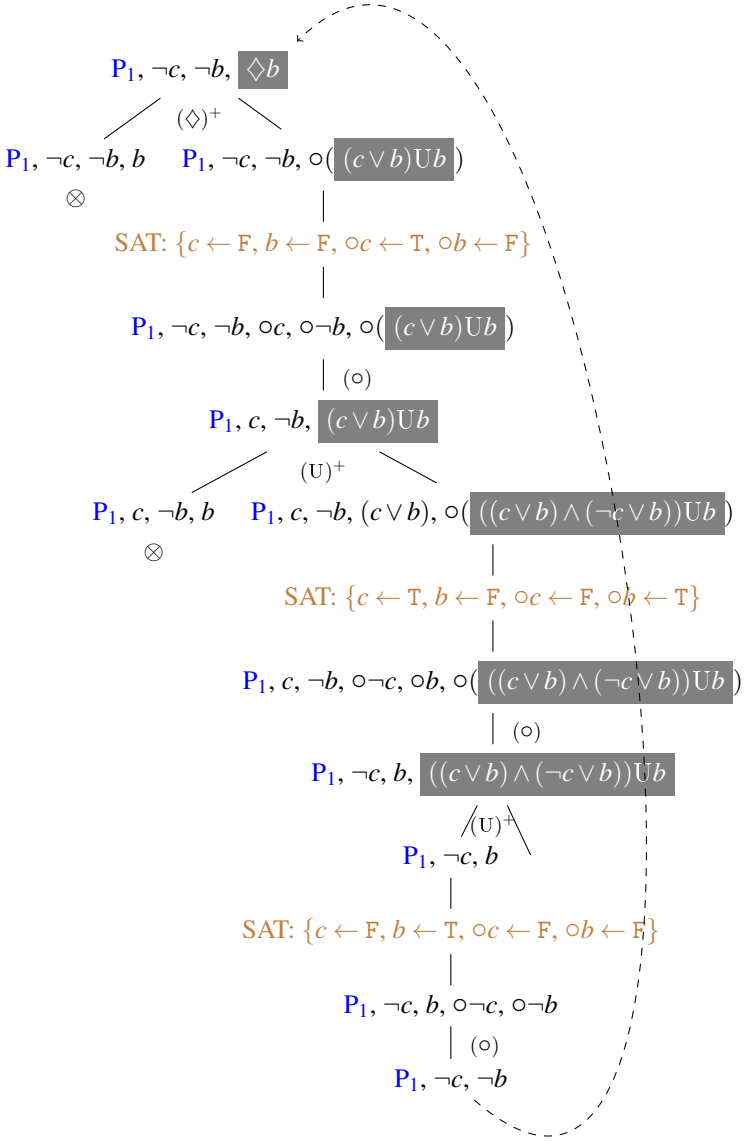


Figure 10: Opened tableau

Figure 10 is the tableau constructed by our algorithm. For readability, the marked eventualities are in gray boxes. The root node contains the set of permanent formulas in addition with the set $S = \{\neg c, \neg b, \Diamond b\}$. The formula $\Diamond b$ is the negation of formula (1). The algorithm stops when the first open branch is found. Then it returns the PLTL-structure that represents the cycle starting at initial vertex. Hence, this PLTL-structure is the counterexample showing that formula (1) is false. It is worth remarking that thanks to the negation of the context introduced by the $(\Diamond)^+$-rule, the left branch is quickly closed.

Remember the usual ASCII syntax for temporal operators are X for $\circ$, G for $\Box$, F for $\Diamond$, R for R and U for U.

To test the previous example through the developed algorithm we would need a file (example-1.txt) in which we would keep the description of the system and the rules under study with the following format:

```
G((-c&-b)->(X(c&-b)))
G((c&-b)->(X(-c&b)))
G((-c&b)->(X(-c&-b)))
-c
-b
Fb
```

Later we would execute the program tableau.py with the result:

```
python tableau.py examples/example-1.txt
Set of formulas
G((-c&-b)->(X(c&-b)))
G((c&-b)->(X(-c&b)))
G((-c&b)->(X(-c&-b)))
-c
-b
Fb
Found Model:

1 -c, -b
2 -b, c
3 -c, b
Cycle at 1

Open Tableau
Resolved 9 Nodes
Elapsed Time was 0.00178599357605 seconds.
```

As you can see the program gives us a counterexample indicating that the temporary formula added is false.

7.2 A closed tableau showing that one formula is True

In this second example we are going to slightly modify the way we feed a baby: if the child cries we can calm him down with some of the typical parents' tricks: games, gestures, etc. The system description would look like in Figure 12.

