



Université
catholique
de Louvain

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN



Implementing Temporal Logic Queries with PyNuSMV

Supervisor: Charles PECHEUR

Readers: Simon BUSARD

Kim MENS

Thesis submitted for the Master's degree

in Computer Science (120 credits)

option Software Engineering and

Programming Systems

by Simon THIBERT

Louvain-la-Neuve
Academic year 2014-2015

Abstract

In recent years, the use of efficient methods that ensure reliable hardware and software systems has been critical, and it will become even more in the future. Indeed, we interact with more and more computing devices in our daily lives, our confidence in these is therefore essential. Model checking is a popular technique to achieve this goal of reliability in hardware and software systems. It is used to verify, in an automatic way, that a given system satisfies or not given properties. This technique is successfully applied in practice on various systems to verify their correctness. However, model checking does not appear to be effective in the process of understanding systems behaviors when their design is opaque. Indeed, while such a process is still possible with model checking, it takes the form of an inefficient trial-and-error approach.

Temporal logic query solving was therefore proposed by William Chan at CAV 2000 to fill this gap. This technique is an extension of model checking whose main aim is to understand a model as opposed to merely verifying its correctness. A temporal logic query basically consists of a temporal logic formula where some subformulas are replaced by the special symbol $?$ representing a “hole” in the formula. The query solving problem then consists of finding the right subformula to fill the hole(s) and make the initial formula satisfied in the considered system. His researches have been well-received in the model checking community, and important developments have been initiated by his paper. Among them, Samer and Veith systematically investigated temporal logic queries, and corrected and extended Chan’s work.

For the time being, no public implementation of temporal logic queries – as defined by Chan and Samer and Veith – is known. This thesis therefore proposes a first implementation of such queries, in the form of a new Python package, called PyTLQ. In order to fit the needs of this practical implementation in Python, we present adaptations to Chan’s and Samer and Veith’s algorithms, and we use PyNuSMV – a Python library that gives access to the rich BDD-based functionalities of the well-known model checker NuSMV. Initial experiments demonstrate the applicability of PyTLQ with concrete systems, and show how it can be used to better apprehend systems design and correct potential faulty behaviors.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Charles Pecheur, for the support on the way, for his remarks, wise advices, and expertise, providing me all the necessary explanations in regard to the theoretical aspect of the thesis.

Besides my supervisor, I would like to express all my gratitude to Simon Busard, for his insightful comments and clarifications, but also for his availability at all times, and valuable help through the development of PyTLQ.

I thank Professor Kim Mens for his early advices and his time as second reader of this thesis.

My sincere thanks also goes to my cousin, Thomas Lecocq, for passing on his passion for IT to me, and for his valuable advices on the home stretch.

I take this opportunity to thank all my friends for all the fun we had since we know each other, and particularly for the last five years in Louvain-la-Neuve.

Last but not least, I would like to thank my loved ones for supporting me throughout writing this thesis and in my life in general.

Contents

Abstract	i
Acknowledgements	iii
Contents	v
1 Introduction	1
1.1 Model Checking	2
1.2 Temporal Logic Query Solving	3
1.3 Related Work	4
1.4 A Simple Demonstration Example	5
1.5 Contributions	7
1.6 Overview	7
2 Background	9
2.1 Foundations	9
2.1.1 Systems and Properties	9
2.1.2 Binary Decision Diagrams	13
2.2 Temporal Logic Queries	16
2.2.1 Fragment CTLQ^x	17
2.2.2 The Extended Chan Algorithm	19
2.2.3 Chan’s Simplification Algorithm	23
2.3 PyNuSMV	25
2.3.1 Origins	25
2.3.2 Structure	26
3 PyTLQ	29
3.1 System Architecture	29
3.2 System Functionalities	31
3.2.1 Parsing CTL Queries	31
3.2.2 Checking the Membership to Fragment CTLQ^x	32
3.2.3 Solving CTL Queries	34
3.2.4 Simplifying Solutions	35
3.3 Implementation Details	37
3.3.1 Abstract Syntax Trees	37
3.3.2 Standalone Script	38
3.4 Limitations	38

4	Applications	41
4.1	A Simple Print Server	42
4.2	A Cache Consistency Protocol	43
5	Conclusion and Perspectives	47
	Bibliography	49
A	User Manual	55
A.1	Input Language	55
A.2	Installation	56
A.3	Usage	56
A.4	Application Programming Interface	57
A.4.1	The <code>ast</code> module	57
A.4.2	The <code>parser</code> module	57
A.4.3	The <code>checker</code> module	58
A.4.4	The <code>solver</code> module	59
A.4.5	The <code>simplifier</code> module	59
A.4.6	The <code>exception</code> module	60
A.4.7	The <code>utils</code> module	60
B	Source Code Metrics	63
B.1	Coding Standard	63
B.2	Raw Metrics	64
B.3	Cyclomatic Complexity	64
B.4	Maintainability Index	67
B.5	Tests Coverage	68

Chapter 1

Introduction

Today, hardware and software systems are found in a myriad of applications where failure is unacceptable such as banking industry, air traffic control systems, embedded systems in automobiles, or even medical instruments. Nevertheless, over the past decades, numerous incidents involved failures in such systems, and showed the severe consequences they can provoke (for example, the explosion of the Ariane 5 rocket just after lift-off on June 4, 1996 [25], or the lives lost because of the Therac-25 malfunction in the late 1980s [38]). These also proved that hardware and software errors can be expensive. Actually, a study commissioned by the National Institute of Standards and Technology in 2002 [55] found that the costs of software errors amount to an estimated \$59.5 billion to the economy of the United States of America annually. The study also revealed that more than a third of these costs, or an estimated \$22.2 billion, could be eliminated by improving the testing process of software systems.

As a result, the use of efficient methods that ensure reliable hardware and software systems has been critical for many years now, and this will certainly increase in the future with the growth of the Internet and the omnipresence of technology in everyday life. Indeed, one will become even more dependent on the proper functioning of computing devices, and it will therefore become even more important to use methods that increase our confidence in the correctness of those systems. Although there exist several validation methods to achieve this goal of reliability in hardware and software systems such as simulation (that is, experiments on an abstraction of the system), testing (that is, experiments on the actual system), and deductive verification (that is, a manual proof using axioms and proof rules), this thesis focuses on model checking, the cradle in which temporal logic queries were born.

Actually, temporal logic query solving tackles the Achilles' heel of model checking, namely its use for understanding systems behaviors. Indeed, despite being particularly popular and efficient as validation technique, model checking does not appear to be effective for comprehension purposes. This observation has been first pointed out by Chan in 2000 [13], when he realized that model checking was not only used to verify properties, but also to better understand a model when its design is unclear.

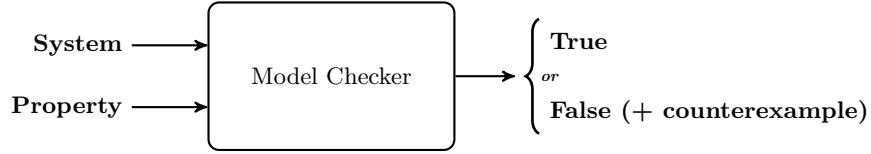


Figure 1.1. Simplified overview of a model checker.

Back then, the process of model understanding basically consisted of the identification of a few key properties, followed by an iteration over them with model checking in order to validate hypotheses and develop a more detailed set of properties that the model satisfies or should satisfy. To speed up this process and avoid the trial-and-error method, Chan introduced temporal logic queries, which were later corrected and extended by Samer and Veith [51, 53].

The goal of this thesis is to implement a temporal logic query solver by following Chan’s ideas, as corrected and extended by Samer and Veith. To achieve this objective, we use PyNuSMV [11], a Python framework for implementing model-checking algorithms developed in the Louvain Verification Lab of the Université Catholique de Louvain.

1.1 Model Checking

Model checking is a well-known technique used to verify, in an automatic way, that a given system (modeled as a Kripke structure) satisfies or not a given property (modeled as a temporal logic formula). The answer of model checking is “True” or “False”, followed by a counterexample in the latter case (see Figure 1.1). This technique appeared at the beginning of the 1980s through the independent works of Clarke and Emerson [21] and Queille and Sifakis [46]. It has several important advantages over simulation, testing, and deductive verification: it is fully automatic, usually quite fast, and it produces a counterexample if a violation of the property is found in the model.

Model checking has been successfully applied in practice to verify complex sequential circuits and network protocols [19, 22]. However, the method faces a main challenge: the *state space explosion problem*. Succinctly, this problem appears when the number of state variables in the system increases, which causes an exponential growth of the size of the system state space, and limits the size of the systems that can be verified. Let us take an example (adapted from [64]) to illustrate this. Imagine a simple system with only one boolean variable, a . This system has two possible states: $a = \text{true}$ or $a = \text{false}$. Adding another boolean variable, b , will give the system four possible states: $a = \text{true}$ and $b = \text{true}$, $a = \text{true}$ and $b = \text{false}$, $a = \text{false}$ and $b = \text{true}$, $a = \text{false}$ and $b = \text{false}$. With three boolean variables, the number of possible states increases up to eight, and so on. We can easily infer that a system with n boolean

variables has 2^n possible states, and a system that allows variables with K possible values (as opposed to boolean variables that have only two possible values) has K^n possible states.

In 1987, McMillan [42] came up with the idea of *symbolic model checking* to attack the state space explosion problem. Symbolic model checking basically refers to a much more efficient traversal of the state space by considering large numbers of states at each step. Following Bryant’s work [7], McMillan proposed to symbolically represent the states sets and transition relations as boolean functions (instead of manipulating states and transitions explicitly), which are implemented with binary decision diagrams (BDDs). This improvement dramatically increased the size of the systems that could be verified by model checking. Practically, model checking moved from manipulating systems with at most 10^8 reachable states to systems with more than 10^{20} states [9]. Later, improvements of the BDD-based techniques further increased the state count up to more than 10^{120} [10].

Note that other approaches exist to cope with the state space explosion problem such as partial-order reduction [29], bounded model checking [5], and counterexample-guided abstraction refinement [20]. Yet, this thesis concentrates on symbolic model checking and BDDs. For simplicity, we write “model checking” to denote symbolic model checking in the following.

1.2 Temporal Logic Query Solving

Temporal logic query solving is an extension of model checking whose principal aim is to understand a model as opposed to merely verifying its correctness. In practical terms, a temporal logic query is defined as a temporal logic formula containing the special symbol $?$, called *placeholder*, that represents a “hole” in the formula. The idea of temporal logic query solving is to give the propositional formula (that is, a formula built only from atomic propositions and boolean operators) that, when replacing the placeholder(s) in the temporal logic query, makes the initial formula satisfied by the system. In other words, a temporal logic query solver would take a system and a temporal logic query as inputs, and return a propositional formula as output (see Figure 1.2). Then, if we substitute the placeholder of the temporal logic query by the propositional formula, we obtain a property that is satisfied in the given system (that is, if we launch a model checker like the one depicted in Figure 1.1 with this newly built property, the output will be “True”).

In his paper [13], Chan defines temporal logic queries as Computation Tree Logic (CTL) specifications, called “CTL queries”. Furthermore, he focuses on CTL queries having *exactly one* strongest solution (that is, a single solution that summarizes all solutions), called “valid” queries. (Note that a CTL query can have, in general, many

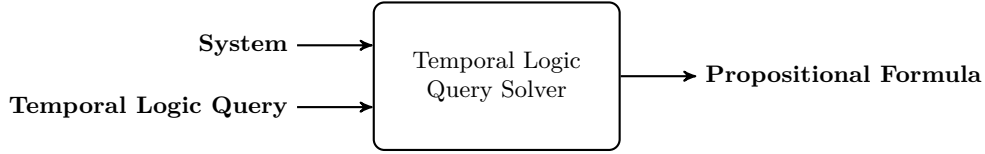


Figure 1.2. Simplified overview of a temporal logic query solver.

strongest solutions.) Unfortunately, identifying valid queries is EXPTIME-complete¹ [13]. Consequently, in order to avoid coping with this complexity, Chan defines a syntactic class of queries that are guaranteed to be valid, CTL^v . He also proposes an efficient symbolic BDD-based algorithm for solving CTL queries that belong to CTL^v , with the same complexity as CTL model checking (that is, a complexity linear in the size of the model and the length of the query).

For their part, Samer and Veith systematically investigate Chan’s work and valid queries. The starting point of their work is the correction of Chan’s syntactic class after discovering a counterexample to his claim, namely the existence of a unique strongest solution for CTL queries in CTL^v . In their first paper about temporal logic queries [51], Samer and Veith define two new syntactic classes, CTL_{new}^v and CTL^d , the former being the correction of Chan’s syntactic class, and the latter being an extension to queries that have *at most one* strongest solution, called “exact” queries. In their second paper about CTL queries [53], they investigate deterministic CTL query solving (that is, the CTL query solving “reduced in a deterministic manner to solving subqueries at appropriate system states” [53]). They define a new fragment for this purpose, CTLQ^x , and propose an extension of Chan’s algorithm for solving CTL queries that belong to fragment CTLQ^x . Following this approach, Chan’s work becomes a special case of Samer and Veith’s work.

Our work in this thesis is based on Samer and Veith’s paper about deterministic CTL query solving [53] since it corrects and extends Chan’s paper [13]. In other words, when details about formalisms differ between Samer and Veith’s and Chan’s papers, Samer and Veith’s version is preferred.

1.3 Related Work

Chan’s work has been well received in the model checking community. In addition to Samer and Veith’s contributions, most important developments initiated by Chan’s paper are carried out by Bruns and Godefroid [6], Hornus and Schnoebelen [34], Gurfinkel, Chechik and Devereux [14, 30, 31, 32], and Gheorghiu, Gurfinkel and Chechik [27, 28]. However, their main focus is on generalizing Chan’s work in the

¹ The complexity class EXPTIME is the set of all decision problems solvable in exponential time by a deterministic Turing machine. EXPTIME-complete decision problems denote the hardest or most expressive problems in EXPTIME. [65]

sense that they investigate temporal logic queries in a broader framework, without restricting themselves to valid queries.

Bruns and Godefroid are interested in all CTL queries, even those that have multiple strongest solutions. Furthermore, they do not restrict their attention to CTL, their automata-theoretic approach – inspired by Kupferman, Vardi and Wolper’s paper [35] – for tackling the query solving problem is defined for an arbitrary temporal logic.

Hornus and Schnoebelen study the problem of computing the set of all strongest solutions to arbitrary temporal logic queries. In particular, they show that computing a strongest solution and determining if this solution is unique over a given system can be solved in polynomial-time.

Gurfinkel et al. propose a multi-valued model checking approach [15] to solve temporal logic queries with multiple *distinct* placeholders. Following this approach, they implement a temporal logic query solver, TLQSolver, on top of their existing multi-valued model checker XChex [16].

Gheorghiu et al. provide a symbolic algorithm on top of the model checker NuSMV [18] for finding states solutions to any CTL queries. Their approach – based on the framework described by Gurfinkel et al. [32] – generalizes previous specialized techniques such as computing procedure summaries [3, 47] and finding dominators and postdominators in program analysis [2].

1.4 A Simple Demonstration Example

To illustrate CTL query solving, we use Clarke, Grumberg and Peled’s microwave oven example [22] as running case throughout this thesis.

The microwave oven functioning is informally specified as follows:

“To cook food in the oven, open the door, put the food inside, and close the door. Do not put metal containers in the oven. Press the start button. The oven will warmup for 30 seconds, and then it will start cooking. When the cooking is done, the oven will stop. The oven will stop also whenever the door is opened during cooking. If the oven is started while the door is open, an error will occur, and the oven will not heat. In such a case, the reset button may be used.” [43]

Figure 1.3 depicts this model as a Kripke structure. The Kripke structure consists of seven states, each of which being labeled with both the atomic propositions that are true in the state and the negations of the atomic propositions that are false in the state. The initial state is s_0 . For clarity, the arcs are also labeled to indicate

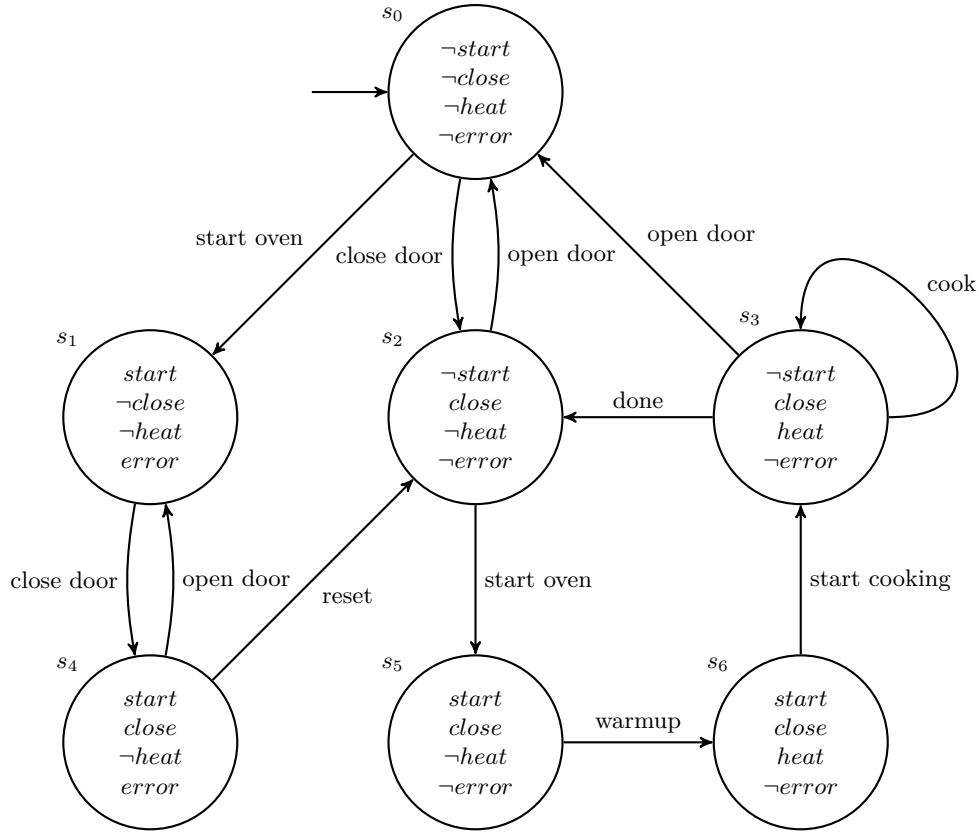


Figure 1.3. Microwave oven example, taken from [22].

the actions that cause transitions between states, but these labels are not part of the Kripke structure. [22]

In addition to the Kripke structure, consider these three CTL formulas denoting simple requirement properties (taken from [43]):

1. $AG (start \rightarrow AF heat)$ which means “Whenever the start button is pushed, eventually the oven will heat.”
2. $AG (heat \rightarrow close)$ which means “If the oven heats, then the door is closed.”
3. $AG (error \rightarrow EF heat)$ which means “Whenever an error occur, it will be still possible to cook.”

