Verification of temporal properties of infinite state systems

Author: Cristina Luengo Agulló
Director: Albert Rubio

Degree in Computer Science
Computing specialization

Universitat Politècnica de Catalunya
Facultat d'informàtica de Barcelona

29 June 2015

Abstract

It is no secret that computer software programs, computer hardware designs, and computer systems in general exhibit errors. Testing and simulation methods can identify many significant problems, but for systems that have safety or economically critical requirements, exhaustive verification is indispensable. Such exhaustive analysis can be performed with the use of formal verification methods.

One approach to formal verification is model checking, which is a fully automated process based on the construction of abstract models to represent systems. These models are then checked against desired properties defining a specification, usually expressed in some temporal logic, such as linear temporal logic (LTL). Temporal properties can describe the ordering of events in time without introducing time explicitly, thereby being useful when verifying the possible executions of a system.

This project aims to implement model checking algorithms that determine whether an LTL formula expressing a desired property is satisfied in a computing system.

Resumen

No es ningún secreto que tanto los sistemas software como hardware generalmente presentan errores. Los métodos de testeo y simulación pueden identificar muchos problemas importantes, pero para sistemas que tienen requerimientos de seguridad o que son económicamente críticos, es indispensable llevar a cabo una verificación exhaustiva. Tal análisis se puede realizar utilizando métodos de verificación formal.

Un enfoque de la verificación formal es la verificación de modelos, que es un proceso totalmente automático basado en la construcción de modelos abstractos para representar sistemas. Posteriormente, sobre estos modelos se comprueban propiedades deseadas del sistema, normalmente expresadas en alguna lógica temporal, como por ejemplo lógica linear temporal. Las propiedades expresadas con fórmulas de lógica linear temporal pueden describir el orden de los eventos en el tiempo sin describir el tiempo explícitamente. Por eso mismo, son útiles a la hora de verificar las posibles ejecuciones de un sistema.

Este proyecto pretende implementar algoritmos de verificación de modelos que determinen si una fórmula de lógica linear temporal que exprese una propiedad de un cierto sistema es satisfecha por éste.

Resum

No és cap secret que tant els sistemes software com hardware generalment presenten errors. Els mètodes de testeig i simulació poden identificar molts problemes importants, però per sistemes que tenen requeriments de seguretat o que són econòmicament crítics, és indispensable realitzar una verificació exhaustiva. Aquest anàlisi es pot fer utilitzant mètodes de verificació formal.

Un enfocament de la verificació formal és la verificació de models, que és un procés totalment automàtic basat en la construcció de models abstractes per representar sistemes. Posteriorment, sobre aquests models es comproven propietats desitjades del sistema, normalment expressades en alguna lògica temporal, com per exemple la lògica lineal temporal. Les propietats expressades amb fórmules de lògica lineal temporal poden descriure l'ordre dels esdeveniments en el temps sense descriure el temps explícitament. Per això mateix, són útils a l'hora de verificar les possibles execucions d'un sistema.

Aquest projecte pretén implementar algorismes de verificació de models que determinin si una fórmula de lògica lineal temporal que expressi una propietat d'un cert sistema és satisfeta per aquest.

Acknowledgements

First and foremost, I would like to thank my project director, Albert Rubio. Working alongside with him has not only been an enriching experience, but also a fun time. His enthusiasm for research and for the work that he does is truly inspiring, and this translated to motivation for me throughout the project. I can say that I have learned many new things while doing this project, and I always had full support from him when struggling with something.

I would also like to thank Alicia Villanueva and Laura Titolo for providing us with examples to test.

Finally, I would specially like to thank Javi as well, who always supported me and helped me with the design of this document.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the ubiquity and increasing complexity of hardware and software systems, the likelihood of errors is high. Such errors can be sometimes hard to find, and its consequences can be catastrophic in terms of time, money or even human life, depending on the systems. Therefore, it is extremely important to find all possible errors and doing it as early in the design process as possible, since every late-found errors may delay the whole production phase.

Different methods have been extensively used to help developers find errors in the systems they design. Two of such methods are testing and simulation, which are useful in early phases of the debugging process. Nevertheless, their effectiveness dramatically decreases when the system hides only a small number of bugs. Moreover, these methods are not able to confirm that the design is fully correct, since they explore only some of the possible behaviors of the system. Due to this, there is a growing demand for methods that can increase confidence in correct system design and construction. One of these methods is formal verification.

Formal verification techniques create a mathematical model of a system, using a language to specify desired properties of the system in a concise and unambiguous way, and applying a method of proof to verify that the specified properties are satisfied by the model. The main advantage shared by various formal verification techniques is the ability to ensure that the system is correct (i.e. the system satisfies a certain specification), given that they explore all its possible behaviors.

In a concurrent system, different processes can be involved in the execution of a task, which means that the order in which events occur in the execution of such systems is mostly unpredictable and hence, correct behavior is harder to track. For this reason, formal verification techniques have been extensively applied in the design of these kind of systems.

Concurrent systems are usually represented using finite-state machines which enumerate all the state space. Apart from the fact that only finite-state systems can be considered, this has an inherent disadvantage, commonly known as state-space explosion. In order to cope with that, there exist different methods that either perform on-the-fly operations to save up memory and computation time, or follow a different approach when modeling the system. The latter makes it possible to deal with bigger and more complex models, and it is the approach that we are going to follow in this project.

Therefore, we aim to implement formal verification methods for infinite-state systems in order to allow automatic and rigorous verification of complex systems with an infinite (or huge) state space, providing new robust and reliable tools for software developers.

1.2 Context

The context in which the project is defined covers different areas of interest. Our main focus will be on automated formal verification and automata theory. The following sections describe the concepts covered in each one.

1.2.1 Automated formal verification

Formal verification [1] is the act of proving or disproving the correctness of a system with respect to a certain specification or property, using formal methods of mathematics. The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system.

One approach to formal verification is model checking [2], which consists of a systematically exhaustive exploration of the abstract model. It aims to solve the following problem: Given a model of a system and a specification, exhaustively and automatically check whether the specification can be applied. The specification is usually given as a formula in a temporal logic, such as *linear temporal logic* (LTL) [3], which relates elements of the system model in time.

Model checking was invented more than 30 years ago by Emerson & Clarke [4] and Queille & Sifakis [5] independently. The first algorithms used are known as *explicit model checking*, which represent all the possible states of a system and suffer from the state-space explosion problem. Roughly speaking, the problem is that the number of global states of a system is very large in contrast to the length of the system's high level definition (e.g. its source code).

Later on, other methods arose to try to cope with that problem, and *symbolic model checking* [6] became popular because of its non-explicit representation of the states of a system. This allowed to verify bigger models where explicit model checking would be unfeasible.
Following that line, other approaches to try to avoid the infamous state-space explosion problem started being used and studied. One of them was *bounded model checking* [7], where properties are proven for computation paths of bounded length only.

Therefore, many years of research led to different methods and improvements, and nowadays formal verification is still an active field of study in the research community.

1.2.2 Automata theory

Automata theory [8] is the study of abstract machines and automata, as well as the computational problems that can be solved using them.
Automatons are abstract models of machines that perform computations on an input by moving through a series of states. At each state of the computation, a transition function determines the

next state. Some of the states are accepting, which means that the input is accepted if there exists a path in the automaton where one of those states is reached after reading all the input. Hence, automatons can define languages by determining the words they accept.

Example 1. Figure 1.1 shows an automaton that accepts the words composed of any combination of letters from the alphabet $\{a, b\}$ that end in letter 'a':

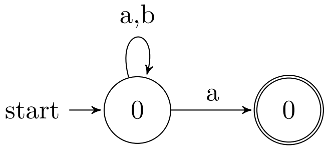


Figure 1.1: Automaton example

Automatons can be used to model both a computing system and a property expressed in LTL. Therefore, understanding concepts such as the language defined by an automaton and automata intersection will be essential to carry out the verification task.

1.3 State of the art

Formal verification techniques such as model checking are aimed at validation of finite-state as well as infinite-state systems. The motivation for verification of finite-state systems lies in the fact that computing systems have finite memory, thereby having a finite number of possible states. However, the number of states needed to model a finite-state system can be huge, so this is a good reason to deal with verification techniques for infinite-state systems. Since they do not rely on a given set of states, these techniques can be used for the verification of general or parameterized systems. Moreover, infinite-state abstractions of large finite-state systems can be smaller and the verification of such abstractions can be easier.

The following sections provide a brief overview of the state of the art on verification methods for finite-state systems, and enumerate known techniques for infinite-state systems.

1.3.1 LTL model checking for finite-state systems

One of the most prominent instances of the model checking problem is the explicit model checking problem, which aims to decide whether a given finite-state system satisfies specifications expressed by an LTL formula. As already stated, a linear temporal logic formula can express the order of events in time. So as an example, it could be used to express properties like mutual exclusion for concurrent systems, among many others.

Example 1. Consider a concurrent system with two processes p_1 and p_2, which can access a shared resource. The access to the resource is indicated by $p_x.hasResource$, where $x \in \{1, 2\}$. The LTL formula $\Box\neg(p_1.hasResource \wedge p_2.hasResource)$ expresses that p_1 and p_2 can never access the shared resource at the same time.

LTL model checking for finite-state systems deals with the problem: Given a model M of a system, and an LTL formula φ expressing a property of such system, check whether M satisfies φ. If φ is satisfied, the model checking algorithm will return a positive answer. Otherwise, a counterexample will be given in the shape of a trace in M (i.e. an example of the system execution) where the property is violated.

The main approach to model checking is automata-based [3], which means that the system is represented by a Transition System and the LTL property with an automaton that accepts infinite-length words, namely a Büchi Automaton. Note that, Transition Systems can be viewed as automatons and, as explained in Section 1.2.2, automatons accept input words. Therefore, both the system and the property models will accept words representing executions of the system.
Thus, considering M to be a Transition System, and B_φ the automaton representation of φ, intersecting M and B_φ will result in another automaton that accepts the executions accepted by both.

The key idea proposed in the literature [9–11], is to try to find a counterexample by performing the intersection between M and the negation of the property automaton $B_{\neg}\varphi$, rather than B_φ. If the resulting automaton accepts some word, this means that the system accepts an execution that is also accepted by $\neg\varphi$ and hence, violates the property. In order to check whether the intersection automaton accepts some word, an emptiness test is performed, which determines if the language accepted by the automaton is empty.
There are many model checkers which apply explicit model checking tools, such as **SPOT** [12] or **LTL2Buchi** [13].

Further detail on Transition Systems and Büchi Automata will be given in Section 3.

1.3.2 LTL model checking for infinite-state systems

The main technical challenge in the area of model checking is to come up with methods and structures that handle large state spaces. With the appearance of new model checking approaches, such as symbolic model checking, the size of systems that can be handled has increased considerably.
For instance, *Binary Decision Diagrams* [14] (BDDs) have been used to represent transition relations of the system efficiently. For the systems that are regular in some sense (e.g. hardware systems), BDDs are much smaller than the representation by explicit enumeration. This fact and the existence of fast algorithms for manipulation of BDDs have led to the development of recent model checkers, such as **NuSMV** [15] or **SAL** [16](Symbolic Analysis Laboratory).

Another approach to cope with state-space explosion is based on *reduction*, which consists of reducing the size of the state space that is explored. *Partial order reduction* [17] is such a technique, which avoids the generation of all the paths formed by interleaving the same set of transitions .

Recent research in the area of program analysis [18, 19] has shown that program termination methods can be used to adapt LTL model checking for finite-state systems to infinite-state systems. This project is concerned with such adaptation and with the satisfiability check of LTL formulas.

1.4 Project contribution

We aimed to implement tools for handling linear temporal logic (LTL) formulas and checking their validity as properties of computing systems.
Therefore, the contribution of this project can be divided in two parts:

- The development of a toolkit for checking satisfiability, which includes tools to process LTL formulas, build models for them in the shape of Büchi automata and check their emptiness. These tools have been applied to check LTL formulas modeling the behavior of concurrent systems as described in [20].

- The use and adaptation of our LTL tools to combine LTL properties with infinite-state computing systems by performing the intersection between Büchi automata and transition systems. The transition system resulting of the combination of a Büchi automaton and a transition system is sent to $VeryMax$ [21], a program analysis tool which is intended to apply termination analysis techniques to the resulting transition system.

Finally, this project provides a general survey on LTL, Büchi Automata and other ingredients and techniques involved in LTL model checking.

1.5 Project outline

This document is organized as follows:

Chapter 2: Describes the project management in terms of its objectives, obstacles, temporal planning, budget, and sustainability.

Chapter 3: Provides an overview on the formalisms used during this document and conceptually in the development of the project. It formally describes linear temporal logic, formal languages, Büchi automata and transition systems.

Chapter 4: Summarizes how we carry out LTL satisfiability checking in order to check the validity of LTL formulas coming from logic programs. It reviews the steps to follow and gives experimental results.

Chapter 5: Reviews how we perform the combination of transition systems representing computing systems with Büchi automata representing LTL formulas defined over such systems.

Chapter 6: Recapitulates the contents of this document and highlights the most important results derived from the development of the project.

Chapter 7: Provides information about the most important topics for future work.

Chapter 2

Project management

The following sections describe all the concepts related to the management of the project. We provide a description of the objectives and obstacles found, the temporal planning followed, a budget estimation and a sustainability analysis.

2.1 Objectives

The main goals of these project are to implement tools for handling linear temporal logic formulas and checking their validity as properties of computing systems. We aim to implement methods and tools that can be easily introduced into the system development process, and are intuitive both to specify LTL formulas and to interpret the results. Therefore, our objectives can be summarized in the following:

- **LTL satisfiability checking**: Development of tools that perform LTL satisfiability checking in order to determine if an LTL formula modeling a temporal property on a given concurrent system is valid.
- **Combination of computing systems and LTL formulas**: Application and development of tools that build models for both a given system and an LTL property, and combine them. The results are intended to be applied to the program analysis tool $VeryMax$, but that is out of the scope of this project.

Further sections describe the required elements to fulfill such objectives, and detail the methodology that was used during the development of the project.

2.2 Obstacles

The main obstacle found during the implementation of the project has been the fact that some of the Büchi automata representing LTL formulas were untreatable due to their sizes. The sizes of Büchi automata are exponential with respect to the sizes of the LTL formulas they model, so we were not able to generate Büchi automata for some formulas.

Moreover, the conceptual complexity of the algorithm that translates LTL formulas to Büchi automata also came out as an obstacle, since we had to spend more time fully understanding the process and implementing it. Further detail on the actions taken to face or adapt to these obstacles can be found in Section 2.4.3.