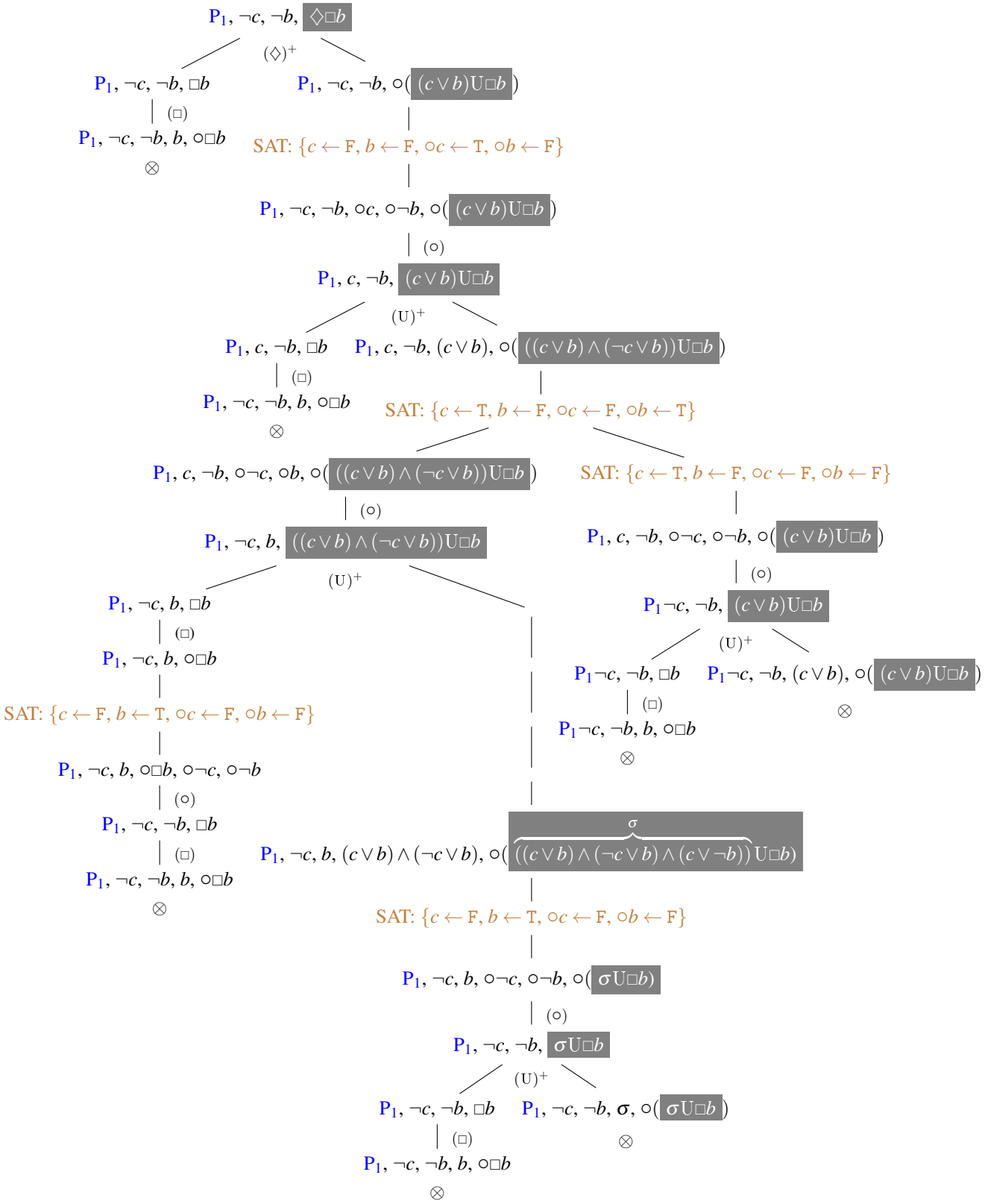


Figure 11: Closed tableau

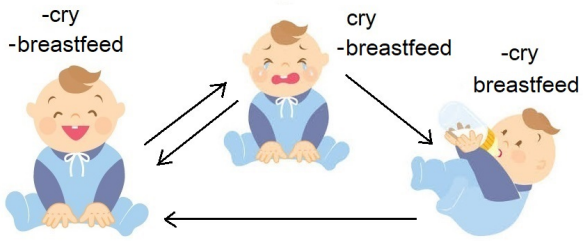


Figure 12: Another way to breastfeed babies

And the set of TRANS-formulas:

$$\begin{aligned}\mathrm{TS}_1 = \{\ & \Box((\neg c \wedge \neg b) \rightarrow (\circ(c \wedge \neg b))),\\ & \Box((c \wedge \neg b) \rightarrow (\circ(\neg c \wedge b) \vee \circ(\neg c \wedge \neg b))),\\ & \Box((\neg c \wedge b) \rightarrow (\circ(\neg c \wedge \neg b)))\ \}\end{aligned}$$

We are going to make the next question.

$$\Box\Diamond\neg b \tag{2}$$

Figure 11 is the tableau constructed by our algorithm. The root node contains the set of permanent formulas in addition with the set $S = \{\neg c, \neg b, \Diamond\Box b\}$. The formula $\Diamond\Box b$ is the negation of formula (2). All its branches are closed as expected. Hence, the constructed tableau is a formal proof of the statement, that is (2) is true. Again, the $(\mathrm{U})^+$ rule allows the algorithm to close all branches.

7.3 More examples

This subsection shows two examples about checking whether a finite directed graph (a representation of a state machine or transition system) contains a cycle starting at some particular vertex. For example, in Figure 13, the left graph G_1 contains a cycle where the node a is reachable from a, while the right one, G_2, does not contain any cycle.

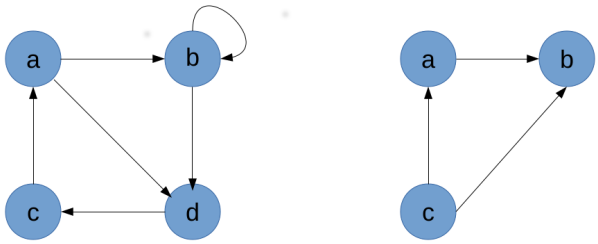


Figure 13: Two finite directed graphs G_1 and G_2.

There exist many ways of encoding directed graphs onto transition systems (TS) depending on the nature of the problem to be solved. However, all of them simulate the adjacency matrix for a given graph G. Essentially, states of TS are the vertices in G and the transitions in TS correspond to the edges in G.