(Note that the notions of Kripke structure and CTL formulas will be formally defined in the next chapter.)

Now, imagine that we run a model checker (see Figure 1.1) with the Kripke structure in Figure 1.3 and these three CTL properties as inputs. The output we get is:

```
AG (start -> AF heat) is False
Error trace: s0 > s1 > s4 > s1 > s4 > ...
AG (heat -> close) is True
AG (error -> EF heat) is True
```

Surprisingly, the first property does not hold in the model (suppose we do not know why). This is where temporal logic query solving can come into play. Thanks to this technique, we can discover what makes the oven eventually heat without using a trial-and-error approach. To do that, we express the following CTL query: $\text{AG } (? \rightarrow \text{AF } \textit{heat})$, which asks “What can guarantee that the oven will heat?” We then launch a temporal logic query solver with the Kripke structure in Figure 1.3 and this CTL query as inputs, and get the following output:

```
close & !error & (start | heat)
```

This output means that, to ensure that the oven will eventually heat, the door must be closed, there must be no error, and either the start button is pushed or the oven already heats. We can verify this solution by modifying the first property that did not hold with the new property $\text{AG } ((\textit{close} \wedge \neg \textit{error} \wedge (\textit{start} \vee \textit{heat})) \rightarrow \text{AF } \textit{heat})$, and relaunching the model checker. The answer we get is now:

```
AG ((close & !error & (start | heat)) -> AF heat) is True
```

This simple example exposes *one way* of using temporal logic queries to help users of model checking to better understand systems behaviors. Indeed, we might as well have expressed other queries to increase our knowledge about the model, even if all the given properties hold (for example, the query $\text{AG } (\neg \textit{close} \rightarrow ?)$ would tell us what is guaranteed if the door of the oven is open).

1.5 Contributions

We develop PyTLQ – an original Python package for solving temporal logic queries using PyNuSMV – by following Chan’s ideas, as corrected and extended by Samer and Veith. We adapt their fragment and algorithms to fit the needs of a practical implementation with PyNuSMV, and show how PyTLQ can be used to better understand concrete systems designs.

1.6 Overview

The remainder of this thesis is organized as follows: Chapter 2 summarizes all the theoretical concepts on which this thesis is based, as well as PyNuSMV structure. In Chapter 3, we present the architecture, functionalities and limitations of PyTLQ. Chapter 4 exposes applications of PyTLQ on models representing concrete systems in order to validate our solution. Finally, Chapter 5 concludes this thesis and describes some avenues of future work.

Chapter 2

Background

This chapter presents the context of this thesis, following two main axis: the *substance* (that is, temporal logic queries) and the *form* (that is, PyNuSMV).

Before getting to the heart of the matter, we remind some basics of (symbolic) model checking on which temporal logic query solving relies. Then, the whole theory of Chan – corrected and extended by Samer and Veith – is exposed, as well as the syntactic fragment and algorithms that we use. Finally, the third section presents the PyNuSMV library.

2.1 Foundations

In this section, we remind the foundations on which temporal logic queries were built. We begin by reviewing the modeling of systems and properties in model checking, as well as important concepts and definitions. We then briefly present the binary decision diagrams theory, as it plays an inherent role in symbolic model checking and, by extension, temporal logic query solving.

2.1.1 Systems and Properties

The first step in evaluating temporal logic queries consists of modeling the *system* we want to analyse. As stated by Clarke, Grumberg and Peled [22], model checking is primarily used with reactive systems (that is, “systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination” [39]). A common formalism for the representation of this kind of systems is the *Kripke structure*.

Definition 1 (Kripke Structure, adapted from [13] and [22]). *A Kripke structure $\mathcal{K}$ is a tuple $\mathcal{K} = (Q, Q_0, \Delta, AP, L)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times Q$ is a total transition relation (that is, for every state*

$s \in Q$, there exists a state $s' \in Q$ such that $(s, s') \in \Delta$), AP is a finite set of atomic propositions, and $L : Q \rightarrow 2^{AP}$ is an interpretation (or labeling) function (that is, a function that labels each state with the set of atomic propositions that are true in that state).

Definition 2 (State, from [22]). A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time.

For example, the Kripke structure representing the microwave oven example is composed as follows:

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$
- $Q_0 = \{s_0\}$
- $\Delta = \{(s_0, s_1), (s_0, s_2), (s_1, s_4), (s_2, s_0), (s_2, s_5), (s_3, s_0), (s_3, s_2), (s_3, s_3), (s_4, s_1), (s_4, s_2), (s_5, s_6), (s_6, s_3)\}$
- $AP = \{start, close, heat, error\}$
- $L = \{s_0 \rightarrow \{\}, s_1 \rightarrow \{start, error\}, s_2 \rightarrow \{close\}, s_3 \rightarrow \{close, heat\}, s_4 \rightarrow \{start, close, error\}, s_5 \rightarrow \{start, close\}, s_6 \rightarrow \{start, close, heat\}\}$

On their side, *properties* about a model are conveniently expressed in temporal logics. Temporal logics are well-suited in model checking because they can describe the sequence of events in time without introducing it explicitly in properties. For example, in a temporal logic, we can express statements like “I am *always* tired”, “I will *eventually* be tired”, or “I will be tired *until* I sleep a few hours” [67].

Following Chan, we use the *Computation Tree Logic* (CTL) [21] to specify properties and, by extension, temporal logic queries. CTL is a branching-time logic, meaning that the time is modeled as a tree-like structure (see Figure 2.1a). In other words, there exist multiple possible computation paths (or simply *paths*) from a given state of the model (see Figure 2.1b).

Definition 3 (Path, adapted from [13] and [53]). A path $\pi = (s_0, s_1, s_2, \dots)$ in $\mathcal{K}$ is an infinite sequence of states in which each consecutive pair of states belongs to Δ . Formally, $\pi : \mathbb{N} \rightarrow Q$ such that $(\pi(i), \pi(i+1)) \in \Delta$ for all $i \in \mathbb{N}$.

Practically, *CTL formulas* consist of atomic propositions, boolean operators, path quantifiers, and temporal operators. There are two path quantifiers: A (“for all paths”) and E (“for some path”), and four conventional temporal operators: X (“next”), F (“eventually”), G (“globally”), and U (“until”). Intuitively, $X \varphi$ means that φ is true in the next state of the path, $F \varphi$ means that φ will be true at some state on the path, $G \varphi$ means that φ is true at every state on the path, and $\varphi U \psi$ means that φ remains true until ψ becomes true. Temporal operators must always be immediately preceded by a path quantifier in CTL, which means that there are eight conventional CTL operators in all: AX, EX, AF, EF, AG, EG, AU, and EU. Figure 2.2 graphically summarizes the meaning of each CTL operator.

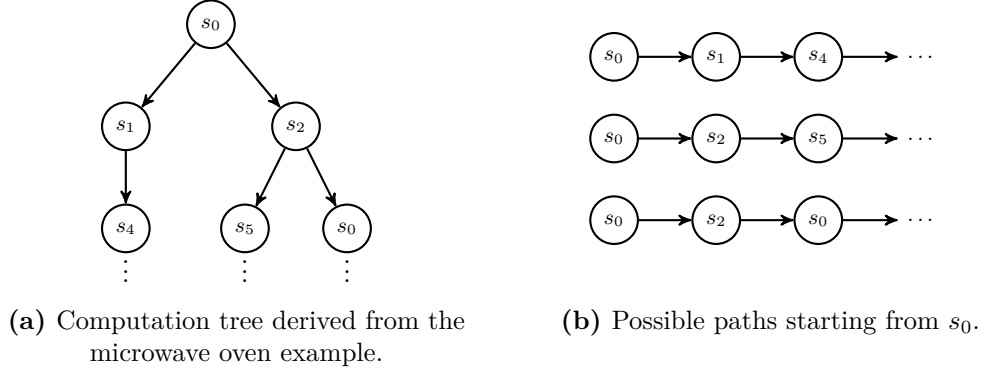


Figure 2.1. Illustration of the model of branching-time logic. The state s_0 is reached before states s_1 and s_5 , but the states s_1 and s_5 are not in any temporal relation.

Definition 4 (CTL Formulas). CTL formulas are formally defined as follows:

- Any atomic proposition $p \in AP$ and *true* are CTL formulas.
- If φ and ψ are CTL formulas, then $\neg\varphi$, $\varphi \vee \psi$, $\text{EX } \varphi$, $\text{EG } \varphi$, and $\text{E}(\varphi \text{ U } \psi)$ are also CTL formulas.

The *false* constant and the other operators can be derived from Definition 4:

- $\text{false} \equiv \neg \text{true}$
- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$
- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\text{EF } \varphi \equiv \text{E}(\text{true} \text{ U } \varphi)$
- $\text{AX } \varphi \equiv \neg \text{EX } \neg\varphi$
- $\text{AG } \varphi \equiv \neg \text{EF } \neg\varphi$
- $\text{AF } \varphi \equiv \neg \text{EG } \neg\varphi$
- $\text{A}(\varphi \text{ U } \psi) \equiv \neg[\text{E}(\neg\psi \text{ U } \neg(\varphi \vee \psi)) \vee \text{EG } \neg\psi]$

Note that the symbols $\top$ and $\perp$ are often used to respectively denote *true* and *false*.

In order to be consistent with Chan [13] and Samer and Veith [51, 53], we define five additional temporal operators. The first one is the “weak until” operator:

$$\varphi \text{ W } \psi \equiv (\text{G } \varphi) \vee (\varphi \text{ U } \psi).$$

The four others are variants of the (strong) until and weak until operators:

$$\begin{aligned} \varphi \overset{\circ}{\text{U}} \psi &\equiv \varphi \text{ U } (\varphi \wedge \psi) && \text{ (“overlapping strong until”)} \\ \varphi \overset{\circ}{\text{W}} \psi &\equiv \varphi \text{ W } (\varphi \wedge \psi) && \text{ (“overlapping weak until”)} \\ \varphi \bar{\text{U}} \psi &\equiv \varphi \text{ U } (\neg\varphi \wedge \psi) && \text{ (“disjoint strong until”)} \\ \varphi \bar{\text{W}} \psi &\equiv \varphi \text{ W } (\neg\varphi \wedge \psi) && \text{ (“disjoint weak until”)} \end{aligned}$$

Note that these new operators do not increase the expressive power of CTL [51, 53].

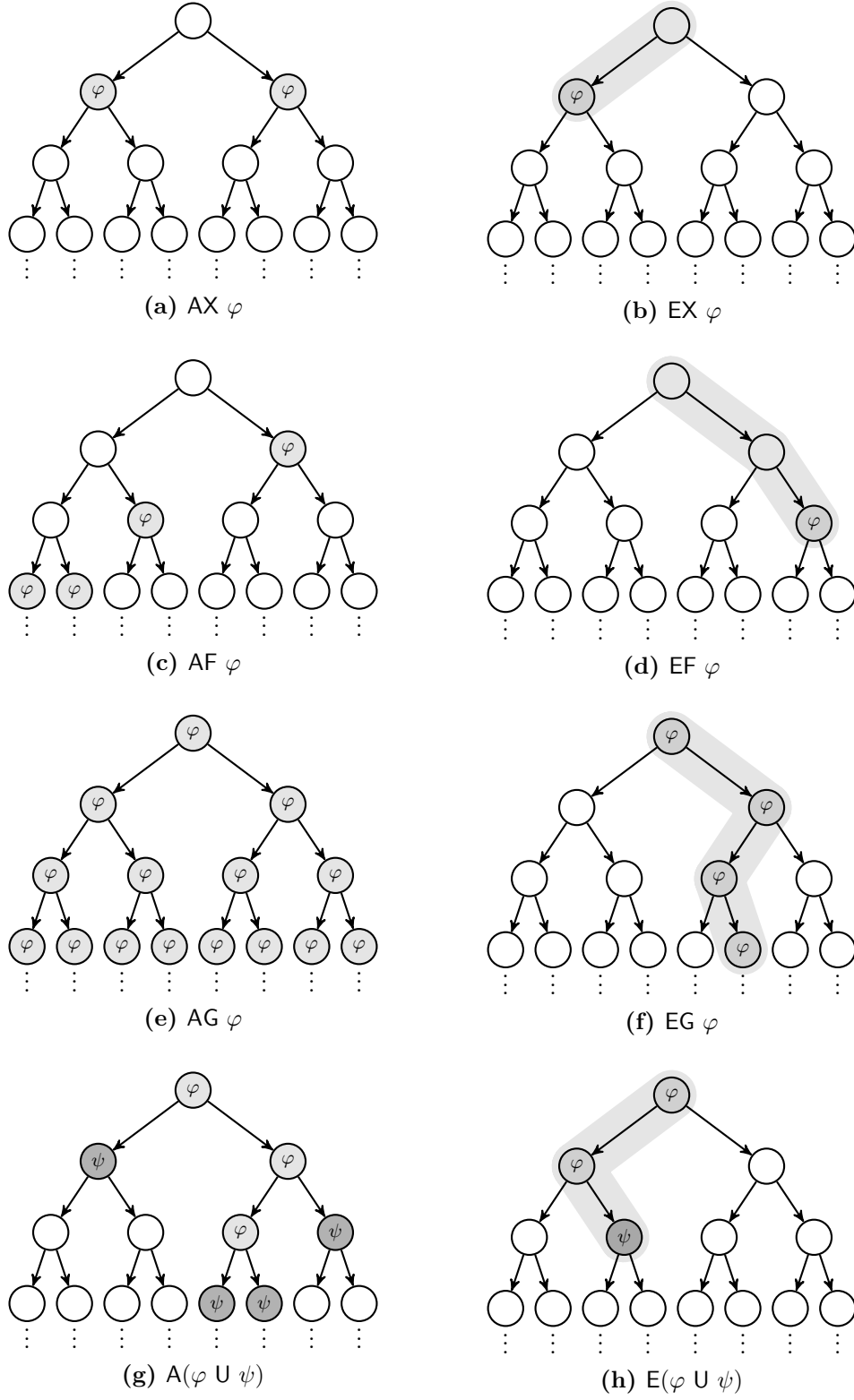


Figure 2.2. Graphical representation of the CTL operators.

Before going on, let us highlight some additional definitions that will be useful to fully understand the temporal logic queries theory:

Definition 5 ($\mathcal{K} \models \varphi$, from [13]). *We write $\mathcal{K}, s \models \varphi$ to denote that the CTL formula φ is satisfied at state s in the Kripke structure $\mathcal{K}$, and we write $\mathcal{K} \models \varphi$ if and only if $\forall s_0 \in Q_0, \mathcal{K}, s_0 \models \varphi$.*

Definition 6 ($\llbracket \varphi \rrbracket$, from [13]). *For any CTL formula φ , $\llbracket \varphi \rrbracket$ denotes the set of states in which φ is true*

Definition 7 (Set of Predecessors). *For any state set S , the set of predecessors of states in S is defined as:*

$$pre_{\exists}(S) = \{ s \in Q \mid \exists s' . (s, s') \in \Delta \text{ and } s' \in S \}$$

Definition 8 (Set of Successors). *For any state set S , the set of successors of states in S is defined as:*

$$post_{\exists}(S) = \{ s' \in Q \mid \exists s . (s, s') \in \Delta \text{ and } s \in S \}$$

Definition 9 (Fixed Points, from [68]). *For any states set S , let $\mathcal{P}(S)$ be the power set of S (that is, the set of all subsets of S), and $\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, an arbitrary function that maps sets of states to sets of states. A fixed point of τ is a set X such that $\tau(X) = X$. The least fixed point of τ , denoted $\mu Z. \tau(Z)$ is a set X such that $\tau(X) = X$, and, for all Z , if $\tau(Z) = Z$, then $X \subseteq Z$. The greatest fixed point of τ , denoted $\nu Z. \tau(Z)$ is a set X such that $\tau(X) = X$, and, for all Z , if $\tau(Z) = Z$, then $Z \subseteq X$.*

In addition to Definition 9, it should be noted that if τ is continuous, then it has a unique least and unique greatest fixed points respectively defined as: $\mu Z. \tau(Z) = \bigcup_i \tau^i(\emptyset)$ and $\nu Z. \tau(Z) = \bigcap_i \tau^i(S)$, with $\tau^i(X)$ denoting i applications of τ to X [22]. In practice, they are computed in an iterative way (see Figure 2.3). Least and greatest fixed points are commonly used in CTL model checking, and will be used to solve CTL queries. Indeed, provided an appropriate function, the least or greatest fixed point of this function can be used to characterize the set of states satisfying a CTL formula [26]. For example, if f is such appropriate function, and $EX\ Z = pre_{\exists}(Z)$:

- $EF\ f = \mu Z. f \vee EX\ Z$
- $EG\ f = \nu Z. f \wedge EX\ Z$

Intuitively, least fixed points correspond to properties that should hold eventually, while greatest fixed points correspond to properties that should hold globally.

2.1.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) were first introduced by Lee in the 1950s [37], and further studied by Akers in the 1970s [1]. However, the original representation of

<pre> function LFP($\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$) $Q \leftarrow \emptyset$ $Q' \leftarrow \tau(Q)$ while $Q \neq Q'$ do $Q \leftarrow Q'$ $Q' \leftarrow \tau(Q)$ end while return Q end function </pre>	<pre> function GFP($\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$) $Q \leftarrow S$ $Q' \leftarrow \tau(Q)$ while $Q \neq Q'$ do $Q \leftarrow Q'$ $Q' \leftarrow \tau(Q)$ end while return Q end function </pre>
--	--

Figure 2.3. Algorithms for computing the least and greatest fixed points, adapted from [68].