2.3 Scope

As explained in Section 1.3.1, LTL formulas specify properties that vary over time, so we can use them to express the desired behavior of a system during its execution. Model checking consists in the verification of those formulas on computing systems, and it requires the creation of abstract models for both the systems and LTL formulas.

In order to build those models, we will base our implementation in the fact that transition systems can be used to represent computing systems, and to model LTL properties we can use automata on infinite words, namely Büchi automata. This type of automata has great expressive power, and yet its emptiness can be checked fairly efficiently, which will be of great use when checking the satisfiability of LTL formulas (see Sections 3 and 4 for more details).

Example 1. The transition system and Büchi automaton in Figure 2.1 are models of a system representing a switch, and an LTL formula $\Box(on \rightarrow \Diamond off)$ which specifies that whenever the switch is on, it will eventually go off.
An execution will be defined by a sequence of labels of the reachable transitions at each state (e.g. sequence $on \rightarrow off \rightarrow on$ in the system model). Therefore, LTL models will represent executions of the system with the desired behavior, and system models will express all possible executions.

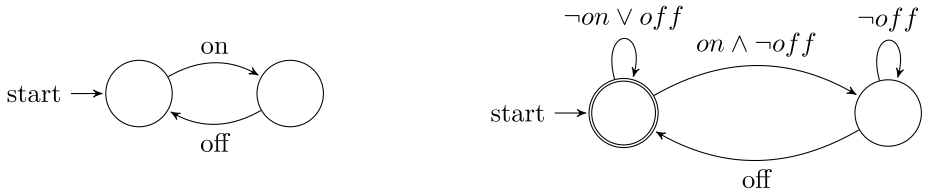


Figure 2.1: System and property automatons that model the behavior of a switch.

Once the models are obtained, we consider two possible approaches in order to apply model checking methods to verify LTL properties on computing systems.

The first one is by trying to verify formulas directly created from computing systems (e.g. from software programs). In such case, it would be enough to check whether the given LTL formulas are valid, thereby confirming that the systems they represent meet the desired properties. Our approach in this case will be to use LTL satisfiability in order to check the validity of the given formulas (see Section 4 for more details).

On the other hand, the second approach consists in adapting the model checking method for finite-state systems to infinite-state systems. In order to do so, given a model M for a system and a Büchi automaton B_φ modeling an LTL property φ over the system, it is necessary to perform $T = M \cap B_{\neg\varphi}$. Later on, the transition system T can be fed to a program analysis tool, which will check whether $M \models \varphi$ by analyzing T. However, this is beyond the scope of this project and we will only be concerned with the generation of transition systems resulting from the intersection.

Considering the process described, the different functionalities we will implement are the following:

- **Specification of LTL formulas**: Users will be able to specify LTL formulas, either coming from software programs or directly embedded in the parts of the computing system they want to verify.
- **Visual output of the models**: In order to obtain more feedback about the verification process, users will be able to see the system and property models, and if necessary, the resulting automaton of their intersection.
- **Satisfiability feedback**: Our tools will provide results for the satisfiability of LTL formulas.

These functionalities are part of the functional objectives defined for our project, and are also an important requirement. Table 2.1 shows the identification of the requirements in relation to the life cycle of the project.

	Life cycle			
Project value	**Planning**	**Implementation**	**Experimentation**	**Closing**
Cost	Human resources costs	Hardware resources costs		
Time	Start: 09-02-2015			End: 29-06-2015
Functional objectives		LTL formulas specification Models visual output Satisfiability feedback		
Quality		Possible optimizations		
Risks		Non-scalable		

Table 2.1: Identification of the requirements

2.3.1 Methodology

For the development of this project we followed a rigorous methodology which helped us carry out the different tasks incrementally and more efficiently.
We started out by implementing the parts that worked as a basis for other parts, and did not move on to a new task until the previous one was completed and validated.
The different stages we followed are listed below in order:

1. **Parsing LTL formulas**: We defined a method to read input LTL formulas and represent them using an internal structure of the parsing algorithm, which helped us transform them into a Büchi automaton later on.
2. **LTL to Büchi automaton**: Once the formulas could be parsed, we were able to build models for them in the shape of Büchi automata.

3. **Visualization methods**: In order to check the automatons built and be confident about their correctness, we implemented visualization methods using the *dot* library.
4. **Emptiness test**: To determine whether the language accepted by a Büchi automaton is empty, we implemented an emptiness test. This allowed us to check the satisfiability of LTL formulas (see Section 4).
5. **Büchi automata intersection**: We implemented a method to intersect transition systems with Büchi automata. This allowed us to obtain results which later on would be fed to a program analysis tool in order to verify LTL properties on computing systems (see Section 5).

The methodology was based on setting short cycles, where we aimed to achieve a certain goal related to the stage we were in. Therefore, we had weekly meetings to discuss the goals that were planned, and the outcome of the weeks compared to the initial planning. This is similar to the $SCRUM$ methodology. This methodology allowed us to have regular control over the different phases of the project, and prevent or solve deviations at an early stage. Moreover, a version control tool was used to keep track of all the different versions of the project that we implemented.

2.4 Planning

In this section, we summarize our temporal planning for the development of the project. This project was carried out in a time frame of five months approximately, starting on the 2^{nd} of February 2015 and finishing on the 29^{th} of June 2015. The following sections describe the breakdown of tasks, the general time planning, resources used, and the deviations that occurred while developing the project.

2.4.1 Description of tasks

Some general tasks have been defined in order to sum up the different steps of the project regarding the organization and implementation. Their definition and detailed explanations will be given in the following sections.

Project definition

This comprises the definition of the subject for the project. An initial stage of research in the LTL verification field was carried out in order to decide whether the project would be viable. And indeed, after documenting ourselves we came to the conclusion that the project would be viable and interesting to develop. Furthermore, we decided that we were going to use some already working verification tools to model the computing systems.

Project management

It involves the project description, project planning, control meetings and bureaucratic tasks.

- **Project description and planning**: It constitutes all the GEP tasks. These include from the description of the project, its scope and resources needed, to a state of the art research and budget calculation. This stage helped us have a better overview of what was necessary to develop the project and also, organize the tasks so it could be done in the stipulated time.

- **Control meetings**: As described in Section 2.3.1, we kept control of the evolution of the project by regularly meeting to discuss the situation at each stage. This practice allowed us to adapt our planning better and prevent critical deviations.

- **Bureaucratic tasks**: They comprise the project inscription, the GEP submission, the final presentation inscription and the final submission of the project.

Environment set up

Before we could start implementing the project, we needed to install the required software. A list of all the software needed is provided in Section 2.4.4.

Main development

The following tasks comprise the implementation of the project.

- **Parsing of LTL formulas**: LTL formulas will determine desired properties of a system, so we needed a way to save and represent formulas provided by the user. Therefore, this task involved programming in *C++* and also using the *Flex* and *Bison* utilities, which generate a scanner and a parser algorithm. These read the LTL formulas and build an intermediate representation for them, respectively.

  Example 1. Consider a system modeling the behavior of a robot. One possible property to verify could be that the robot does not stop until the distance to an obstacle is less than a given threshold. Such property would be modeled by the LTL formula

  $$\neg stop \text{ U } (distance < threshold)$$

  And the corresponding intermediate representation would be the *Abstract Syntax Tree* (AST)-like structure shown in Figure 2.2.

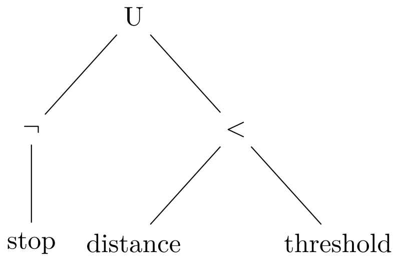


Figure 2.2: Intermediate representation of $\neg stop \text{ U } (distance < threshold)$

- **Generation of Büchi automata**: It comprises the implementation of algorithms that build Büchi automata from LTL formulas, and adapt a system model to a Büchi automaton representation. This involved *C++* programming and looking for a visualization tool that could show the results in a more readable way.

  Example 2. Following Example 1, Figure 2.3 shows the Büchi automaton for the formula *¬ stop U (distance < threshold)*. This automaton accepts all executions where the robot does not stop if it is far enough from an obstacle.

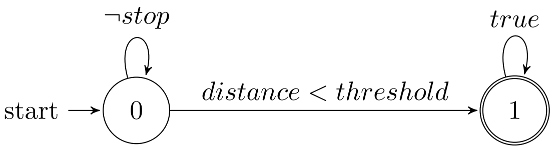


Figure 2.3: Büchi automaton for the formula $\neg stop$ U ($distance < threshold$)

- **Emptiness check**: This test is performed on a Büchi automaton in order to determine if the language it accepts is empty. If the automaton comes from an LTL formula φ modeling a program, then checking the satisfiability of $\neg\varphi$ will determine whether the formula is satisfiable. If that is the case, the LTL property has been verified on the system.
- **Intersection of Buchi automata**: The intersection of Büchi automata is performed in order to obtain a combined model from the computing system and LTL property models. This resulting model could then be fed to a program analysis tool in order to check the satisfiability of the LTL property on the system.

Development control

These tasks were carried out at the end of every development task.

- **Validation**: Our implementation was validated after each development task finished, thereby avoiding the propagation of errors to subsequent stages.
- **Technical documentation**: Documentation was written as implementation tasks finished, and it was extended once the implementation of the project was done.

Experimentation

A stage of experimentation was carried out in order to assess the performance of our tools with different kinds of LTL formulas.

Final stage

This stage was used to gather everything up and finish the technical report, as well as preparing the final presentation.

The following sections describe the temporal planning followed for the development of the project.

2.4.2 Temporal planning

In this section we describe how the temporal planning was organized, the deviations we had to face during the implementation phase, and the resources needed when developing the project.

Timetable

Table 4.1 shows the breakdown of tasks along with the estimated hours spent in each task, their dependencies, and the resources needed to carry them out. The dependencies between tasks are finish to start, except for tasks 8, 9 and 10, which were performed once an implementation task (i.e. tasks 4-7) was done. The hours allocated for those tasks are the total sum of hours spent at each implementation stage.

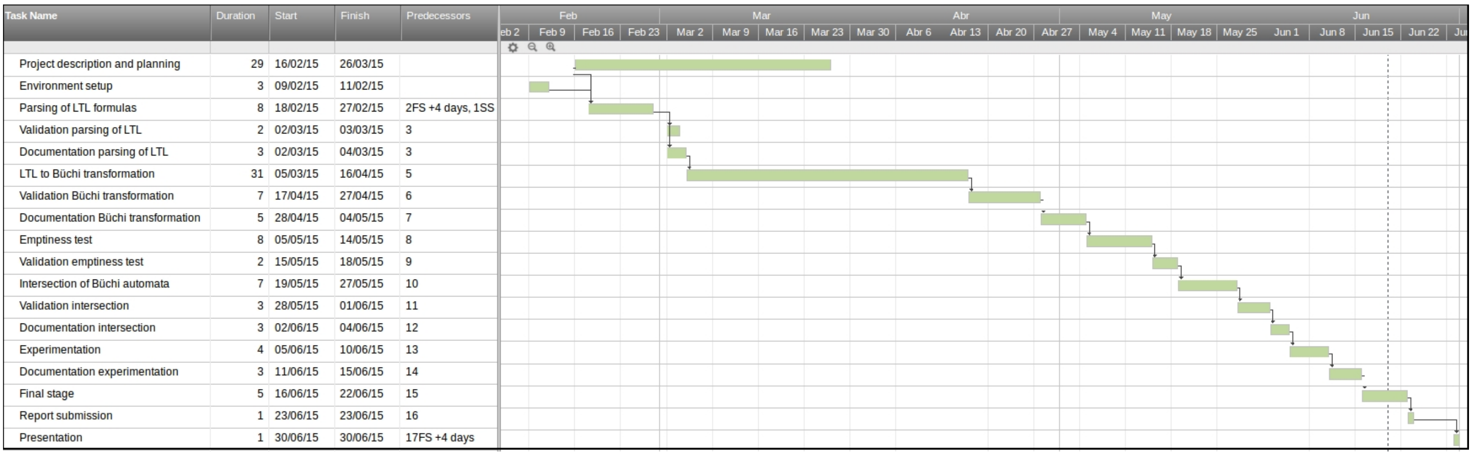
	Tasks	Time (hours)	Dependencies	Resources
1	Project definition	5	—	—
2	Project description and planning	70	1	—
3	Environment set up	5	2	H1
4	Parsing of LTL formulas	40	3	S1, S2, S3, S7, H1
5	Generation of Buchi automatons	90	4	S1, S2, S4, S6, S7, H1
6	Emptiness check	60	5	S1, S2, S4, S7, H1
7	Intersection of Buchi automatons	70	5	S1, S2, S4, S5, S7, H1
8	Experimentation	30	6, 7	S1, S2, S4, S7 H1
9	Validation	30	4, 5, 6, 7 [1]	S1, S2, S4, S7, H1
10	Technical documentation	70	4, 5, 6, 7, 8	S8, H1
11	Control meetings	20	—	—
12	Final stage	30	8	S8, H1
	Total	**520**		

Table 2.2: Tasks breakdown

As can be seen, most of the hours are allocated to implementation tasks, since they comprise the main part of the project.

Gantt chart

Figure 2.4 shows the final Gantt chart. Task 1 was omitted for better visualization, since it was carried out a month earlier than all the other tasks.
Tasks 9 and 10 are broken down into smaller tasks for better understanding, since they were performed right after the corresponding tasks they depended on finished.
The amount of hours dedicated to the project per day was of about five hours on average, and we consider working days.



Task Name	Duration	Start	Finish	Predecessors
Project description and planning	29	16/02/15	26/03/15	
Environment setup	3	09/02/15	11/02/15	
Parsing of LTL formulas	8	18/02/15	27/02/15	2FS +4 days, 1SS
Validation parsing of LTL	2	02/03/15	03/03/15	3
Documentation parsing of LTL	3	02/03/15	04/03/15	3
LTL to Büchi transformation	31	05/03/15	16/04/15	5
Validation Büchi transformation	7	17/04/15	27/04/15	6
Documentation Büchi transformation	5	28/04/15	04/05/15	7
Emptiness test	8	05/05/15	14/05/15	8
Validation emptiness test	2	15/05/15	18/05/15	9
Intersection of Büchi automata	7	19/05/15	27/05/15	10
Validation intersection	3	28/05/15	01/06/15	11
Documentation intersection	3	02/06/15	04/06/15	12
Experimentation	4	05/06/15	10/06/15	13
Documentation experimentation	3	11/06/15	15/06/15	14
Final stage	5	16/06/15	22/06/15	15
Report submission	1	23/06/15	23/06/15	16
Presentation	1	30/06/15	30/06/15	17FS +4 days

Figure 2.4: Gantt chart defining the project planning

[1] Validation and technical documentation was done at the end of each indicated task.