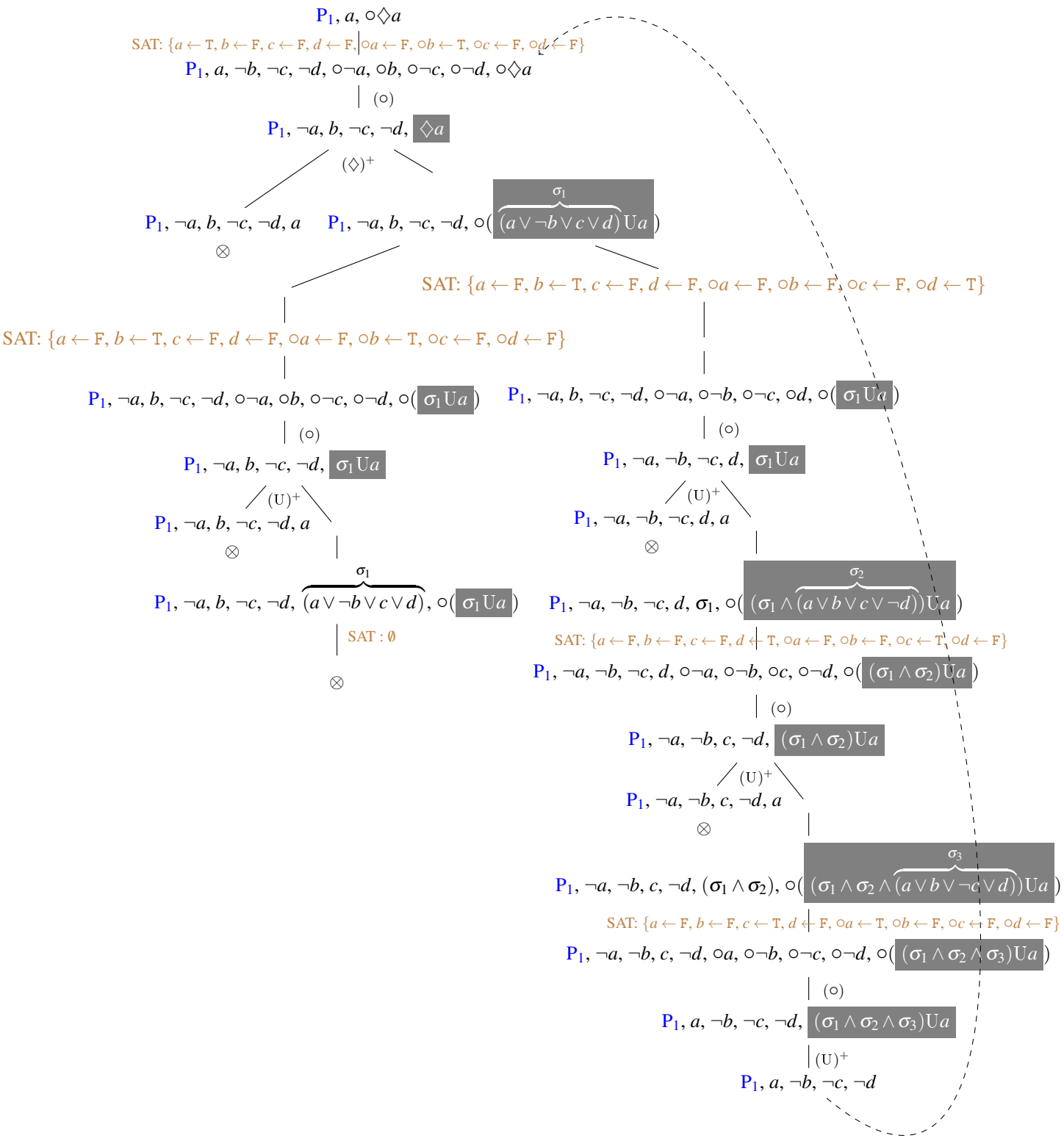


Figure 14: Opened Tableau for G_1

Graph G_1: In the case of G_1 the adjacency matrix is as follows. Note that these formulas correspond to a set of TRANS-formulas

$$\begin{aligned}\mathrm{TS}_1 = \{\ & \Box(a \rightarrow ((\circ\neg a \wedge \circ b \wedge \circ\neg c \wedge \circ\neg d) \vee (\circ\neg a \wedge \circ\neg b \wedge \circ\neg c \wedge \circ d))),\\ & \Box(b \rightarrow ((\circ\neg a \wedge \circ b \wedge \circ\neg c \wedge \circ\neg d) \vee (\circ\neg a \wedge \circ\neg b \wedge \circ\neg c \wedge \circ d))),\\ & \Box(c \rightarrow (\circ a \wedge \circ\neg b \wedge \circ\neg c \wedge \circ\neg d))\\ & \Box(d \rightarrow (\circ\neg a \wedge \circ\neg b \wedge \circ c \wedge \circ\neg d))\ \}\end{aligned}$$

To make the transition system above more realistic (in terms of directed graphs behavior), we add the INVAR-formula that forces to stay on a single vertex each time:

$$\Box[\ (a \rightarrow (\neg b \wedge \neg c \wedge \neg d) \wedge (b \rightarrow (\neg a \wedge \neg c \wedge \neg d) \wedge (c \rightarrow (\neg a \wedge \neg b \wedge \neg d) \wedge (d \rightarrow (\neg a \wedge \neg b \wedge \neg c)\ ]$$

The above five TRANS/INVAR formulas are used as before to construct the set P_1 of permanent formulas: the $\Box$ operator is omitted, and $\circ a$, $\circ b$, $\circ c$, and $\circ d$ are renamed by a', b', c', and d' respectively. Finally, to say that the initial vertex is a (in our example), we initialize the variable a as $true$.