BDDs was quite inefficient and unpopular at the time. It was the work of Bryant [7], in 1986, that attracted the attention and renewed the interest about BDDs. Bryant observed that a reduced and ordered representation of BDDs can both significantly improve their efficiency and ease their use. He then proposed a new useful data structure and efficient algorithms to represent and manipulate what is denoted as ordered binary decision diagrams. Nowadays, ordered binary decision diagrams are often referred to as BDDs, the original definition of the latter being completely shadowed by Bryant's work. Following the established use, we will write “BDDs” to denote Bryant's ordered binary decision diagrams in the following.

Structure and Benefits

Fundamentally, BDDs are data structures used to efficiently represent boolean functions. They are represented as rooted, directed, acyclic graphs, with several decision nodes and two terminal nodes (namely, 0 and 1). Each decision node is labeled by a boolean variable and has two child nodes, the first that represents an assignment of the labeled boolean variable to 0, and the second that represents an assignment of the labeled boolean variable to 1 (see Figure 2.4). [61] In order to decide whether a particular assignment of the variables makes the function true or not, we must then traverse the graph from the root to a terminal node. For example, in Figure 2.4, the assignment $a = 1$, $b = 0$, $c = 1$, and $d = 0$ leads to the terminal node 0, hence we know that the function is false for this assignment.

Put differently, BDDs can be compared to binary decision trees (see Figure 2.5) to which all the redundancy is removed by a reduction process. We then obtain a very concise canonical representation for the boolean functions (that is, a representation such that every function has a unique representation *for a given variables ordering*).

Note that the chosen ordering of the variables is critical when building BDDs. Indeed, the size of the BDD for a given function depends on this (see the difference between

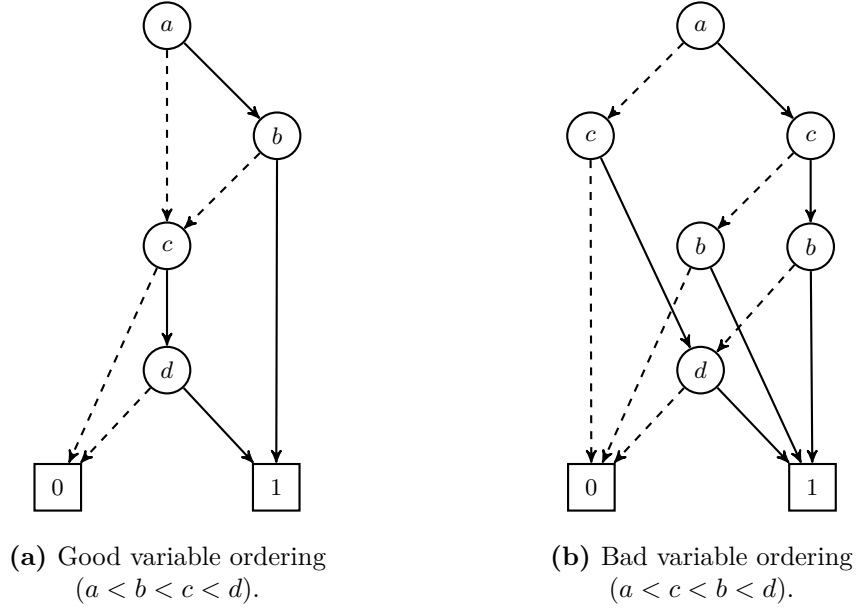


Figure 2.4. Binary decision diagrams representing the function $f = (a \wedge b) \vee (c \wedge d)$. Dashed arrows represent an assignment of the labeled boolean variables to 0, and solid arrows represent an assignment of the labeled boolean variables to 1.

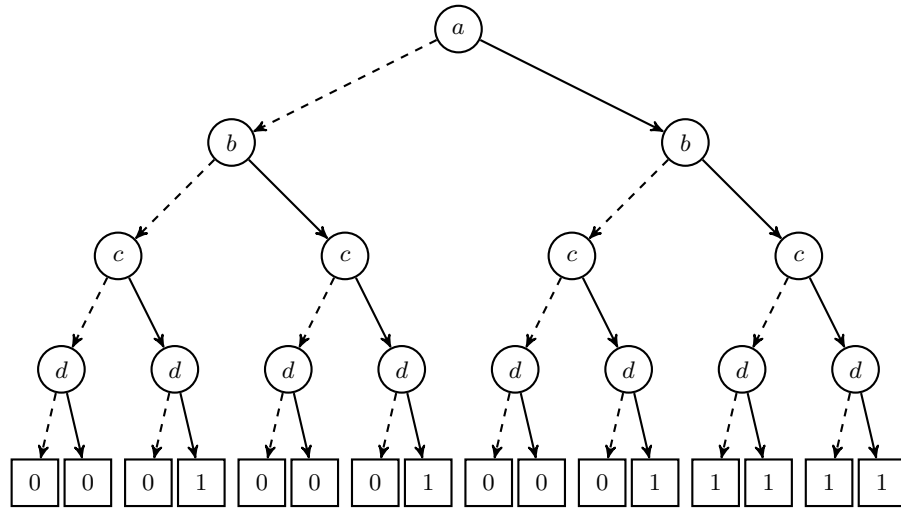


Figure 2.5. Binary decision tree representing the function $f = (a \wedge b) \vee (c \wedge d)$. Dashed arrows represent an assignment of the labeled boolean variables to 0, and solid arrows represent an assignment of the labeled boolean variables to 1.

Figure 2.4a and Figure 2.4b). Although finding the optimal ordering is NP-complete² [8], there fortunately exist efficient heuristics to tackle this problem such as dynamic variable reordering [49].

² The complexity class NP is the set of all decision problems solvable in polynomial time (relative to the size of the input) by a non-deterministic Turing machine. NP-complete decision problems denote the hardest or most expressive problems in NP. Note that, in practice, any given solution to an NP-complete decision problem can be verified in polynomial time, but there is no known efficient way to find a solution in the first place. [66]

In addition to its efficiency regarding to the space required for storing large boolean functions, another main advantage of BDDs lies in the efficient computation of the sixteen³ binary logical operations on boolean functions. Practically, the complexity of these operations is linear and depends on the size of the input BDDs [22].

Implications in Symbolic Model Checking

Following Bryant's work, McMillan [42] proposed to symbolically represent the states sets and transition relations of a system as boolean values and boolean formulas (instead of manipulating them explicitly) in order to tackle the state space explosion problem. Each state is encoded by an assignment of boolean values, and the transition relations are expressed as boolean formulas in terms of two sets of variables (one set encoding the old state and the other encoding the new one) [22]. Practically, this innovation is reflected in the use of BDDs to represent the boolean formulas. Indeed, their canonical representation of boolean functions and efficiency in the computation of logical operations make them optimal for this purpose.

2.2 Temporal Logic Queries

In this section, we survey the fundamental principles of temporal logic queries theory. As stated previously, we define temporal logic queries as CTL specifications, and call them *CTL queries*.

Definition 10 (CTL Query, from [53]). *A CTL query γ is a CTL formula where some subformulas are replaced by the special symbol $?$, called placeholder. We write $\gamma[\varphi]$ to denote the result of substituting all occurrences of the placeholder in γ by φ . We denote the set of all CTL queries by $CTLQ$.*

For example, provided the CTL formulas φ and ψ , $\gamma = A(\varphi \cup AG ?)$ is a CTL query (that is, $\gamma \in CTLQ$), and $\gamma[\varphi \wedge \psi]$ leads to the property $A(\varphi \cup AG (\varphi \wedge \psi))$. According to this general definition, $AG(? \leftrightarrow AF \varphi)$ and $A((? \vee \psi) \dot{W} \neg ?)$ are also CTL queries.

Definition 11 (Solution, from [53]). *Let γ be a CTL query, $\mathcal{K}$ a Kripke structure, and φ a CTL formula. If $\mathcal{K} \models \gamma[\varphi]$, then we say that φ is a solution to γ in $\mathcal{K}$. We denote the set of all solutions to γ in $\mathcal{K}$ by $\text{sol}(\mathcal{K}, \gamma) = \{ \varphi \mid \mathcal{K} \models \gamma[\varphi] \}$. A solution ξ to a CTL query γ in a Kripke structure $\mathcal{K}$ is exact if and only if it holds that $\text{sol}(\mathcal{K}, \gamma) = \{ \varphi \mid \xi \Rightarrow \varphi \}$.*

³ Binary logical operations have two arguments, each with two possible values, which gives $2^2 = 4$ possible combinations of inputs. Each of these four possible combinations leads to an output which can have two possible values. Consequently, there is a total of $2^4 = 16$ possible binary logical operations. [62]

Consider the Kripke structure $\mathcal{K}$ of the microwave oven example shown in Figure 1.3, and the CTL query $\gamma = \text{AG } (\text{heat} \rightarrow ?)$, meaning “What is guaranteed if the oven heats?” It is easy to see that close , $\neg \text{error}$, and $\text{close} \wedge \neg \text{error}$ are solutions to γ in $\mathcal{K}$. For its part, the exact solution to $\text{AG } (\text{heat} \rightarrow ?)$ in $\mathcal{K}$ is $\text{close} \wedge \neg \text{error}$ because it implies all the other solutions (namely, close and $\neg \text{error}$). Note that the notion of exact solution is equivalent to the notion of *strongest* solution we mentioned in Chapter 1.

Definition 12 (Exact Query, from [53]). *A CTL query is exact if and only if it has an exact solution in every Kripke structure where the set of solutions is not empty.*

In analogy to the EXPTIME-complete complexity for deciding the validity of a CTL query [13], identifying exact queries is also EXPTIME-complete [50]. To avoid coping with this complexity, Samer and Veith proceed the same way as Chan did, and define a syntactic fragment, CTLQ^x , in which all queries are exact [53]. Thanks to this fragment, we know that the query $\text{AG } (\text{heat} \rightarrow ?)$ is exact.

As stated previously, we only consider Samer and Veith’s exact queries as they are an extension to Chan’s valid queries.

2.2.1 Fragment CTLQ^x

Samer and Veith describe the fragment CTLQ^x as a deterministic context-free template grammar [53]. The production rules of this grammar are listed in Table 2.1. Accordingly, CTLQ^x is defined as $\bigcup_{i=1}^{10} \text{CTLQ}^i$, with CTLQ^1 denoting the language derived from non-terminal $\langle Q1 \rangle$, CTLQ^2 the language derived from $\langle Q2 \rangle$, and so on.

Although the general definition of CTL queries (see Definition 10) allows to have multiple occurrences of the placeholder, CTLQ^x only recognizes queries with *one* occurrence of the placeholder. In addition to this restriction to a single occurrence of the placeholder, CTL queries must also be expressed in *negation normal form* (NNF) (that is, expressions where the negation appears only in front of the atomic propositions and the placeholder). Note that any CTL formula can be put in NNF thanks to the extended set of operators, with respect to the following equivalences (from [54]):

- $\neg \text{A } \varphi \equiv \text{E } \neg \varphi$
- $\neg \text{E } \varphi \equiv \text{A } \neg \varphi$
- $\neg \text{X } \varphi \equiv \text{X } \neg \varphi$
- $\neg \text{G } \varphi \equiv \text{F } \neg \varphi$
- $\neg \text{F } \varphi \equiv \text{G } \neg \varphi$
- $\neg(\varphi \text{ U } \psi) \equiv \neg \psi \overset{\circ}{\text{W}} \neg \varphi$
- $\neg(\varphi \text{ W } \psi) \equiv \neg \psi \overset{\circ}{\text{U}} \neg \varphi$
- $\neg(\varphi \overset{\circ}{\text{U}} \psi) \equiv \neg \psi \text{ W } \neg \varphi$
- $\neg(\varphi \overset{\circ}{\text{W}} \psi) \equiv \neg \psi \text{ U } \neg \varphi$
- $\neg(\varphi \bar{\text{U}} \psi) \equiv (\varphi \vee \neg \psi) \overset{\circ}{\text{W}} \neg \varphi$
- $\neg(\varphi \bar{\text{W}} \psi) \equiv (\varphi \vee \neg \psi) \overset{\circ}{\text{U}} \neg \varphi$

$\langle Q1 \rangle ::=$	$?$	$\neg?$	$\star \wedge \langle Q3 \rangle$	$\star \wedge \langle Q4 \rangle$
	$\star \vee \langle Q2 \rangle$	$AX \langle Q3 \rangle$	$AX \langle Q4 \rangle$	$AX \langle Q6 \rangle$
	$AX \langle Q7 \rangle$	$A(\langle Q3 \rangle \dot{U} \star)$	$A(\langle Q4 \rangle \dot{U} \star)$	$A(\star \dot{U} \langle Q4 \rangle)$
	$A(\star \dot{U} \langle Q5 \rangle)$	$A(\star \bar{U} \langle Q2 \rangle)$	$A(\star \bar{U} \langle Q3 \rangle)$	$A(\star \bar{U} \langle Q4 \rangle)$
	$A(\star \bar{U} \langle Q5 \rangle)$	$A(\langle Q3 \rangle \dot{W} \star)$	$A(\langle Q4 \rangle \dot{W} \star)$	$A(\star \bar{W} \langle Q2 \rangle)$
	$A(\star \bar{W} \langle Q3 \rangle)$	$A(\star \bar{W} \langle Q4 \rangle)$	$A(\star \bar{W} \langle Q5 \rangle)$	$\star \wedge \langle Q1 \rangle$
	$\star \vee \langle Q1 \rangle$	$AX \langle Q1 \rangle$	$A(\langle Q1 \rangle \dot{U} \star)$	$A(\star \bar{U} \langle Q1 \rangle)$
	$A(\langle Q1 \rangle \dot{W} \star)$	$A(\star \bar{W} \langle Q1 \rangle)$		
$\langle Q2 \rangle ::=$	$\star \wedge \langle Q5 \rangle$	$AX \langle Q5 \rangle$	$A(\langle Q5 \rangle \dot{U} \star)$	$A(\star \dot{U} \langle Q3 \rangle)$
	$A(\langle Q5 \rangle \dot{W} \star)$	$A(\star \dot{W} \langle Q3 \rangle)$	$A(\star \dot{W} \langle Q4 \rangle)$	$A(\star \dot{W} \langle Q5 \rangle)$
	$\star \wedge \langle Q2 \rangle$	$AX \langle Q2 \rangle$	$A(\langle Q2 \rangle \dot{U} \star)$	$A(\langle Q2 \rangle \dot{W} \star)$
$\langle Q3 \rangle ::=$	$AF \langle Q6 \rangle$	$A(\langle Q1 \rangle U \star)$	$A(\langle Q2 \rangle U \star)$	$A(\langle Q4 \rangle U \star)$
	$A(\langle Q5 \rangle U \star)$	$A(\langle Q6 \rangle U \star)$	$A(\langle Q7 \rangle U \star)$	$A(\star U \langle Q6 \rangle)$
	$\star \vee \langle Q3 \rangle$	$AF \langle Q3 \rangle$	$A(\langle Q3 \rangle U \star)$	$A(\star U \langle Q3 \rangle)$
$\langle Q4 \rangle ::=$	$\star \vee \langle Q5 \rangle$	$AF \langle Q5 \rangle$	$AF \langle Q7 \rangle$	$A(\langle Q6 \rangle \dot{U} \star)$
	$A(\langle Q7 \rangle \dot{U} \star)$	$A(\star U \langle Q5 \rangle)$	$A(\star U \langle Q7 \rangle)$	$A(\star \dot{U} \langle Q7 \rangle)$
	$A(\star \bar{U} \langle Q6 \rangle)$	$A(\star \bar{U} \langle Q7 \rangle)$	$A(\langle Q1 \rangle W \star)$	$A(\langle Q2 \rangle W \star)$
	$A(\langle Q3 \rangle W \star)$	$A(\langle Q5 \rangle W \star)$	$A(\langle Q6 \rangle W \star)$	$A(\langle Q7 \rangle W \star)$
	$A(\langle Q6 \rangle \dot{W} \star)$	$A(\langle Q7 \rangle \dot{W} \star)$	$A(\star W \langle Q3 \rangle)$	$A(\star W \langle Q5 \rangle)$
	$A(\star W \langle Q6 \rangle)$	$A(\star W \langle Q7 \rangle)$	$A(\star \bar{W} \langle Q6 \rangle)$	$A(\star \bar{W} \langle Q7 \rangle)$
	$\star \vee \langle Q4 \rangle$	$AF \langle Q4 \rangle$	$A(\star U \langle Q4 \rangle)$	$A(\langle Q4 \rangle W \star)$
	$A(\star W \langle Q4 \rangle)$			
$\langle Q5 \rangle ::=$	$A(\star \dot{U} \langle Q6 \rangle)$	$A(\star \dot{W} \langle Q6 \rangle)$	$A(\star \dot{W} \langle Q7 \rangle)$	
$\langle Q6 \rangle ::=$	$A(\langle Q8 \rangle U \star)$	$A(\langle Q9 \rangle U \star)$	$\star \vee \langle Q6 \rangle$	
$\langle Q7 \rangle ::=$	$\star \wedge \langle Q6 \rangle$	$\star \vee \langle Q8 \rangle$	$\star \vee \langle Q9 \rangle$	$A(\langle Q8 \rangle W \star)$
	$A(\langle Q9 \rangle W \star)$	$\star \wedge \langle Q7 \rangle$	$\star \vee \langle Q7 \rangle$	
$\langle Q8 \rangle ::=$	$AF \langle Q9 \rangle$	$AG \langle Q1 \rangle$	$AG \langle Q3 \rangle$	$AG \langle Q4 \rangle$
	$AG \langle Q6 \rangle$	$AG \langle Q7 \rangle$	$A(\star U \langle Q9 \rangle)$	$A(\star \dot{U} \langle Q9 \rangle)$
	$A(\star \bar{U} \langle Q9 \rangle)$	$A(\star W \langle Q9 \rangle)$	$A(\star \bar{W} \langle Q9 \rangle)$	$\star \wedge \langle Q8 \rangle$
	$AX \langle Q8 \rangle$	$AF \langle Q8 \rangle$	$AG \langle Q8 \rangle$	$A(\langle Q8 \rangle \dot{U} \star)$
	$A(\star U \langle Q8 \rangle)$	$A(\star \dot{U} \langle Q8 \rangle)$	$A(\star \bar{U} \langle Q8 \rangle)$	$A(\langle Q8 \rangle \dot{W} \star)$
	$A(\star W \langle Q8 \rangle)$	$A(\star \bar{W} \langle Q8 \rangle)$		
$\langle Q9 \rangle ::=$	$A(\star \dot{W} \langle Q8 \rangle)$	$\star \wedge \langle Q9 \rangle$	$AX \langle Q9 \rangle$	$A(\langle Q9 \rangle \dot{U} \star)$
	$A(\langle Q9 \rangle \dot{W} \star)$	$A(\star \dot{W} \langle Q9 \rangle)$		
$\langle Q10 \rangle ::=$	$AG \langle Q2 \rangle$	$AG \langle Q5 \rangle$	$AG \langle Q9 \rangle$	$\star \wedge \langle Q10 \rangle$
	$\star \vee \langle Q10 \rangle$	$AX \langle Q10 \rangle$	$AF \langle Q10 \rangle$	$AG \langle Q10 \rangle$
	$A(\langle Q10 \rangle U \star)$	$A(\langle Q10 \rangle \dot{U} \star)$	$A(\star U \langle Q10 \rangle)$	$A(\star \dot{U} \langle Q10 \rangle)$
	$A(\star \bar{U} \langle Q10 \rangle)$	$A(\langle Q10 \rangle W \star)$	$A(\langle Q10 \rangle \dot{W} \star)$	$A(\star W \langle Q10 \rangle)$
	$A(\star \dot{W} \langle Q10 \rangle)$	$A(\star \bar{W} \langle Q10 \rangle)$		

Table 2.1. CTLQ^x production rules, adapted from [53]. $\star$ is a wildcard symbol representing any CTL formula.