2.4.3 Deviations

We started performing the tasks according to the initial planning until the implementation of the LTL to Büchi automaton transformation task. During this task, we had to face the following problems:

- **Conceptual complexity of the algorithm:** The algorithm chosen had some parts that were difficult to understand, so an extra effort was needed when developing those parts.
- **The algorithm does not scale well:** While for small formulas we could obtain correct and fast results, for bigger formulas there was a blowup in the execution time of the algorithm. This is due to the fact that the size of the automata generated is exponential in the size of the LTL formula. Thus, the bigger the formula the more calculations need to be done.

On one hand, the first problem was solved by doing research on the topic and finding other sources of information. Once we had a clear idea of how all the components worked and we fully understood the algorithm, we finished the implementation.

On the other, the second problem was harder to tackle, since we could not avoid the fact that bigger formulas involve more computations, thereby leading to larger execution times. Hence, we contemplated two possible options: either trying to optimize the algorithm by reusing calculations, or trying another algorithm with a different approach. But since we had already implemented the algorithm, we decided to carry out the optimization. However, it did not show any improvement in the execution times, so we proceeded to the next task and decided to leave the optimizations for the final stage.

The next task we carried out after that task was the emptiness check, rather than the intersection of Büchi automatons. This alteration on the original planning was done because after doing some research, we assumed that it could be done in less time than the stipulated. Thus, we could have a better overview of the time we had left if we finished that task before expected.

2.4.4 Resources

We needed the following software and hardware resources to develop the project:

Software

S1. Ubuntu 14.04: Used in all tasks

S2. Emacs: Editor used to program in C++. Used in all tasks.

S3. Flex and Bison: Programs needed for the LTL formulas parsing.

S4. Visualization tool: We used the *dot* tool.

S5. VeryMax: Used for building transition systems.

S6. BarceLogic: Used when building Büchi automata.

S7. Subversion: Used in all tasks to keep track of the versions of the project.

S8. LaTeX: Used for project documentation.

Hardware

H1. PC: Used in all tasks.

2.5 Budget

In this section we estimate the budget for the project considering the different tasks that were carried out and the resources needed. Even though there were some deviations in the implementation phase, the estimation of costs is the same that we initially performed, since those deviations did not affect the budget. We contemplate the direct, indirect and unforeseen costs associated to the project.

2.5.1 Direct costs

These costs are directly related to the development of the tasks described in Section 2.4.2. Therefore, they are computed considering the different resources needed to carry out the project, which are introduced in the following sections.

Human resources

The different roles that participated in the development of the project were:

1. **Project Manager**: Responsible for establishing the project plan and coordinating resources to complete the plan on time and budget.

2. **Software designer**: Helps creating software that meets the client's needs, in an effective and cost-efficient manner.

3. **Software developer**: Responsible for the technical implementation of the software.

Considering the tasks described in Section 2.4.1, the assignment of tasks for the different roles and the budget allocated for them is shown in tables 2.3 and 2.4, respectively.

	Tasks	Hours	Role 1	Role 2	Role 3
1	Project definition	5	5	—	—
2	Project description and planning	70	70	—	—
3	Environment set up	5	—	—	5
4	Parsing of LTL formulas	40	—	10	30
5	Generation of Büchi automatons	90	—	30	60
6	Emptiness check	60	—	15	45
7	Intersection of Büchi automatons	70	—	25	45
8	Experimentation	30	—	—	30
9	Validation	30	—	15	15
10	Technical documentation	70	—	35	35
11	Control meetings	20[2]	20	20	20
12	Final stage	30	10	10	10
	Total	**520**	**105**	**160**	**295**

Table 2.3: Tasks and hours assignment for each role

Role	Hours	Price per hour	Cost
Project Manager	105	50 €	5.250 €
Software designer	160	35 €	5.600 €
Software developer	295	35 €	10.325 €
Total			**21.175 €**

Table 2.4: Human resources budget

Hardware resources

The only hardware resource we needed was a computer, so the estimated cost for hardware is the one associated to the depreciation, which is computed as follows:

$$4\ years \times 12\ months/year \times 20\ days/month \times 5\ hours/day = 4800\ hours$$
$$Depreciation = (1000\ \text{€}/4800\ hours) \times 520\ hours = 108,33\ \text{€}$$

The results are summarized in Table 2.5.

Product	Price	Useful life	Usage	Depreciation
Computer	1000 €	4 years	520h	108,33 €
Total				**108,33 €**

Table 2.5: Hardware resources costs

Software resources

The cost for software resources is described in table 2.6. Since we will use open-source software, no costs will be associated to it.

Product	Price
Ubuntu 14.04	0 €
Emacs	0 €
Flex and Bison	0 €
Visualization tool	0 €
LaTeX	0 €
Total	**0 €**

Table 2.6: Software resources costs

2.5.2 Indirect costs

Other resources unrelated to the project were also needed for its development and hence, must be considered in the budget as well. Table 2.7 shows the indirect costs associated to electricity and paper.

[2] Each role is assigned the total amount of hours dedicated to this task, since everyone will participate in the meetings. Therefore, the total sum of hours allocated to the roles will be different than the total sum of hours dedicated to the tasks.

On one hand, we assumed that the computer consumed in average 250 W per hour, so the total energy that it consumed during the project (520 hours) was $energy = 250W \times 520h = 130\ kWh$. The rest of the energy (32 kWh) was consumed by electricity.

Product	**Price**	**Units**	**Cost**
Electricity	0.12 €/kWh	162kWh	19,44 €
Paper	4,5 €/pack €	500	4,50 €
Total			**23,94 €**

Table 2.7: Electricity and paper costs

2.5.3 Unforeseen costs

This section includes the costs associated to unexpected events that might have increased the price of the project. We consider them here because they were part of the initial budget estimation. We contemplate one possible scenario which would have slightly deviated the budget.

1. **The computer needing repair**: Since the computer we will use was purchased a few months before starting the project, we will assign a probability of 5% to this event.

Table 2.8 shows the resulting costs.

Event	**Probability**	**Price**	**Cost**
1	5%	200 €	10 €
Total			**10 €**

Table 2.8: Unforeseen costs

2.5.4 Total budget

The total estimated budget for the project is defined in table 2.9. A 5% level of contingency is added in order to contemplate possible deviations not considered in unforeseen costs.

Type	**Cost**
Human resources	21.175 €
Hardware resources	108,33€
Software resources	0 €
Total direct costs	**21.283,33 €**
Paper & Electricity	23,94 €
Total indirect costs	**23,94€**
Total unforeseen costs	**10 €**
Contingency	**1.062,86 €**
Total	**22.383,13 €**

Table 2.9: Total costs

2.6 Sustainability analysis

We introduce the economic, social and environmental sustainability of this project in the following sections.

2.6.1 Economic sustainability

Previous sections show the costs associated to the development of the project. These are mainly covered by the budget required to carry out the project, but they don not contemplate maintenance or future updates. This will be something to consider in future versions since, as it has been already stated, some optimizations might be needed. Therefore, once the development of the project finishes, more research could be done in order to optimize even more. So, costs for future updates should be considered after the project is done.

The price of the project can be considered viable, since most of the costs are covered by human resources, and given the complexity of the software they are implementing, a reasonable budget has been destined for them. Thus, we believe that this price would be competitive if other companies offered the same tools, given that human resources costs cannot vary significantly. Furthermore, we consider that the estimated time to develop the project is rather compressed, so we strongly think that this project cannot be developed in less time.

Also, all resources are essential and therefore none of them could be discarded. Specifically, human resources have been assigned a number of hours according to the difficulty of the tasks that need to be performed, so assigning less hours would result in a project of lesser quality.

Table 2.10 shows the points awarded to this project regarding its economic sustainability. These have been given based on the fact that the price is viable considering the difficulty in the implementation of the project. Moreover, there are no unnecessary resources used and the budget did not change with respect to the initial planning.

2.6.2 Environmental sustainability

The resources needed to develop this project are described in Section 2.4.4. The only resources that could affect the environmental sustainability of this project are the computer, paper and electricity used. These are used in every stage of the project, since the computer will always be running during its development, and electricity and paper will be used by the human resources working on it.

As explained in Section 2.5.2, the energy consumed will be of 162 kWh. This results in 62,37 kg of CO_2, which is a big measure, but still significantly smaller than the average consumption of energy per capita for a year, in comparison. Thus, even though it is a big measure, the resources used are essential, and cannot be reduced. However, we will at least have environmental conscience when using paper, which will be recycled.

Once the project can be used, our users will run our tools in the computers they already have. Therefore, the energy used to apply our tools will be significantly smaller than the energy used to develop the project and the software that uses the tool.

Finally, we would like to state that already implemented software and libraries have been used in the implementation of our project, which means that less hours of development have been needed to develop the project, thereby reducing the level of energy consumption.

Table 2.10 shows the points awarded to this project regarding its environmental sustainability. These have been assigned based on the fact that its environmental impact is mostly due to the energy spent on running the computer and light. So the total consumption is significantly smaller than the average consumption per capita.

2.6.3 Social sustainability

Our tools will be used in developed countries, where people can have access to a computer, and implement applications that need verification. These applications would need to be verified by testing or by hand if it was not for the use of automated verification tools. Therefore, we consider that our project would benefit many computer systems that need verification, avoiding tedious checking processes and giving correct results. This means that people would be using more secure and reliable systems, given that more properties could be easily checked on them.

Hence, software developers would considerably benefit from our tools, since verification would become an easy and fast task that complements the development of their projects, and does not take up as many hours of their implementation work.

We also consider that the development of this project cannot harm any collective, since the collectives involved in systems verification need automated tools that ease their jobs.

Table 2.10 shows the points awarded to this project regarding its social sustainability. These have been assigned considering the fact that it will make a part of the job of some software developers easier.

Sustainable?	**Economically**	**Environmentally**	**Socially**	**Total rows**
Planning	9	8	8	**25**
Results	9	8	8	**25**
Risks	0	0	0	**0**
Total cols.	**18**	**16**	**16**	**50**

Table 2.10: Sustainability matrix

Chapter 3

Preliminaries

This chapter introduces the formalisms used in the project. We provide the definition of linear temporal logic, including a comprehensive survey of the temporal operators and a brief introduction to formal languages and automata theory.

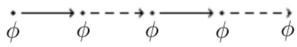
3.1 Linear temporal logic

Linear temporal logic reasons over linear sequences of instants, and extends propositional logic by modal operators that refer to points in time. Hence, it is useful to specify correctness properties about the executions of a system.
Given a set of propositions AP where $p \epsilon AP$, the syntax of LTL formulas is defined as:

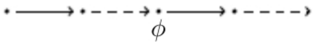
$$\phi ::= \mathit{true} \mid p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \mid \Box\phi \mid \Diamond\phi \mid \phi \text{ U } \phi \mid \phi \text{ R } \phi \mid \text{X } \phi$$

Operators $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are boolean connectives, and $\Box$, $\Diamond$, U, R, X define the temporal modalities of LTL formulae.
Intuitively, temporal operators are defined as follows:

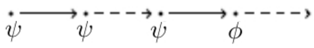
- $\Box\phi$: Called *"always"*. States that ϕ is true at an instant of time and in all the following ones.



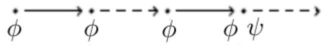
- $\Diamond\phi$: Called *"eventually"*. It is the dual of $\Box$ and states that ϕ is true either at a given instant of time or in the future.

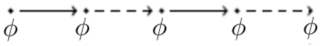


- ψ U ϕ : Called *"until"*. Informally, ψ U ϕ means that ψ is true until ϕ first becomes true. After that, ψ and ϕ can take any value.

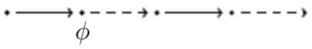


- ψ R ϕ : Called *"release"*. It is the dual of U, and it means that either ϕ is true until and including the point where ψ first becomes true, or ϕ is always true. If ψ and ϕ are true at an instant of time, they can take any value in the following ones.





- X ϕ : Called *next*. States that ϕ is true in the next instant of time.



Some of the operators can be derived from others as defined below:

$$
\begin{aligned}
&\Box\phi = false \text{ R } \phi \\
&\Diamond\phi = true \text{ U } \phi \\
&\phi \rightarrow \psi = \neg\phi \vee \psi \\
&\phi \text{ U } \psi = \neg(\neg\phi \text{ R } \neg\psi) \\
&\phi \text{ R } \psi = \neg(\neg\phi \text{ U } \neg\psi) \\
&\Box\phi = \neg\Diamond\neg\phi
\end{aligned}
$$

An interpretation of an LTL formula is an infinite sequence $\sigma = \sigma_0\sigma_1\sigma_2...$ over the alphabet $\Sigma = 2^{AP}$. Thus, each σ_i defines the propositions that are true at a moment in time.
To determine whether σ satisfies an LTL formula (i.e. it is a model), the semantics are inductively defined. We consider $\sigma|_k = \sigma_k\sigma_{k+1}\sigma_{k+2}...$ for the definition:

$$
\begin{array}{ll}
\sigma \models true & \\
\sigma \models p & \text{iff } p \epsilon \sigma_0 \\
\sigma \models \neg\phi & \text{iff } \sigma \not\models \phi \\
\sigma \models \psi \vee \phi & \text{iff } \sigma \models \psi \vee \sigma \models \phi \\
\sigma \models \psi \wedge \phi & \text{iff } \sigma \models \psi \wedge \sigma \models \phi \\
\sigma \models \psi \rightarrow \phi & \text{iff } \sigma \models \neg\psi \vee \sigma \models \phi \\
\sigma \models \Box\phi & \text{iff } \forall k \geq 0\text{: } \sigma|_k \models \phi \\
\sigma \models \Diamond\phi & \text{iff } \exists k \geq 0\text{: } \sigma|_k \models \phi \\
\sigma \models \psi \text{ U } \phi & \text{iff } \exists k \geq 0\text{: } \sigma|_k \models \phi \wedge \forall 0 \leq i \leq k\text{: } \sigma_i \models \psi \\
\sigma \models \psi \text{ R } \phi & \text{iff } (\exists k \geq 0\text{: } \forall i \leq k\text{: } \sigma|_i \models \phi \wedge \sigma|_k \models \psi) \vee (\forall j \geq 0 : \sigma|_j \models \phi) \\
\sigma \models \text{X } \phi & \text{iff } \sigma|_1 \models \phi
\end{array}
$$

Example 1. Consider a system that receives and processes requests, which has the set of propositions $AP = \{received, processed\}$. One possible property to verify could be:

$$\varphi = \Box(received \rightarrow \Diamond processed)$$

Which states that every time a request is received, it is eventually processed. A model for this property is:
$received, processed, received, received, processed, processed, received, processed, ...$

Note that in Example 1, AP contains propositions where a boolean value is assigned depending on the activation of the actions they represent (i.e. when a request is received, $received = true$). However, we could also have propositions where the boolean value is assigned depending on linear integer arithmetic.