We want to check if "Vertex a is unreachable in the future". That formula corresponds to the temporal formula

$$\circ\Box\neg a \tag{3}$$

Figure 14 is the tableau constructed by our algorithm. The root node contains the set of permanent formulas P_1 in addition with the set $S = \{a, \circ\Diamond a\}$. The formula $\circ\Diamond a$ is the negation of formula (3). The algorithm stops when the first open branch (in Figure 14, the right most one) is found. Then, it returns the PLTL-structure that represents the cycle starting at vertex a. Hence, this PLTL-structure is the counterexample showing that formula (3) is false. It is worth remarking that thanks to the negation of the context introduced by the $(\Diamond)^+$-rule, the left branch is quickly closed.

As in a previous example we will test the validity of the developed application with this new problem. First we need a file (example-3.txt) in which we keep the description of the system and the rules under study with the following format:

```
G(a->((X-a&Xb&X-c&X-d)|(X-a&X-b&X-c&Xd)))
G(b->((X-a&Xb&X-c&X-d)|(X-a&X-b&X-c&Xd)))
G(c->(Xa&X-b&X-c&X-d))
G(d->(X-a&X-b&Xc&X-d))
G((a->(-b&-c&-d))&(b->(-a&-c&-d))&(c->(-a&-b&-d))&(d->(-a&-b&-c)))
a
XFa
```

Later we execute the program tableau.py with the result:

```
python tableau.py examples/example-3.txt

Set of formulas
G(a->((X-a&Xb&X-c&X-d)|(X-a&X-b&X-c&Xd)))
G(b->((X-a&Xb&X-c&X-d)|(X-a&X-b&X-c&Xd)))
G(c->(Xa&X-b&X-c&X-d))
G(d->(X-a&X-b&Xc&X-d))
```

```
G((a->(-b&-c&-d))&(b->(-a&-c&-d))&(c->(-a&-b&-d))&(d->(-a&-b&-c)))
a
XFa

Found Model:
1 a, -b, -c, -d
2 -a, b, -c, -d
3 d, -c, -b, -a
4 -d, c, -b, -a
Cycle at 1

Open Tableau
Resolved 15 Nodes
Elapsed Time was 0.00477814674377 seconds.
```

As you can see the program gives us a counterexample indicating that the temporary formula analyzed is false. Graphically, the model is shown in the next figure.

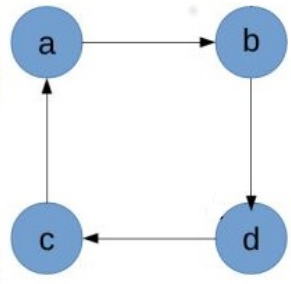


Figure 15: A counterexample.

Graph G_2: Now we study the directed graph without cycles G_2. In general, transition systems do not have terminal states, but the graph G_2 does. To ensure that the codification of G_2 has no terminal states, we need to add the temporal formula $\Box(a \vee b \vee c \vee (\circ\neg a \wedge \circ\neg b \wedge \circ\neg c))$. Thus, from state b, we can only move to a sort of *sink* state where we stay forever.

Hence, the set of TRANS-formulas TS_2 is as follows.

$$\begin{aligned}\mathrm{TS}_2 = \{\ & \Box(a \to (\circ\neg a \wedge \circ b \wedge \circ\neg c)),\ \Box(b \to (\circ\neg a \wedge \circ\neg b \wedge \circ\neg c)),\\ & \Box(c \to ((\circ a \wedge \circ\neg b \wedge \circ\neg c) \vee (\circ\neg a \wedge \circ b \wedge \circ\neg c))),\\ & \Box((\neg a \wedge \neg b \wedge \neg c) \to (\circ\neg a \wedge \circ\neg b \wedge \circ\neg c))\ \}\end{aligned}$$

The INVAR-formula that forces to stay on a single vertex each time is:

$$\Box[\ (a \to (\neg b \wedge \neg c)\ \wedge\ (b \to (\neg a \wedge \neg c)\ \wedge\ (c \to (\neg a \wedge \neg b)\ ]$$

So, the set of permanent formulas P_2 is TS_2 plus the INVAR-formula where the $\Box$-operator is omitted and $\circ a, \circ b$, and $\circ c$ are renamed by a', b', and c' respectively. Figure 16 is the tableau constructed by our algorithm. The root node is the set of permanent formulas P_2 together with $S = \{a, \circ\Diamond a\}$. All its branches