In their paper [53], Samer and Veith choose to ignore – without loss of generality – the case where there is a negation in front of the placeholder. Despite this choice, we find it valuable to keep it for clarity and completeness. Indeed, theoretically, this assumption can be made easily since there is no loss of generality (the case being easily handled by returning the complement of the solution). Nevertheless, as our work essentially consists of the practical implementation of temporal logic queries, it seems essential to clearly handle the negation in front of the placeholder. Hence, we make a slight modification of the production rules to take the negation of the placeholder into account. In this case, we extended the non-terminal $\langle Q1 \rangle$ to recognize $\neg?$.

Note that, due to the EXPTIME-completeness proof for deciding the exactness of CTL queries, CTLQ^x is naturally not a maximal characterization of exact CTL queries. This limitation means that some exact queries are not captured by the fragment (for example, provided the propositional formulas p and q , the query $\gamma = \text{AF}(p \wedge \text{AF}(q \vee \text{AG } ?))$ is not in CTLQ^x although it is exact).

To solve queries that belongs to fragment CTLQ^x , Samer and Veith propose an extension to Chan’s symbolic BDD-based algorithm: the *extended Chan algorithm*.

2.2.2 The Extended Chan Algorithm

The original CTL query solving algorithm was introduced by Chan [13] in order to solve queries in his syntactic fragment of valid queries, CTL^v . Regrettably, he did not formally prove the correctness of his algorithm. For this reason, Samer and Veith [53] propose – along with a formal proof – the extended Chan algorithm, a generalization of the original algorithm in order to solve queries in their (larger) fragment of exact queries, CTLQ^x (see Algorithm 1). Note that Q represents the set of all the (reachable) states of the model. In addition, following Chan, three auxiliary sets are defined (from [13, 53]):

$$\begin{aligned}\mathcal{R}_\varphi &= \mu Z.((S \cup \text{post}_\exists(Z)) \cap \llbracket \varphi \rrbracket) \\ \mathcal{C}_\varphi^\gamma &= \nu Z.(\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]} \cap \text{post}_\exists(Z)) \\ \mathcal{B}_\varphi^\gamma &= (S \cup \text{post}_\exists(\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}) \setminus (\llbracket \varphi \rrbracket \cup \llbracket \gamma[\perp] \rrbracket))\end{aligned}$$

Informally, the set $\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}$ consists of the states of the model that are *reachable* from states in S by going only through the states that satisfy $\varphi \wedge \neg \gamma[\perp]$, the set $\mathcal{C}_\varphi^\gamma$ consists of all states within a *cycle* in $\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}$, and the set $\mathcal{B}_\varphi^\gamma$ represents the *boundary* of $\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}$ (that is, the first states on each path starting from S that are not in $\mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}$ and that do not satisfy $\gamma[\perp]$). The intuition of these three sets is given in Figure 2.6, where the initial set S is assumed to gather the four double-circled states. [53]

Algorithm 1 The extended Chan algorithm, adapted from [53].

```

1: function FSOL( $\gamma, \varphi, S, cycle$ )
2:   if  $cycle$  then
3:      $C \leftarrow \mathcal{C}_\varphi^\gamma$ 
4:   else
5:      $C \leftarrow \emptyset$ 
6:   end if
7:    $U_1 \leftarrow \nu \mathcal{Z}.((C \setminus \llbracket \gamma[\top] \rrbracket) \cap post_\exists(\mathcal{Z}))$ 
8:    $U_2 \leftarrow U_1 \cup (\mathcal{B}_\varphi^\gamma \setminus \llbracket \gamma[\top] \rrbracket)$ 
9:    $U_3 \leftarrow \mu \mathcal{Z}.((U_2 \cup pre_\exists(\mathcal{Z})) \cap \mathcal{R}_{\varphi \wedge \neg \gamma[\perp]} \setminus \llbracket \gamma[\top] \rrbracket)$ 
10:  return  $((pre_\exists(U_3) \cap \mathcal{R}_{\varphi \wedge \neg \gamma[\perp]}) \cup \mathcal{B}_\varphi^\gamma \cup C) \cap \llbracket \gamma[\top] \rrbracket$ 
11: end function
12:
13: function ESOL( $\gamma, S$ )
14:  switch  $\gamma$  do
15:    case ?
16:      return  $S$ 
17:    case  $\neg?$ 
18:      return  $Q \setminus S$ 
19:    case  $\theta \wedge \bar{\gamma}$ 
20:      return ESOL( $\bar{\gamma}, S$ )
21:    case  $\theta \vee \bar{\gamma}$ 
22:      return ESOL( $\bar{\gamma}, S \setminus \llbracket \theta \rrbracket$ )
23:    case AX  $\bar{\gamma}$ 
24:      return ESOL( $\bar{\gamma}, post_\exists(S)$ )
25:    case AF  $\bar{\gamma}$ 
26:      return ESOL( $A(\top \cup \bar{\gamma}), S$ )
27:    case AG  $\bar{\gamma}$ 
28:      return ESOL( $A(\bar{\gamma} \dot{W} \perp), S$ )
29:    case  $A(\bar{\gamma} \cup \theta)$ 
30:      return ESOL( $A((\theta \vee \bar{\gamma}) \dot{W} \theta), S$ )
31:    case  $A(\bar{\gamma} \dot{U} \theta)$ 
32:      return ESOL( $A(\bar{\gamma} \dot{W} \theta), S$ )
33:    case  $A(\theta \cup \bar{\gamma})$ 
34:      return ESOL( $\bar{\gamma}, FSOL(\bar{\gamma}, \theta, S, true)$ )
35:    case  $A(\theta \dot{U} \bar{\gamma})$ 
36:      return ESOL( $A(\theta \cup (\theta \wedge \bar{\gamma})), S$ )
37:    case  $A(\theta \bar{U} \bar{\gamma})$ 
38:      return ESOL( $A(\theta \bar{W} \bar{\gamma}), S$ )
39:    case  $A(\bar{\gamma} W \theta)$ 
40:      return ESOL( $A((\theta \vee \bar{\gamma}) \dot{W} \theta), S$ )
41:    case  $A(\bar{\gamma} \dot{W} \theta)$ 
42:      return ESOL( $\bar{\gamma}, S \cup post_\exists(\mathcal{R}_{\neg \theta})$ )
43:    case  $A(\theta W \bar{\gamma})$ 
44:      return ESOL( $\bar{\gamma}, FSOL(\bar{\gamma}, \theta, S, false)$ )
45:    case  $A(\theta \dot{W} \bar{\gamma})$ 
46:      return ESOL( $A(\theta W (\theta \wedge \bar{\gamma})), S$ )
47:    case  $A(\theta \bar{W} \bar{\gamma})$ 
48:      return ESOL( $\bar{\gamma}, (S \cup post_\exists(\mathcal{R}_\theta)) \setminus \llbracket \theta \rrbracket$ )
49:  end function

```

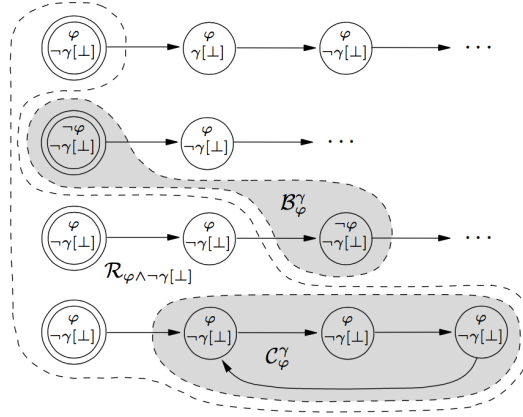


Figure 2.6. Intuitive meaning of the auxiliary sets, taken from [53].

Recall the principal claim of Samer and Veith in their second paper about CTL queries [53]: all queries in CTLQ^x can be reduced in a deterministic manner to solving their subqueries at appropriate system states. They implemented this deterministic solving through the ESOL function of Algorithm 1.

Unfortunately, this function is not sufficient in itself, because some CTL operators intrinsically cause non-determinism in the process of finding the appropriate set of states while others do not. Samer and Veith respectively classify such operators as *existential* and *universal* ones (see Table 2.2). For example, the AG operator is universal because, for every path π and CTL formula φ , the query $\gamma = \text{AG } \bar{\gamma}[\varphi]$ is satisfied at path π if and only if $\bar{\gamma}[\varphi]$ is satisfied at every state of π . “Therefore, solving γ can be reduced to solving its subquery $\bar{\gamma}$ at universally quantified positions on a path.” [53] In contrast, the AU operator with respect to its second argument is existential because, for every path π and CTL formulas θ and φ , the query $\gamma = \text{A}(\theta \cup \bar{\gamma}[\varphi])$ is satisfied at path π if and only if there exists a state of π from which $\bar{\gamma}[\varphi]$ is satisfied before θ stops being satisfied. But, we do not know which state it is beforehand, hence the non-determinism. “Therefore, solving γ can be reduced to solving its subquery $\bar{\gamma}$ at existentially quantified positions on a path.” [53]

In light of this classification, we can now establish the actual use of the auxiliary sets: they ensure the deterministic solvability of CTL queries that contain existential operators. Indeed, according to Samer and Veith, “solving a query in fragment CTLQ^x can be reduced to solving its existentially occurring subqueries at the set of states $\mathcal{B}_\varphi^\gamma$ and $\mathcal{C}_\varphi^\gamma$ ” [53]. However, in practice, this set of states has to be computed in a more sophisticated way. This is the purpose of the FSOL function. This function computes the set of states with highest indices (that is, the states that are the furthest away from the initial set S) among the states at which the temporal logic query has a solution. “Intuitively, the sets U_1 and U_2 together consist of all states with highest indices but on which γ has no solution. The set U_3 consists then of all states that can be reached by going backwards as long as γ does not have a solution. Therefore,

Universal		Existential	
AX ?	AG ?	AF ?	$A(\varphi \cup ?)$
$A(? \cup \varphi)$	$A(? \mathbin{W} \varphi)$	$A(\varphi \mathbin{W} ?)$	$A(\varphi \mathbin{\dot{U}} ?)$
$A(? \mathbin{\dot{U}} \varphi)$	$A(? \mathbin{\dot{W}} \varphi)$	$A(\varphi \mathbin{\dot{W}} ?)$	
$A(\varphi \mathbin{\bar{U}} ?)$	$A(\varphi \mathbin{\bar{W}} ?)$		

Table 2.2. Classification of CTL operators, adapted from [53].

by making a further step backwards, we obtain the desired set of states (that is, the states with highest indices at which γ has a solution).” [53]

For example, when solving the query $A(\text{heat} \cup \text{AG } ?)$, we first enter in the condition $A(\theta \cup \bar{\gamma})$. The algorithm tells us that the query may be solved by solving the subquery $\bar{\gamma}$ at the set of states given by $\text{FSOL}(\bar{\gamma}, \theta, S, \text{true})$, because $A(\theta \cup \bar{\gamma})$ is an existential operator. Next, we enter in the condition $\text{AG } \bar{\gamma}$. The algorithm then tells us that the query may be solved by transforming the query into $A(\bar{\gamma} \mathbin{\dot{W}} \perp)$ at the same set of states. In the next recursive call, we naturally enter in the $A(\bar{\gamma} \mathbin{\dot{W}} \theta)$ condition, which tells us that the query may be solved by solving the subquery $\bar{\gamma}$ at the set of states given by $S \cup \text{post}_{\exists}(\mathcal{R}_{-\theta})$. Finally, in the last recursive call, we enter in the ? condition and the current set of states (which represents the unique set of solution states; see Theorem 1) is returned.

Note that, as for the production rules of CTLQ^x , we also extended Samer and Veith’s algorithm to take the negation of the placeholder into account. This change is reflected in the addition of lines 17 and 18 in Algorithm 1. Basically, when the algorithm encounters the negation of the placeholder, it returns the complement of the states set S (in relation to the set of all the reachable states Q).

The following theorem states Samer and Veith’s main result regarding to the extended Chan algorithm.

Theorem 1 (from [50, 53]). *Let $\gamma \in \text{CTLQ}^x$ and S be a set of states in a Kripke structure. Moreover, let ESOL be the function defined in Algorithm 1. Suppose that $S \models \gamma[\top]$ (that is, γ has a solution at each state in S). Then, $\text{ESOL}(\gamma, S)$ returns the unique set of solution states to γ at S .*

Definition 13 (Set of Solution States, from [53]). *Let γ be a CTL query and S a set of states in a Kripke structure. A set of states R is the unique set of solution states to γ at S if and only if it holds that $R \models \varphi$ if and only if $S \models \gamma[\varphi]$ for all CTL formulas φ .*

Consider the microwave oven example one more time. If we launch the extended Chan algorithm with $\gamma = \text{AG } (? \rightarrow \text{AF } \text{heat})$ and the set of initial states of the model $Q_0 = \{s_0\}$ as inputs, the algorithm returns the set $\{s_3, s_5, s_6\}$. It is easy to see that this set is indeed the unique set of solution states, since it gathers the only three states from which the oven eventually heats. As shown as foretaste with this example,

the extended Chan algorithm makes complete sense when we give Q_0 (that is, the set of initial states of the model) instead of an arbitrary set S as input. Indeed, according to Chan and Samer and Veith [13, 50, 53], the *characteristic function* of the set returned by $\text{ESOL}(\gamma, Q_0)$ is an exact solution to γ in the model.

Definition 14 (Characteristic Function, adapted from [13]). *The characteristic function of a set of states R is a propositional formula defined as:*

$$\bigvee_{s \in R} \left(\bigwedge_{x \in L(s)} x \wedge \bigwedge_{x \in AP \setminus L(s)} \neg x \right)$$

In the previous example, the unique set of solution states is represented, in detail, by: $\{\{\neg \text{start}, \text{close}, \text{heat}, \neg \text{error}\}, \{\text{start}, \text{close}, \neg \text{heat}, \neg \text{error}\}, \{\text{start}, \text{close}, \text{heat}, \neg \text{error}\}\}$. The characteristic function of this set is therefore: $(\neg \text{start} \wedge \text{close} \wedge \text{heat} \wedge \neg \text{error}) \vee (\text{start} \wedge \text{close} \wedge \neg \text{heat} \wedge \neg \text{error}) \vee (\text{start} \wedge \text{close} \wedge \text{heat} \wedge \neg \text{error})$, and is an exact solution to $\text{AG} (? \rightarrow \text{AF heat})$ in the model of the microwave oven.

Nevertheless, this exact solution – although giving complete information – is likely to be too complex to be easily understood. According to Chan’s principal motivation, the real interest of temporal logic queries is however to give an answer that is easy to comprehend for the user in order to actually help him to understand the system behaviors. For this purpose, Chan proposes a *simplification algorithm*, which takes the form of a decomposition of the solution into a (possibly large) number of “simple” pieces [13].

2.2.3 Chan’s Simplification Algorithm

In this subsection, we survey Chan’s strategy to cope with the simplification of the exact solution derived from the output of the extended Chan algorithm. Specifically, Chan’s approach consists of an *approximate conjunctive decomposition* of a propositional formula (see Algorithm 2). Intuitively, this greedy algorithm tries to encapsulate a given propositional formula into a conjunction of (simpler) propositional formulas that are the smallest possible. It is important to note that Chan’s decomposition is an approximation, since the conjunction obtained may be weaker than the given propositional formula [13].

Chan’s simplification algorithm operates as follows. It takes a propositional formula s (typically, the characteristic function of the output of the extended Chan algorithm) and an integer k with $0 < k \leq |\text{atoms}(s)|$ as inputs. The function $\text{atoms}(s)$ returns the set of atomic propositions appearing in s , and k denotes the maximum number of atomic propositions that must appear in the conjuncts of the simplification. The result is accumulated in the variable C , which has the form of a set of propositional formulas. Consequently, the simplification of s is defined as $\bigwedge C$ (that is, the conjunction of the elements of C). Basically, the two nested loops of the algorithm search for nontrivial propositional formulas (that is, propositional formulas that are not equal to *true* or s) that contain only j atomic propositions, with increasing j up

Algorithm 2 Approximate conjunctive decomposition of a propositional formula, taken from [13].

