UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Adding SAT-based model checking to the PyNuSMV framework

Dissertation presented by
Xavier GILLARD

for obtaining the Master's degree in
Computer Science

Supervisor
Charles PECHEUR

Readers
Charles PECHEUR, Simon BUSARD, Ramin SADRE

Academic year 2015-2016

Abstract

NuSMV is an open-source reimplementation and extension of the well known SMV model checker developed at Carnegie Mellon University. While this tool provides a rich set of high performance, state of the art verification features; the size and complexity of its code base make the extension and customization of this tool uneasy. In order to alleviate that problem, PyNuSMV [10] was developed at Louvain Verification Lab as a simpler platform to implement verification tools for new logics in Python. However, in its current version, PyNuSMV is limited to symbolic model checking with BDDs.
This thesis proposes an extension of PyNuSMV which features SAT-BMC capabilities available in NuSMV. After that, it describes two experiments that were made to validate the usefulness and performance of our addition to the framework. The first one consists in an implementation of an LTL bounded model checker in Python while the second implements a *diagnosability* test tool which shows how SAT-BMC tools working with multiple parallel traces can be developed using our addition to the framework. The performance evaluation made in the scope of both experiments revealed that the verification tools developed using our addition to PyNuSMV only add a low overhead compared to an equivalent verification done with NuSMV and, in some cases significantly outperforms what is feasible with the regular NuSMV.

Acknowledgments

Before entering in the core subject of this dissertation, I would like to express my gratitude to all the people that helped me during the realization of this master's thesis. In particular, I would like to thank:

- **Prof. Charles Pecheur** for his guidance, support and accessibility as well as opening my mind to the verification of new kind of logics and their potential applications.
- **Simon Busard** for his invaluable support on PyNuSMV, his patience and his many remarks that helped me to stay on track during the development phase of the project and to improve this very dissertation by challenging its content.
- **My wife and family** whose support have been essential in the accomplishment of these two years of study.

Contents

Introduction

Dependable software

In the world of today, software has pervaded in virtually every aspect of our lives: for instance, nobody imagines traveling to an unknown region without a GPS guiding system. Moreover, over the last decade, the cars we are driving have become incredibly software dependent for the best (safer equipment) or for the worst as it's been illustrated by the Volkswagen scandal revealed by end of 2015. Besides that, men also rely on software for life critical matters where the guarantees provided by the systems need to be more stringent in order to avoid disasters such as the death of patients in 2001 in Panama [2] who were exposed to lethal levels of radiations because of the poor quality of the software operating the radiation machine.

Unfortunately, contrary to the hardware manufacturing industry, the software ecosystem critically lacks from a corpus of direct evidences about the causes and effects of software failures [24]. Because of this, it has often been widely accepted that software systems should be thoroughly tested in order to be depended upon.

Testing is not enough

However, as Brooks put it in *The Mythical Man Month*, testing has never been considered the panacea because "testing must be extensive, for the number of cases grows combinatorially. It is time-consuming, for subtle bugs arise from unexpected interactions of debugged components" [9, p. 6]. In addition to that, as of 1970, Dijkstra made a famous argument against testing stating that "program testing can be used to show the presence of bugs, but never to show their absence!" [17, p. 6]. Moreover, he proved in [16] that exhausting the complete spectrum of all possible test cases would be infeasible even for a program as trivial as a simple floating point multiplication. From there, he advocates that formal proofs of program correctness provide much stronger dependability guarantees. Based on that idea, many formal methods have been developed to ease and automate the establishment of such program proof of correctness amongst which, NASA's Java Path Finder (JPF)[35] turned out to be one of the most successful examples.

Need to verify the requirements

Meanwhile, many others acknowledged the fact that a *program* in itself only constitutes the final output of a development process started long before its delivery. Because of this, coding errors - the only kind of faults testing is able to detect - only account for a fraction of the problems that may impact a software solution and those are typically the ones that are the easiest to fix. Based on that statement came the idea that the complete development process should be kept under control and that the very specifications of a

software to be developed had to be validated and their ability to fulfill some pre-established condition should be proved. However, such specification are usually not expressed in a form amenable to automated processing and hence the verification of such specification could only be done by hand and were almost never complete.

Why focus on model checking

In order to relieve analysts from doing such an exhaustive, expensive and time consuming proof of their analysis, many modeling languages have been proposed to express the specifications of software abstractions in a more formal and structured way. Which one could then be verified by automated tools in order to check that desired properties hold on the proposed model. Those tools are usually called model checkers and let the analyst express the properties that need to be satisfied by a model using a *temporal logic formalism* (such as the *Linear* Temporal Logic: LTL). These automated formal methods have already been used a lot: mostly by hardware manufacturers such as Intel for whom an error in the design of a chip can have dramatic financial consequences [4]; but also by software engineers e.g. by Ongaro and Ousterhout who proved the correctness of their distributed consensus algorithm (RAFT) using *TLA+* [36].

The main advantages of this technique come from the fact that it is *fully* automated and requires no human guidance. Moreover, whenever a violation of the property is detected, a model checker is able to report a scenario leading to the property violation. Those scenarios turn out to be invaluably helpful to the analyst since they often witness subtle bugs that would otherwise have been very hard to identify and fix. However, the kind of verification that can be performed by the model checker is limited by the intrinsic (exponential) computational complexity of the verification problem. Hence, despite the fact that algorithms are known and the formalisms are sound and clear, the existing tools fail to assess the validity of these properties on large model instances.

... and in particular SAT-based bounded model checking

In that context, some people have developed a variation of the model checking known as *SAT-based bounded model checking* that can handle problems significantly larger than the traditional approaches. To that end, they leveraged the many performance improvements that have been made in the field of SAT solvers [23, 19, 22] and used them for verification purpose. However, this scalability gain came at the expense of another limitation of the verification that can be carried out: only finite executions of a bounded length can be verified. While this bounded length restriction looks a priori quite limiting, it turns out that the state space that can be exhausted by SAT solvers is usually one or several order of magnitudes bigger than the maximum size explorable with a classic BDD[1] approach. Moreover, as stated by Jackson with his *Small scope hypothesis*: "most bugs have small counterexamples" [25, p. 143] which means that in practice, the kind of verification performed with SAT-BMC is really useful and efficiently complements the classical model checking techniques.

[1] Binary Decision Diagram.

Going beyond LTL : PyNuSMV

Over the years, many authors have decided to go beyond the capabilities of LTL and extended its semantics to change the assumptions it makes in order to reason about other types of systems. Because of this, the Louvain Verification Lab (UCL) has developed PyNuSMV, a Python library that aims at giving developers an easy access to the main functionalities of the well known NuSMV model checker while shielding them from the internal complexity of NuSMV. For instance, using PyNuSMV, one no longer needs to care about the low level memory management while manipulating NuSMV data structures. Thanks to this framework, it becomes much simpler for someone to experiment with new verification algorithms, or new functionalities. However, in its current version, the framework is limited to the BDD-related features.

Contribution

In that context, the contribution of this master's thesis is to continue the effort that has already been done on PyNuSMV and complement it with the SAT bounded model checking related features that are available in NuSMV.

To that end, this thesis will start presenting the principles of model checking and detail the functioning of SAT-BMC. Then, in a second part, we will explain the developments that have been made in order to achieve the goal of completing PyNuSMV with SAT-BMC features. In particular, we will detail the architecture of the proposed solution and explain the technological choices that have been made. After that, we will demonstrate the usefulness and usability of this addition to PyNuSMV with two illustrative case studies. Finally we will finalize this work with an integrative conclusion and propose some possible topics for further research.

Part I

Background

Chapter 1

Principles of model checking

The objective of this part of the master's thesis is to present the model checking technique in order to precisely and formally define the problem it solves. To this end, we will start from the informal definition proposed by Baier and Katoen: "Model Checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model" [4, p. 11]. As it shines from this definition, model checking needs two distinct inputs in order to operate: a finite-state model representing the system under analysis and a property to be verified expressed in the form of a temporal logic formula.

The structure of this chapter closely reflects that of the aforementioned definition and will therefore start defining the kind of system-specification a model checker can operate on. After that, we will give an overview of the kind of properties that can be verified on those models and present the language and semantics of the *Linear Temporal Logic* which is one of the most commonly used logics to express the properties verified by a model checker. Then, an illustration will show how these kind of models and properties are expressed in a practical tool, NuSMV. Finally, we will conclude this chapter with a precise definition of the model checking problem.

1.1 Finite-state model

The very first thing one needs to do when studying the properties of an actual (existing or not) system via automated verification techniques, is to produce a formal, machine-processable model standing for that system. Doing so, one needs to strive to achieve two objectives: first he needs to produce a model which is faithful to its real-world counter part. Meanwhile, it must also abstract away all the details that are not relevant to the ongoing analysis. For instance, in order to prepare the analysis of a distributed system, it is useful to be able to keep track of the different *kinds* of exchanged messages. However, in order to facilitate the analysis, it is probably also useful to remove the implementation-level details regarding the structure of those messages such as which bit serves what purpose.