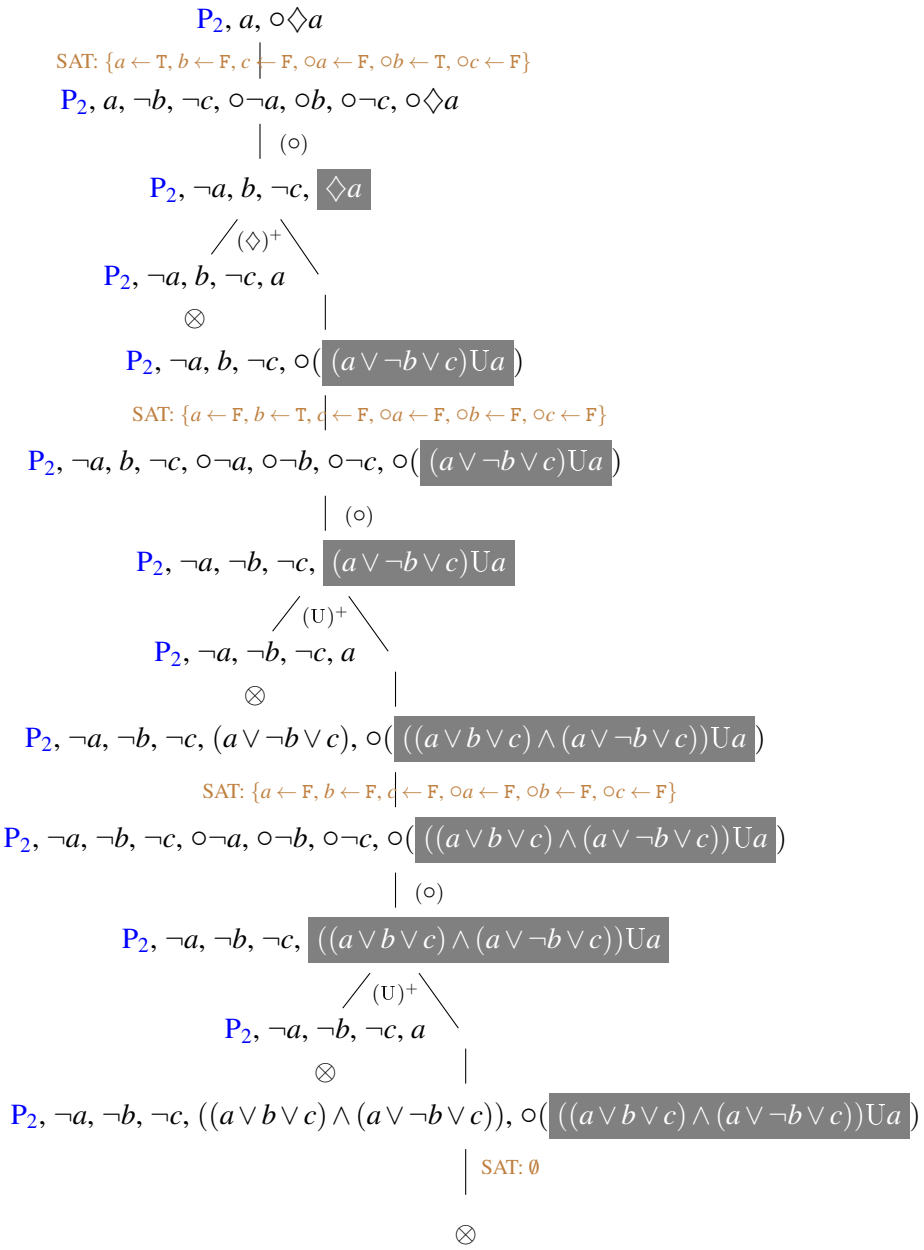


Figure 16: Closed Tableau for G_2

are closed as expected. Hence, the constructed tableau is a formal proof of the statement "Vertex *a* is unreachable in the future".

In this last example we will test the algorithm forcing the application to show us all the nodes that the tableau is generating. First we write in a file (example-4.txt) the description of the system and the rules under study. Later we execute the program tableau.py which the option verbose.

```
python tableau.py examples/example-4.txt -v

Set of formulas

G(a->(X-a&Xb&X-c))
G(b->(X-a&X-b&X-c))
G(c->(Xa&X-b&X-c)|(X-a&Xb&X-c))
G((-a&-b&-c)->(X-a&X-b&X-c))
G((a->(-b&-c))&(b->(-a&-c))&(c->(-a&-b)))
a
XFa

1 ************************** from 0 .
a
XFa
--------------
Applying NEXT Rule SAT 1
2 ************************** from 1 .
-a
b -c
Fa
--------------
Applying rule:  F+
3 ************************** from 2 .
-a
b
-c
a
--------------
X no Satisfiable
4 ************************** from 2 .
X((c|(-b|a))Ua)
-a
b
-c
--------------
Applying NEXT Rule SAT 1
5 ************************** from 4 .
-a
```

```
-b
-c
((c|(-b|a))Ua)
--------------
Applying rule:  U+
6 *************************** from 5 .
-a
-b
-c
a
--------------
X no Satisfiable
7 *************************** from 5 .
X(((c|(b|a))&(c|(-b|a)))Ua)
(c|(-b|a))
-a
-b
-c
--------------
Applying NEXT Rule SAT 1
8 *************************** from 7 .
-a
-b
-c
(((c|(b|a))&(c|(-b|a)))Ua)
--------------
Applying rule:  U+
9 *************************** from 8 .
-a
-b
-c
a
--------------
X no Satisfiable
10 *************************** from 8 .
X(((c|(b|a))&((c|(b|a))&(c|(-b|a))))Ua)
((c|(b|a))&(c|(-b|a)))
-a
-b
-c
-------------- X in SAT 1

Closed Tableau
Resolved 10 Nodes
Elapsed Time was 0.00246214866638 seconds.
```

As can be seen in the results on the screen, in verbose mode the application shows in detail what is doing node by node. For each node it initially shows its number and that of its parent, then the formulas under study and finally the rule that must be applied or a message saying that the set of formulas is unsatisfiable. In this case as we expected, we will not obtain any model.

8 Implementation

8.1 Modules

For the development of the program we have decided to divide the problem into several independent modules. The Advantages of Modular Programming are known: simplify the design, decrease the complexity of the algorithms, decrease the total size of the program, it saves on programming time because it promotes the reusability of the code, promotes teamwork, facilitates debugging and testing, It facilitates maintenance and it allows the structuring of specific libraries. The modules are:

- Parser: the module takes as input the description of the grammar that allows us to describe the system under study and returns that system in a more manageable format: trees. As we saw earlier formula like $\square(\circ a \wedge \circ b)$ is written in text format like G(Xa&Xb). This formula must be transformed into a tree like the one in the figure that would translate into a list in python. The list will be [G[&,[X,a],[X,b]]].