Input: Propositional formula s

Input: k with $0 < k \leq |atoms(s)|$

1: $C \leftarrow \emptyset$

2: **for** $j \leftarrow 1$ to k **do**

3: **for all** $Y \subseteq atoms(s)$ with $|Y| = j$ **do**

4: $r \leftarrow (\exists \overline{Y}. s) \downarrow C$

5: **if** $r \not\Leftarrow true$ and $r \not\Leftarrow s$ **then**

6: $C \leftarrow C \cup \{r\}$

7: **end if**

8: **end for**

9: **end for**

Output: C with $s \Rightarrow \bigwedge C$

to k . Such propositional formulas are computed as the projection of s onto a set Y of atomic propositions ($proj = \exists \overline{Y}. s$). (Note that this operation can also be called an existential abstraction.) To be more specific, Y denotes subsets of $atoms(s)$ with $|Y| = j$, and $proj$ represents a propositional formula that contains only the atomic propositions in Y . In order to remove eventual redundant information, the result of the projection is minimized using the other propositional formulas already computed ($proj \downarrow C$). Formally, $proj \wedge (\bigwedge C) \Leftrightarrow (proj \downarrow C) \wedge (\bigwedge C)$. This minimization is performed thanks to the *restrict* operator defined by Coudert, Berthet and Madre [24]. Each time a nontrivial propositional formula is found, it is added to the set C , and, at the end, the algorithm outputs C . Note that the algorithm runs in exponential-time in relation to k . However, as reported by Chan, “this is not a serious problem in practice, because the result will be too complicated to understand for large k anyway” [13]. Typical values for k are $k \leq 4$.

To illustrate Chan’s simplification algorithm, consider the exact solution we found in previous subsection $(\neg start \wedge close \wedge heat \wedge \neg error) \vee (start \wedge close \wedge \neg heat \wedge \neg error) \vee (start \wedge close \wedge heat \wedge \neg error)$. With $k = 2$, this exact solution is simplified as follows: $\neg error \wedge close \wedge (start \vee heat)$. As expected, the simplification of the solution is easier to understand. It is therefore easier to draw conclusions about it (recall what we learned at the end of the demonstration example in Chapter 1).

Note that we may not be interested in all the atomic propositions of the system, and may want to restrict those that are present in the solution. In this case, we just have to project the propositional formula s on the atomic propositions we are interested in (recall the projection $\exists \overline{Y}. s$), and use the result as input to Algorithm 2.

In practice, our approach for simplifying solutions differs in some ways from Chan’s. The differences will be deeply explained in the next chapter.

2.3 PyNuSMV

The implementation of temporal logic queries has been made using *PyNuSMV* [11], a Python framework for implementing BDD-based model-checking algorithms developed in the Louvain Verification Lab of the Université Catholique de Louvain.

2.3.1 Origins

PyNuSMV is the descendant of a lineage of symbolic model verifiers. Its origin lies in the SMV system, which was succeeded by NuSMV.

Symbolic Model Verifier (SMV)

Besides having suggested the symbolic model checking approach to cope with the state space explosion problem, McMillan also introduced the *SMV system* [42], a tool for checking finite state systems against CTL specifications. Beyond the tool, McMillan proposed a language to define models: the *SMV language* [42]. Its primary purpose was to provide a symbolic description of the transition relation of a finite Kripke structure. The system remained at the experimental stage, but the language evolved and is still widely used today.

New Symbolic Model Verifier (NuSMV)

NuSMV [17] appeared less than a decade after the SMV system. Its development met the need for a well structured, flexible and documented platform for model checking, designed to be applicable in technology transfer projects. NuSMV is the result of the reengineering, reimplementing and extension of the SMV system. The main changes include a textual interaction shell, a system architecture designed to be highly modular and open, and a strongly enhanced implementation. All this makes NuSMV a robust and maintainable state-of-the-art symbolic model checker. As for the system functionalities, NuSMV principally supports Linear Temporal Logic (LTL) and CTL model checking by implementing BDD-based symbolic model checking.

Two years later, the version 2 of NuSMV tool (NuSMV 2 in the following) has been released [18]. The main novelty of interest here is the new development and license model of the tool: NuSMV 2 is distributed under an Open Source License [57]. This change is crucial in the context of this thesis, since it led to the development of PyNuSMV. Note that NuSMV 2 inherits all the functionalities of the previous version, and extends them in two main directions: the integration of model checking techniques based on propositional satisfiability, and the support of bounded model checking.

Both versions of NuSMV use a slightly modified version of the SMV language to describe models. We invite the reader to refer to NuSMV user manual [12, pp. 6-47] for a complete description of this language. Note that our work is based on this version of the SMV language, and we will call “SMV models” the models that are modeled with it.

PyNuSMV: NuSMV as a Python Library

Despite being robust, powerful and widely-used, NuSMV is hard to extend or customize because of the size and complexity of its code base (more than 200,000 lines of C code). Starting from this fact, Busard and Pecheur [11] began the implementation of PyNuSMV, a Python library whose aim is to simplify the use/extension/customization of the rich BDD-based model-checking functionalities of NuSMV, without substantially affecting the performance of model checking. Python has been chosen because “it comes with a full standard library and a full-fledged programming language supporting high-level programming (garbage collection, functional closures)” [11], which makes PyNuSMV a modern and easy-to-use model-checking library.

2.3.2 Structure

The architecture of PyNuSMV (depicted in Figure 2.7) consists of three layers:

1. The code of NuSMV.
2. The lower interface.
3. The upper interface.

The lower interface – which makes the link between the code of NuSMV and the upper interface – is generated by SWIG [4], a wrapper generator for C code. In other words, SWIG automatically generates the bindings between C and Python, giving access to all NuSMV functions in PyNuSMV. Unfortunately, SWIG does not provide the high-level Python capabilities such as memory management. For this reason, the upper interface has been implemented upon the lower one in order to cope with this issue. The upper interface basically consists of a library of classes and modules giving access to NuSMV’s main data structures and functionalities, while abstracting implementation details by taking care of the memory management (through garbage collection) and wrapping pointers.

As suggested by Busard and Pecheur in their paper [11], we only interact with the upper interface in the context of this thesis. The latter is composed of eleven modules (namely, `init`, `glob`, `model`, `node`, `fsm`, `prop`, `dd`, `parser`, `mc`, `exception`, and `utils`), each one proposing some functionalities. For our part, we mainly work with the modules that give access to BDDs and states of the model, and to the model



Figure 2.7. PyNuSMV three-layer architecture, taken from [11].

itself (respectively, `dd` and `glob`), as well as `init`, `mc`, `prop`, and `utils` modules. Their use will be detailed in the next chapter.

Chapter 3

PyTLQ

PyTLQ is an original Python package for solving temporal logic queries – as defined by Chan [13] and corrected/extended by Samer and Veith [51, 53] – using the BDD-based functionalities of the PyNuSMV model-checking library [11].

PyTLQ is licensed under the GNU Lesser General Public License version 2.1 [56], and is available at <https://github.com/sthibert/PyTLQ>.

This chapter describes the architecture and functionalities of PyTLQ, reviews technical aspects relating to important implementation details, and covers the tool limitations.

3.1 System Architecture

PyTLQ has been developed as an independent package, uncoupled from PyNuSMV. Indeed, in the same idea as the other tools that have already been implemented with PyNuSMV (for example, ARCTL and CTLK model checkers), PyTLQ only uses the latter as an application programming interface (API). Nevertheless, efforts have been made to follow up the structure and implementation choices of these other tools in the interest of consistency with the PyNuSMV project.

The high-level design of PyTLQ basically consists of an *API* providing classes and functions for external programs, and a *standalone script* that uses all the functionalities defined in the API for solving CTL queries per se.

The internal structure of PyTLQ (depicted in Figure 3.1) is composed of four main blocks:

- In the first place, the *parser* parses the input string representing a CTL query and creates the corresponding abstract syntax tree (AST).

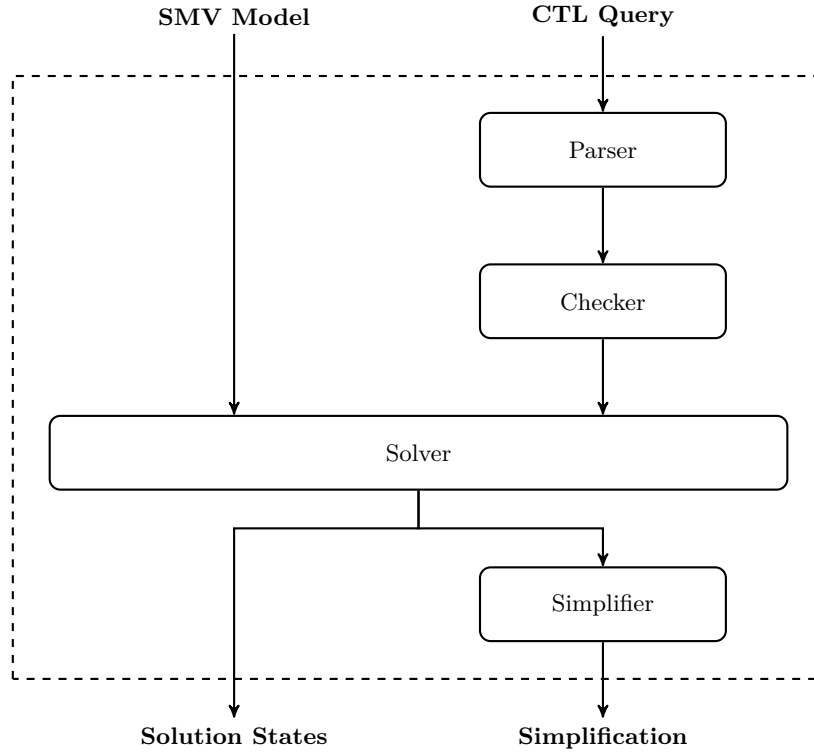


Figure 3.1. The internal structure of PyTLQ.

- Secondly, the *checker* verifies that the AST-based CTL query belongs to the syntactic fragment CTLQ^x . It follows the corresponding production rules to determine the membership to the fragment, and returns a truth value.
- Then, the *solver* applies the solving algorithm on the SMV model and the CTL query that passes the verification, and returns the unique set of solution states that represents an exact solution to the query in the model (if there is one).
- Finally, if a simplification is required, the *simplifier* takes care to transform the set of solution states – which is likely to be too complex – in a more understandable form.

We made the choice of separating the parsing from the verification of the membership to fragment CTLQ^x for several reasons. First of all, CTL queries that belong to CTLQ^x must be in negation normal form (NNF). The transformation in NNF at parsing time would consequently have increased the complexity of the parser significantly. Then, the query must satisfy the complex production rules of CTLQ^x , which would have increased the complexity of the parser even more. Finally, this architecture allowed us to implement a parser that is as general as possible (see Subsection 3.2.1), and could therefore be reused for eventual future work about CTL queries that do not need to belong to CTLQ^x .

PyTLQ aiming to be used in the context of hardware and software verification, the source code of these four blocks is therefore – naturally – entirely tested thanks to unit tests.

3.2 System Functionalities

In this section, we describe the main functionalities of PyTLQ, by following the typical execution path followed by both an SMV model and a CTL query.

Let us consider the microwave oven example and the CTL query $AG (? \rightarrow AF \textit{heat})$ to illustrate our words throughout the section. As a reminder, this query asks “What can guarantee that the microwave oven will heat?”

The first step in evaluating $AG (? \rightarrow AF \textit{heat})$ in the model is to *parse* the query in order to be able to further manipulate it.

3.2.1 Parsing CTL Queries

The `parser` module takes care of the parsing process. It provides the `parse_ctlq` function to parse strings representing CTL queries and return corresponding ASTs.

As CTL queries require the additional CTL operators defined by Chan (see Subsection 2.1.1), we needed to implement a custom parser – denoted as the *CTLQ parser* (recall Definition 10 and the set CTLQ) – that is able to parse these specific operators. Indeed, the parser of PyNuSMV cannot be used to parse CTL queries because it does not recognize the additional operators, and PyNuSMV does not provide functionalities to extend it.

For the sake of consistency with the tools that are already present, we chose to use PyParsing [41] – a state-of-the-art Python parsing tool for creating recursive-descent parsers – to implement the CTLQ parser. We also tried to follow the structure of the custom parsers defined in the other PyNuSMV tools, and reused (from these other tools) the private function that parses logical expressions, with usual precedence and associativity (see Table 3.1).

Accordingly, the CTLQ parser recognises the classical CTL grammar supplemented with the placeholder and the additional temporal operators (see Table 3.2). Note that it correctly assigns a higher precedence to unary CTL operators (namely, **AX**, **EX**, **AF**, **EF**, **AG**, and **EG**) in relation to logical operators (for example, the query $AX \neg ? \wedge \textit{true}$ is evaluated as $((AX (\neg ?)) \wedge \textit{true})$). Unfortunately, the current implementation of the CTLQ parser cannot detect at parsing time that a CTL query given as input does not contain any placeholder. This kind of validation being hard to implement in the grammar itself with PyParsing, we then post-process the parse

Operators in order of precedence	Associativity
$\neg$	Right-to-left
$\wedge$	Left-to-right
$\vee$	Left-to-right
$\rightarrow$	Right-to-left
$\leftrightarrow$	Left-to-right

Table 3.1. Precedence and associativity of logical operators.

$\langle CTLQ \rangle ::=$	$?$	$\neg \langle CTLQ \rangle$	$\langle CTLQ \rangle \wedge \langle CTLQ \rangle$
	$\langle CTLQ \rangle \vee \langle CTLQ \rangle$	$\langle CTLQ \rangle \rightarrow \langle CTLQ \rangle$	$\langle CTLQ \rangle \leftrightarrow \langle CTLQ \rangle$
	$AX \langle CTLQ \rangle$	$EX \langle CTLQ \rangle$	$AF \langle CTLQ \rangle$
	$EF \langle CTLQ \rangle$	$AG \langle CTLQ \rangle$	$EG \langle CTLQ \rangle$
	$A(\langle CTLQ \rangle \cup \langle CTLQ \rangle)$	$E(\langle CTLQ \rangle \cup \langle CTLQ \rangle)$	$A(\langle CTLQ \rangle \mathcal{W} \langle CTLQ \rangle)$
	$E(\langle CTLQ \rangle \mathcal{W} \langle CTLQ \rangle)$	$A(\langle CTLQ \rangle \bar{\cup} \langle CTLQ \rangle)$	$E(\langle CTLQ \rangle \bar{\cup} \langle CTLQ \rangle)$
	$A(\langle CTLQ \rangle \bar{\mathcal{W}} \langle CTLQ \rangle)$	$E(\langle CTLQ \rangle \bar{\mathcal{W}} \langle CTLQ \rangle)$	$A(\langle CTLQ \rangle \bar{\cup} \langle CTLQ \rangle)$
	$E(\langle CTLQ \rangle \bar{\cup} \langle CTLQ \rangle)$	$A(\langle CTLQ \rangle \bar{\mathcal{W}} \langle CTLQ \rangle)$	$E(\langle CTLQ \rangle \bar{\mathcal{W}} \langle CTLQ \rangle)$

Table 3.2. Grammar recognized by the CTLQ parser.

results (that is, the ASTs) to do the additional validation of the presence of the placeholder.

In summary, if the string representing the CTL query is well-formed and contains at least one occurrence of the placeholder, the parser returns the corresponding AST, otherwise an exception is raised with an explicit error message. For example, the string "AG 'heat'" will lead to a `pytlq.NoPlaceholderError` exception, while the string "AG (? $\rightarrow$ 'heat'" will lead to a `pyparsing.ParseException` exception.

In practice, the query $AG (? \rightarrow AF \text{heat})$ is then parsed as (see Subsection 3.3.1 for a complete description of the structure of the ASTs):

```
AG(child=ImPLY(left=Placeholder(), right=AF(child=Atom(value='heat'))))
```

Once the parsing is done, we need to *check* that the query does belong to the fragment $CTLQ^x$. Indeed, in the context of this thesis, we only implemented the solving of CTL queries that belong to $CTLQ^x$.

3.2.2 Checking the Membership to Fragment $CTLQ^x$

The `checker` module provides the `check_ctlqx` function (together with the `non_terminal_ctlqx` auxiliary function) to verify whether a CTL query belongs to fragment $CTLQ^x$ or not.

According to Samer and Veith, before using this function, the AST-based CTL query must first be transformed in negation normal form. This transformation is handled

externally by the `negation_normal_form` utility function, which must be called before `check_ctlqx`. This choice has been made in order to separate the role of each function to keep a coherent API. Note that the `negation_normal_form` function performs the transformation in NNF on the entire AST (that is, it pushes the negation in the CTL subformulas as well, even if the latter does not have any influence on the production rules). Indeed, with this implementation, the function is more general and can be used to transform CTL formulas if needed.

The CTL query $AG (? \rightarrow AF \textit{heat})$ is therefore transformed in NNF before going through the verification process of the `check_ctlqx` function:

```
AG(child=Or(left=Not(child=Placeholder()),
            right=AF(child=Atom(value='heat'))))
```

The first step of the verification then consists of checking that there is only one occurrence of the placeholder in the query, because the production rules do not allow zero or more than one occurrence of the placeholder. Here, the query (in negation normal form) $AG (\neg ? \vee AF \textit{heat})$ does effectively contain only one occurrence of the placeholder. Note that since the negation normal form of $\varphi \leftrightarrow \psi$ is $(\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$ (that is, each argument is duplicated), CTL queries that contain this operator with the placeholder in one of its arguments are directly rejected at this step.

The second and final step consists of actually verifying that the CTL query effectively stands in the fragment $CTLQ^x$ such as defined in the production rules. This “sub-verification” is actually performed by the `non_terminal_ctlqx` auxiliary function, which returns the number of the non-terminal in which the query stands if it belongs to $CTLQ^x$, 0 otherwise. A number between 1 and 10 therefore means that the query belongs to the fragment, and more particularly to the language defined by the non-terminal whose number is returned. In order to find the non-terminal in which the query lies, we first create the list of the operators constituting the path from the root of the AST to the placeholder. Indeed, we only need to consider the operators that are directly related to the placeholder, and we may omit the rest of the AST. Then, we reverse the list to represent the path from the placeholder to the root of the AST, because the only starting points that are clearly identifiable in the production rules are the placeholder and its negation. Consequently, we check the membership of the query to $CTLQ^x$ starting from the placeholder, in a bottom-up fashion.

For example, the AST of the running example engenders the following list:

```
['Placeholder', 'Not', '_Or', 'AG']
```

Note that we chose to use strings to build this list in order to easily indicate in which argument of the binary operators the placeholder stands. This indication is made via the dedicated symbol `_` (that is, the underscore symbol) next to the operator

name (for example, herebefore, the placeholder stands in the left argument of the $\vee$ operator).