Example 2. Consider a system that manipulates integer variables x and y. An arbitrary LTL formula using those variables and integer arithmetic could be:

$$\varphi = ((x + 1 \leq 0) \text{ U } (y - 2 \geq 2))$$

which states that $x + 1$ has a value smaller or equal than zero until $y - 2$ becomes greater or equal than two.

3.2 Formal languages and automata

Regular languages are formal languages [22] that can be expressed using regular expressions and are recognizable by a finite automaton.

Example 1. A regular language over the alphabet Σ=$\{a, b\}$ is L=$\{ab^*\}$, which represents all the words starting with an a and finishing with a finite sequence of b's.

3.2.1 Finite Automaton

Formally, a finite automaton is a 5-tuple $< Q, \Sigma, \delta, q_0, F >$, where:

Q: A finite set of states

Σ: A finite set of input symbols

δ: A transition function, $Q \times \Sigma \rightarrow Q$

q_0: The initial state, $q_0 \ \epsilon \ Q$

F: A set of accepting states, $F \subseteq Q$

A finite word w over the alphabet Σ is accepted if and only if there exists a finite execution of the automaton on w that ends in an accepting state.

Example 2. Following Example 1, the automaton that represents language L is shown in Figure 3.1.

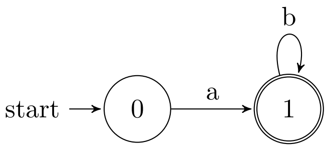


Figure 3.1: Automaton example

On the other hand, an *w-regular language* generalizes the definition of regular language to infinite-length words, meaning that they can not be recognized by finite automata.

Example 3. Following Example 1, an w-regular language over Σ is L=$\{ab^w\}$, which represents all the words starting with an a an finishing with an infinite sequence of b's.

The models of an LTL formula define an w-regular language, and they can be recognized by a special type of automaton which accepts infinite inputs, called Büchi automaton.

3.2.2 Büchi automaton

Formally, a Büchi automaton is a 5-tuple BA $= < Q, \Sigma, \Delta, q_0, F >$, where:

Q: A finite set of states

Σ: A finite set of labels

Δ: A transition relation, $Q \times \Sigma \to Q$

q_0: The initial state, $q_0 \,\epsilon\, Q$

F: A set of accepting states, $F \subseteq Q$

An infinite word w over the alphabet Σ is accepted by a BA if and only if there exists an infinite execution of BA on w that reaches states in F an infinite number of times.

Therefore, an LTL formula ϕ can be translated into a Büchi automaton such that an infinite word w is accepted if and only if $w \models \phi$.

Example 4. Following Example 1, Figure 3.2 shows a Büchi automaton that accepts all the words that satisfy the formula $\Box(received \to \Diamond processed)$. Note that an execution on the automaton is accepted if and only if it goes through state 0 infinitely often.

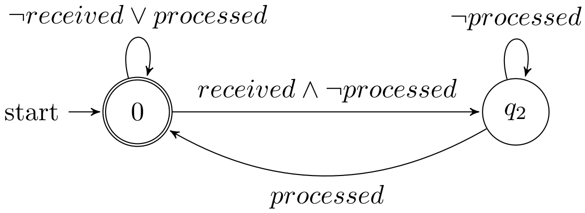

$\neg received \vee processed$
$\neg processed$
start
0
$received \wedge \neg processed$
q_2
$processed$

Figure 3.2: Büchi automaton for the formula $\Box(received \to \Diamond processed)$

In order to translate an LTL formula into a Büchi automaton, our work relies on the use of another type of w-automaton as an intermediate structure: a transition-based generalized Büchi automaton. Such automaton is used to consider eventualities introduced by U formulae in the final Büchi automaton. Further details are provided in Section 4.2.

3.2.3 Transition-based generalized Büchi automaton

Formally, a transition-based generalized Büchi automaton is a 5-tuple TGBA $= < Q, \Sigma, \Delta, q_0, F >$, where:

Q: A finite set of states

Σ: A finite set of labels

Δ: A transition relation, $Q \times \Sigma \to Q$

q_0: The initial state, $q_0 \epsilon Q$

F: A set of sets of accepting transitions, $F \subseteq 2^{\Delta}$

An infinite word w over the alphabet Σ is accepted by a TGBA if and only if there exists an infinite execution of the TGBA on w that contains at least one transition from each accepting set of the sets in F an infinite number of times.
If $F = \emptyset$ then a word w is accepted if and only if there exists and infinite execution of the TGBA on w.

Example 5. Following Example 1, Figure 3.3 shows the TGBA that accepts all the words that satisfy the formula $\Box(received \to \Diamond processed)$.
Note that in this case, $F = \{\{\tau_1, \tau_4\}\}$, and for better visualization transitions labeled with a $\{0\}$ belong to the only set of F. Therefore, an execution on the automaton is accepted if and only if it goes through at least one of the transitions labeled with $\{0\}$ infinitely often.

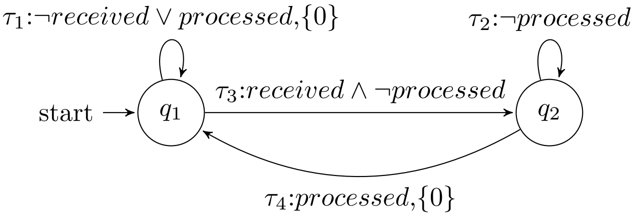

τ_1:¬received ∨ processed,{0}
τ_2:¬processed
τ_3:received ∧ ¬processed
start
q_1
q_2
τ_4:processed,{0}

Figure 3.3: TGBA for the formula $\Box(received \to \Diamond processed)$

So far, we have introduced how LTL properties can be modeled. Next section describes how computing systems can be modeled with the use of Transition Systems.

3.3 Transition System

A Transition System is a 3-tuple TS $= < S, R, I >$ where:

S: A finite set of states

R: A transition relation, $R \subseteq S \times S$

I: A finite set of initial states, $I \subseteq S$

Transition systems are used to describe dynamic processes with configurations representing states, and transitions saying how to go from one state to another. Therefore, a transition system generates a set of sequences of labels from its transitions, which describe all the possible execution paths of the system it is modeling. They can be used to model either finite-state or infinite-state systems, with slight variations.

3.3.1 Finite-state systems

Finite-state systems rely on bounded domains in order to represent computer systems. These domains are obtained through finite abstractions, which apply a limit on the number of elements the state space has. For instance, if a system had to deal with integer numbers, one possible abstraction would be to use only a subset (e.g. 32-bit numbers).
Therefore, these systems provide an explicit representation of the state space, meaning that they enumerate every possible state change the system could perform.

Example 1. Consider a system modeling a counter between 0 and N, which increments and decrements. An explicit representation of such system for $N = 3$ is shown in Figure 3.4. As can be seen, the domain is bounded to $D = \{0..N\}$, and there is a state for each increment/decrement of the counter.

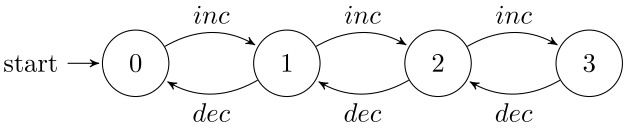


Figure 3.4: Transition System for counter system

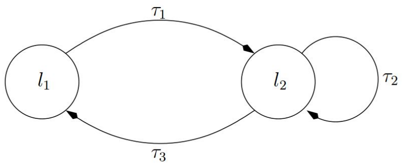
Since transition systems can define infinite traces, they can be considered Büchi automata where all states are accepting states. Intuitively, a transition system defines all the possible executions of a system, so any infinite run in the transition system would be an accepted run. Moreover, those transition systems that do not represent infinite paths can be turned into Büchi automata by adding self-loops to ending states, which do not change to another state anymore.

3.3.2 Infinite-state systems

As seen in the previous section, finite-state systems need to bound their domain through finite abstractions. This clearly imposes a limit on the power of representation transition systems can provide when modeling computer systems. For instance, following Example 1, if we had to represent a system which performs an undefined number of increments/decrements, it would not be possible to use the representation given. Moreover, if a system needed to perform a huge amount of increments, the state space of the model would increase dramatically.
On the other hand, infinite-state systems do not impose a bound on their state space domain. Hence, they allow one to represent more complex systems, such as software programs manipulating integer variables. Such systems require more general models which, for instance, admit the representation of assignments between variables.

Example 2. Figure 3.5 shows an example of a program manipulating integer variables and a transition system representing it.

```
int main()
{
  int x=undet(),y=undet(),z=undet();
  l1: while (y>=1) {
       x--;
  l2:  while (y<z) {
         x++; z--;
       }
       y=x+y;
     }
}
```



$$\begin{array}{lllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}$$

Figure 3.5: Transition system for program

As can be seen, transitions of the transition system have two parts: conditions and assignments. The former determine whether a transition can be activated, and the latter express the changes variables x, y and z experiment once the transition is fired. Hence, primed variables indicate the new values of x, y and z after the transition is activated, and unprimed ones represent their values before the transition, as explained in [23].

Moreover, initial states have one or more associated entry transitions which are applied to the initial values, before reaching the initial location. In example of Figure 3.5, the only initial location is 1_1, which has a single entry transition with condition *true* and no assignments.

Chapter 4

Satisfiability of LTL formulas

In this section we introduce a method to check the validity of LTL formulas through LTL satisfiability.

An LTL formula is valid if every possible interpretation is a model (i.e. satisfies the formula). Formally, given an LTL formula φ defined over a set of propositions AP, the formula specifies a language $L(\varphi) = \{\sigma \mid \sigma \in (2^{AP})^w\text{: } \sigma \models \varphi\}$ with all the traces that satisfy it. Thus, φ is valid if and only if all possible interpretations of φ belong to $L(\varphi)$, that is:

$$(2^{AP})^w = L(\varphi)$$

Such condition implies checking that all the elements of $(2^{AP})^w$ are contained in $L(\varphi)$.

To this end, we can check the satisfiability of $\neg\varphi$. Intuitively, if $\neg\varphi$ has a model, this means that there is an interpretation that does not satisfy φ (as it satisfies $\neg\varphi$), and thus φ is not valid. This can be performed by checking whether the language defined by $\neg\varphi$ is empty. If that is the case, there will be no interpretations that satisfy $\neg\varphi$, and φ is proven to be valid. Formally:

$$\varphi \equiv \mathit{true} \leftrightarrow L(\neg\varphi) = \emptyset$$

One way to check whether $L(\neg\varphi) = \emptyset$ is by using a Büchi automaton that accepts the words in $L(\neg\varphi)$. Once the automaton is built, it is possible to perform an emptiness test in order to see whether the language it accepts is empty or not.
Therefore, the steps to follow are:

1. Negate the formula
2. Translate from LTL to Büchi automaton
3. Perform an emptiness test

The following sections describe how we parse an LTL formula φ, translate $\neg\varphi$ into a Büchi automaton and perform the emptiness test on the latter in order to check the satisfiability of $\neg\varphi$ (i.e. the validity of φ).

4.1 Parsing the formulas

The first step is to parse the LTL formulas specified by the user in the system code.
LTL properties will be specified as code statements among the system's source code. Hence, they will be evaluated in the point where they are inserted, rather than globally for the whole system.
We will use the *Flex* and *Bison* utilities to parse the formulas and will save and represent them with an Abstract Syntax Tree (AST) structure.

Once the formulas are parsed, the Büchi automaton representing each formula will be built following the method introduced in the following section.

4.2 Automaton transformation

As stated in Section 3.2, given an LTL formula ϕ defined over a set of propositions AP, it is possible to build a Büchi automaton representing such formula.
The transition labels of the automaton will be propositional formulas over AP, and they will possibly represent multiple elements of 2^{AP}.

Example 1. For $AP = \{a, b\}$ and $\Sigma = 2^{AP}$, the label *true* would correspond to any of the elements of Σ (i.e: $\{a\}$, $\{b\}$, $\{a, b\}$), the label a would correspond to either $\{a\}$ or $\{a, b\}$, and the label $\neg a \wedge b$ would correspond to $\{b\}$.

Therefore, transition labels on the Büchi automaton will determine the propositions that need to be true for a transition to be activated, regardless of the value of the other propositions.
This means that a run in the Büchi automaton will be an infinite sequence of interpretations over AP that will make the LTL formula satisfiable.

Note that the size of the generated Büchi automaton might be exponential in the size of the LTL formula, as explained in [24]. Hence, Büchi automata can end up being huge, depending on the formulas they are generated from.

For the translation, there are different approaches in the literature [25, 26], but the algorithm used in this project is based in that of [13, 27], and it is carried out in the following steps:

1. Formula rewriting.

2. Transformation of the formula into a TGBA.

3. Degeneralization of the TGBA to obtain a Büchi automaton.

The first step is performed in order to obtain smaller automata and simplify the transformation algorithm by using less temporal operators. The second one takes into consideration eventualities introduced by U formulae, and the third generates the resulting Büchi automaton. Each step is explained in the following sections.

4.2.1 Formula rewriting

Formulas will be rewritten into their Negation Normal Form (NNF), where the negation operator $\neg$ is only found in front of propositions.

Example 2. Let p and q be propositions. Examples of the Negation Normal Form of LTL formulas are:

$$\begin{array}{ll}
\neg\Box p = \Diamond\ \neg p & \neg(\mathrm{X}\ p) = \mathrm{X}\ \neg p \\
\neg\Diamond p = \Box\ \neg p & \neg(p \wedge q) = \neg p \vee \neg q \\
\neg(p\ \mathrm{U}\ q) = \neg p\ \mathrm{R}\ \neg q & \neg(p \vee q) = \neg p \wedge \neg q \\
\neg(p\ \mathrm{R}\ q) = \neg p\ \mathrm{U}\ \neg q &
\end{array}$$

Furthermore, the only operators that will be used are U, R, $\vee$ and $\wedge$, since the other ones can be derived from these (see Section 3.1).

In order to avoid an exponential blow up in the size of the formula rewritten, we will consider both U and R, even though they can be derived from each other.