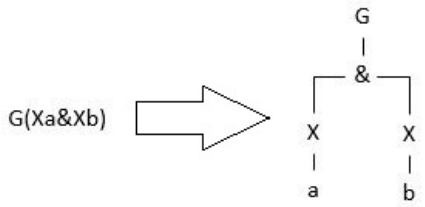


Figure 17: Example of conversion.

- Tree: The module contains dozens of utilities for handling trees, such as: isImplication, isAnd, simplifyFormula, isUnsatisfiable, etc. The objective of the module is to facilitate the numerous operations that higher level modules must perform by simplifying the code. All the advantages that have been listed in describing the modular programming are evident in this module: simplify the design, decrease the complexity of the algorithms, etc. The most important utility of this module is nnf that allows us to pass any tree to nnf format with the advantages described above: code reduction and facilitate comparisons among others advantages.
- CNF: The objective of this module is to convert the formulas to the CNF format, wich expresses as conjunctions of clauses with an AND or OR. Each clause connected by a conjunction, or AND, must be either a literal or contain a disjunction, or OR operator. CNF is the only format accepted by the different SATs that can be used in the application.
  To convert any propositional formula into its equivalent CNF it is necessary to perform operations such as eliminate biconditionals and implications, move negations inwards, Distribute $\wedge$ over $\vee$, among others operations. Below is an example of conversion to the CNF format.

$$(p \leftrightarrow q) \rightarrow r \quad \equiv \quad (p \vee q \vee r) \wedge (\neg q \vee \neg p \vee r)$$

Figure 18: Example of conversion.

- SAT: This module implements several SATs that are used by the tableau module to simplify the resolution processes. We use tableaux method for temporal logics to check whether a system satisfies a property. Modern SAT-Solvers are used for the non-temporal reasoning part of the tableau construction. We use in this module PySAT which is Python library providing a simple interface to a number of state-of-art Boolean satisfiability (SAT). The purpose of PySAT is to enable researchers working on SAT and its applications and generalizations to easily prototype with SAT oracles in Python while exploiting incrementally the power of the original low-level implementations of modern SAT solvers. Most of SAT solvers use DIMACS CNF format which is widely accepted as the standard format for boolean formulas in CNF. Since the format provided by the CNF module is not DIMACS we must transform it to the format required by pysat library. Logically, at the end of the process we must return the results to the format used by the program. Here there may be an improvement area.

- Tableau: the core of the algorithm. The main method of this module is tableau. This function is strongly recursive, as are most of the methods of this application: nnf, cnf and among others. This can have a high impact on memory if the data describing the system is not well managed. A bad approach to solving any problem can dramatically decrease program performance. In addition, for larger systems, model checkers need too many resources, and an explosion of states may occur.

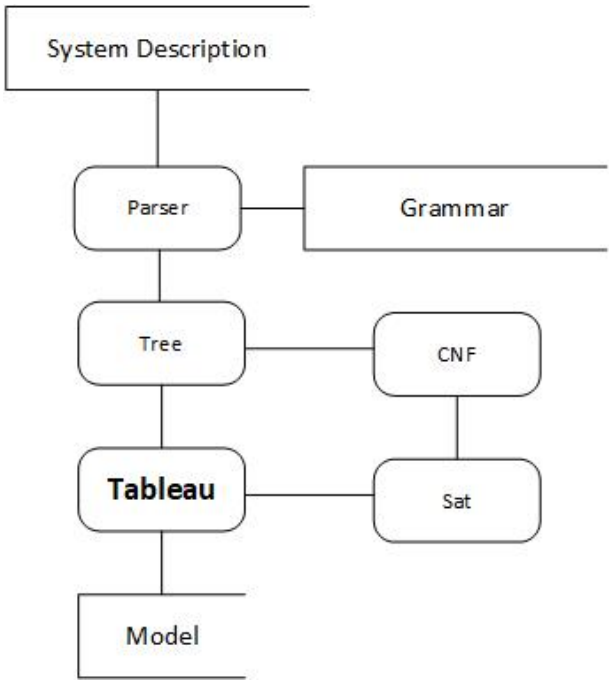


Figure 19: Application scheme.

As described in the figure, the application takes as input a description of the system under study and returns a model consistent with the description or indicates that there is no such model. Next we will describe the modules that require some additional explanation.

9 Description of the stages

9.1 Parser

To validate the formulas that the application analyzes we use a parser developed in python with parsimonious name. This parser aims to be the fastest arbitrary-lookahead parser written in pure PythonâĂŤand the most usable. It's based on parsing expression grammars (PEGs), which means you feed it a simplified sort of EBNF notation. In the figure you can read the full description of the PLTL grammar used in the development of this project.