Then, we iterate over this list following the production rules of the fragment CTLQ^x to find if the query belongs to it. In the running example, we first encounter the placeholder (which is, naturally, always the case), so the query momentarily stands in the language defined by the non-terminal $\langle Q1 \rangle$. Then, we encounter a negation that is directly related to the placeholder, so the query stays in the language defined by the non-terminal $\langle Q1 \rangle$. The $\vee$ operator leaves things as they are (because $\langle Q1 \rangle ::= \star \vee \langle Q1 \rangle$). Finally, the **AG** operator moves the query in the language defined by the non-terminal $\langle Q8 \rangle$ (because $\langle Q8 \rangle ::= \text{AG } \langle Q1 \rangle$). The `non_terminal_ctlqx` function returns 8 and we know that the query **AG** ($? \rightarrow \text{AF } \textit{heat}$) belongs to CTLQ^8 , and therefore to CTLQ^x . The `check_ctlqx` function then finally returns *true*.

The query having passed the verification process, it can now be *solved* in the model of the microwave oven.

3.2.3 Solving CTL Queries

The `solver` module – which is the core of PyTLQ – provides the `solve_ctlqx` function to evaluate CTL queries that belong to fragment CTLQ^x .

The `solve_ctlqx` function calls the extended Chan algorithm with an AST-based CTL query that passes the previous verification and the initial states of the considered SMV model in order to get the unique set of solution states that represents an exact solution to the query in the model. Formally, if γ is any CTL query that belongs to CTLQ^x , $\mathcal{K}$ any Kripke structure, and R is the result of `solve_ctlqx`($\mathcal{K}$, γ), then the characteristic function of R is an exact solution to γ in $\mathcal{K}$.

The two functions composing the extended Chan algorithm (namely, FSOL and ESOL) and the three auxiliary sets (namely, $\mathcal{R}_\varphi$, $\mathcal{B}_\varphi^\gamma$ and $\mathcal{C}_\varphi^\gamma$) are almost verbatimly translated into Python code. Indeed, as stated in the previous chapter, only a slight change has been made to support the negation of the placeholder in the ESOL function.

The `solver` module is the one that uses – directly or indirectly (via utilities) – nearly all the PyNuSMV functionalities used in PyTLQ. Indeed, to use the extended Chan algorithm, we need a BDD representation of the SMV model and, especially, of its states. Furthermore, we need functions to manipulate these states. `pynusmv.glob` and `pynusmv.dd` modules provide those features. We also need the `eval_ctl_spec` and `fixpoint` functions, provided by `pynusmv.mc` and `pynusmv.utils` modules, to respectively retrieve the set of states in which a CTL formula φ is true (that is, $\llbracket \varphi \rrbracket$) and compute fixed points. Finally, we use the `pynusmv.prop` module to create PyNuSMV specifications from AST-based CTL formulas. Note that we do not directly manipulate Kripke structures with PyNuSMV. Instead, a finite state machine

(FSM) is built from the given SMV model, and various functions allow us to manipulate it and its states. For example, provided an FSM `fsm`, we can retrieve its initial states thanks to the function call `fsm.init`.

In the running example, the unique set of solution states of $\text{AG } (\neg ? \vee \text{AF } \textit{heat})$ at the initial states of the model of the microwave oven returned by the `solve_ctlqx` function is:

```
{{'close': 'TRUE', 'error': 'FALSE', 'heat': 'TRUE', 'start': 'FALSE'},
 {'close': 'TRUE', 'error': 'FALSE', 'heat': 'FALSE', 'start': 'TRUE' },
 {'close': 'TRUE', 'error': 'FALSE', 'heat': 'TRUE', 'start': 'TRUE' }}
```

If we suppose that we cannot analyse a graphical representation of the model, this solution is likely to be too complex to be easily understood. Moreover, for large SMV models, the set of solution states may be too large to be printed in the terminal. In such cases, one may want to *simplify* complex solutions.

3.2.4 Simplifying Solutions

The `simplifier` module provides the `simplify` and `project` functions. The former implements an adapted version of Chan's approximate conjunctive decomposition, and the latter – which is a more general function – implements the projection of a set of states on a list of variables of interest. These two functions may – optionally – be used to simplify a result obtained with the `solve_ctlqx` function.

First of all, the `project` function implements the projection of a set of states on a subset of the variables of the SMV model. Intuitively, this function may be compared to the projection of a propositional formula on a set of atomic propositions that we have described in Subsection 2.2.3. Practically, the `project` function enumerates all the possible values of the listed variables, in all states of the given set of states. For example, the projection of the set of solution states computed in the previous subsection on the set of variables $\{\textit{start}, \textit{close}\}$ gives:

```
(close = TRUE & start = TRUE) | (close = TRUE & start = FALSE)
```

For its part, the `simplify` function implements an adapted version of Chan's approximate conjunctive decomposition algorithm (see Algorithm 3). The adapted simplification algorithm operates as follows. It takes a BDD representing a set of states s (typically, the output of the `solve_ctlqx` function), a list of variables $\textit{vars}$ (by default, all the variables of the system), and an integer k with $0 < k \leq |\textit{vars}|$ as inputs. k denotes the maximum number of variables that must appear in the conjuncts of the simplification (by default, $k = 1$). The result of the simplification is accumulated in the list $\textit{result}$, and the simplification of s is defined as $\bigwedge \textit{result}$. Indeed, we cannot use the variable c for this purpose anymore, because it is represented

Algorithm 3 Adapted version of Chan’s approximate conjunctive decomposition.**Input:** BDD representing a set of states s **Input:** List $vars$ **Input:** Integer k with $0 < k \leq |vars|$

```

1:  $result \leftarrow []$ 
2:  $c \leftarrow true$ 
3: for  $j \leftarrow 1$  to  $k$  do
4:   for all  $Y \subseteq vars$  with  $|Y| = j$  do
5:      $r \leftarrow (\exists \overline{Y}. s) \downarrow c$ 
6:     if  $r \Leftrightarrow s$  then
7:        $result \leftarrow result + project(r, Y)$ 
8:       return  $result$ 
9:     end if
10:    if  $r \not\Leftrightarrow true$  then
11:       $c \leftarrow c \wedge r$ 
12:       $result \leftarrow result + project(r, Y)$ 
13:      if  $c \Leftrightarrow s$  then
14:        return  $result$ 
15:      end if
16:    end if
17:  end for
18: end for

```

Output: $result$ with $s \Rightarrow \bigwedge result$

as a BDD in PyNuSMV, and the latter only allows to visualize a BDD by enumerating all its states. Consequently, such representation will not help us to improve the understandability of the solution since it is already an enumeration of states. c then only becomes useful for the minimization of the candidate r (see Algorithm 3, line 5). Note that c here represents the conjunction of the candidate sets of states. Therefore, c is initialized with *true* because the empty conjunction is defined as such. Basically, the two nested loops of the algorithm search for useful sets of states (that is, sets of states that are equal to s or not equal to *true*) that contain only j atomic propositions, with increasing j up to k . Such sets of states – also in the form of BDDs – are computed in a similar way as in Chan’s simplification algorithm, namely as the projection of s onto a set Y of variables ($proj = \exists \overline{Y}. s$). More precisely, Y denotes sublists of $vars$ with $|Y| = j$, and $proj$ represents a set of states that contains only the variables in Y . In order to remove eventual redundant information, the result of the projection is minimized using the conjunction of the other sets of states already computed ($proj \downarrow c$). Formally, $p \wedge c \Leftrightarrow (p \downarrow c) \wedge c$. This minimization is still performed thanks to the *restrict* operator defined by Coudert et al. [24]. Each time a useful set of states is found, it is added to the conjunction c , and, in parallel, to the list $result$ via the **project** function in order to have a meaningful representation of the candidate (recall the BDD visualization issue in PyNuSMV). At the end, the algorithm outputs $result$. It is important to note that, in the case of $r \Leftrightarrow s$ (resp. $c \Leftrightarrow s$), it is useless to continue the algorithm since we found a (simpler) set of states (resp. a conjunction of (simpler) sets of states) that can represent the solution to its own. Continuing the algorithm would then lead to redundancy, we therefore directly return $result$ in those cases.

To illustrate PyTLQ simplification algorithm, consider the set of solution states computed in the previous subsection. With $k = 2$, and if we take all the variables of the system into account, this set of solution states is simplified as follows:

```
(close = TRUE) & (error = FALSE) & ((heat = TRUE & start = TRUE) |
                                     (heat = FALSE & start = TRUE) |
                                     (heat = TRUE & start = FALSE))
```

As already stated in the introduction and confirmed here, this output means that, to ensure that the microwave oven will eventually heat, the door must be closed, there must be no error, and either the start button is pushed or the oven already heats. In conclusion, thanks to the CTL query $\text{AG } (? \rightarrow \text{AF } \textit{heat})$, PyTLQ correctly discovered the following invariant in the model of the microwave oven: $\text{AG } ((\textit{close} \wedge \neg \textit{error} \wedge (\textit{start} \vee \textit{heat})) \rightarrow \text{AF } \textit{heat})$.

Note that the `simplifier` module also uses functions from the `pynusmv.dd` module in order to manipulate BDDs. Among others, this module provides the restrict ($\downarrow$) and the existential abstraction ($\exists \overline{Y}. s$) operators.

3.3 Implementation Details

In this section, we review important implementation details that have not been covered in previous sections, namely the structure of the abstract syntax trees and the standalone script of PyTLQ.

3.3.1 Abstract Syntax Trees

In order to be consistent with the tools that are distributed with PyNuSMV, we use ASTs to manipulate CTL queries.

The `ast` module of PyTLQ provides classes to represent such CTL formulas/queries as ASTs. An AST can be made up of 27 possible classes (namely, `Placeholder`, `TrueExp`, `FalseExp`, `Atom`, `Not`, `And`, `Or`, `ImPLY`, `Iff`, `AX`, `AF`, `AG`, `AU`, `AW`, `EX`, `EF`, `EG`, `EU`, `EW`, `AoU`, `AoW`, `AdU`, `AdW`, `EoU`, `EoW`, `EdU`, and `EdW`), each of which symbolizing a specific component of a CTL formula/query (clearly denoted by its name). Note that `oU` denotes the overlapping strong until operator, `oW` the overlapping weak until, `dU` denotes the disjoint strong until, and `dW` the disjoint weak until.

All the 27 classes are build as Python namedtuples, and extends the `AST` superclass. The choice of the namedtuple data structure to produce ASTs lies, once again, in the concern of consistency in regards to the other PyNuSMV tools. Indeed, all of them use namedtuples to build ASTs. Moreover, a namedtuple has several advantages: it is an easy to create, lightweight, and immutable data structure. Unfortunately for

us, as a namedtuple is basically a tuple with a name (and whose elements can have names as well), the comparison between two namedtuples then only takes care of the values of the tuple (that is, the namedtuple `AF(child=Placeholder())` is equal to the namedtuple `AG(child=Placeholder())`). For this reason, we implemented the `AST` superclass that redefines the equality comparison between namedtuples (by taking their names into account) in order to correctly compare the ASTs.

3.3.2 Standalone Script

We used the Click – which stands for “Command Line Interface Creation Kit” – package [48] to implement the command line interface (CLI) of PyTLQ with as little code as necessary. We chose this package because it automatically generates operational CLIs. We could therefore focus on the actual implementation of PyTLQ and not on the side-features of the CLI. For example, Click takes care of the usage instructions, and handles arguments and options (that is, optional arguments) intuitively.

The standalone script is the last place where we used PyNuSMV in PyTLQ. Indeed, this is where NuSMV is started, the SMV model is loaded, and the corresponding FSM is built (thanks to `pynusmv.init` and `pynusmv.glob` modules).

The usage of PyTLQ as a standalone script is given in Appendix A.

3.4 Limitations

PyTLQ has four main limitations:

1. Since the development of the main feature of PyTLQ – that is, the CTL query solver – entirely relies on Samer and Veith’s extended Chan algorithm, PyTLQ suffers from the main limitation outlined in their paper (that is, the CTL queries must belong to the $CTLQ^x$ fragment). Moreover, PyTLQ only considers CTL queries, and not queries formed with other temporal logics. Finally, PyTLQ only accepts queries with one occurrence of a single distinct placeholder.
2. The error management being a limitation in PyNuSMV, and PyParsing raising somewhat unclear exceptions, the standalone script of PyTLQ suffers from poor error explanations.
3. The projection and the adapted version of Chan’s approximate conjunctive decomposition algorithms cannot provide a simplified visualization of boolean formulas. For example, a typical output has the following form:

```
((heat = TRUE  & start = TRUE) |
 (heat = FALSE & start = TRUE) |
 (heat = TRUE  & start = FALSE))
```

Instead of being simply represented as:

```
(heat | start)
```

4. In addition to the previous limitation, these two algorithms do not take the `DEFINE` variables of the SMV models into account. `DEFINE` declarations are used to make descriptions more concise in SMV models, thanks to a symbol that is associated with a common expression (see NuSMV user manual [12, pp. 26-27]). PyNuSMV not allowing to retrieve those variables, we cannot consider them in the projection and simplification algorithms. Solutions are therefore not “projectable/restrictable” to such variables, and the algorithms might consequently lose a bit of their power.

Chapter 4

Applications

In addition to inferring temporal logic properties, temporal logic query solving can also be used to gather more information for the user in model checking. Indeed, this technique can be used to obtain an additional explanation when a checked property holds in the model, and diagnostic information when a checked property does not.

In Chapter 1, we already considered the case of a property that does not hold in the model of the microwave oven (see Section 1.4), and showed the efficiency of temporal logic query solving for gathering diagnostic information about the model.

Now, still in the microwave oven example, consider the invariant $\text{AG } (heat \rightarrow close)$, meaning that “if the oven heats, then the door is closed.” We know that this invariant holds in the model because it has been verified with a model checker. However, we would like to be sure that this invariant is the strongest one regarding to what is implied when the microwave oven heats. In order to do this, we can evaluate the query $\text{AG } (heat \rightarrow ?)$. The answer of a temporal logic query solver with this query and the initial states of the model of the microwave oven given as inputs is:

```
close & heat & !error
```

We are therefore informed of a stronger invariant, and can trivially explain that $heat \rightarrow close$ is invariant because $heat \rightarrow (close \wedge heat \wedge \neg error)$ is. This simple example shows that, even if a property holds, temporal logic query solving can help the user to learn more about a studied system.

In the remainder of this chapter, we report initial experiments of applying PyTLQ on SMV models coming from the NuSMV 2.5.4 [44] and Carnegie Mellon University’s (CMU) SMV 2.5.3 [23] distributions. (The source code of the used SMV models is also available in the `examples/` directory of the PyTLQ distribution.) We begin this chapter with the well-known simplified model of a print server in order to describe every step of temporal logic query solving with PyTLQ. Then, we experiment the technique on a large, complex and realistic model in order to show how PyTLQ can effectively help the user to better understand systems behaviors.

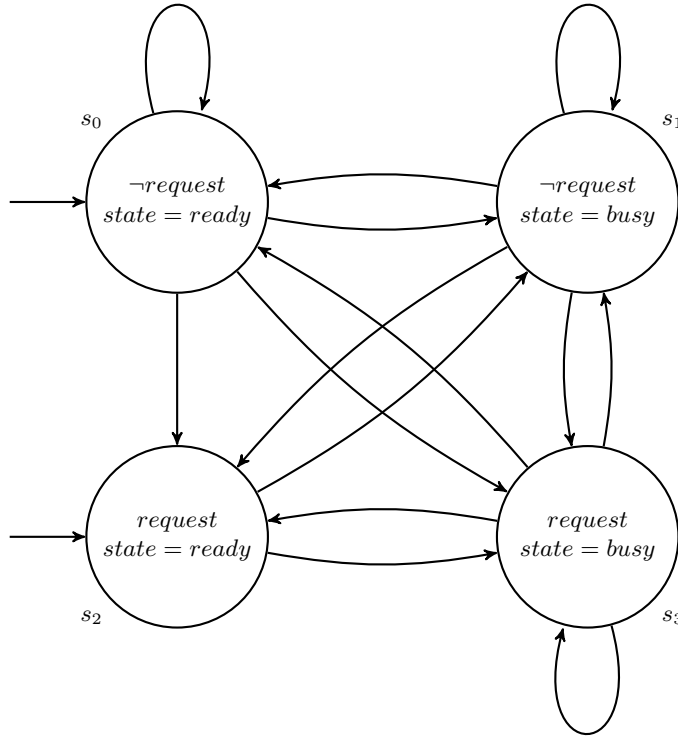


Figure 4.1. Simple print server example.

4.1 A Simple Print Server

The simple print server model coming from the NuSMV 2.5.4 distribution (see `examples/short.smv`), graphically depicted in Figure 4.1, consists of four states, including two initial states (namely, s_0 and s_2). In its initial states, the print server is in the state “ready”. If the print server is ready and it receives a request, it becomes busy. If it does not receive any request, the print server may be either in state “ready” or “busy”.

Suppose that we want to discover the strongest invariant of the system. As seen previously, such query is expressed as $\text{AG } ?$. We then launch PyTLQ with the following command:

```
$ pytlq examples/short.smv "AG ?"
```

And get the following unique set of solution states:

```
{{'request': 'TRUE', 'state': 'ready'},
 {'request': 'FALSE', 'state': 'ready'},
 {'request': 'TRUE', 'state': 'busy' },
 {'request': 'FALSE', 'state': 'busy' }}
```

This set means that the invariant $\text{AG } ((\text{request} \wedge \text{state} = \text{ready}) \vee (\neg \text{request} \wedge \text{state} = \text{ready}) \vee (\text{request} \wedge \text{state} = \text{busy}) \vee (\neg \text{request} \wedge \text{state} = \text{busy}))$ holds in the model.

Actually, this solution regroups all the reachable states of the model. This is the expected answer, but it does not help us much. So, let us simplify it. By considering all the variables of the model in the simplification algorithm, we get the following simplification:

```
((state = ready) | (state = busy))
```

This answer means that the algorithm found a variable (namely, *state*) that can represent the solution on its own. Actually, in such case, we can conclude that the other variables have no impact on the veracity of the solution. The solution can therefore be represented as its states projected on this variable, and not all the states of the solution. Note that, in this example, the simplification is non-deterministic (that is, we could have `((request = FALSE) | (request = TRUE))` as well). Thanks to the simplification algorithm of PyTLQ, we then discovered that the simple print server can be represented by only considering one of the two variables constituting its model.