For the sake of really standing for the system it represents, the formal model first needs to capture the *state* of the system, that is to say, an "instantaneous description of the system that captures the values of its variables at a particular instant of time" [14, p. 12]. Moreover, the model also needs to showcase the same *behavior* as the original system. That is to say, the possible changes in the state of the model need to be the same in the model and in the real system.

In order to group these requirements related to the model, men most of the time use a *Kripke Structure (KS)*. Which one can be understood as a finite-state machine (FSM aka automaton) equipped with an interpretation function [38, s. 2]. To be complete, let us also note that an other formalism is also widely used in the model checking community: namely, *labeled transition systems (LTS)* which are e.g. used by LTSA, the model checker proposed by Magee and Kramer in [31]. Given that NuSMV and hence PyNuSMV rely on the Kripke Structure representation, and because De Nicola et al. have shown in [15] that these two kinds of transition systems essentially have the same expressive power, we will not detail LTS any further. However let us mention that as opposed to KS, LTS do not represent the internal state of the system[1] and therefore only model its external (observable) behavior.

1.1.1 Kripke Structure

In order to be able to give a sensible meaning to the different states of a Kripke Structure, a finite set of *atomic proposition* with predefined meaning is used. The truth value associated with each of these propositions is set by the state of the system. That is to say, the truth value of each of these proposition depends on the value of the variables composing the internal state of the system. For instance, considering a model whose state consists of the sole variable $favourite\_drink \in \{coffee, tea, juice\}$, one could define the following atomic propositions:

- $stressed \Leftrightarrow (favourite\_drink \leftarrow coffee)$
- $british \Leftrightarrow (favourite\_drink \leftarrow tea)$.

Formal definitions

In the context of a Kripke Structure, a *state* is defined as an assignment of values to each of the variables composing the system state. Formally, assuming that $\mathcal{V} = \{v_1, v_2, v_3, \cdots, v_n\}$ is the set of system variables ranging over the finite set $\mathcal{D}$ (the domain); a *state* is a valuation $s : \mathcal{V} \to \mathcal{D}$ for the set of variables in $\mathcal{V}$ [14, p. 14].

A *Kripke Structure* is a 5-tuple $K = \langle AP, S, S_0, T, V \rangle$ with:

- AP being the set of atomic propositions discussed above.
- S is the set of possible states in the system. That is to say, the set of possible ways to assign a value belonging to its domain to each of the variables composing the system state.
- $S_0 \subseteq S$ the set of initial states.
- $T \subseteq S \times S$ is a transition relation representing the possible evolution of the system over time. Note that, the Kripke structure formalism imposes this relation to be *total* [14]. This means that $\forall s \in S \cdot \exists s' \in S : (s, s') \in T$.
- $V : S \to 2^{AP}$ is an *interpretation function* which maps to each state $s \in S$ the subset of the atomic propositions of AP that hold in s.

In addition to the notions of state and KS, is is also useful to precisely define a path or *trace* of the structure K. Formally, it is an infinite sequence $\pi = s_0, s_1, s_2, s_3, \cdots$ of states of K such that and $\forall i > 0 : (s_{i-1}, s_i) \in T$. By convention, we denote Σ_K the set of all the traces of a Kripke Structure K, $\Sigma_{K_{prefix}}$ the set of all prefixes of Σ_K and $\Sigma_{K_{suffix}}$ the set of all suffixes of Σ_K.

[1] Whenever it is useful to access the internal state of the system, Magee and Kramer propose to use *fluents* which basically serve the same purpose.

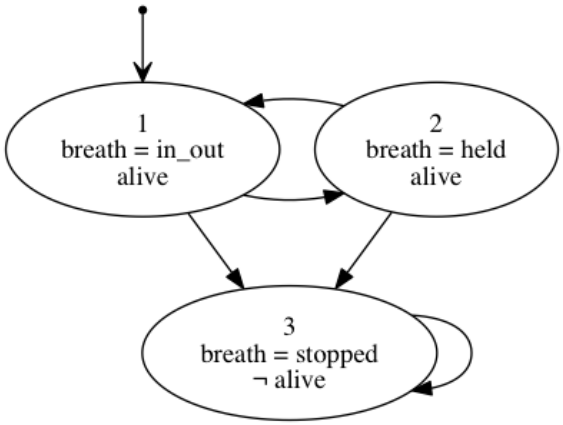

Figure 1.1: Graphical representation of the Kripke structure detailed in the example of subsection 1.1.1. In this graph, the first line of each node corresponds to the node number used to facilitate further reference. The second line corresponds to the internal state of the system (the assignments of its variable when in that state), and the third line represents the truth value of the proposition in AP.

Example

If we were to model (very summarily), the life of an animal; assuming the state variable $breath \in \{in\_out, held, stopped\}$ it could be done with the following structure $K = \langle AP, S, S_0, T, V \rangle$ with:

- $AP = \{alive\}$ with $alive \Leftrightarrow (breath \neq stopped)$
- $S = \{1 = (breath \leftarrow in\_out), 2 = (breath \leftarrow held), 3 = (breath \leftarrow stopped)\}$
- $S_0 = \{1\}$
- $T = \{(1,2), (1,3), (2,1), (2,3), (3,3)\}$
- $V =$
  - $V(1) = \{alive\}$
  - $V(2) = \{alive\}$
  - $V(3) = \{\}$

In the Kripke Structure defined this way, the sequence $1, 2, 3, 3, 3, \cdots$ would be a valid path of K whereas $2, 2, 2, \cdots$ would not.

Graphically, this structure can be represented by the graph shown in Figure-1.1. However, this kind of representation is seldom used in the reality since it turns out to be very impractical for a human to reason about a large system when presented in such a way. This is especially striking in the case of concurrent systems since the Kripke structure corresponding to the compound system is usually very large[2]. Therefore, most model checkers and NuSMV in particular allow the user to provide the description of the model under a textual form, close to a programming language. This text is then compiled into an equivalent Kripke structure.

[2]This stems from the fact that composition of two models in the interleaved (asynchronous) scheme is made as a product between the two structures.

1.2 Temporal Logics

Because model checking is a technique that aims at verifying properties on *reactive systems* which are not necessarily meant to terminate (for instance, a web server), the properties to be verified cannot simply be expressed using pairs of *pre/post* conditions as would be the case with theorem proving[3]. Instead, they are expressed with "*temporal logic* [which] is a formalism for describing sequences of transitions between states in a reactive system." [14, p. 27] Because of this, they give us the ability to reason in terms of temporal relations such as "before" or "after" [48] to verify not only that a program output is correct but also the fact that this result was *correctly produced*.

1.2.1 The kind of properties that can be expressed

Before to dig in the syntax and semantics of actual temporal logics like LTL, it is important to clarify the kind of properties that one might want to express about the system modeled with a Kripke structure. To this end, we will use the informal classification established by Lamport in [30] where he distinguishes *safety* and *liveness* properties and define each of these two sets[4].

Safety

Informally, a *safety* property is one stating "that something will *not* happen" [30, p. 125]. For instance, saying that a program will not be stuck in a deadlock is an example of such property. More formally, a safety property P is one such that:

$$\forall \pi \in \Sigma_K : \pi \not\models P \implies (\exists \alpha \in \Sigma_{K_{prefix}}, \beta \in \Sigma_{K_{suffix}} : \pi = \alpha\beta \text{ and } \forall \gamma \in \Sigma_{K_{suffix}} : \alpha\gamma \not\models P)$$

That is to say, if any execution prefix α of a trace of K violates the property P, then it is *irremediably* falsified in all the possible executions starting with that prefix. From this definition one easily sees the two characteristics of safety properties, namely: a safety property is *violated* in *finite* time but *verified* in *infinite* time.

Liveness

Liveness properties are those stating that "something *must* happen" [30, p. 125]. The most common example of a liveness property is probably the one aiming at verifying that a program eventually terminates. More formally, a property P is a liveness property iff:

$$\forall \alpha \in \Sigma_{K_{prefix}} \cdot \exists \beta \in \Sigma_{K_{suffix}} : \alpha\beta \models P$$

That is to say, for any finite execution prefix α on the studied Kripke struture, there exists a potentially infinite suffix β that permits the satisfaction of that property. This definition lets us establish the two characteristics of liveness properties: they are *verified* in *finite* time but *violated* in *infinite* time.

[3] See for instance [3] for a detailed discussion on that topic.

[4] It is interesting to note that safety and liveness define *disjoint* classes of path properties.