In addition to that, we have also implement a simplification step using the following rewriting rules introduced in [13, 27]. Let ϕ, ψ and φ be LTL formulas. The rewriting rules applied are:

$$\begin{array}{ll}
\phi \wedge \phi \rightarrow \phi & X\ true \rightarrow true \\
\phi \wedge true \rightarrow \phi & \phi\ \mathrm{U}\ false \rightarrow false \\
\phi \wedge false \rightarrow false & (\Box\Diamond\ \phi) \vee (\Box\Diamond\ \psi) \rightarrow \Box\Diamond(\phi \vee \psi) \\
\phi \wedge \neg\phi \rightarrow false & \Diamond X\ \phi \rightarrow X\Diamond\ \phi \\
\phi \vee \phi \rightarrow \phi & \Box\Box\Diamond\ \phi \rightarrow \Box\Diamond\ \phi \\
\phi \vee true \rightarrow true & \Diamond\Box\Diamond\ \phi \rightarrow \Box\Diamond\ \phi \\
\phi \vee false \rightarrow \phi & X\Box\Diamond\ \phi \rightarrow \Box\Diamond\ \phi \\
\phi \vee \neg\phi \rightarrow true & \Diamond(\phi \wedge (\Box\Diamond\ \psi)) \rightarrow (\Box\ \phi) \wedge (\Box\Diamond\ \psi) \\
(X\phi)\ \mathrm{U}\ (X\psi) \rightarrow X(\phi\ \mathrm{U}\ \psi) & \Box(\phi \vee (\Box\ \Diamond\ \psi)) \rightarrow (\Box\ \phi) \vee (\Box\Diamond\ \psi) \\
(\phi\ \mathrm{R}\ \psi) \wedge (\phi\ \mathrm{R}\ \varphi) \rightarrow \phi\ \mathrm{R}\ (\psi \wedge \varphi) & X(\phi \wedge \Box\Diamond\ \psi) \rightarrow (X\ \phi) \wedge (\Box\Diamond\ \psi) \\
(\phi\ \mathrm{R}\ \psi) \vee (\varphi\ \mathrm{R}\ \psi) \rightarrow (\phi \wedge \varphi)\ \mathrm{R}\ \psi & X(\phi \vee \Box\Diamond\ \psi) \rightarrow (X\ \phi) \vee (\Box\Diamond\ \psi) \\
(X\phi) \wedge (X\psi) \rightarrow X(\phi \wedge \psi)\ \mathrm{R}\ \psi &
\end{array}$$

4.2.2 LTL to TGBA

Given an LTL formula φ, the algorithm relies on the construction of an intermediate graph of nodes that somehow contain subformulae of φ. Transitions and states of the TGBA come from the nodes of such temporal graph, and the algorithm returns the TGBA generated.

Intuitively, a transition in the TGBA will determine the literals [1] that hold at a given moment in time, and a state will represent formulae that hold on an execution starting from it. Therefore, during the TGBA construction, a node will keep information about the formulas that must be satisfied in the current step, and formulas that must be satisfied from the next step in time on. So, in the end, all intermediate nodes are processed and some of them provide transitions and states of the TGBA.

The algorithm will create and expand nodes by applying recursive rules that decompose subformulae of φ. These are based on LTL semantics (see Section 3.1) and the expansion of temporal operators U and R:

$$\phi\ \mathrm{U}\ \psi = \psi \vee (\phi \wedge\ \mathrm{X}(\phi\ \mathrm{U}\ \psi)) \qquad \phi\ \mathrm{R}\ \psi = \psi \wedge (\phi \vee\ \mathrm{X}(\phi\ \mathrm{R}\ \psi))$$

[1] Literal: A proposition or its negation.

This decomposition will define the temporal interpretations that satisfy φ by creating different paths in the automaton. Therefore, nodes will expand by applying these rules on the formulas that must hold on them in the current step of time.

The information that nodes will keep is the following:

- **Id**: A unique node identifier.
- **Incoming**: A set of identifiers of all the nodes that have an edge pointing to the current node.
- **New**: A set of LTL formulae that must be satisfied immediately but that have not yet been processed (proven to hold). This set is used during the construction and is empty for all nodes of the final TGBA.
- **Old**: A set of literals that must be satisfied immediately and have already been processed.
- **Next**: A set of LTL formulae that must be satisfied in all successor nodes.
- **Untils**: Bitmap of size equal to the number of U subformulas in the formula being translated. A bit is set if the corresponding subformula has been processed in this node.
- **Right-of-untils**: A bitmap that records, for each subformula ϕ U ψ, whether ψ has been processed in this node.

Furthermore, states of the TGBA will hold the following information:

- **Id**: A state identifier.
- **Transitions**: A set containing the incoming transitions.
- **Next**: A set of LTL formulae that must hold in all the immediate successors of the state.

And transitions will keep the following records:

- **Source**: A set of Ids of the source states. These have a transition to the same state with the same transition label.
- **Label**: The set of literals that must hold for the transition to be triggered.
- **Accepting**: A bitmap that records to which accepting sets the transition belongs.

Thus, *New* will only be used to expand a node until nodes where all formulae have been processed are reached. These nodes will represent transitions of the TGBA labeled with the formulas in their *Old* field. Furthermore, they will define the states where any temporal interpretation starting from them satisfies formulae in their *Next* set.

Algorithm 4.3 shows how node expansion is performed. The line numbers in the following description refer to that algorithm.

As can be seen, there are two possible cases:

- **There are no formulas left to process** (lines 2 to 12): Which implies that *New* is empty. In this case, if the node is equivalent to an existing state of the TGBA, they will merge. This means that the state will receive all the information regarding the transition defined on the node by its *Old* field. So, merging allows us to generate all the different transitions that lead to the same state. Algorithm 4.1 shows how this step is performed.

Algorithm 4.1 Node and state merging. Function called from a state.

```
function MERGE(Node n)
    $acc = \sim(n.Untils) \mid (n.Right\text{-}of\text{-}Untils)$
    if $\exists t \in Transitions$ s.t. $(t.Label = n.Old)$ and $(t.Accepting = acc)$ then
        $t.Source \cup \{n.Incoming\}$
    else
        $Transitions = Transitions \cup \{$new $Transition\ nt$ with $nt.Source = n.Incoming,$
        $nt.Label = n.Old,\ nt.Accepting = acc\}$
```

On the other hand, if the node is not equivalent to any state, a new state and transition will be created from its *Next* and *Old* fields, respectively. Moreover, a new node will be created in order to process the formulas in the *Next* field.

- **There are formulas left to process** (lines 14 to 33): Formulas are processed one by one, and if no contradictions or redundancies are found, the node is expanded according to the rules. Both the test for contradiction and redundancy check are based on the ideas of [13, 27]. Therefore, fields *Old* and *Next* of the current node are searched in order to find possible conflicts or redundancies.

  Example 3. Let $\varphi = \phi \text{ U } \psi$ be a formula being processed by a node n during its expansion. A contradiction would be found if $\neg\varphi = \neg\phi \text{ R } \neg\psi$ had already been processed in the node, meaning that either $\neg\psi$ belongs to $n.Old$ and $(\neg\phi \text{ R } \neg\psi)$ belongs to $n.Next$, or $\neg\phi$ and $\neg\psi$ belong to $n.Old$.
  Furthermore, the formula would be redundant if either ϕ belonged to $n.Old$ and $(\phi \text{ U } \psi)$ belonged to $n.Next$, or ψ belonged to $n.Old$.

  Fields *Right-of-Untils* and *Untils* are updated as necessary (lines 15-17 and 22-24) in order to keep track of the accepting sets to which the node belongs. Further explanation is provided below in this section.

  Once the formula is checked against contradictions and redundancies, the node is either split or updated, depending on the type of formula that is being processed.

  Therefore, if the formula is either a U, R or $\vee$ formula (lines 25 to 27), the node will be split in order to create different paths that can satisfy the formula. This task is performed as shown in Figure 4.1, and it is based in the decomposition provided in Table 4.1.

Algorithm 4.2 Node splitting. Function called from a node.

```
function SPLIT(Formula f)
    Create new node $node2$ with new $Id$ s.t.
    ($node2.Incoming = Incoming$, $node2.New = New \cup (\mathbf{New2}(\mathrm{f}) \setminus Old)$,
    $node2.Old = Old \cup \{f\}$, $node2.Next = Next$, $node2.Untils = Untils$ and
    $node2.Right\text{-}of\text{-}Untils = Right\text{-}of\text{-}Untils$)
    Modify this (current node) as follows:
    $New = New \cup (\mathbf{New1}(\mathrm{f}) \setminus Old)$, $Old = Old \cup \{f\}$, $Next = Next \cup \mathbf{Next1}(f)$
    return $node2$
```

Formula	**New1**	**Next1**	**New2**
φ U ψ	$\{\varphi\}$	$\{\phi$ U $\psi\}$	$\{\psi\}$
ϕ R ψ	$\{\psi, \varphi\}$	$\{\psi, \phi\}$	$\{\phi$ R $\psi\}$
$\phi \vee \psi$	$\{\phi\}$	$\emptyset$	$\{\psi\}$
$\phi \wedge \psi$	$\{\psi, \phi\}$	$\emptyset$	$\emptyset$

Table 4.1: Definition of New and Next for non-literals

Example 4. Consider the formula ϕ U ψ being processed in a node. Figure 4.1 shows how during its expansion, the node is split in two different nodes. The new nodes will also expand until their *New* set is empty.

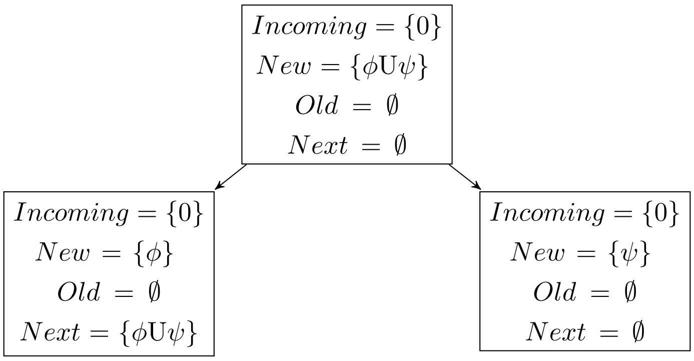


Figure 4.1: Node splitting

On the other hand, if the formula is of the type $\phi \wedge \psi$ (lines 28 to 30) the node will not split, since ϕ and ψ need to hold at the same time.

Finally, if the formula is a literal (lines 32 to 33) it can be directly added to the *Old* set, since it has already been checked against contradictions and redundancies, .

Accepting sets

In the end, the TGBA will represent all the possible models of φ. Nevertheless, by construction there will be paths where ϕ subformulae for ϕ U ψ is satisfied, but no node satisfies ψ along the path. Such paths are not models of ϕ U ψ, which requires ψ to be true eventually.

Therefore, given that for the rest of operators any infinite path in the TGBA would be accepting, an acceptance family is defined over U subformulae. This will allow us to discard those paths that do not represent models of U formulas.

Hence, every different U subformula will define an acceptance set. These acceptance conditions will be applied to nodes during the construction (lines 6 to 8), and finally defined over TGBA transitions when merging nodes and states (see Algorithm 4.1).
A node will belong to an accepting set if either the U formula that defines that set does not hold in that node, or the right side of the formula holds. Formally, given a node n, the acceptance condition over a U formula is defined as: $(\phi \text{ U } \psi \textit{ processed in } n) \rightarrow (\psi \textit{ processed in } n)$, which is equivalent to $\neg(\phi \text{ U } \psi \textit{ processed in } n) \vee (\psi \textit{ processed in } n)$.

Fields *Untils* and *Right-of-untils* are used in order to keep track of the U formulas being processed in a node and their right subformulas, respectively. So, in order to obtain the accepting sets to which a node belongs, the acceptance condition stated above has to be applied for all the U formulas represented in the bitmaps. This can be done by carrying out the following bitwise operation:

$$\sim (Untils) \mid (Right\text{-}of\text{-}Untils)$$

Where $\sim$ represents bitwise negation and $\mid$ represents bitwise or.

Example 5. Let p, q and r be propositions and $\varphi = p \text{ U } (q \text{ U } r)$. There will be an accepting set defined for φ and another one for $q \text{ U } r$. Therefore, *n.Untils* and *n.Right-of-untils* will have size equal to 2 for all nodes and initially, all of their bits will be set to 0.
Each position of the bitmaps will reference the accepting set associated to that index, so every time one node processes φ, the bit 0 of *n.Untils* will be set to 1, and bit 1 will remain the same. The same applies to *n.Right-of-untils* when the right side of one of the U subformulas is processed.

Once the TGBA has been built, a degeneralization process will be applied in order to assure that all the accepting conditions are met. The result will be a Büchi automaton which represents φ.

Algorithm 4.3 Node expansion. Function called from a node.

```
function EXPAND(List states)
    if $New = \emptyset$ then
        if $\exists s \in states$ s.t $Next = s.Next$ then
            $s$.merge(this)
            return states
        $acc = \sim(n.Untils) \mid (n.Right\text{-}of\text{-}Untils)$
        create new state $s$ with $s.Id = Id$, $s.Transitions = \{$new $Transition\ t$ with
        $t.Source = Incoming$, $t.Label = Old$, $t.Accepting = acc\}$ and $s.Next = Next$
        $states = states \cup s$
        create new node $node$ with new Id, $node.Old = \emptyset$, $node.Next = \emptyset$,
        $node.Incoming = Id$ and $node.New = Next$
        return $node$.expand($states$)
    else
        let $formula \in New$ and remove $formula$ from $New$
        if $formula$ is the right side child of a $U$ formula then
            Set to 1 the bits in $Right\text{-}of\text{-}Untils$ corresponding to the $U$ formulas for which
            $formula$ is their right side.
        if testForContradiction($formula$) then
            return states
        if isRedundant($formula$) then
            return $this$.expand($states$)
        if $formula$ is a $U$ formula then
            Set to 1 the bits in $Untils$ corresponding to the $U$ formula that $formula$ represents.
        if $formula$ is not a literal then
            if $formula$ is a $U$, $R$ or $\vee$ formula then
                $node2$ = split($formula$)
                return $node2$.expand(expand($states$))
            if $formula$ is a $\varphi_1 \wedge \varphi_2$ formula then
                $New = New \cup (\{\varphi_1, \varphi_2\} \setminus Old)$
                return expand($states$)
        else
            $Old = Old \cup formula$
            return expand($states$)
```

4.2.3 Degeneralization

The TGBA generated in the previous step is translated into a regular Büchi automaton by performing a "Degeneralization" process. This process will deal with the accepting conditions defined over the TGBA, and it will assure that the final Büchi automaton meets all of them.