We can verify this simplification by expressing the CTL specification `SPEC AG ((state = ready) | (state = busy))`, and checking it with the CTL model checker of PyNuSMV. As expected, the output is:

```
Specification AG ((state = ready) | (state = busy)) is True
```

In conclusion, this example is too simple to make the temporal logic query solver really helpful, but it still illustrates the usefulness of the simplification algorithm in the process of understanding a system behaviors.

4.2 A Cache Consistency Protocol

Now that we have seen how to use PyTLQ with a simple example, let us consider the model of the GIGAMAX cache consistency protocol. This is a famous model found in most of the SMV and NuSMV distributions. However, in this section, we focus on the SMV model coming from the CMU's SMV 2.5.3 distribution (see `examples/gigamax.smv`).

The model of the cache consistency protocol consists of 3048 reachable states, and has more than 15×10^6 initial states. Without going into details, the latter is composed of three processors p_0 , p_1 and p_2 , and a shared global memory m . The purpose of a cache consistency protocol is to provide the illusion to the programmer of a distributed computer that all the processors in the system have access to the shared global memory, despite the fact that the physical storage is distributed [42]. In order to provide this illusion, each processor is provided with a local cache. The GIGAMAX protocol is therefore intended to ensure that these local caches remain consistent between each other.

The analysis of the model of the GIGAMAX cache consistency protocol has already been made by Chan in his paper [13] as application of his work. This section therefore aims to compare Chan's results with those obtained with PyTLQ in order to determine if we come to the same conclusions. Yet, we cannot perform exactly the same analysis as Chan. Indeed, the limitation of the PyTLQ simplification algorithm regarding to the **DEFINE** variables prevents us from manipulating the *readable* and *writable* variables of the model. Instead, we analyse the latter in relation to the *state* and *waiting* variables, since those are used to define the *readable* and *writable* ones:

DEFINE

```
readable := ((state = shared) | (state = owned)) & !waiting;
writable := (state = owned) & !waiting;
```

The author of the model defines three temporal logic properties:

1. $AG\ EF\ (p_0.readable)$
2. $AG\ EF\ (p_0.writable)$
3. $AG\ \neg(p_0.writable \wedge p_1.writable)$

However, in order to learn more about it, we would like to discover the strongest invariant of the system. We therefore express the same CTL query as Chan, namely $AG\ ?$, and take into account only the variables $p_i.state$ and $p_i.waiting$ for $i \in \{0, 1, 2\}$. By considering all singleton sets of the variables of interest (that is, $k = 1$ in Algorithm 3), the simplification algorithm of PyTLQ returns the following propositional formula:

```
(p2.state = invalid) & (p2.waiting = FALSE)
```

From the latter, we can straightforwardly infer – with the definition of *readable* and *writable* variables – that the following invariants hold in the system:

$$AG\ \neg p_2.readable$$

$$AG\ \neg p_2.writable$$

Encouragingly, those two invariants were also found by Chan. Yet, such invariants are surprising since they mean that the processor p_2 is never readable or writable in the system. After a deep examination of the SMV model, we come to the same conclusion as Chan: there is a typographical error in the SMV model. At line 158 of the `gigamax.smv` file, instead of:

```
p0.cmd = idle & p1.cmd = idle & m.cmd = idle : p0.cmd;
```

We should find:

```
p0.cmd = idle & p1.cmd = idle & m.cmd = idle : p2.cmd;
```

As expected, with this error fixed, the two surprising invariants no longer hold in the model, and the latter has now a correct behavior regarding to the specifications of the protocol.

The limitation regarding the **DEFINE** variables makes Chan's other finding (that is, the invariant $\text{AG} ((p_i.\text{readable} \wedge p_j.\text{readable}) \rightarrow \neg p_k.\text{writable})$ holds for every distinct $i, j, k \in \{0, 1, 2\}$) harder to find with PyTLQ. Nevertheless, we may argue that the critical issue present in the model of the GIGAMAX cache consistency protocol – as defined in the CMU's SMV 2.5.3 distribution – has been detected quickly.

In conclusion, this example allows us to confirm the applicability of PyTLQ on a concrete SMV model, and the correctness of the results obtained with PyTLQ in comparison with Chan's.

Chapter 5

Conclusion and Perspectives

In a world where the technology around us takes an increasing part in our lives, it is essential to ensure the correctness of the computing devices that we use every day. In addition to model checking – a well-known technique used to verify, in an automatic way, that a given system satisfies or not given properties – temporal logic query solving allows to help designers of such hardware and software systems to better understand their behaviors (whether they satisfy the eventual defined properties or not).

Throughout this thesis, we presented the concept of temporal logic query solving – an extension of model checking whose principal aim is to understand a system as opposed to merely verifying its correctness. Our work is based on Chan’s paper [13], as corrected and extended by Samer and Veith [51, 53]. In order to fit the needs of a practical implementation of temporal logic queries, we proposed the extensions of the syntactic fragment CTLQ^x , the extended Chan algorithm, and the Chan’s simplification algorithm.

All these theoretical aspects then led to the development of PyTLQ, an original Python package for solving temporal logic queries – as defined by Chan and corrected/extended by Samer and Veith – using the BDD-based model checking functionalities of the PyNuSMV library [11]. PyTLQ allowed us to evaluate temporal logic query solving on existing SMV models. The initial experience of applying PyTLQ on concrete SMV models showed the applicability of the package, and demonstrated the discovery of a faulty behaviors in the complex SMV model of the GIGAMAX cache consistency protocol coming from the CMU’s SMV 2.5.3 distribution [23].

However, temporal logic query solving is not a magic bullet. It may indeed help the user to better understand system behaviors, but additional work will still be necessary to understand a considered system down to the smallest detail.

Our objective of implementing a temporal logic query solver has been achieved, but it is obvious that such a project is never entirely finished. New ideas and optimizations will always come up to improve its quality and applicability. For example, a logical

follow-up of our work would be to address the limitations of PyTLQ. In particular, it could be interesting to clear up the `DEFINE` variables issue, as well as the limitation related to the representation of boolean formulas. Furthermore, in order to extend the temporal logic queries to the Linear Temporal Logic (LTL), one could investigate Samer and Veith's papers about LTL query solving [52, 54]. Finally, in a long-term perspective, we could imagine a deep investigation of the related works about temporal logic query solving in order to decide their applicability in Python using PyNuSMV.

Bibliography

- [1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.
- [2] B. Aminof, T. Ball, and O. Kupferman. Reasoning About Systems with Transition Fairness. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2005.
- [3] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer Berlin Heidelberg, 2000.
- [4] D. M. Beazley. SWIG: An Easy To Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.
- [5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic Model Checking Without BDDs*. Springer, 1999.
- [6] G. Bruns and P. Godefroid. Temporal Logic Query Checking. In *Logic in Computer Science*, pages 409–417, 2001.
- [7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [8] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.
- [9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Logic in Computer Science*, pages 428–439. IEEE, 1990.
- [10] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

- [11] S. Busard and C. Pecheur. PyNuSMV: NuSMV as a Python Library. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 453–458. Springer Berlin Heidelberg, 2013.
- [12] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV 2.5 User Manual. <http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf>. [Online; accessed 14-August-2015].
- [13] W. Chan. Temporal-Logic Queries. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer Berlin Heidelberg, 2000.
- [14] M. Chechik and A. Gurfinkel. TLQSolver: A Temporal Logic Query Checker. In J. Hunt, Warren A. and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 210–214. Springer Berlin Heidelberg, 2003.
- [15] M. Chechik, S. Easterbrook, and V. Petrovykh. Model-Checking Over Multi-Valued Logics. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 72–98. Springer Berlin Heidelberg, 2001.
- [16] M. Chechik, A. Gurfinkel, and B. Devereux. xChek: A Multi-valued Model-Checker. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 505–509. Springer Berlin Heidelberg, 2002.
- [17] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer Berlin Heidelberg, 1999.
- [18] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002.
- [19] E. Clarke. The Birth of Model Checking. *25 Years of Model Checking*, pages 1–26, 2008.
- [20] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.
- [21] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In D. Kozen, editor, *Logics*

- of Programs, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.
- [22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [23] CMU’s SMV 2.5.3 Distribution. <http://cs.cmu.edu/~modelcheck/smv/smv.r2.5.3.1d.tar.gz>. [Online; accessed 14-August-2015].
- [24] O. Coudert, C. Berthet, and J. C. Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer Berlin Heidelberg, 1990.
- [25] M. Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2), 1997.
- [26] E. A. Emerson and E. M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin Heidelberg, 1980.
- [27] M. Gheorghiu and A. Gurfinkel. Tlq: A Query Solver for States. In *Tools and Posters Session at the 14th International Symposium on Formal Methods, FM 2006, Hamilton, Canada, August 21-27, 2006*.
- [28] M. Gheorghiu, A. Gurfinkel, and M. Chechik. Finding State Solutions to Temporal Logic Queries. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 273–292. Springer Berlin Heidelberg, 2007.
- [29] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032. Springer Heidelberg, 1996.
- [30] A. Gurfinkel and M. Chechik. Temporal Logic Query Checking through Multi-Valued Model Checking. Technical report, 2002.
- [31] A. Gurfinkel, B. Devereux, and M. Chechik. Model Exploration with Temporal Logic Query Checking. *ACM SIGSOFT Software Engineering Notes*, 27(6): 139–148, 2002.
- [32] A. Gurfinkel, M. Chechik, and B. Devereux. Temporal Logic Query Checking: A Tool for Model Exploration. *IEEE Transactions on Software Engineering*, 29(10):898–914, 2003.
- [33] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.

- [34] S. Hornus and P. Schnoebelen. On Solving Temporal Logic Queries. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin Heidelberg, 2002.
- [35] O. Kupferman, M. Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.
- [36] M. Lacchia. Radon Documentation. <https://radon.readthedocs.org>. [Online; accessed 14-August-2015].
- [37] C.-Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.
- [38] N. Leveson. Medical Devices: The Therac-25. *Appendix of: Safeware: System Safety and Computers*, 1995.
- [39] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Science & Business Media, 2012.
- [40] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.
- [41] P. T. McGuire. PyParsing Wiki. <https://pyparsing.wikispaces.com>. [Online; accessed 14-August-2015].
- [42] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.
- [43] A. Montanari. An Introduction to Model Checking. <https://users.dimi.uniud.it/~angelo.montanari/MCclasses.pdf>. [Online; accessed 14-August-2015].
- [44] NuSMV 2.5.4 Distribution. <http://nusmv.fbk.eu/distrib/NuSMV-2.5.4.tar.gz>. [Online; accessed 14-August-2015].
- [45] P. Oman and J. Hagemeister. Metrics for Assessing a Software System’s Maintainability. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 337–344. IEEE, 1992.
- [46] J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982.
- [47] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.
- [48] A. Ronacher. Click Documentation. <http://click.pocoo.org>. [Online; accessed 14-August-2015].

- [49] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.
- [50] M. Samer. *Reasoning about Specifications in Model Checking*. PhD thesis, TU Vienna, 2004.
- [51] M. Samer and H. Veith. Validity of CTL Queries Revisited. In M. Baaz and J. A. Makowsky, editors, *Computer Science Logic*, volume 2803 of *Lecture Notes in Computer Science*, pages 470–483. Springer Berlin Heidelberg, 2003.
- [52] M. Samer and H. Veith. A Syntactic Characterization of Distributive LTL Queries. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 1099–1110. Springer Berlin Heidelberg, 2004.
- [53] M. Samer and H. Veith. Deterministic CTL Query Solving. In *Temporal Representation and Reasoning*, pages 156–165, 2005.
- [54] M. Samer and H. Veith. On the Distributivity of LTL Specifications. *ACM Transactions on Computational Logic*, 11(3):20, 2010.
- [55] G. Tasse. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology, RTI Project*, 7007 (011), 2002.
- [56] The GNU Lesser General Public License. <http://gnu.org/licenses/old-licenses/lgpl-2.1.html>. [Online; accessed 14-August-2015].
- [57] The Open Source Organization. <http://opensource.org>. [Online; accessed 14-August-2015].
- [58] A. van Deursen. Think Twice Before Using the “Maintainability Index”. <http://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index>. [Online; accessed 14-August-2015].
- [59] G. van Rossum, B. Warsaw, and N. Coghlan. Style Guide for Python Code. <https://python.org/dev/peps/pep-0008>. [Online; accessed 14-August-2015].
- [60] Verifysoft. Halstead Metrics – Measurement of Halstead Metrics with Testwell CMT++ and CMTJava (Complexity Measures Tool). http://verifysoft.com/en_halstead_metrics.html. [Online; accessed 14-August-2015].
- [61] Wikipedia. Binary Decision Diagram – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Binary_decision_diagram&oldid=648110417, 2015. [Online; accessed 14-August-2015].
- [62] Wikipedia. Boolean Algebra – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Boolean_algebra&oldid=673677592, 2015. [Online; accessed 14-August-2015].

- [63] Wikipedia. Cyclomatic Complexity – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=672406586, 2015. [Online; accessed 14-August-2015].
- [64] Wikipedia. Combinatorial Explosion – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Combinatorial_explosion&oldid=663497580, 2015. [Online; accessed 14-August-2015].
- [65] Wikipedia. EXPTIME – Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=EXPTIME&oldid=674305975>, 2015. [Online; accessed 14-August-2015].
- [66] Wikipedia. NP (Complexity) – Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=NP_\(complexity\)&oldid=673470694](https://en.wikipedia.org/w/index.php?title=NP_(complexity)&oldid=673470694), 2015. [Online; accessed 14-August-2015].
- [67] Wikipedia. Temporal Logic – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Temporal_logic&oldid=663677277, 2015. [Online; accessed 14-August-2015].
- [68] T. Willemse. Algorithms for Model Checking (2IW55) – Lecture 2: Symbolic Model Checking for CTL. <http://www.win.tue.nl/~timw/downloads/amc2014/lecture2.pdf>. [Online; accessed 14-August-2015].

Appendix A

User Manual

PyTLQ is an original Python package for solving temporal logic queries, as defined by Chan [13] and corrected/extended by Samer and Veith [51, 53]. Such queries have the form of CTL specifications and must belong to the syntactic fragment CTLQ^x.

To run, PyTLQ needs three external packages:

- Click (<https://pypi.python.org/pypi/click>) for the command-line interface.
- PyParsing (<https://pypi.python.org/pypi/pyparsing/2.0.3>) for the parsing of CTL queries.
- PyNuSMV (<http://lvl.info.ucl.ac.be/Tools/PyNuSMV>) for implementing the solving and simplification algorithms.

A.1 Input Language

The syntax of CTL queries recognized by PyTLQ is as follows:

```
ctl_query ::= ?                -- placeholder
            | True              -- true constant
            | False             -- false constant
            | 'atom'            -- atomic proposition
            | ( ctl_query )      -- parentheses
            | ~ ctl_query        -- logical not
            | ctl_query & ctl_query -- logical and
            | ctl_query | ctl_query -- logical or
            | ctl_query -> ctl_query -- logical implies
            | ctl_query <-> ctl_query -- logical equivalence
            | EX ctl_query        -- exists next state
            | EG ctl_query        -- exists globally
            | EF ctl_query        -- exists eventually
```

```

| E [ ctl_query U ctl_query ]      -- exists until
| E [ ctl_query oU ctl_query ]     -- exists overlapping until
| E [ ctl_query dU ctl_query ]     -- exists disjoint until
| E [ ctl_query W ctl_query ]      -- exists weak until
| E [ ctl_query oW ctl_query ]     -- exists overlapping weak until
| E [ ctl_query dW ctl_query ]     -- exists disjoint weak until
| AX ctl_query                     -- forall next state
| AG ctl_query                     -- forall globally
| AF ctl_query                     -- forall eventually
| A [ ctl_query U ctl_query ]      -- forall until
| A [ ctl_query oU ctl_query ]     -- forall overlapping until
| A [ ctl_query dU ctl_query ]     -- forall disjoint until
| A [ ctl_query W ctl_query ]      -- forall weak until
| A [ ctl_query oW ctl_query ]     -- forall overlapping weak until
| A [ ctl_query dW ctl_query ]     -- forall disjoint weak until

```

Note that the fragment CTLQ^x does not allow all these operators (see Table 2.1).

A.2 Installation

Remark. Make sure you are using Python 3 before installing PyTLQ.

Simply use the following command to install PyTLQ on your computer:

```
$ pip install https://github.com/sthibert/PyTLQ/zipball/master
```

Note that Click and PyParsing are *automatically* installed during the installation of PyTLQ, but PyNuSMV must be *manually* installed before using PyTLQ.

A.3 Usage

PyTLQ is meant to be called from the command line. The usage is:

```
$ pytlq <model_path> <query> [--order <order_path>]
```

pytlq command needs two arguments: `model_path` and `query`, and accepts three options (that is, optional arguments): `--order`, `--help`, and `--version`.

- `model_path` represents the path to the SMV model you want to analyse.
- `query` represents your CTL query as a string that follows the input language defined in Section A.1.

- `--order` is necessary when an order file (`.ord`) is provided with the considered SMV model (for efficiency reasons). This option requires the path to the order file.
- `--help` displays the usage instructions.
- `--version` displays the version of PyTLQ.

This command computes the unique set of solution states that represents an exact solution to `query` in the system defined in `model_path` (if there is one). Then, you have the choice of projecting the solution on a subset of the variables of the system, simplifying the solution thanks to an approximate conjunctive decomposition, or you can quit PyTLQ.

A.4 Application Programming Interface

PyTLQ is composed of several modules, each one proposing some functionalities:

- `ast` defines classes to represent CTL formulas/queries as abstract syntax trees.
- `parser` provides functions to parse strings representing CTL queries.
- `checker` provides functions to check that CTL queries belong to syntactic fragment $CTLQ^x$.
- `solver` provides functions to solve CTL queries that belong to fragment $CTLQ^x$.
- `simplifier` provides functions to simplify solutions returned by the solver.
- `exception` groups all the PyTLQ-related exceptions.
- `utils` provides classes and functions used by PyTLQ internals, and utilities for manipulating CTL formulas/queries.

A.4.1 The `ast` module

The `pytlq.ast` module provides 27 classes: `Placeholder`, `TrueExp`, `FalseExp`, `Atom`, `Not`, `And`, `Or`, `ImPLY`, `Iff`, `AX`, `AF`, `AG`, `AU`, `AW`, `EX`, `EF`, `EG`, `EU`, `EW`, `AoU`, `AoW`, `AdU`, `AdW`, `EoU`, `EoW`, `EdU`, and `EdW`, to build ASTs representing CTL formulas/queries.