1.2.2 Linear Temporal Logic (LTL)

The Linear Temporal Logic is a logic formalism extending the propositional logic with temporal modalities that permit to reason about execution traces of a reactive system in terms of the transitions that have been used. In this logic, the nature of time is considered *linear* because it is interpreted over individual execution traces (sequences of states). Therefore, it only considers one single successor to each of the states in an execution trace. "Stated differently, in the linear-time perspective, we select one execution and look at it holistically, as one monolithic entity" [4, p. 226].

In its standard setup, LTL defines five temporal operators : *Globally* ($\Box\ \cdot$), *Finally* ($\Diamond\ \cdot$), *Next* ($\bigcirc\ \cdot$), *Until* ($\cdot\ \mathbf{U}\ \cdot$), *Unless* also known as *Weak Until* ($\cdot\ \mathbf{W}\ \cdot$)[5]. In this context, formulas are interpreted over computations that is to say paths, sequences of states in the Kripke Structure; and are said to hold in a state s if and only if *all the computations starting in s satisfy the given property.*

Syntax

The complete syntax of LTL can be described as follows, where $p \in AP$ is an atomic proposition:

$$\begin{aligned}\phi := \quad & p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \implies \phi \mid \phi \iff \phi \\ & \mid \Box\ \phi \mid \Diamond\ \phi \mid \bigcirc\ \phi \mid \phi\ \mathbf{U}\ \phi \mid \phi\ \mathbf{W}\ \phi\end{aligned}$$

Semantics

An informal definition of the LTL semantics is given in Figure-1.2. Intuitively, it expresses that the meaning of $\Box\ \phi$ is a safety property, an invariant. It is therefore verified in the current state if ϕ holds in that state and all the subsequent states of the considered trace. Generalizing that idea, this property holds *for a model* iff ϕ holds for each of the states of each of the possible executions starting in an initial state of that model. The formula $\Diamond\ \phi$, on the other hand defines a liveness property and is therefore verified if ϕ holds in the current state of the trace or in some future state. This means that there can be an arbitrarily long sequence of states and transitions between the current state and the one where ϕ holds but that state will *eventually* be reached. In the same order of idea, a property of the form $\bigcirc\ \phi$ is verified if and only if ϕ holds for the next state of the considered trace. Besides that, the property places no constraint on the value of ϕ in the current state nor on any state subsequent to the following one. The semantics of $\cdot\ \mathbf{U}\ \cdot$ and $\cdot\ \mathbf{W}\ \cdot$ are slightly more complex: indeed, both $\phi\ \mathbf{U}\ \psi$ and $\phi\ \mathbf{W}\ \psi$ impose that ϕ be true all along the path leading to a state where ψ holds after which nothing is known about the values of ϕ and ψ. There is however a major difference between these two modalities: $\phi\ \mathbf{U}\ \psi$ imposes that ψ eventually become true whereas $\phi\ \mathbf{W}\ \psi$ does not and is considered to hold if ϕ remains true forever.

Formally, assuming a Kripke Structure $K = \langle AP, S, S_0, T, V\rangle$ and an execution trace $\pi = s_0, s_1, s_2, \cdots \in \Sigma_K$, whose suffix starting at state s_i is denoted π^i, if ϕ and ψ denote

[5] For the sake of completeness, let us also mention that the operator *Release*($\cdot\mathbf{R}\cdot$) is often defined as the exact dual of $\cdot\ \mathbf{U}\ \cdot$.

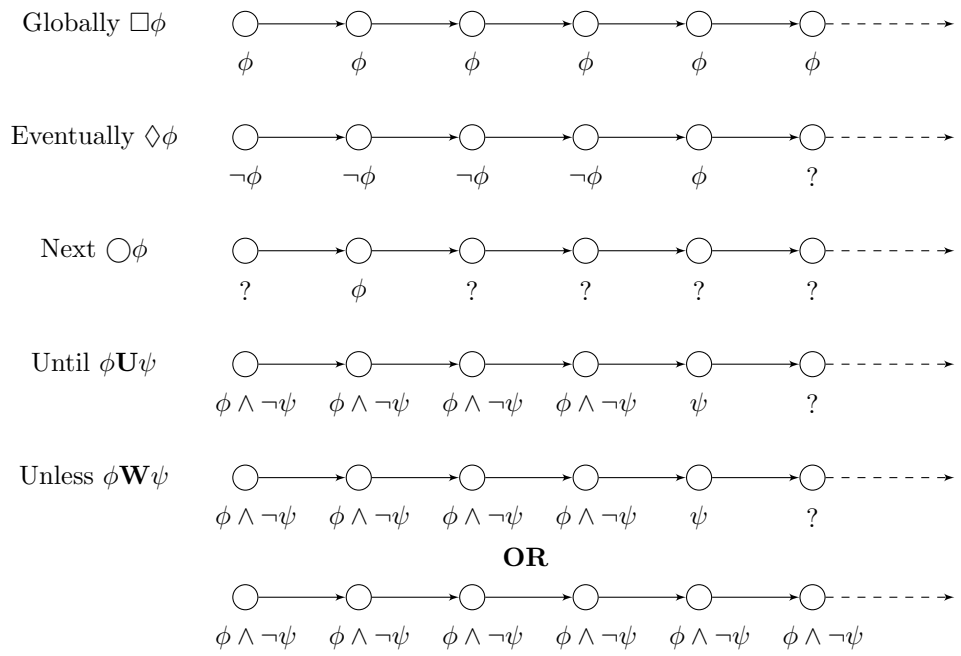

Figure 1.2: Informal semantics of LTL temporal operators. (Adapted from [4, p. 229].)

LTL formulas; the semantics of LTL is described by a relation $\models$ defined as follow:

$$
\begin{array}{lcl}
\pi \models \phi & \iff & \phi \in V(s_0) \\
\pi \models \neg\phi & \iff & \pi \not\models \phi \\
\pi \models \phi \vee \psi & \iff & \pi \models \phi \; or \; \pi \models \psi \\
\pi \models \phi \wedge \psi & \iff & \pi \models \phi \; and \; \pi \models \psi \\
\pi \models \bigcirc \phi & \iff & \pi^1 \models \phi \\
\pi \models \Box \phi & \iff & \forall i : \pi^i \models \phi \\
\pi \models \Diamond \phi & \iff & \exists i : \pi^i \models \phi \\
\pi \models \phi \; \mathbf{U} \; \psi & \iff & \exists i : \pi^i \models \psi \; and \; \forall j, 0 \leq j < i : \pi^j \models \phi \\
\pi \models \phi \; \mathbf{W} \; \psi & \iff & \forall i : \pi^i \models \phi \; or \; (\exists i : \pi^i \models \psi \; and \; \forall j, 0 \leq j < i : \pi^j \models \phi)
\end{array}
$$

From these definitions, it is useful (and sometimes more intuitive) to notice that the following equivalences hold:

$$
\begin{array}{lcl}
\Box \phi & = & \phi \; \mathbf{W} \; \bot \\
\Diamond \phi & = & \top \; \mathbf{U} \; \phi \\
\phi \; \mathbf{W} \; \psi & = & \Box \phi \; or \; \phi \; \mathbf{U} \; \psi \\
\phi \; \mathbf{U} \; \psi & = & \Diamond \psi \; and \; \phi \; \mathbf{W} \; \psi \\
\phi \; \mathbf{W} \; \psi & = & \neg(\neg\phi \; \mathbf{U} \; \neg(\phi \vee \psi)) \\
\phi \iff \psi & = & (\phi \implies \psi) \wedge (\psi \implies \phi) \\
\phi \implies \psi & = & \neg\phi \vee \psi \\
\phi \vee \psi & = & \neg(\neg\phi \wedge \neg\psi)
\end{array}
$$

The consequence of which being that $\Box \cdot$, $\Diamond \cdot$ and $\cdot \; \mathbf{W} \; \cdot$ do not add to the expressive power of LTL and are only useful for the convenience of expressing temporal properties. Therefore, it suffices for an algorithm to be able to verify $\wedge, \vee, \neg, \bigcirc \; \cdot \; and \; \cdot \; \mathbf{U} \; \cdot$ in order to be able verify the complete LTL language.

Therefore, the complete syntax of LTL can be reduced to the following minimal grammar because the remaining operators can be expressed in terms of the others using the above equivalences:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc \; \phi \mid \phi_1 \mathbf{U} \; \phi_2$$

It is also worth noting that the temporal modalities are also often denoted using an alternate *letter* symbol. However, the symbol used to express a temporal operator has absolutely no impact on the semantic of that operator. The alternate symbols are the following : *Globally* (**G** $\cdot$), *Finally* (**F** $\cdot$), *Next* (**X** $\cdot$).

Examples

In order to illustrate the use and semantics of the different LTL operators, we will translate some assertions in plain English to their LTL equivalent formulation. Note, in the following examples all verbs and nouns are assumed to be defined as atomic propositions of the set AP.