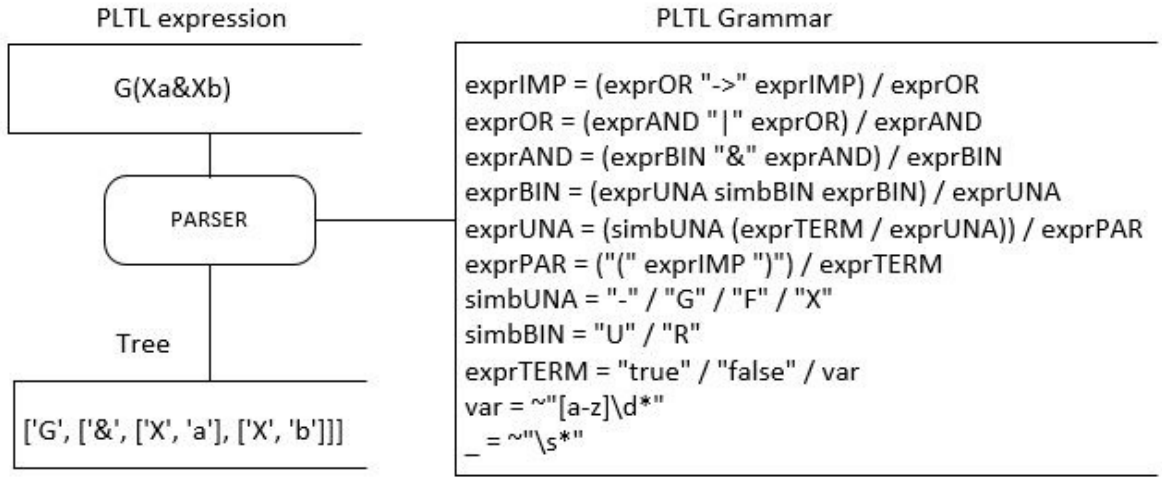


Figure 20: Scheme of operation of the parser's module.

The purpose of the module is to validate the formulas that the user writes in the system description. In addition, if they are correct, they will be converted into a list that expresses their tree format. The grammar takes into account the precedence of the operators; The operators defined in the first lines have a lower precedence than those described at the end. In this way "a AND b UNTIL -c OR d" would be equivalent to" (a AND (b UNTIL (-c))) OR d".
The module is developed in object oriented programming and contains the Formula class and the isValid and text methods among others. The next example explains how to use the Formula class to manipulate a temporal expresion: We create a formula of name f with the expression: (a -> (bUc)). Later we print it first denied and finally it will be denied and passed to the normal negative form.

```
>from parser import Formula
>f=Formula("(a->(bUc))")
>print f.no()
-(a->(bUc))
>print f.no().nnf()
(a&(-bR-c))
```

9.2 CNF-SAT

In this section we will describe two modules simultaneously because they are used in this way in the program.

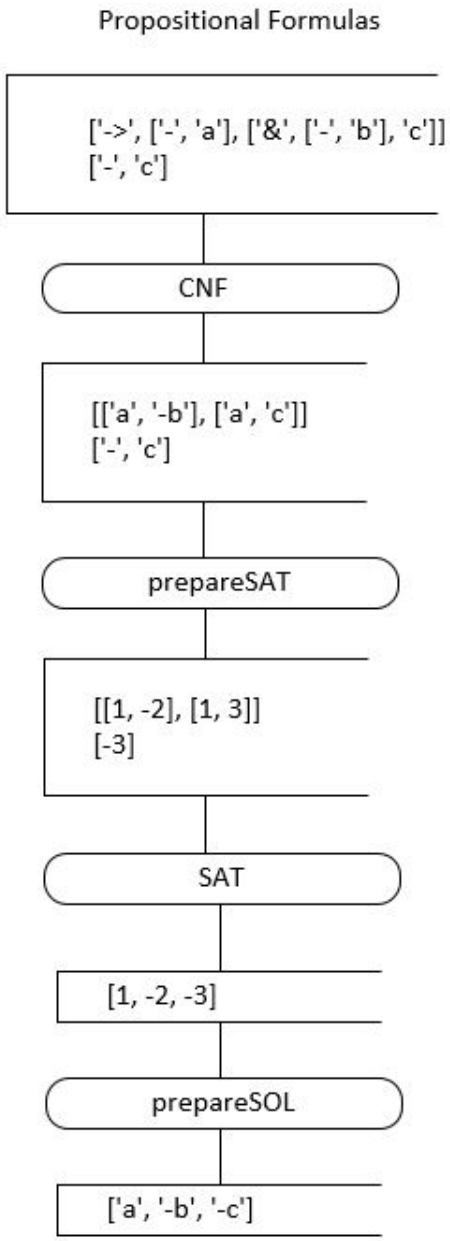


Figure 21: Scheme of operation of the SAT and CNF modules.

As we see, any expression passed to the sat module must be expressed in CNF format. On the other hand, this transformation is not enough; each atom must be represented by an integer, which requires

having a dictionary that relates the names and their numbers. Finally, the solution that you provide with the SAT module must be rewritten with the names of the variables, so that the process continues.
As stated earlier we use the Python PySAT library. PySAT integrates a number of widely SAT solvers. All the provided solvers are the original low-level implementations installed along with PySAT.

Currently, the following SAT solvers are supported (at this point, for Minisat-based solvers only core versions are integrated):

- Glucose (3.0)
- Glucose (4.1)
- Lingeling (bbc-9230380-160707)
- MapleLCMDistChronoBT (SAT competition 2018 version)
- MapleCM (SAT competition 2018 version)
- Maplesat (MapleCOMSPSLRB)
- Minicard (1.2)
- Minisat (2.2 release)
- Minisat (GitHub version)

Boolean variables in PySAT are represented as natural identifiers. A clause is a list of literals, e.g. [-3, -2] may be a clause $\neg x_3 \vee \neg x_2$.

The following is a trivial example of PySAT usage:

```
> from pysat.solvers import Glucose3
> g = Glucose3()
> g.add_clause([-1, 2])
> g.add_clause([-2, 3])
> print g.solve()
> print g.get_model()
...
True
[-1, -2, -3]
```

As you can see, changing SAT solver only requires changing one line of code, assuming that the corresponding library has already been imported. This allows us to test the performance of each SAT in different circumstances and perhaps dynamically choose which one we want to use.

9.3 Tableau

This module contains the core of the model checking algorithm described above. The model description is obtained from a file and after the process the program returns a model or indicates that there is no solution.