A.4.2 The `parser` module

The `pytlq.parser` module provides the `parse_ctlq` function to parse strings representing CTL queries.

```
pytlq.parser.parse_ctlq(query)
```

Parse query and return its corresponding AST.

Parameters:

`query` – a string representing a CTL query (that is, a CTL formula where some subformulas are replaced by the special symbol `?`, called placeholder)

Returns: an AST-based CTL query

Raise: a `pyparsing.ParseException` if a parsing error occurs

Raise: a `pytlq.exception.NoPlaceholderError` if `query` does not contain any placeholder

Note: The parser uses `pytlq.ast` module’s classes to build ASTs.

A.4.3 The checker module

The `pytlq.checker` module provides the `check_ctlqx` and `non_terminal_ctlqx` functions to check that CTL queries belong to syntactic fragment $CTLQ^x$.

```
pytlq.checker.check_ctlqx(query)
```

Check that `query` belongs to fragment $CTLQ^x$.

Parameters:

`query` – an AST-based CTL query *in negation normal form*

Returns: True if `query` belongs to fragment $CTLQ^x$, False otherwise

Note: See `pytlq.utils.negation_normal_form()` for the transformation in negation normal form.

```
pytlq.checker.non_terminal_ctlqx(query)
```

Return the number of the non-terminal (defined in the production rules, adapted from Samer and Veith’s definition at TIME’05) in which `query` stands if `query` belongs to fragment $CTLQ^x$, 0 otherwise.

Parameters:

`query` – an AST-based CTL query *with exactly one occurrence of the placeholder*

Returns: a number between 1 and 10 (representing a non-terminal) if `query` belongs to fragment $CTLQ^x$, 0 otherwise

Note: See `pytlq.utils.count_placeholders()` for the verification that `query` contains exactly one occurrence of the placeholder.

A.4.4 The solver module

The `pytlq.solver` module provides the `solve_ctlqx` function to solve CTL queries that belong to fragment $CTLQ^x$.

```
pytlq.solver.solve_ctlqx(fsm, query)
```

Compute the unique set of solution states of `fsm` to `query` at the initial states of `fsm`, as defined by Samer and Veith at TIME'05.

Parameters:

`fsm` – the concerned FSM

`query` – an AST-based CTL query *that belongs to fragment $CTLQ^x$*

Returns: the unique set of solution states of `fsm` to `query` at the initial states of `fsm`

Return type: `pynusmv.dd.BDD`

Note: The characteristic function of the output is an exact solution to `query` in `fsm`.

Note: See `pytlq.checker.check_ctlqx()` for the verification that `query` belongs to fragment $CTLQ^x$.

A.4.5 The simplifier module

The `pytlq.simplifier` module provides the `project` and `simplify` functions to simplify the solutions returned by the solver.

```
pytlq.simplifier.project(fsm, states, variables=None)
```

Project `states` on `variables` (in other words, enumerate all the possible values of the variables in `variables`, in all states of `states`).

Parameters:

`fsm` – the concerned FSM

`states` – a set of states

`variables` – the list of variables on which `states` is projected (by default: all the variables)

Returns: a String representing the projection of `states` on the variables of `variables`

Raise: a `pytlq.exception.VariableNotInModelError` if a variable of `variables` is not present in the model

```
pytlq.simplifier.simplify(fsm, states, maximum=1, variables=None)
```

Compute the approximate conjunctive decomposition of `states`.

Parameters:

`fsm` – the concerned FSM

`states` – a set of states

`maximum` – the maximum number of variables that must appear in the conjuncts of the approximate conjunctive decomposition (by default: 1)

`variables` – the list of variables to which the solution is restricted (by default: all the variables)

Returns: a String representing the approximate conjunctive decomposition of `states`, if a simplification is possible

Raise: a `pytlq.exception.ValueOutOfBoundsError` if `maximum` is not in the correct bounds

Raise: a `pytlq.exception.VariableNotInModelError` if a variable of `variables` is not present in the model

Note: This function is adapted from Chan’s approximate conjunctive decomposition of a propositional formula, defined at CAV 2000.

A.4.6 The exception module

The `pytlq.exception` module gathers the `PyTLQError`, `NoPlaceholderError`, `ValueOutOfBoundsError`, and `VariableNotInModelError` exceptions.

```
Exception pytlq.exception.PyTLQError
```

Base class for PyTLQ exceptions.

```
Exception pytlq.exception.NoPlaceholderError
```

Exception raised when there is no placeholder in CTL query.

```
Exception pytlq.exception.ValueOutOfBoundsError
```

Exception raised when the input value is out of bounds.

```
Exception pytlq.exception.VariableNotInModelError
```

Exception raised when a variable is not present in the model.

A.4.7 The utils module

The `pytlq.utils` module provides classes and functions used by PyTLQ internals, and utilities for manipulating CTL formulas/queries.

```
Class pytlq.utils.HashableDict
```

Define a hashable dictionary.

```
pytlq.utils.ast_to_spec(ast)
```

Return a PyNuSMV specification representing `ast`.

Parameters:

`ast` – an AST-based CTL formula

Returns: a PyNuSMV specification representing `ast`

Return type: `pynusmv.prop.Spec`

Raise: a `NotImplementedError` if an operator is not implemented

```
pytlq.utils.bdd_to_set(fsm, bdd)
```

Return a Set representing `bdd`.

Parameters:

`fsm` – the concerned FSM

`bdd` – a `pynusmv.dd.BDD` representing a set of states

Returns: a Set representing `bdd`

```
pytlq.utils.negation_normal_form(ast)
```

Transform `ast` in negation normal form.

Parameters:

`ast` – an AST-based CTL formula/query

Returns: an AST-based CTL formula/query in negation normal form

Raise: a `NotImplementedError` if an operator is not implemented

```
pytlq.utils.replace_placeholder(query, formula)
```

Replace all the occurrences of the placeholder of `query` with `formula`.

Parameters:

`query` – an AST-based CTL query

`formula` – an AST-based CTL formula

Returns: an AST-based CTL formula

Raise: a `NotImplementedError` if an operator is not implemented

```
pytlq.utils.count_placeholders(query, counter=0)
```

Count how many occurrences of the placeholder there are in `query`.

Parameters:

`query` – an AST-based CTL query
`counter` – the counter (an accumulator variable)

Returns: the number of occurrences of the placeholder in `query`

Raise: a `NotImplementedError` if an operator is not implemented

```
pytlq.utils.path_to_placeholder(query)
```

Compute the path from the root of `query` to the placeholder.

Parameters:

`query` – an AST-based CTL query *with exactly one occurrence of the placeholder*

Returns: a list of strings representing the path from the root of the AST to the placeholder

Raise: a `NotImplementedError` if an operator is not implemented

Note: See `pytlq.utils.count_placeholders()` for the verification that `query` contains exactly one occurrence of the placeholder.

Appendix B

Source Code Metrics

This chapter presents some metrics of the source code of PyTLQ, collected with three Python tools:

- PyLint (<https://pypi.python.org/pypi/pylint/1.4.4>) looks for programming errors and code smells, and helps enforcing the PEP 8 [59] coding standard.
- Radon (<https://pypi.python.org/pypi/radon/1.2.2>) computes various metrics from the source code such as raw metrics, cyclomatic complexity, and maintainability index.
- PyTest Coverage Plugin (<https://pypi.python.org/pypi/pytest-cov>) generates tests coverage reports.

B.1 Coding Standard

Used command:

```
$ pylint pytlq/ --ignore=tests
```

PyLint correctly detected the 9 modules (note that it takes the `__init__` module into account), 33 classes, 33 methods, and 27 functions constituting PyTLQ. The analysis revealed that all of them are at 100% documented, and none of them have bad names (following PEP 8). Furthermore, PyLint did not detect any duplicated lines in the source code of PyTLQ.

Regarding to the coding standard per se, PyLint returned:

- 2 warning messages, due to a `global` statement in the `parser` module and a too general exception catching in the standalone script.

- 13 convention messages, principally due to too short variable names in the `parser` and `solver` modules.
- 13 refactor messages, due to too many statements, return statements, branches, and local variables in the complex functions of PyTLQ (such as the `negation_normal_form` and `non_terminal_ctlqx` functions).

However, we consider these 28 messages as minor warnings, the 2 warning messages being necessary for the good functioning of PyTLQ, the 13 convention messages not hurting the understandability of the code, and the 13 refactor messages resulting from the complexity of the problem at hand.

Finally, according to PyLint, the source code of PyTLQ has a global evaluation of 9.69/10.

B.2 Raw Metrics

Used command:

```
$ radon raw pytlq/ --ignore=tests --exclude=pytlq/__init__.py -s
```

According to Radon, PyTLQ consists of 1952 lines of codes (LOC), 1041 logical lines of code (LLOC), 1646 source lines of code (SLOC), 222 line comments (COM), 290 lines of multi-line comments (M-COM), and 306 blank lines (BL). The specific metrics of each PyTLQ module are given in Figure B.1. Note that $LOC = SLOC + BL$, and a logical line of code is a line of code that contains exactly one statement.

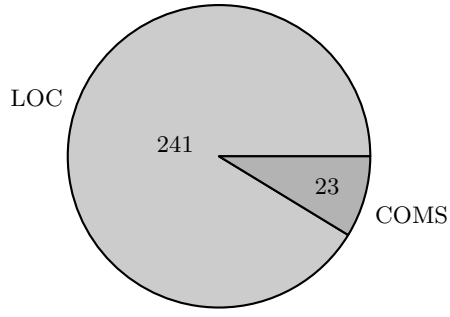
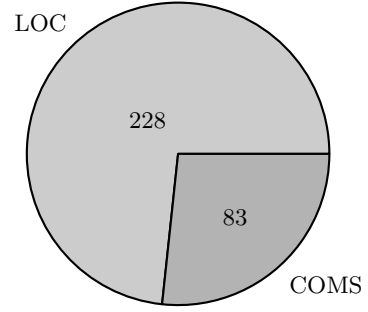
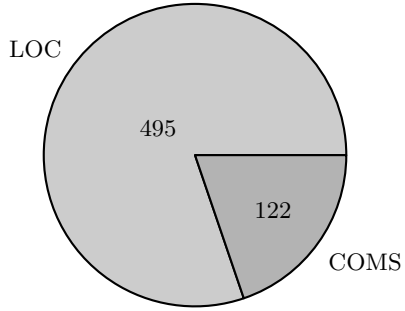
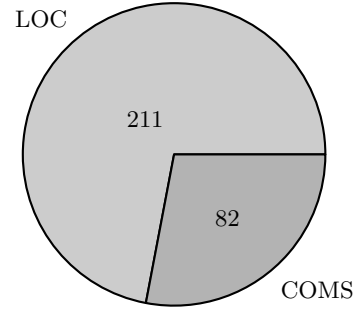
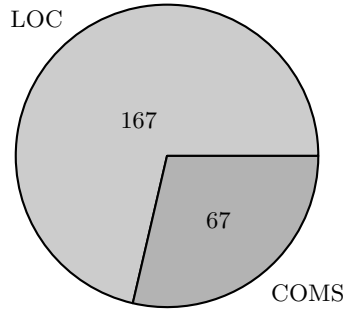
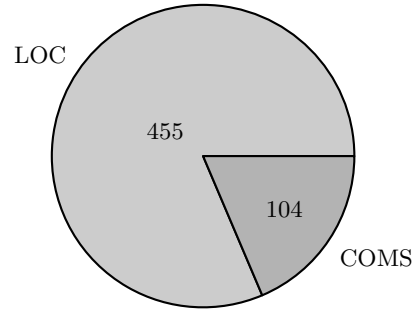
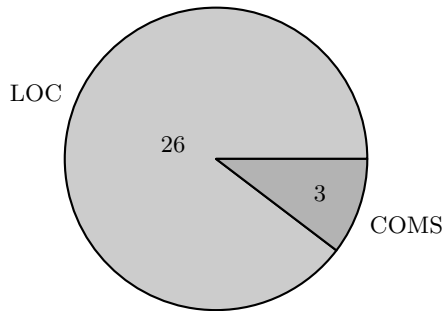
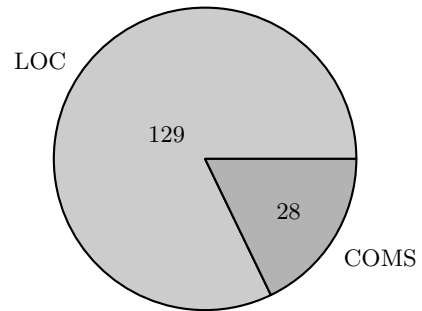
The important difference between LOC and LLOC metrics comes from the PEP 8 coding standard. Particularly, the latter recommends to limit all lines to a maximum of 79 characters [59]. Long lines are therefore divided into multiple shorter lines that do not always contain a statement.

The source code of PyTLQ has an average of 26.23% of comment lines (in relation to the number of lines of code). Apart from the `ast` and `exception` modules, this percentage stands above 20% for each module.

B.3 Cyclomatic Complexity

Used command:

```
$ radon cc pytlq/ --ignore=tests --exclude=pytlq/__init__.py -a -s
```

(a) `ast` module. LLOC = 126.(b) `parser` module. LLOC = 132.(c) `checker` module. LLOC = 351.(d) `solver` module. LLOC = 86.(e) `simplifier` module. LLOC = 59.(f) `utils` module. LLOC = 207.(g) `exception` module. LLOC = 14.(h) `pytlq` module. LLOC = 66.**Figure B.1.** Raw metrics of the PyTLQ modules. COMS = COM + M-COM.

Construct	Effect on CC	Reasoning
<code>if</code>	+1	An <code>if</code> statement is a single decision.
<code>elif</code>	+1	The <code>elif</code> statement adds another decision.
<code>else</code>	+0	The <code>else</code> statement does not cause a new decision. The decision is at the <code>if</code> .
<code>for</code>	+1	There is a decision at the start of the loop.
<code>while</code>	+1	There is a decision at the <code>while</code> statement.
<code>except</code>	+1	Each <code>except</code> branch adds a new conditional path of execution.
<code>finally</code>	+0	The <code>finally</code> block is unconditionally executed.
<code>with</code>	+1	The <code>with</code> statement roughly corresponds to a <code>try/except</code> block (see PEP 343 for details).
<code>assert</code>	+1	The <code>assert</code> statement internally roughly equals a conditional statement.
Comprehension	+1	A list/set/dictionary comprehension of generator expression is equivalent to a <code>for</code> loop.
Lambda	+1	A <code>lambda</code> function is a regular function.
Boolean Operator	+1	Every boolean operator (<code>and</code> , <code>or</code>) adds a decision point.

Table B.1. Effects of the statements on the cyclomatic complexity (CC), taken from [36].

The cyclomatic complexity corresponds to the number of linearly independent paths through the source code of a program. This software metric has been proposed by McCabe [40] in 1976, and is used to indicate the complexity of a program [63]. Basically, a high cyclomatic complexity (that is, greater than 20) indicates a complex block⁴ which may be error-prone, and a low cyclomatic complexity (that is, between 1 and 20) indicates a simple, stable block.

Radon inspects the abstract syntax tree of the input Python program to compute the cyclomatic complexity. The effects of the statements on the result is given in Table B.1.

Radon analyzed the 93 blocks of PyTLQ, and their average cyclomatic complexity is evaluated to 4.71.

The two functions that have the worst results are the `non_terminal_ctlqx` and `negation_normal_form` functions, with cyclomatic complexities of, respectively, 170 and 57. Follow the `ast_to_spec` and `_e_sol` functions with a cyclomatic complexity

⁴ A “block” here denotes a class, function, or method.

of 27. (Note that the latter is the private function that implements the ESOL function of the extended Chan algorithm.) All the other blocks have cyclomatic complexities below 20.

Remark. We must use this metric carefully because measuring an abstract concept such as the complexity of a program with a single simple measurement can lead to oversimplified conclusions.

B.4 Maintainability Index

Used command:

```
$ radon mi pytlq/ --ignore=tests --exclude=pytlq/__init__.py -s
```

The maintainability index measures how maintainable (that is, easy to support and change) the source code of a program is [36]. This software metric has been proposed by Oman and Hagemeister [45] in 1992, and consists of a blend of several metrics: the Halstead's volume⁵ (HV), the cyclomatic complexity (CC), the number of source lines of code (SLOC), and the percentage of comments in radians (PCOM). Basically, a high maintainability index (that is, between 20 and 100) indicates a high maintainability of the code, and a low maintainability index (that is, between 0 and 19) indicates a low maintainability of the code.

Radon uses the following formula to compute the maintainability index of the input Python program (adapted from [36]):

$$MI = \max \left[0, 100 \frac{171 - 5.2 \ln HV - 0.23CC - 16.2 \ln SLOC + 50 \sin \left(\sqrt{2.4PCOM} \right)}{171} \right]$$

Radon analyzed the 8 modules of PyTLQ, and their average maintainability index is evaluated to 60.94.

The module that has the worst maintainability index is the **checker** module (namely, 22.96), most likely because of the **non_terminal_ctlqx** function. The module that has the best maintainability index is the **exception** module (namely, 100), because it is the simplest one (it contains only four exception classes). All the other modules have a maintainability index that lies between 40 and 70.

Remark. Just as for the cyclomatic complexity, we must use this metric carefully because it is still very experimental. Moreover, there is no clear explanation about the used formula, which is computed over averaged metrics confounded by size [58].

⁵ In substance, the Halstead's volume describes the size of the implementation of an algorithm. Its computation is based on the number of operations performed and operands handled in the algorithm [60]. This metric has been proposed by Halstead in 1977 [33].

B.5 Tests Coverage

Used command:

```
$ py.test --cov pytlq/ --cov-report term
```

We additionally used a `.coveragerc` configuration file to ignore the unit tests and the `pytlq/__init__.py` and `pytlq/pytlq.py` files.

The tests coverage analysis (coverage criteria: statements) returned the following report:

Name	Statements	Misses	Coverage
pytlq/ast	96	0	100%
pytlq/checker	345	0	100%
pytlq/exception	9	0	100%
pytlq/parser	93	0	100%
pytlq/simplifier	50	0	100%
pytlq/solver	74	0	100%
pytlq/utils	181	8	96%
TOTAL	848	8	99%

Note that the eight misses in the `utils` module come from the `NotImplementedError` exceptions that could not be tested.