After the rain comes the sun	$\square\ (rain \implies \bigcirc\ sun)$
Some day, you will die	$\lozenge\ die$
You won't get married unless you propose	$\neg married\ \mathbf{W}\ proposed$
You're only alive until you're dead	$alive\ \mathbf{U}\ dead$

Discussion about temporal logics

Albeit the most widely used, LTL is not the sole temporal logic that has been defined: indeed there exists a wide array of formalisms which are distinguished from one an other by their time structure (linear as in LTL or branching) and the operators they define. For instance, the very popular Computation Tree Logic (CTL) was defined with a branching time structure and evaluates over trees of executions instead of single paths[6].

1.3 NuSMV

Given that NuSMV is a reimplementation and extension[13, 26, 12] of the original SMV model checker developed at Carnegie Mellon University[33], its input language owes much to that of its predecessor. Since the exact syntax and semantics of the SMV language is considered peripheral to the explanation of the principles of model checking and is none of the main focus of this master's thesis, this paragraph will not provide an in-depth description of the language syntax and its associated semantics. Instead, it will focus on giving the reader an informal overview of the relevant constructs and show how such models relate to the concepts defined so far[7]. To that end, we will simply build upon the breathing animal example which was discussed when introducing the notion of Kripke structure and show the kind of properties that can be shown to hold or not on these models.

The SMV model reproduced in Figure 1.3 is a raw translation of the aforementioned example. From this snippet, one can observe that the automaton is described in the SMV text as a *module* whose state is characterized by *variables*. Interestingly enough, the SMV language has no explicit notion of a state when describing a model therefore, the set of states S of the Kripke structure defined this way is simply considered to be the set of possible valuations of the variables. In this example, S contains only three states since there is one single variable which can only take one of the three values $\{in\_out, held, stopped\}$. Besides that, it is also interesting to note that the set of initial states S_0 and the transition relation are described using the $init(\cdot)$ and $next(\cdot)$ statements introduced by the `ASSIGN`

[6] This means that CTL is able to differentiate the behavior of deterministic vs non-deterministic automata that feature the same set of traces.

[7] Meanwhile, the interested reader will find in appendix A a more detailed summary of the syntax and semantics of NuSMV. Apart from that, a thorough discussion on this matter can be found in chapter 4 of [33] and in [11].

directive. Also, the language offers the `DEFINE` facility which is used to name some complex conditions of the set AP with an intelligible name.

```
MODULE main
VAR
  breath : {in_out, held, stopped};
DEFINE
  alive := (breath = in_out) | (breath = held);
ASSIGN
  init(breath) := in_out;
  next(breath) := case
                    breath = in_out : {held, stopped};
                    breath = held   : {in_out, stopped};
                    breath = stopped : stopped;
                  esac;

-- Example of LTL properties that can be verified on the above model.
LTLSPEC
  -- At any time, if it is breathing then it is alive
  G breath = in_out -> alive
LTLSPEC
  -- Eventually, the animal dies. (Note: this property does not hold).
  F ! alive
```

Figure 1.3: A summarizing example which represents the same Kripke structure as the one detailed in the example of subsection 1.1.1 but encoded in the SMV language.

Despite its extreme simplicity, this model is a good vehicle to illustrate the meaning of the various LTL operators. For instance $\Box\ (breath = in\_out \implies alive)$ which is true since an animal can only breathe when alive. However, analyzing the truth value of $\Diamond\ \neg alive$ yields a result which might seem a little bit surprising at first sight. Indeed, this property is easily falsified by a trace that loops infinitely between the states 1 and 2. Similarly, $(alive\ \mathbf{W}\ breath = stopped)$ evaluates to true, and $(alive\ \mathbf{U}\ breath = stopped)$ to false for the same reason.

1.4 The model checking problem

Based on all the concepts we have introduced in the previous sections, defining precisely what model checking is about is a relatively straightforward matter. Indeed, given a Kripke structure $K = \langle AP, S, S_0, T, V \rangle$ and a formula expressed in a temporal logic ϕ, the whole problem of model checking consists in verifying whether ϕ holds for K by determining the set of states of K for which the property ϕ holds. That is to say : $\{x \in S | K, x \models \phi\}$ [14].

However, the *bounded model checking* technique studied in the scope of this master's thesis aims at solving a slightly different problem. Indeed, rather than identifying the set of *states* for which the property ϕ holds; it determines whether $K \models \phi$. That is to say, it determines whether $\pi \models \phi$ for all paths π of K starting in an initial state of K.

Chapter 2

Model checking techniques

The objective of this chapter is to introduce bounded model checking (BMC) and in particular the *SAT-based bounded model checking* technique which is at the core of the work realized in the scope of this master's thesis. To this end, we will start explaining the general idea that underlies the various flavors of model checking. Then we'll give a word about the peculiarities of the most classical model checking techniques. After this, the BMC and more specifically SAT-BMC[1] technique will be presented as it was proposed by Biere et al. in [7, 8][2].

2.1 The general idea : refutation

Fundamentally, all model checking methods are based on the refutation technique. So, instead of trying to build a constructive proof that demonstrates that $K \models \phi$ like a theorem prover would; the model checkers try to find a counterexample proving that $K \not\models \phi$. This approach offers the following two advantages: first, whenever a counterexample is found; it can be reported to the end user and provides him with a concrete scenario leading to the property violation. This often constitutes an invaluable source of information when one needs to adapt the specification in order to enforce some property. Second, this technique requires no human guidance and is thus able to work in a fully automatic fashion while providing the same proof-strength guarantees as theorem proving.

2.1.1 Automata-theoretic approach

In the context of the verification of LTL formulas, proving $K \not\models \phi$ means that the tool needs to find at least one trace in the Kripke structure for which the evaluated formula is falsified. Concretely, many ways to achieve that goal have been proposed over time. The first technique that appeared is now called the *automata-theoretic approach* since it basically implements a reachability test in a special automaton to verify if it is possible to falsify the property under verification. Practically, this is accomplished as follows: first an automaton $B_{\neg\phi}$ representing the formula $\neg\phi$ is constructed. However, because the computations of K and $B_{\neg\phi}$ are potentially infinite, and because $\neg\phi$ may contain the operators $\Box$ and $\Diamond$ which are only verified resp. falsified at infinity; the automaton $B_{\neg\phi}$

[1] Although in the practice, BMC and SAT-BMC tend to be used as synonyms, this master's thesis operates a slight distinction between the two as nothing prevents to invent an alternative technique based on the BMC ideas without relying on a SAT solver to perform the bulk of the verification.

[2] The technique described in these papers has later been extended by Penczek et al in [42, 41] to cover other logics than LTL; namely, CTL and its universal fragment ACTL.

may not simply be a regular DFA or NFA. Therefore, it is converted to a special kind of automaton called *Büchi automaton* which accepts traces passing *infinitely often* through an accepting state[37, slide 12][3]. Once $B_{\neg\phi}$ is built, it is combined with K to produce a new automaton V which is used to perform the actual verification. Despite the fact that the verification of ϕ can be realized in linear time ($O(|V|)$) using a repeated reachability test[4] in V, this approach fails to scale to large models because the size of the constructed automaton $B_{\neg\phi}$ is exponential in the size of $\neg\phi$ and can therefore have a huge impact on the size of the combined automaton V.

2.1.2 Symbolic model checking

In order to palliate for the limitations of the automata-theoretic approach, an other technique called *symbolic model checking* has been proposed. It differentiates itself from the previous technique by not considering each and every state and transition of the Kripke structure individually but instead manipulating sets of them which are seen as boolean formulas. Indeed, sets of states can be seen as the *extension* of a boolean formula where each state belonging to the set verifies the formula[5]. The meaning of these formulas naturally derives from the interpretation function $V : S \to AP$ which is part of the Kripke Structure. Typically the sets of states representing boolean functions are encoded as *Binary Decision Diagrams* (BDD)[6] which provide a very compact representation of both the model and the formula to be verified. In order to accommodate the *infinite-time* nature of the $\Box$ and $\Diamond$ operators to the set-based approach used by symbolic model checking, a slight change was introduced in the characterization of the semantics of the temporal operators which are then considered as greatest and least fixpoints[7] of the sub-formulas they define[8]. Intuitively, this means that in order to verify a property ϕ, a symbolic model checker computes the *smallest* set containing *all reachable states* satisfying $\neg\phi$ and checks the emptiness of the obtained set rather than considering each possible computation of the model.

While this approach has dramatically increased the size of the models that could be verified using model checking, it nevertheless turned out to be impractical to use on large real-life models because of the sheer amount of memory required in order to represent and manipulate the BDDs used to perform the verification.