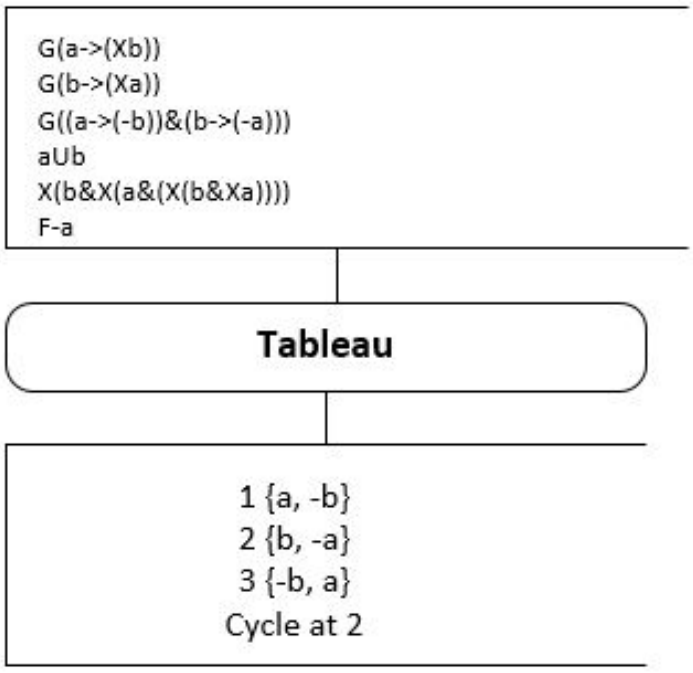


Figure 22: Scheme of operation of the Tableau module.

The following example shows the execution of a program with a group of input formulas.

```
$ python tableau.py examples/example.txt
Set of formulas

Fw
((a|c)&(b|c))U((d|e)&(f|e))
((-a|-b)&(-c))R((-d|-f)&(-e))

Found Model:
1 -a, c, -b, w, -d, -f, -e
2 -w, -a, c, -b, -d, -f, -e
Cycle at 2

Open Tableau
Resolved 22 Nodes
Elapsed Time was 0.00774097442627 seconds.
```

10 Use of the program

The program can be downloaded freely from the github platform at the link: https://github.com/acebuche/mal

10.1 Connectives

The description of the system, as well as the rules to be checked, must be included in a text file using the following directives.

- propositional symbols = ‘[a-z][0-9]*‘
- and = ‘&‘
- or = ‘|‘
- implication = ‘->‘
- negation = ‘-‘
- next = ‘X‘
- always = ‘G‘
- eventually = ‘F‘
- until = ‘U‘
- release = ‘R‘

Parentheses can be used to force precedence as well as using spaces to make formulas more readable.

10.2 Usage

Below is the description of the use of the program as shown by its help.

```
Usage:  tableau.py <file> [-s | -stop] [-v | -verbose]
tableau.py -version
Options:
-s -stop Step by step execution, one step per tableau.
-v -verbose Show tableau's information.
-version Show version.
-h -help Show this.
```

Examples: to execute the model checker step by step with the description of the system stored in the file example1.txt of the folder examples you would have to write:

```
python tableau.py -s examples/example1.txt
```

To execute the model checker in verbose mode with the description of the system stored in the file example2.txt of the folder examples you would have to write:

```
python tableau.py -v examples/example2.txt
```

More options can be seen running:

```
python tableau.py -h
```

10.3 Dependecies

The following libraries are necessary for the proper work of the program.

- Python2.7
- parsimonious
- numpy
- python-sat
- docopt==0.6.2

11 Conclusion

In this master thesis we have made a brief summary of the fundamentals of propositional logic. We have also reviewed the behavior of temporal logics, applying this type of logics in software verification, specifically in symbolic model checking. We provide detailed foundations, algorithms and examples on a new idea for using SAT-solvers in combination with a one-pass tableau technique for implementing a certifying model-checker. The use of SAT-solvers makes the process of model-checking more efficient. Future advances in SAT solvers will directly improve the algorithm implementation and performance. The implementation approaches the interaction between the algorithm and the SAT-solvers, putting background for future implementation work.

We have finished the implementation of a first prototype of the algorithm and experiment with it. This implementation has been made using Python language. This makes the implementation even more interesting as we have used one of the most expanded programming languages in the market right now, with powerful tools and libraries. It also makes it possible for this tool to be used with all the facilities that python offers, such as an integration features, extensive support community or ease of maintenance.

In the future we can improve algorithm performance using multiple approaches. We can use heuristics to decide in each branch of the tableau which rule is most promising to reach a quick solution reducing runtime and expanded branches. It is possible to check if a previously expanded branch reappears on the tableau; in part it has already been tested in this algorithm implementation, with very good results. In multiprocessor equipment, different branches could be executed in different processors to take advantage of this hardware capacity. In the same line as the previous case when it is necessary to use SAT we can simultaneously launch several SATs and take the first one that ends. In summary, there are many lines of improvement that would make the algorithm more efficient and that can be explored in the future.

References

[1] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer-Verlag, London, 2012.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, pages 52–71, 1981.

[5] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 463–478. Springer Verlag, 2013.

[7] Georgios E. Fainekos, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for mobile robots. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation, ICRA 2005, April 18-22, 2005, Barcelona, Spain*, pages 2020–2025, 2005.

[8] Joxe Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro, and Fernando Orejas. Dual systems of tableaux and sequents for PLTL. *J. Log. Algebr. Program.*, 78(8):701–722, 2009.

[9] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[10] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[11] Alain Mebsout and Cesare Tinelli. Proof certificates for SMT-based model checkers for infinite-state systems. In R. Piskac and M. Talupur, editors, *Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design (Mountain View, CA)*, pages 117–124. IEEE, 2016.

[12] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 2–13, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[13] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[14] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[15] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.