This is performed by computing the synchronous product between the TGBA and what the literature calls a *degeneralizer* [28].

A *degeneralizer* is a deterministic Büchi automaton that expresses the fact that a path can only be accepting if it contains infinitely often at least one accepting transition from each of the accepting sets. Such automaton has $|F| + 1$ states, where one of them is accepting.
Moreover, each transition is labeled with either numbers referring to the accepting sets, or an *else*. This distinction defines priorities over transitions.

Example 6. Figures 4.2 and 4.5 show examples of degeneralizers for a different number of accepting sets. Thick lines denote the highest priority, whilst dashed lines define the lowest.

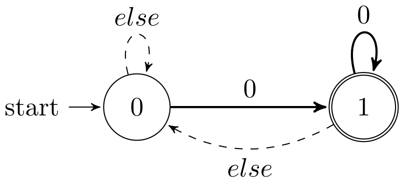


Figure 4.2: Degeneralizer for a TGBA with one accepting set

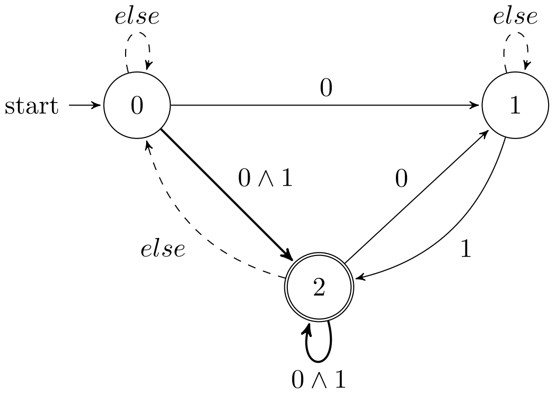


Figure 4.3: Degeneralizer for a TGBA with two accepting sets

In order to perform the synchronized product, the TGBA is considered as a Büchi automaton where all its states are accepting. This is done so the accepting states in the final Büchi automaton only depend on the accepting state of the degeneralizer (see Section 5.1 for more details).
That way, the product is performed as a regular synchronous product between Büchi automata, but having the following considerations:

- A joint transition (t_1, t_2), where t_1 belongs to the degeneralizer and t_2 belongs to the TGBA, can only be fired if and only if t_2 belongs to the accepting set or sets that the predicate on t_1 requires.
- Transitions on the degeneralizer are explored by priority, so transitions from the TGBA are only combined with those that have the highest priority possible on the degeneralizer.

Therefore, the final Büchi automaton will have as accepting states the ones where a state from the TGBA is combined with the accepting state from the degeneralizer. Furthermore, transitions will be labeled with transition labels from the TGBA, where all accepting-related labels are removed.

Example 7. Let $\varphi = p \text{ U } q$. Figure 4.4 shows the TGBA representing φ.

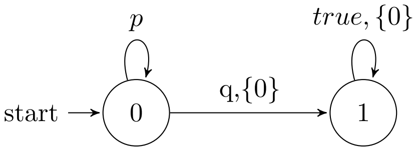


Figure 4.4: TGBA representing p U q

In order to obtain the final Büchi automaton we need to perform the synchronized product between this TGBA and the degeneralizer in Figure 4.2. The result is the following:

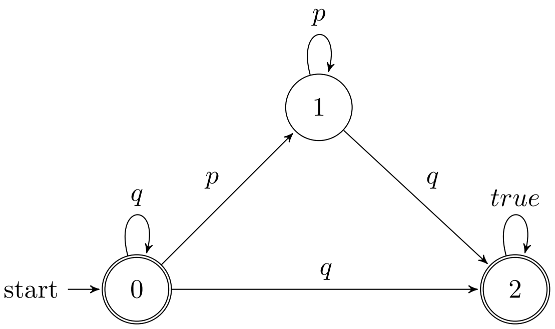


Figure 4.5: Büchi automaton representing p U q

Finally, note that because of the deterministic behavior of the degeneralizer, an order is imposed on the fulfillment of the accepting conditions. Nevertheless, since accepting transitions have to be visited infinitely often, the order does not affect the language accepted by the final automaton. This means that we could choose any order for the accepting conditions to be explored, but some of the them might lead to bigger Büchi automata. In our case we chose to set an incremental order starting from 0.

Once the Büchi automaton is built from the negation of the desired LTL formula, an emptiness test is performed in order to check its satisfiability.

4.3 Emptiness test

The emptiness test checks whether a Büchi automaton defines an empty language or not. Recall that a Büchi automaton accepts an infinite input if there exists a trace that visits an accepting state infinitely often. Therefore, the algorithm used for the emptiness test must look for some kind of cycles in the automaton which contain accepting states. These cycles ensure that the trace found will be infinite, and the accepting states in them will be visited infinitely often. Thus, the language is empty if and only if there are no such cycles.

Example 1. Figure 4.6 shows a Büchi automaton that accepts the language $L = \{a(ba)^w\}$ and, as can be seen, there exists a cycle between states 1 and 2.

There are different methods proposed in the literature [29], but most of them are based on either applying reachability methods or looking for Strongly Connected Components (SCC) [30] in the

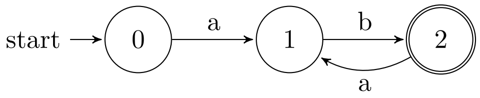


Figure 4.6: Büchi automaton example

Büchi automaton. The algorithm we implemented is based on the former, and it is described in the following section.

Nested DFS

DFS stands for *Depth First Search* [31], which is a method for traversing the states of a graph. The Nested DFS algorithm [32] traverses a Büchi automaton by doing a DFS starting from the initial state. When it encounters an accepting state, it performs another DFS from that state in order to see if there is a path that leads back to it (i.e. a cycle).
The first DFS procedure is called *blue* and the second one (i.e. the nested) is called *red*. While each procedure traverses the Büchi automaton, it marks the states it visits as either *blue* or *red*. This avoids visiting the same states repeatedly for any of the two DFS procedures.

Finally, if a path that leads to a cycle containing accepting states is found, this means that the language the automaton recognizes is non-empty. Otherwise, the automaton does not accept any word.

Example 2. Following Example 1, the Nested DFS would start by running a *blue* DFS from state 0. The procedure would visit state 1 and state 2, and then from state 2 a *red* DFS would start running. It would first visit 1 and then 2, realizing that there exists a path in the Büchi automaton that holds a cycle which contains accepting states. The algorithm would then finish and state that the language accepted by the Büchi automaton is not empty.

4.4 Experiments

We carried out a series of experiments in order to assess the performance of our tools with different kinds of formulas. LTL formulas found in the literature were rather small, and did not allow us to test our tools to the limit. Hence, we not only tried formulas found in the literature, but also bigger examples provided by [20], which represented concurrent logic programs. In order to deal with these formulas, we manipulated them keeping their semantics intact, but turning them into formulas with linear integer arithmetic expressions in their propositions.

4.4.1 LTL over linear integer arithmetic expressions

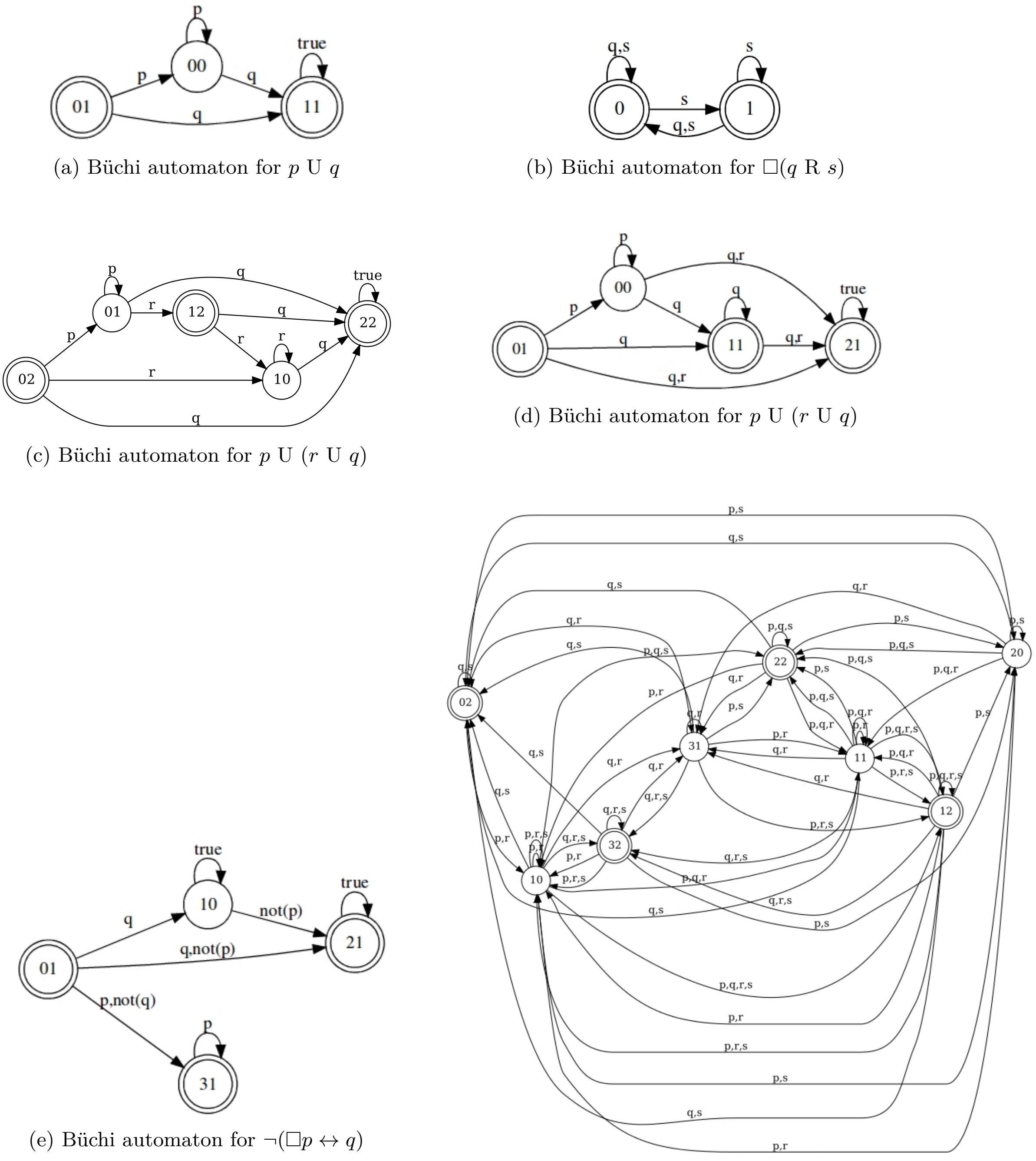
In order to ease the description of concepts related to linear temporal logic, we have assumed that LTL formulas only had boolean propositions. However, our LTL formulas can be built over atoms in linear integer arithmetic as well, and all we have described in Section 3 applies in the same way if the language is enriched in this manner. Moreover, a *SAT Modulo Theory* (SMT) [33] solver is used to check that transitions in Büchi automata are feasible, meaning that every transition has a satisfiable formula. The SMT solver used in our case is the *Barcelogic* [34] solver.

Table 4.2 shows the number of states and transitions for some of the formulas we used, as well as their results after performing the emptiness test. The formulas are not negated throughout the process, they are considered to be already negated when building Büchi automata. Moreover, all the subformulas hold propositions.

As can be seen, even with small formulas one can get a sense of how big the Büchi automata generated can be with respect to the size of the formula. For instance, take formula 20, for which the size of the Büchi automaton generated is considerably bigger than the size of the formula. Figure 4.7 provides examples of Büchi automata generated for some of the example formulas.

	Formula	**# states**	**# trans.**	**Satis.?**
1	p U (q U (r U s))	4	10	yes
2	p U (r U q)	3	6	yes
3	p U q	2	3	yes
4	p U $(q \wedge \neg q)$	1	0	no
5	p U (r R q)	3	6	yes
6	$(\Box\Diamond p)$ U q	6	20	yes
7	p R q	2	3	yes
8	p R (q R (s R t))	8	43	yes
9	$\Box p$	1	1	yes
10	$\Box(q$ R $s)$	2	4	yes
11	$\Box((p$ U $q) \wedge (r$ U $s))$	4	25	yes
12	$\Box\Diamond(p \rightarrow q)$	2	8	yes
13	$\Box(p$ U $q)$	2	5	yes
14	$\Box(p \rightarrow Xq)$	2	4	yes
17	$\Box\Diamond p \leftrightarrow \Box\Diamond q$	15	54	yes
18	$\Box\Diamond p \rightarrow \Box(q \rightarrow \Diamond r)$	5	15	yes
16	$\Diamond(p$ U $q)$	3	6	yes
19	$((\Box p) \wedge (\Box\Diamond q)) \vee ((\Box\Diamond r) \wedge (\Box\Diamond s))$	7	36	yes
15	$\neg(\Box p \leftrightarrow q)$	4	7	yes
20	$\neg(p_1$ U $(p_2$ U $(p_3$ U $(p_4$ U $(p_5$ U $(p_6$ U $(p_7$ U $p_8)))))))$	128	10923	yes

Table 4.2: Examples of formulas



(a) Büchi automaton for p U q

(b) Büchi automaton for $\Box(q$ R $s)$

(c) Büchi automaton for p U (r U q)

(d) Büchi automaton for p U (r U q)

(e) Büchi automaton for $\neg(\Box p \leftrightarrow q)$

(f) Büchi automaton for $\Box((p$ U $q) \wedge (r$ U $s))$

Figure 4.7: Examples of Büchi automata

One of the biggest formulas we have tested is:

$\varphi = ((((((X(\Box(IError = n0 \wedge FError = FE3))) \wedge (X(\Box(IDoor = n1 \wedge FDoor = FD1)))) \wedge (X(\Box(IButton = n2 \wedge FButton = FB2)))) \wedge (((\Box(IDoor = 2 \wedge FDoor = FD1)) \wedge (\Box(IButton = 3 \wedge FButton = FB2)) \wedge ((X(\Box(IE3 = 4 \wedge FE3 = FE14))) \wedge (X(\Box(IB2 = 1 \wedge FB2 = FB15))))) \vee ((\neg(\Box(IDoor = 2 \wedge FDoor = FD1)) \wedge (\Box(IButton = 3 \wedge FButton = FB2))) \wedge (X(\Box(IE3 = 0 \wedge FE3 = FE16)))))) \wedge (X(FB2 = 3 \rightarrow ((X^5 FD1 = 2) \vee FE3 = 4)))) \wedge \neg(FButton = 3 \rightarrow ((X^5 FDoor = 2) \vee FError = 4)))$

This formula represents a logic program, and holds atoms over integer linear arithmetic. The Büchi automaton generated from φ has 34 states and 55 transitions, and after running the emptiness test on it, we could confirm that φ is satisfiable.