2.2 Bounded model checking

Departing from the observation that *eventualities* (formulas that acquire a truth value after an infinite amount of time) are the main causes of intractability for the previous model checking techniques, an alternative approach has been proposed: *bounded model checking* (BMC). As opposed to the automata-theoretic approach and the BDD-driven symbolic model checking, BMC is not *complete* in the sense that is only provides an approximation of the answer of a *complete* model checker by considering only the first k steps of each possible computations. Therefore, when a BMC model checker fails to report a counterexample,

[3] The translation from LTL formulas to Büchi automata proposed by Vardi and Wolper is out of the scope of this master thesis. Details can however be found in [47].

[4] This test can for instance be realized using a double-DFS search or an SCC reachability test which can be efficiently implemented using one of Tarjan's or Kosaraju's algorithms.

[5] Hence, the empty set is used to denote the proposition $false$.

[6] Not discussed here but [14, Chap 5, p.51+] gives a gentle introduction to the subject.

[7] Here, the term fixpoint of a formula f must be understood as a set of value s such that $f(s) = s$.

[8] Intuitively, $\Diamond\ \phi$ is going to be the least fixpoint of ϕ because it is falsified at infinity and similarly, $\Box\ \phi$ is going to be its greatest fixpoint.

it does not necessarily mean that the verified property is valid for the examined model. However, whenever BMC reports a counterexample in the form of a violation witness, there is a strong guarantee that the property does not hold for the studied model.

2.2.1 Incompleteness mitigation

While the incompleteness of the results obtained via BMC might seem quite limiting, it turns out that the importance of this issue can be mitigated in many ways:

1. The method can be made complete. Indeed, it is easy to convince oneself that it suffice to consider all possible trace lengths $(0, 1, 2, 3 \cdots)$ in order to make BMC yield the same result as one of the complete methods. Moreover, as it has been shown in [7, 8] the finite state nature of the considered systems implies that there exists an upper bound[9] to the length of the traces that need be considered in order to ensure a complete result.
2. It has been experimentally shown in [7, 8] that the size of the problems that can be model checked with implementations of BMC (and in particular SAT-BMC) is an order of magnitude larger than any of the problems that could be verified with BDD-based symbolic model checking.
3. The *small scope hypothesis* might hold even for system for which complete exhaustion of the state-space is considered out of reach. This hypothesis states that "most bugs have small counterexamples" [25, p. 143] and hence that an incomplete result obtained analyzing a large portion of the state-space might be sufficient to have some confidence in the verified design despite the absence of guarantee. This hypothesis has however not been proved and solely relies on empirical experience.

2.2.2 Reaching Completeness

As it has been mentioned previously, Biere et al. have shown in [7, 8] that it was possible to set an upper-bound on the length of the traces to be considered while performing a property verification in order to ensure the validity of that property. This upper-bound stems from the fact that Kripke structure are *finite state* automata, therefore there exists a *shortest path* between any state reachable from an initial state and that state. Moreover, because all shortest paths between any two states are necessarily of *finite length*, there must exist one such shortest path of maximal length. The length of that "longest shortest path" [8, p. 15] is called *reachability diameter* of the Kripke structure and corresponds to the minimal number of steps that needs to be allowed in a trace in order to ensure that the outcome of BMC is the same as that of any other complete model checking technique.

[9] In this context, the term upper-bound must be understood as the *minimal* number of steps k that should be considered by BMC in order to have a chance to find a counterexample. It is an upper bound since any BMC verification with an higher value of k would find no additional counterexamples.

2.2.3 The main idea: unrolling

The main idea applied in BMC in order to deliver a bounded approximation of the verification of $K \models \phi$ is the unrolling of the temporal operators. Indeed, in the case of LTL, they can all be recursively defined using as such:

$$\begin{aligned} \Box\ \phi &= \phi \wedge \bigcirc \Box\ \phi \\ \Diamond\ \phi &= \phi \vee \bigcirc \Diamond\ \phi \\ \phi\ \mathbf{U}\ \psi &= \psi \vee (\phi \wedge \bigcirc\ [\phi\ \mathbf{U}\ \psi]) \\ \phi\ \mathbf{W}\ \psi &= \psi \vee (\phi \wedge \bigcirc\ [\phi\ \mathbf{W}\ \psi]) \end{aligned}$$

Important observation

From the above equivalences, it is important to note that $\cdot\ \mathbf{U}\ \cdot$ and $\cdot\ \mathbf{W}\ \cdot$ have the same *expansion law* and "the semantic difference between $\cdot\ \mathbf{U}\ \cdot$ and $\cdot\ \mathbf{W}\ \cdot$ is shown by the fact that $\phi\ \mathbf{W}\ \psi$ is the *greatest* solution of $\kappa = \psi \vee (\phi \wedge \bigcirc\ \kappa)$ [whereas] $\phi\ \mathbf{U}\ \psi$ is the least solution of this equivalence" [4, p. 248-249].

2.2.4 Bounded semantics

Even though the unrolled definition of the operators is an useful and intuitive way to think about the way to perform a bounded verification of some property, it cannot be used "as is" to implement BMC. Indeed, considering the example of the verification of a property of the form $\Box\ \phi$ for a model K using a maximum number of allowed steps k, it is perfectly possible to encounter a case where ϕ holds for the first k states of each trace but is violated in state $k+1$. In that context, one should refrain to let the tool reply that $K \models \Box\ \phi$ since this assertion can be falsified in $k+1$ steps. In order to deal with this case, Biere et al. have proposed in [7, 8] a modified semantics for the temporal operators which they called *bounded semantics*.

Generally speaking, the bounded semantics of LTL operators differs depending on the shape of analyzed trace. Indeed, in the case of a looping path (these will be called *k-loop* as of now), the usual unbounded semantics of these operators can be maintained as the trace suffices to denote an infinite behavior of the system. However, in the absence of such a loop, a more defensive line must be taken in order to avoid produce incorrect result because the trace does not give any information regarding the infinite behavior of the system.

Formally, a path π is called a $k-loop$ iff $\exists k \geq l \geq 0$ such that π is a $(k-l)-loop$. And π is called a $(k-l)-loop$ for $k \geq l \geq 0$ if $(\pi(k), \pi(l)) \in T$ and $\pi = u \cdot v^{\omega}$ with $u = (s_0, s_1, \cdots s_{l-1})$ and $v = (s_l \cdots s_k)$. Graphically, the two possible cases of bounded paths are represented in Figure-2.1.

Figure 2.1: The two possible cases of a bounded path [8].

As it was said above, the bounded semantics over k-loops is essentially the same as the unbounded semantics. However, in order to emphasize on the fact that this semantics holds only traces containing at most k-steps, the validity relation is denoted by $\models_k$ instead of $\models$.

Bounded semantics in the absence of loop.

In the event where the considered trace does not feature the k-loop shape, the validity of a formula needs to be evaluated in each of the states composing the trace. Indeed, because the bounded semantics considers only the k first steps of any trace, the truth value of $\bigcirc \phi$ depends on the state on which it is evaluated. In particular, this formula will always be considered to be *false* in the k-th state since it is assumed to have no successor under the bounded semantics. To reflect this fact, the validity relation is denoted $\models_k^i$ under this assumption with i standing for the index of the state on which the formula is evaluated.

More formally, the bounded semantics $\models_k \phi$ of a formula ϕ subject to a bound k evaluates true for a non k-loop path π iff $\pi \models_k^0 \phi$ where:

$$
\begin{array}{lcll}
\pi \models_k^i & \phi & \iff & p \in V(\pi_i) \\
\pi \models_k^i & \neg\phi & \iff & \pi \not\models_k^i \phi \\
\pi \models_k^i & \phi \wedge \psi & \iff & \pi \models_k^i \phi \; and \; \pi \models_k^i \psi \\
\pi \models_k^i & \phi \vee \psi & \iff & \pi \models_k^i \phi \; or \; \pi \models_k^i \psi \\
\pi \models_k^i & \bigcirc \phi & \iff & i < k \; and \; \pi \models_k^{i+1} \phi \\
\pi \models_k^i & \Box \phi & \iff & \bot \\
\pi \models_k^i & \Diamond \phi & \iff & \exists j, i \leq j \leq k \pi \models_k^i \phi \\
\pi \models_k^i & \phi \; \mathbf{U} \; \psi & \iff & \Diamond \psi \; and \; \forall n, i \leq n \leq k : \pi \models_k^n \phi \\
\pi \models_k^i & \phi \; \mathbf{W} \; \psi & \iff & \Diamond \psi \; and \; \forall n, i \leq n \leq k : \pi \models_k^n \phi
\end{array}
$$

Important observations

From the above definition, it is important to outline that the truth value of a formula rooted with $\Box \cdot$ is *always false*. Moreover, it is worth noting that the *bounded* semantics of $\cdot \; \mathbf{U} \; \cdot$ and $\cdot \; \mathbf{W} \; \cdot$ is identical on a straight path. This stems from the fact that the semantics of $\cdot \; \mathbf{U} \; \cdot$ and $\cdot \; \mathbf{W} \; \cdot$ only differ at infinity. Hence, because it is impossible to model an infinite behavior with a finite straight path of a Kripke structure, there can be no difference in the semantics of these two modalities in the absence of loop.