For other formulas of this magnitude we were not able to generate a Büchi automaton, since the algorithm did not provide a result in a reasonable amount of time (four hours) according to [24]. Recall that the size of the corresponding Büchi automaton can be exponential with respect to the size of the LTL formula. All formulas provided in this section were generated within seconds, but for formulas with many nested temporal operators (specially R operators), the execution time of the algorithm and the size of Büchi automata dramatically grew. One example of this fact is formula 20 from Table 4.2. Even though it is much smaller in size than φ, the Büchi automaton generated is significantly bigger, due to the fact that formula 20 has many nested R operators (recall that $\neg(\psi \text{ U } \phi) = \neg\psi \text{ R } \neg\phi$)).

Chapter 5

Verification of LTL properties of computing systems

In this section we provide an overview on the verification of properties of computing systems. The inherent complexity of sequential and specially concurrent computing systems makes it unfeasible to deal with them directly. Therefore, such systems are generally modeled using abstractions that build much simpler representations, while keeping the particular aspects of interest of the system.

As introduced in Section 3.3, transition systems and Büchi automata can model complex systems. These provide a useful abstraction, which allows reasoning about the systems they model by only focusing in the parts of interest. Therefore, given a computing system represented by either a Büchi automaton or a transition system, and an LTL formula, it is possible to adapt the model checking method for finite-state systems described in Section 1.3 to infinite-state systems.

Recall that model checking methods determine whether a system model M satisfies a given LTL property φ of the system. If $M \models \varphi$, the model checking algorithm returns a positive answer. Otherwise, a system execution path that violates φ is returned as a counterexample.

In order to determine whether $M \models \varphi$, it is necessary to ensure that all the executions of M satisfy φ. As previously stated, M will be either a Büchi automaton or a transition system, so it will define a language $L(M)$ with the executions of the system it accepts. Moreover, φ can be represented with a Büchi automaton, so it will also define a language $L(\varphi)$ with the executions of the system that have a correct behavior.

Hence, verifying that M satisfies φ is reduced to checking whether:

$$L(M) \subseteq L(\varphi)$$

To this end, another approach can be followed in order to check the language inclusion. Such approach is based on the use of $L(\neg\varphi)$ rather than $L(\varphi)$, and takes advantage of the fact that:

$$L(M) \subseteq L(\varphi) \leftrightarrow L(M) \cap L(\neg\varphi) = \emptyset$$

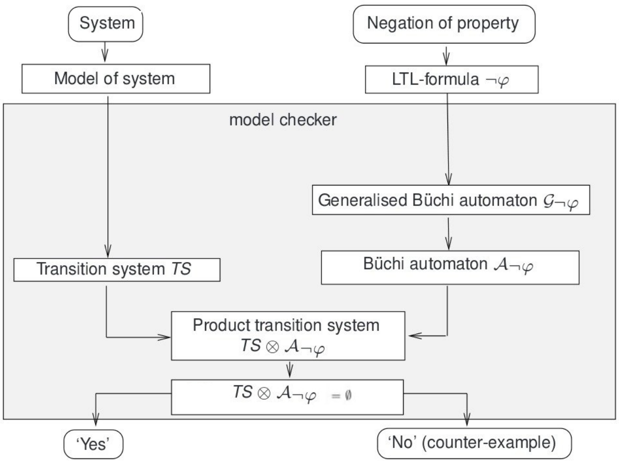

System
Negation of property
Model of system
LTL-formula $\neg\varphi$
model checker
Generalised Büchi automaton $\mathcal{G}_{\neg\varphi}$
Transition system *TS*
Büchi automaton $\mathcal{A}_{\neg\varphi}$
Product transition system
$TS \otimes \mathcal{A}_{\neg\varphi}$
$TS \otimes \mathcal{A}_{\neg\varphi} = \emptyset$
'Yes'
'No' (counter-example)

Figure 5.1: Model checking outline

Hence, model checking of φ on M can be performed by first negating φ, then building a Büchi automaton $B_{\neg\varphi}$ from the latter and finally carrying out the intersection between M and $B_{\neg\varphi}$ and checking the emptiness of language defined by the result of the intersection. Figure 5.1 shows a diagram with the steps of the model checking process. Note that the output of the model checker will be either a 'Yes' or a 'No' along with a counterexample. Such counterexample will be helpful when estimating the possible causes of the violation of the property checked.

For infinite-state systems however, there is a variation in the last step of the process, since the described emptiness test cannot be applied. Given that infinite-state systems do not provide an explicit representation of the system, the emptiness test on the intersection result would not give correct results, since concatenations of several transitions might not be feasible. Thus, other methods need to be used. Further information will be provided in the following sections, which describe the steps followed to carry out the verification of properties of infinite-state computing systems.

5.1 Büchi Automaton intersection

The natural way to intersect two automatons A_1 and A_2 is to construct an automaton A whose state space is the cross product of the state spaces of A_1 and A_2, and let both automatons process the input simultaneously in order to create the transitions of A. For finite words, the input is accepted if each copy can generate a run which reaches an accepting state at the end of the word.

For infinite words it is not that straight forward, since the acceptance condition defined over Büchi automata requires visiting accepting states infinitely often. Therefore, there is no guarantee that runs on A_1 and A_2 will ever visit accepting states simultaneously, as seen in [35].

Example 1. Consider automatons A_1 and A_2 representing languages $L_1 = (a + ba)^w$ and $L_2 = (a^*ba)^w$, respectively. Figure 5.2 shows such automatons and Figure 5.3 shows its intersection as regular automata. As can be seen, state 11 is never reached, which leads to an incorrect behavior, since states 1 are infinitely often visited in A_1 and A_2, and A should reflect that.

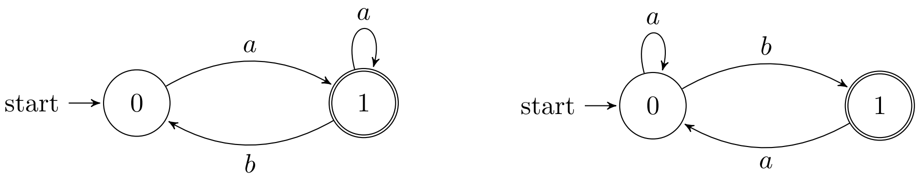


Figure 5.2: Büchi automaton A_1 (left) and A_2 (right)

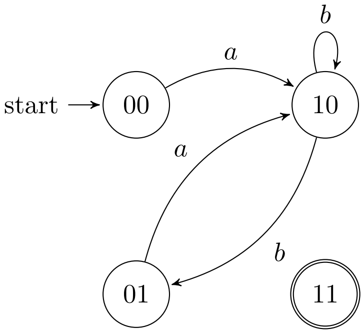


Figure 5.3: Büchi automaton A representing $A_1 \cap A_2$ (regular intersection)

Therefore, in order to obtain an automaton that accepts runs accepted by both A_1 and A_2, it is necessary to follow a different approach. Such approach will start by letting A_1 process the input until it reaches an accepting state. After that, it will focus the execution on A_2 until it reaches an accepting state, and if it does, it will change the focus to A_1 again, and so on.
So, in order to visit infinitely often an accepting state in A_2, it must visit infinitely often some accepting state also in A_1 and viceversa. This will make sure that the resulting automaton accepts infinite runs accepted by both A_1 and A_2.

Formally, let $A_1 = < Q_1,\Sigma,\delta_1,I_1,F_1 >$ and $A_2 = < Q_2,\Sigma,\delta_2,I_2,F_2 >$, then $A = A_1 \cap A_2 = < Q,\Sigma,\delta,F >$, where:

$$Q = Q_1 \times Q_2 \times \{1, 2\}$$

$$I = I_1 \times I_2 \times \{1, 2\}$$

$$F = F_1 \times Q_2 \times \{1\}$$

Given states s_1, s_1' from A_1 and states s_2, s_2' from A_2, transitions in A are defined as follows:

$< s_1, s_2, 1 > \xrightarrow{a} < s_1', s_2', 1 >$ iff $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$ and $s_1 \not\epsilon F_1$
$< s_1, s_2, 1 > \xrightarrow{a} < s_1', s_2', 2 >$ iff $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$ and $s_1 \epsilon F_1$
$< s_1, s_2, 2 > \xrightarrow{a} < s_1', s_2', 2 >$ iff $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$ and $s_1 \not\epsilon F_2$
$< s_1, s_2, 2 > \xrightarrow{a} < s_1', s_2', 1 >$ iff $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$ and $s_1 \epsilon F_2$

Example 2. Following Example 1, the correct Büchi automaton representing the intersection between A_1 and A_2 is shown in Figure 5.4. Unreachable states are left for better understanding of the result.

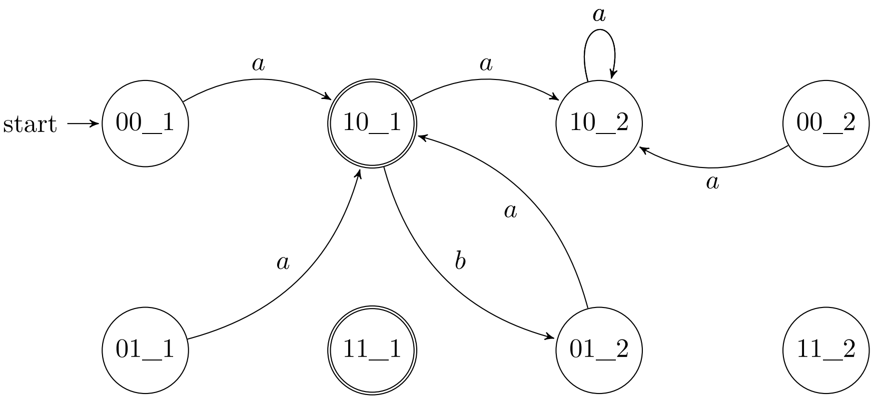


Figure 5.4: Büchi automaton A representing $A_1 \cap A_2$

Once the intersection has been defined for arbitrary automatons, we introduce the special case of either $F_1 = Q_1$ or $F_2 = Q_2$ (i.e. one of the automatons only has accepting states). In this case, we only need to make sure that the resulting automaton accepts infinite runs accepted by the automaton which has some non-accepting states.

Intuitively, if an automaton only has accepting states, it will accept any infinite run. Therefore, when intersecting such automaton with an arbitrary one, the former will restrict the acceptance condition of the resulting automaton.
Thus, if $F_2 = Q_2$ for example, we will have:

$$Q = Q_1 \times Q_2$$

$$I = I_1 \times I_2$$

$$F = F_1 \times Q_2$$

And transitions will be defined such that given states s_1, s'_1 from A_1 and states s_2, s'_2 from A_2:

$$< s_1, s_2 > \xrightarrow{a} < s'_1, s'_2 > \text{ iff } s_1 \xrightarrow{a} s'_1 \text{ and } s_2 \xrightarrow{a} s'_2$$

Example 3. Following Example 1, consider A'_2 is A_2 with only accepting states. Figure 5.5 shows how the intersection between A_1 and A'_2 would be in that case.

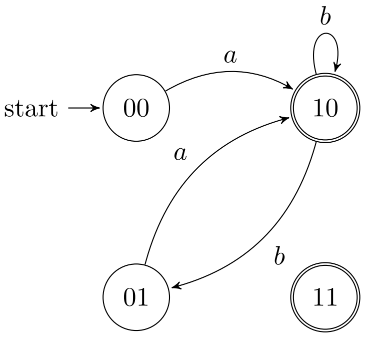


Figure 5.5: Büchi automaton A representing $A_1 \cap A'_2$

For the intersection of a transition system with a Büchi automaton, the same approach is followed, but transitions of the final model will have the assignments found in the transitions of the transition system that were combined with those of the Büchi automaton. Therefore, the final result will be neither a Büchi automaton nor a transition system, but a combination of both.

Example 4. Consider the computing system represented by the program shown below, and the LTL property $\varphi = (j > i)$ U $(j = i)$, which we want to verify on the system. The negation of the LTL property is $\neg\varphi = (\neg(j > i)$ R $\neg(j = i))$. Figure 5.6 shows the Büchi automaton for $\neg\varphi$, and Figures 5.7 and 5.8 show the transition systems for the program and the intersection between such transition system and the Büchi automaton, respectively. The results shown were obtained with the use of our tools.

```
int main() {
  int a;
  int n,m;
  assume(n<=m);
  int j=m;
  int i=0;
  ltl_property((j > i) U (j = i));
  while (j>=0){
    j--;
    i++;
  }
}
```

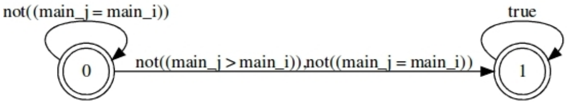


Figure 5.6: Computing system (left) and Büchi automaton for the formula $(j > i)$ U $(j = i)$ (right)

As can be seen in Figure 5.8, the resulting transition system has transitions holding both conditions and assignments, and some of its states are accepting. This transition system represents all the executions of the program that satisfy $\neg\varphi$, if any.