2.3 SAT-Based Bounded model checking

The objective of this section is to gather all the principles that have been explained in the previous pages and use them to define a model checking technique able to perform verification of LTL properties at a scale that could not be approached by the classical methods: the SAT-based bounded model checking. Even though this technique does not solve the intrinsic complexity of the model checking problem; it aims (and often succeeds) at improving the performance of the model checker by leveraging the many improvements that have been made in the field of SAT solvers to perform the bounded verification of LTL property.

In order to achieve that goal, given a maximum bound of k of steps, the tools implementing the SAT-BMC technique generate a boolean propositional formula $[\![K, \neg\phi]\!]_k$ which is satisfiable iff the property $\neg\phi$ holds for the system described by the Kripke structure K. Put an other way, these tools generate a formula which can only be made true if $K \not\models_k \phi$ and there exists a counterexample proving that the property can be violated. After this is done, this propositional formula is fed to a SAT solver in order to determine its satisfiability.

2.3.1 Preliminary note

Because the duality (resp. pseudo-duality) that existed between $\Box \cdot$ and $\Diamond \cdot$ (resp. $\cdot \ \mathbf{W} \cdot$ and $\cdot \ \mathbf{U} \cdot$) is broken by the bounded semantics of these operators, the remainder of this master's thesis makes the assumption that all formulas are converted to an equivalent *negative normal form* (NNF). As will be explained in the next pages, failing to do so would render the reduction of bounded model checking to propositional SAT solving incorrect.

2.3.2 SAT Solvers

From an high level perspective, SAT solvers can be seen as black boxes which given a (potentially very long) propositional boolean formula encoded in *conjunctive normal form* (CNF) are able to determine the existence of a valuation of the propositional variables composing the function that would make the input formula *true*.

Under the hood, those black boxes often implement a variation of the Davis-Putnam algorithm (DPLL)[10] which is essentially a recursive backtracking algorithm enhanced with *early termination*[11], *pure symbols*[12] and *unit clauses*[13] heuristics. In addition to these three heuristics, modern state-of-the art SAT solver also implement the *conflict clause identification*, *minimization* and *restart* heuristics that have been proposed in Berkmin, Zchaff and MiniSat[23, 19, 22].

2.3.3 Reduction of model checking to propositional SAT solving

In conformance with what is done in every other model checking techniques, the formula $[\![K, \neg\phi]\!]_k$ generated by the BMC verification process combines both an element $[\![K]\!]_k$ standing for the model of the system and one standing for the negation of the property to verify $[\![\neg\phi]\!]_k$. Both of which are generated using the *unrolling* metaphor (either for the transition relation or the temporal operators).

Concretely, in order to faithfully stand for the KS it represents, $[\![K]\!]_k$ needs to combine two distinct constraints: the first one, denoted $[\![S_0]\!]$ serves to force the verification process to consider only computations of the KS which actually start in the initial state of the automaton[14]. Whereas the second one $[\![T]\!]_k$ forces the considered paths to only flow according to edges defined by the transition relation.

Moreover, because of the path-shape dependent nature of the bounded semantics that has been explained in section 2.2.4, the generation of $[\![\neg\phi]\!]_k$ must also be decomposed in three sub-formulas:

- $[\![\neg\phi]\!]^0_k$ which stands for the 'straight path' bounded semantics of $\neg\phi$
- ${}_lL_k$ which is a condition evaluating to true iff the considered path is a (k,l)-loop
- ${}_l[\![\neg\phi]\!]^0_k$ which stands for the looping path bounded semantics of $\neg\phi$ when the path is a (k,l)-loop.

[10] Details about this algorithm can be found in [44, pp. 260-262].

[11] Stop searching a value for the variables composing a clause whenever one of the clause disjunct could be made true ($a \vee b$ is true whenever a or b is true, regardless of the value of the other term).

[12] As explained in [44, p. 260], the *pure symbol heuristic* aims at reducing the scope of the search to be carried out by identifying "pure symbols" (symbols which always bear the same sign in all the clauses of the CNF-formula) whose literal can immediately be assigned the value which makes so as to evaluate to true.

[13] A truth value may be assigned to the sole term of such an unit clause since it is critical to the satisfiability of the whole problem.

[14] In the literature, such paths are called *initialized* paths [8].

Hence, the general structure of the sat problem $[\![K, \neg\phi]\!]_k$ to be fed to a SAT solver is described by the following equation:

$$[\![K, \neg\phi]\!]_k := \underbrace{\left([\![S_0]\!] \wedge \bigwedge_{i=0}^{k-1} [\![T]\!]_k^i\right)}_{\text{The automaton formula } [\![K]\!]_k} \wedge \underbrace{\left([\![\neg\phi]\!]_k^0 \vee \bigvee_{l=0}^{k} \left({}_lL_k \wedge {}_l[\![\neg\phi]\!]_k^0\right)\right)}_{\neg\phi \text{ bounded semantics}}$$

Without loss of generality, we can assume that all the variables in the model are boolean. In that case,

- $[\![S_0]\!]$ is defined as the conjunction of the constraints on the value of all the variables in the initial state S_0 of the automaton.
- $[\![T]\!]_k^i$ is defined as the conjunction of the constraints on the value of the variables after the transition $T(i, i+1)$ has been chosen.
- ${}_lL_k$ is true iff $k > l \wedge (k, l) \in T$.

The expressions $[\![\neg\phi]\!]_k^0$ and ${}_l[\![\neg\phi]\!]_k^0$ are recursively defined as follows:

For a straight path

Whenever $i > k$	
$[\![\phi]\!]_k^i$	$:= \bot$
Whenever $i \leq k$	
$[\![p]\!]_k^i$	$:= p(\pi_i)$
$[\![\neg p]\!]_k^i$	$:= \neg p(\pi_i)$
$[\![\phi \vee \psi]\!]_k^i$	$:= [\![\phi]\!]_k^i \vee [\![\psi]\!]_k^i$
$[\![\phi \wedge \psi]\!]_k^i$	$:= [\![\phi]\!]_k^i \wedge [\![\psi]\!]_k^i$
$[\![\bigcirc \phi]\!]_k^i$	$:= [\![\phi]\!]_k^{i+1}$
$[\![\Box \phi]\!]_k^i$	$:= \bot$
$[\![\Diamond \phi]\!]_k^i$	$:= [\![\phi]\!]_k^i \vee [\![\Diamond \phi]\!]_k^{i+1}$
$[\![\phi \mathbf{U} \psi]\!]_k^i$	$:= [\![\psi]\!]_k^i \vee \left([\![\phi]\!]_k^i \wedge [\![\phi \mathbf{U} \psi]\!]_k^{i+1}\right)$
$[\![\phi \mathbf{W} \psi]\!]_k^i$	$:= [\![\psi]\!]_k^i \vee \left([\![\phi]\!]_k^i \wedge [\![\phi \mathbf{W} \psi]\!]_k^{i+1}\right)$

For a (k,l)-loop

Assuming $k, l, i \geq 0$ *with* $l, i \leq k$	
${}_l[\![p]\!]_k^i$	$:= p(\pi_i)$
${}_l[\![\neg p]\!]_k^i$	$:= \neg p(\pi_i)$
${}_l[\![\phi \vee \psi]\!]_k^i$	$:= {}_l[\![\phi]\!]_k^i \vee {}_l[\![\psi]\!]_k^i$
${}_l[\![\phi \wedge \psi]\!]_k^i$	$:= {}_l[\![\phi]\!]_k^i \wedge {}_l[\![\psi]\!]_k^i$
${}_l[\![\bigcirc \phi]\!]_k^i$	$:= {}_l[\![\phi]\!]_k^{succ(i)}$
${}_l[\![\Box \phi]\!]_k^i$	$:= {}_l[\![\phi]\!]_k^i \wedge {}_l[\![\Box \phi]\!]_k^{succ(i)}$
${}_l[\![\Diamond \phi]\!]_k^i$	$:= {}_l[\![\phi]\!]_k^i \vee {}_l[\![\Diamond \phi]\!]_k^{succ(i)}$

Where the function $succ(i)$ is used to denote the index of the successor of the state s_i in a (k,l)-loop. This function is defined as

$$succ(i) := \begin{cases} i+1 & when\ i < k \\ l & otherwise \end{cases}$$

In order to make the semantic difference between $\cdot\ \mathbf{U}\ \cdot$ and $\cdot\ \mathbf{W}\ \cdot$ apparent in the definition of ${}_l[\![\phi]\!]^i_k$, the auxiliary expression ${}^c_l[\![\phi]\!]^i_k$ is used. Intuitively, this auxiliary expression must be understood as the value of ${}_l[\![\phi]\!]^i_k$ after c unrolling steps have been performed along the (k,l)-loop. More formally,

$$\textit{Assuming } k, l, i, c \geq 0 \textit{ with } l, i, c \leq k$$

$$
\begin{array}{lll}
{}_l[\![\phi\ \mathbf{U}\ \psi]\!]^i_k & := {}^0_l[\![\phi\ \mathbf{U}\ \psi]\!]^i_k & \\
{}_l[\![\phi\ \mathbf{W}\ \psi]\!]^i_k & := {}^0_l[\![\phi\ \mathbf{W}\ \psi]\!]^i_k & \\
{}^c_l[\![\phi\ \mathbf{U}\ \psi]\!]^i_k & := \begin{cases} \bot & \text{when } c = k \\ {}_l[\![\psi]\!]^i_k \vee \left({}_l[\![\phi]\!]^i_k \wedge {}^{c+1}_l[\![\phi\ \mathbf{U}\ \psi]\!]^{succ(i)}_k \right) & \text{otherwise} \end{cases} \\
{}^c_l[\![\phi\ \mathbf{W}\ \psi]\!]^i_k & := \begin{cases} \top & \text{when } c = k \\ {}_l[\![\psi]\!]^i_k \vee \left({}_l[\![\phi]\!]^i_k \wedge {}^{c+1}_l[\![\phi\ \mathbf{W}\ \psi]\!]^{succ(i)}_k \right) & \text{otherwise} \end{cases}
\end{array}
$$

The correctness of the definition using this auxiliary expression stems from the fact that:

- Any trace is necessarily shorter than k steps.
- In particular, the (k,l)-loop must be shorter than k steps.
- Given that the path is a (k,l)-loop and is shorter than k steps, it suffices to perform k steps in that loop to produce the prefix of an infinite trace of the system (one that never leaves the loop).

Important observations

It is important to emphasize on the fact that, as explained in the preliminary note of this section, the formula $\neg\phi$ must be converted to negation normal form *before* even starting to generate $[\![\neg\phi]\!]_k$. Indeed, because the bounded semantics of LTL breaks the (pseudo-) duality rules that exist between the various operators; it is *not* always the case that $[\![\neg\phi]\!]_k = \neg[\![\phi]\!]_k$. In particular, in the case of the verification of the formula $\Box\ \phi$; the generated SAT problem would combine $[\![K]\!]_k$ and $[\![\neg\Box\ \phi]\!]_k$. With that example, it is easy to see that under the "straight path" assumption; failing to convert the formula to NNF would yield $\neg[\![\Box\ \phi]\!]_k = \top$ (which is *always* satisfied) instead of $[\![\Diamond\ \neg\phi]\!]_k = [\![\phi]\!]^i_k \vee [\![\Diamond\ \phi]\!]^{i+1}_k$ (which may be unsatisfiable).

It is also interesting to note that, the proposed reduction of LTL bounded model checking to propositional SAT solving "is linear (in time and space) with respect to the size of ϕ and the bound k if subterms are shared" [7, p. 12, 13].

Part II

Extension of PyNuSMV and BMC implementation

Chapter 3

The framework

The second part of this work is dedicated to the presentation of the actual development that has been realized in order to extend PyNuSMV with bounded model checking capabilities. Concretely, this presentation of the project will be organized in two chapters. The first one will focus on the way the PyNuSMV framework has been extended so as to give the reader an overview of the technological choices that have been operated, the architecture of the system and the testing methodology that has been applied.

The second chapter, demonstrates the usability of our extension of the framework through two case studies. The first one illustrates how one can use the SAT-BMC extension to PyNuSMV to develop tools that verify custom logics. To this end, a new LTL SAT-based bounded model checking tool will be implemented as if it were for a brand new logic. The second case illustrates how one can use the practical outcome of this master's thesis to write verification tools able to deal with non-linear logics. Concretely, this illustration will be done through the explanation of a tool able to perform a diagnosability test as explained in [39].

3.1 Architecture and technological choices

As a continuation of the effort undertaken by the Louvain Verification Lab to develop PyNuSMV, the work that has been realized in order to extend this framework with SAT-based bounded model checking facilities had to integrate smoothly with the existing code base. To this end, it was decided to keep the same overall architecture and technological choices than for the rest of the source code.

The high level architecture of PyNuSMV is depicted in Figure-3.1. As can be seen on that picture, the implementation of the software library is divided in three layers. The bottom one comprises the NuSMV base code, the two SAT solvers (namely MiniSat [19] and ZChaff [34, 22]) that can be linked with it and some *glue code* meant to facilitate the interactions between Python[1] and the APIs of the previously mentioned tools. The particularity of this layer is that, as opposed to the other two layers, it is entirely written in C language. The so called *lower interface* is composed of all the Python wrappers generated by SWIG which expose the low level APIs of the bottom layer in Python. The topmost layer, called *upper interface*, is the one which required the most effort: it provides modules, classes and functions encapsulating higher level abstractions of the concepts defined in the two other layers; hence raising the productivity of new developers using the library and shielding them from the burden of i.e. memory management.

[1] Actually, Python 3.

Figure 3.1: Architecture of PyNuSMV with SAT-BMC (Adapted from [10]).

3.1.1 SWIG

As mentioned previously, the extension of the framework maintains the technical choices that have been operated by the Louvain Verification Lab when developing PyNuSMV. Amongst these, the most significant one is the use of SWIG [6, 45, 46][2] in order to generate the Python modules composing the *lower interface*. The advantage of this technique stems from the fact that although most of the code is generated entirely automatically, SWIG offers some hooks to customize the generated Python code. Therefore, the modules composing the lower interface could be generated with minimal effort and runtime overhead while still closely matching the needs of the upper interface.

3.1.2 Upper interface

This section succinctly describes the roles and features of the most important classes and modules that have been developed in order to make the SAT-BMC APIs of NuSMV easily understandable and usable by anyone.

Conceptually, the modules that have been added to the existing framework can be partitioned in three sets as shown on Figure-3.2. The *verification* set contains a single package (*bmc*) whose role is to expose some high level features related to the verification of logical properties as performed by NuSMV. The purpose of the packages and modules grouped in the *core* set is to encapsulate the core abstractions which are manipulated while performing bounded model checking. The most representative of these being the *SAT solvers* and the *boolean expressions*. Finally, a third set of modules was developed in order to tackle the need of the former two sets to rely on some *general purpose* abstractions such as a library of collections.

Verification

As explained previously, the *verification set* comprises the *bmc* package only. This package is composed of four distinct submodules:

- `pynusmv.bmc.glob` which has a role analogous to the preexisting `pynusmv.glob` and serves to initialize the NuSMV global structures dedicated to bounded model checking. In particular, this module provides the convenient `BmcSupport` class which acts as a python context manager and initializes/releases any resources held by the BMC subsystem. In addition to that, it also provides an easy access to the `master_global_be_fsm`. That is to say, the global object holding a representation of the FSM encoded with boolean expressions.
- `pynusmv.bmc.ltlspec` which provides the several high level functions related to the verification of LTL properties. For instance, `check_ltl` and `check_ltl_incrementally`

[2]The acronym SWIG stands for Simplified Wrapper and Interface Generator.

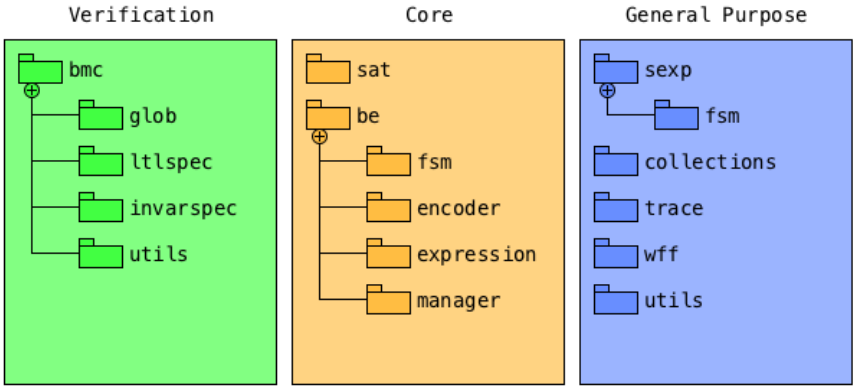

Figure 3.2: The three sets of modules that have been developed in to bring SAT-BMC capabilities to PyNuSMV.

to perform an end-to-end verification of some property passed as argument[3] or the various function related to the generation of the SAT problem like i.e. `generate_ltl_problem` and `bounded_semantics_*`.