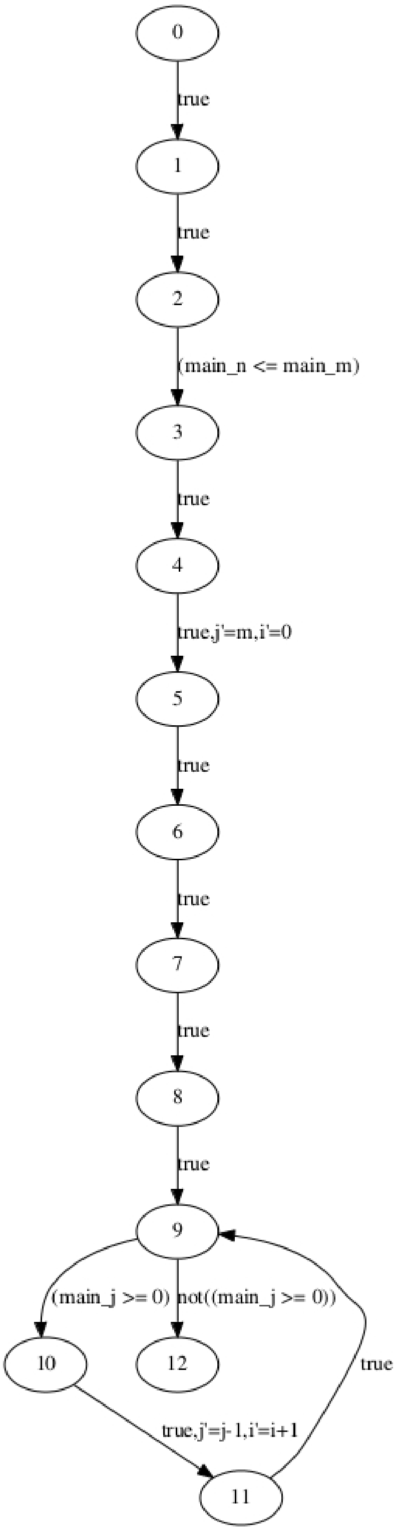


Figure 5.7: Transition system for the program

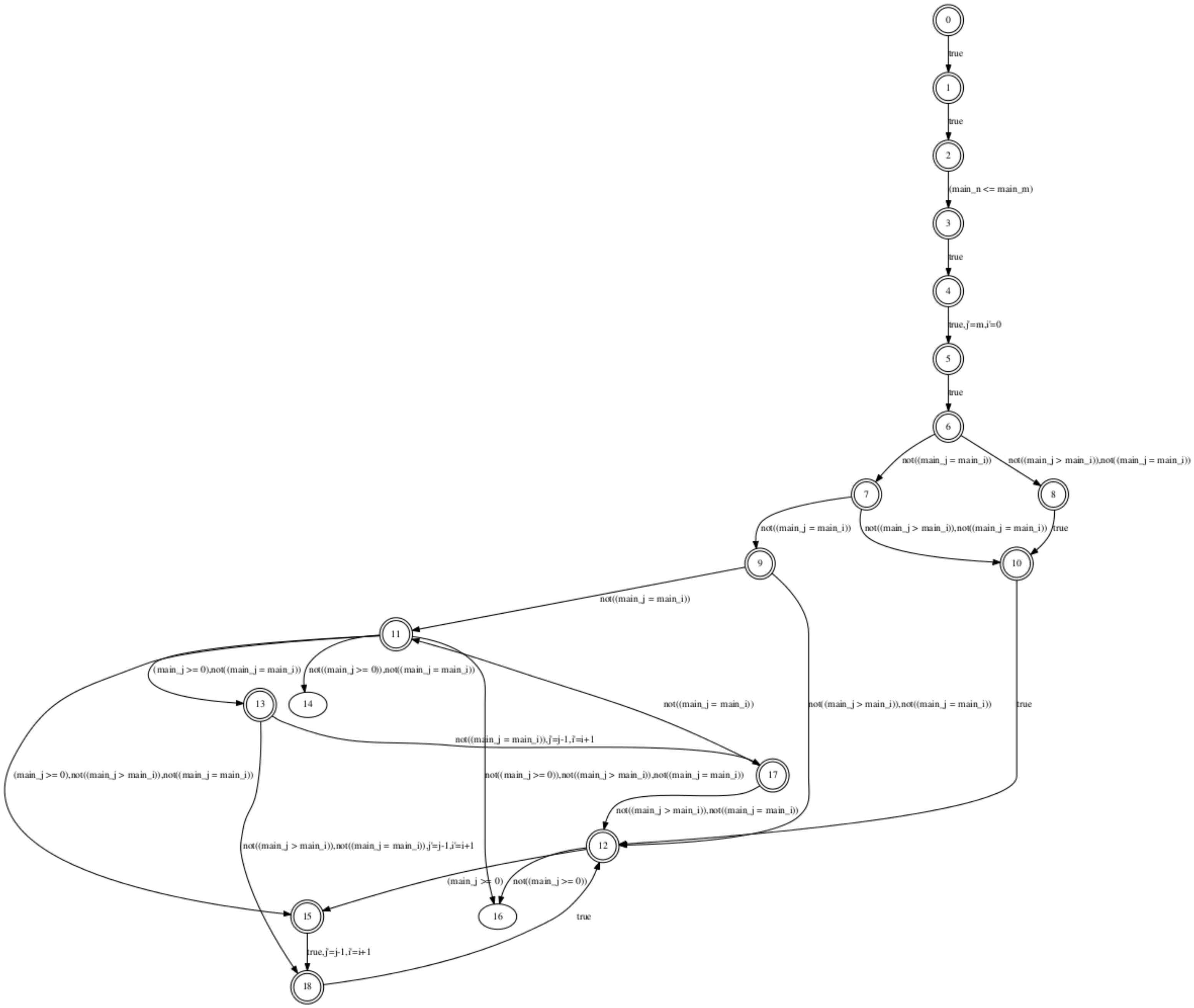

0
true
1
true
2
(main_n <= main_m)
3
true
4
true,j'=m,i'=0
5
true
6
not((main_j = main_i))
not((main_j > main_i)),not((main_j = main_i))
7
8
not((main_j = main_i))
not((main_j > main_i)),not((main_j = main_i))
true
9
10
not((main_j = main_i))
11
(main_j >= 0),not((main_j = main_i))
not((main_j >= 0)),not((main_j = main_i))
13
14
not((main_j = main_i))
not((main_j > main_i)),not((main_j = main_i))
true
not((main_j = main_i)),j'=j-1,i'=i+1
(main_j >= 0),not((main_j > main_i)),not((main_j = main_i))
not((main_j >= 0)),not((main_j > main_i)),not((main_j = main_i))
17
not((main_j > main_i)),not((main_j = main_i))
not((main_j > main_i)),not((main_j = main_i)),j'=j-1,i'=i+1
12
(main_j >= 0)
not((main_j >= 0))
15
16
true
true,j'=j-1,i'=i+1
18

Figure 5.8: Intersection between transition system and Büchi automaton

Finally, once the intersection is performed, it will be necessary to check whether the result determines that the system verifies the property or not.

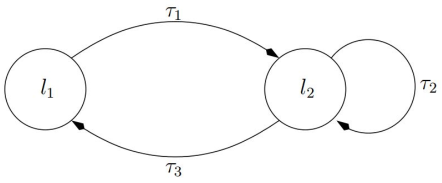
5.2 Verification

We aim to verify LTL properties on infinite state systems, hence the methods that we will use do not rely on an explicit representation of the state space. Rather than that, we will exploit the expressive power transition systems.

The use of such representations implies that methods previously described for the emptiness check (see Section 4.3) cannot be applied, since they could lead to unfeasible accepting traces.

Example 1. Consider the transition system in Figure 5.9 obtained from the program shown. If we considered location l_1 as an accepting state, the Nested DFS algorithm would determine that there always exists an infinite execution that can go through l_1 infinitely often (i.e. an accepting trace). Nevertheless, it can be seen that for instance, with values $x < 0$ and $y \geq 1$, there are no infinite traces and, in fact, this transition system does not have any infinite run.

```
int main()
{
  int x=undet(),y=undet(),z=undet();
  l1: while (y>=1) {
       x--;
  l2:  while (y<z) {
         x++; z--;
       }
       y=x+y;
     }
}
```



$$\begin{array}{llllll} \rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\ \rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\ \rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z \end{array}$$

Figure 5.9: Transition System example

Therefore, instead of performing the emptiness test, one possible adaptation for infinite-state systems could be to use methods based on the notion of program termination [23, 36], which will allow us to find feasible infinite traces in the model. However, the application of such methods is out of the scope of this project, since we aimed to provide a result from which a program analysis tool such as $VeryMax$ could yield a final result .

Chapter 6

Conclusions

We have developed tools which allow us to check LTL satisfiability of formulas coming from logic programs. Moreover, our tools also provide models which can be fed to program analysis methods in order to check the satisfiability of LTL properties on infinite-state systems.
Therefore, our implementation provides models for LTL formulas in the shape of Büchi automata, and allows the combination of such automatons with models representing computing systems.

The satisfiability check for logic programs is performed by analyzing the Büchi automatons obtained from the formulas representing them. This method yields good results when dealing with small formulas, but becomes impractical as the sizes of the formulas increase.
The same applies to the Büchi automatons obtained from the combination of a system model and the Büchi automaton representing an LTL property. Since combining the models means performing an automaton intersection, the size of the result dramatically increases as the models grow.

Hence, our tools efficiently generate models for many LTL formulas used in practice. However, further research should be made in order to find methods that optimize the generation of Büchi automata, thereby making some larger LTL formulas treatable.

Chapter 7

Future work

As the first step, we plan to test the performance of our tools on more examples coming from [20] and compare the behaviour of our solver with the one developed by them, which is based on tableaux, a completely different technique.

On a second step, another topic for future research is the application of program analysis methods on the results obtained from our tools. These techniques will have to be adapted to the models we provide, namely Büchi Automata. Hence, further work needs to be done in order to analyze the Büchi automata and provide concluding results on the satisfiability of LTL properties on computing systems.

Furthermore, given that the size of the generated Büchi automata can be exponential in the size of the formula they represent, further research could be carried out in order to find methods to optimize the size of the Büchi automata created.

Bibliography

[1] J. A. Abraham. Introduction to formal to verification. Lecture notes. URL http://www.cerc.utexas.edu/~jaa/360r/lectures/22-1.pdf. Last visited 2015-03-19.

[2] Mordechai Ben-Ari. A primer on model checking. In *ACM Inroads*, pages 40–47, March 2010.

[3] Christel Baier and Joost Pieter Katoen. *Principles of Model Checking*. The MIT Press, April 2008.

[4] E A Emerson and E M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266, 1982.

[5] J.P. Quielle and J. Sifakis. S. Specification and verification of concurrent systems in cesar. *Lecture Notes in Computer Science*, 137, 1981.

[6] Edmund M. Clarke. Symbolic model checking with bdds. Lecture notes. URL http://www.cs.cmu.edu/~emc/15-820A/reading/lecture_1.pdf. Last visited 2015-03-19.

[7] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, page 2001. Kluwer Academic Publishers, 2001.

[8] Automata theory. Lecture notes. URL https://www.tu-braunschweig.de/Medien-DB/isf/sse/08_temporalmc_vl.pdf. Last visited 2015-03-19.

[9] Stéphane Demri and Paul Gastin. Specification and verification using temporal logics, 2009.

[10] Ina Schaefer. Model checking with temporal logic. Lecture notes, . URL https://www.tu-braunschweig.de/Medien-DB/isf/sse/08_temporalmc_vl.pdf. Last visited 2015-03-14.

[11] M Vardi and P Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*, 1986.

[12] Alexandre Duret-lutz and Denis Poitrenaud. SPOT: An Extensible Model Checking Library Using Transition-Based Generalized Büchi Automata. In *Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 76–83, 2004. doi: 10.1109/MASCOT.2004.1348184.

[13] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of ltl formulae to büchi automata. In *In Proc. FORTE'02., volume 2529 of LNCS*, pages 308–326. Springer, 2002.

[14] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In *FORMAL METHODS IN SYSTEM DESIGN*, pages 415–427. Springer-Verlag, 1994.

[15] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000. ISSN 1433-2779. doi: 10.1007/s100090050046. URL http://dx.doi.org/10.1007/s100090050046.

[16] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center. URL http://www.csl.sri.com/papers/lfm2000/.

[17] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques, 1998.

[18] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. *SIGPLAN Not.*, 42(1): 265–276, January 2007. ISSN 0362-1340. doi: 10.1145/1190215.1190257. URL http://doi.acm.org/10.1145/1190215.1190257.

[19] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to ltl software model checking.

[20] Marco Comini, Laura Titolo, and Alicia Villanueva. Abstract diagnosis for tccp using a linear temporal logic. *Theory and Practice of Logic Programming*, 14:787–801, 7 2014. ISSN 1475-3081. doi: 10.1017/S1471068414000349. URL http://journals.cambridge.org/article_S1471068414000349.

[21] Albert Oliveras Enric Rodríguez-Carbonell Marc Brockschmidt, Daniel Larraz and Albert Rubio. Compositional safety verification with max-smt. URL http://www.cs.upc.edu/~albert/VeryMax.html. Submitted.

[22] Keijo Ruohonen. Formal languages. Lecture notes. URL http://math.tut.fi/~ruohonen/FL.pdf. Last visited 2015-06-21.

[23] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-smt. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 779–796, 2014. doi: 10.1007/978-3-319-08867-9_52. URL http://dx.doi.org/10.1007/978-3-319-08867-9_52.

[24] Kristin Y. Rozier and Moshe Y. Vardi. Ltl satisfiability checking. In *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pages 149–167, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73369-0. URL http://dl.acm.org/citation.cfm?id=1770532.1770548.

[25] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proceedings of 24 IEEE symposium on foundation of computer science*, pages 185–194, 1983.

[26] Marco Daniele, Fausto Giunchiglia, and MosheY. Vardi. Improved automata generation for linear temporal logic. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66202-0. doi: 10.1007/3-540-48683-6_23. URL http://dx.doi.org/10.1007/3-540-48683-6_23.

[27] Dimitra Giannakopoulou and Flavio Lerda. Efficient translation of ltl formulae into büchi automata, 2001.

[28] Pierre Parutto. Improving degeneralization in spot. Technical report, 2011. URL https://www.lrde.epita.fr/dload/20110704-Seminar/parutto1111_degeneralization_report.pdf. Last visited 2015-06-21.

[29] Stefan Schwoon and Javier Esparza. Comparison of algorithms for checking emptiness on büchi automata. In Petr Hliněný, Václav Matyáš, and Tomáš Vojnar, editors, *Proceedings of the 5th Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Znojmo, Czech Republic, November 2009.

[30] Andreas Gaiser and Stefan Schwoon. A note on on-the-fly verification algorithms. In *Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[31] Wikipedia. DFS. Depth first search. URL http://en.wikipedia.org/wiki/Depth-first_search. Last visited 2015-03-14.

[32] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proceedings of the 2nd Spin Workshop, Rutgers University, New Jersey, USA*, 1996.

[33] Albert Oliveras Enric Rodrıguez-Carbonell Albert Rubio, Daniel Larraz. Program analysis using smt and max-smt. In LOPSTR, 2013. URL https://www.cs.upc.edu/~albert/papers/LOPSTR-Slides.pdf. Last visited 2015-06-21.

[34] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic smt solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70543-7. doi: 10.1007/978-3-540-70545-1_27. URL http://dx.doi.org/10.1007/978-3-540-70545-1_27.

[35] Ina Schaefer. Formal modeling with linear temporal logic. Lecture notes, . URL https://www.tu-braunschweig.de/Medien-DB/isf/sse/propositionalandtemporallogic_vl.pdf. Last visited 2015-06-21.

[36] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using max-smt. In *Proc. FMCAD*, 2013.