- `pynusmv.bmc.invarspec` which offers pretty much the same services as those offered by `pynusmv.bmc.ltlspec` except that the functions do not aim at the verification of LTL properties[4] but invariant properties[5]. Technically, the verification technique implemented by the functions in this module is *not* the *SAT-based bounded model checking* which has been presented in the first part of this master thesis. Instead, this technique is called *temporal induction*[6] and will not be detailed in these pages in order to save the reader's time and fit within the limits imposed on the size of this dissertation.
- `pynusmv.bmc.utils` which provides utility functions that can prove to be useful when generating a SAT problem for the bounded verification of a formula expressed in a new kind of logic (an example of this will be given later). For instance, it provides the `loop_condition` function which returns a boolean expression encapsulating all the constraints that need to be satisfied for ${}_l L_k$ to be true.

Core

Because *SAT-solvers* and *boolean expressions (be)* are the two key concepts that intervene when one is elaborating a BMC verification technique for some temporal logic, it was very natural to put these notions at the heart of the development. Therefore, in addition to the `SatSolver` class which stands for the external SAT solving tools, a `Be` class was foreseen to represent boolean expressions. Conceptually, the `Be` objects are the ones used to encode the $[\![\cdot]\!]_k$ expressions presented in the theoretical part of this differtation. Hence, they represent a logical propositions that may refer to the (timed) variables of the model under consideration. In their simplest form, `Be` consist of either one of the constants `Be.true`, `Be.false` or variable of the system state after a given number of 'steps' in a trace.

[3] Either using a simple unrolling of the transition relation or an incremental one.
[4] As would be encoded with *LTLSPEC* in the NuSMV model.
[5] As would be encoded with *INVARSPEC* in the NuSMV model.
[6] Note, although SAT-BMC and temporal induction are similar to one another; the temporal induction technique will not be detailed in the scope of this thesis. Meanwhile, the interested reader will find all the necessary information in [18].

For example, in PyNuSMV, the constraint stating that the boolean variables x and y must be equal at time 3 of a trace is expressed as follows:

```
var_x_3    = enc.by_name['x'].at_time[3].boolean_expression
var_y_3    = enc.by_name['y'].at_time[3].boolean_expression
constraint = var_x_3.iff(var_y_3)
```

Concretely, the implementation of these concepts in PyNuSMV led to the creation of one module (sat) to gather all SAT solving related classes and one package (be) subdivided in four submodules to organize the code related to the manipulation of boolean expressions.

The main contributions of the `pynusmv.sat` module are the standardized interfaces for the various supported solvers[7] enabling their use as either regular or incremental solvers. Roughly speaking, the major difference that exists between a regular solver vs an incremental one is the ability to store CNF clauses in named groups which may then be included (or excluded) from a solving iteration. This makes for very efficient solving of incremental problems when each 'increment' is stored in a named group since it gives the solver the opportunity to remember inferences that were made during a previous iteration. Besides that, the module also provides a handy `SatSolverFactory` which eases the instantiation of these solvers shielding the developer from the need of remembering the low level APIs specific to the instantiation of each such tool.

The internals of the `pynusmv.be` package are somewhat more complex. Indeed, an efficient implementation of SAT-BMC does not only require the ability to combine boolean expressions with one another in order to build complex constraints but also has the need for a *manager (BeManager)* whose role is to encode the formula terms in *conjunctive normal form (CNF)* making sure to share clauses amongst multiple formulas whenever possible[8].

Besides that, the temporal nature of the properties being verified requires the ability to evaluate the value of the variables composing the state of the KS at different moment in time. For instance, this feature is necessary whenever one generates a boolean formula and states that it should be verified at time t (after the considered trace is t steps long). As illustrated in Figure-3.3, this requirement is fulfilled by the *encoder (BeEnc)* which can be seen as an array-like structure. In that structure, each cell represents the state of the automaton at a given moment and stores a set of bits; one for each variable composing the state of the FSM. Thanks to this representation, it is not only possible to efficiently access a variable v at time t but also to declare a boolean expression using the *untimed version* of the variables composing it and then *shift* these to the desired time at which the expression should apply.

Moreover, because there are many different way to identify a single variable in that model; for instance, the symbol used in the SMV text, the number used in the CNF formula standing for that variable, etc... an abstraction *BeVar* was introduced to explicitly represent and manipulate a boolean variable as encoded in the *BeEnc*. As will be shown in the next chapter, this proved to be a very useful choice which helped in defining concise constraints over the paths considered during the verification.

[7] MiniSat, ZChaff.

[8] As explained in [8], this is done to avoid the generation of a CNF formula ϕ' with a length exponential in the size of the formula ϕ it normalizes.

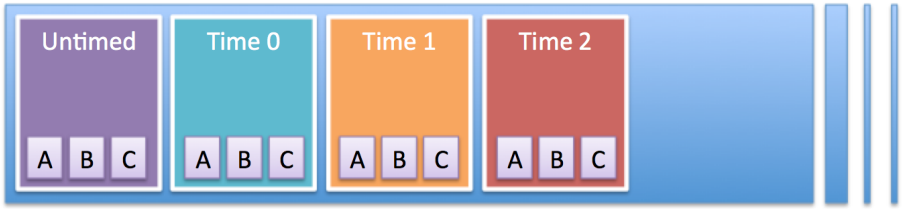

Figure 3.3: Graphical representation of the structure of a BeEnc where A,B,C are the name of variables of the model.

General purpose

The modules that have been grouped in the general purpose set of Figure-3.2 are not *directly* tied to BMC and might therefore be reused in other contexts. For example, the classes *SexpFSM* and *Wff* operate on Nodes which are the PyNuSMV representation of a subtree of the parsed SMV text encoded in the form of an abstract syntax tree (AST). Concretely, *SexpFSM* provides services to interact with the KS described by the SMV text while Wff[9] provides convenient functionalities to build complex logical properties and convert them to negation normal form. Both of these classes could be easily reused, for instance to further extend the set of BDD-related functionalities exposed in PyNuSMV.

Similarly, the classes defined in the `pynusmv.trace` module are not specifically tied to BMC either albeit it is the only place where those classes are used at the time being. Indeed, the combination of *Trace* and *TraceStep* allows to easily create and manipulate counterexample traces to illustrate the violation of an LTL property.

In addition to that, it is also worth mentioning that quite some effort has been invested in porting the most commonly used NuSMV collections to PyNuSMV. The rationale behind this choice is twofolds:

1. Doing so allows to reduce the effort required in order to bring new NuSMV *classes*[10] to the upper interface of PyNuSMV.
2. It avoids unnecessary iterations over the complete collection. Thanks to this approach, it is perfectly possible to retrieve a *NodeList* or an *SList* using an API and pass it on to an other one without modification. Meanwhile, the so called *Python magic methods* [29] used in the development of these classes combined with the fact that they inherit from the Python collections abstract base type make the use of such classes almost transparent to the programmer willing to interact with the content of such collections.

Finally, the `pynusmv.util` was enriched with the *@writeonly* and *@indexed* decorators which act in a fashion similar to the standard `@property` and combine several different methods to give the programmer the illusion of working with a collection field. In addition to the aforementioned decorators, the *StdioFile* context manager was also added to help interacting with legacy API's that are unable to interact with Python file-objects[11].

[9] WFF stands for Well Formed Formula.

[10] Despite its writing in C, NuSMV is heavily geared towards Object Oriented programming. Therefore, many of the OOP concepts (inheritance, polymorphism, etc..) have been used in the definition of the different modules. Hence, it seems perfectly appropriate to talk about NuSMV classes although no such concept exists at the language level.

[11] As of Python3, the *file-objects* (streams) are no longer compatible with the usua FILE* defined in the C standard library.

3.2 Development methodology

Practically, the development of the upper interface was organized as a set of small iterations each of which was designed to perform groundwork and gain a better understanding of the underlying C code in order to ease the development of the next iteration. Therefore, a *bottom up* methodology was chosen: starting with the general purpose classes then the boolean expressions and sat solvers etc. because it reduced the set of dependencies required by each of the developed components. However, this approach turned to be impractical since it failed to reveal the global picture behind the complexity of the NuSMV C code base. Therefore, this strategy was changed in a later phase of the development and a top down, goal oriented approach was adopted. Meanwhile, the iteration level organization of the development was kept intact.

Concretely, each iteration consisted of three phases: first, the NuSMV source code was analyzed in order to highlight the key structures of the module at hand and their interactions. Then, the python programming activity took place paying close attention to providing thorough documentation for each of the modules, classes and functionalities exposed. Finally, unit tests were written based on the documentation in order to validate the behavior of the code. In addition to that, a code coverage tool (namely the `coverage.py` tool for python [5]) was used in order to help identify the portions of the source code that required more extensive testing[12]. Those tests were run on a regular basis during the development iterations in order to lower the risk of regression and the introduction of new bugs.

3.3 Challenges encountered and trade-offs

Although the source code of NuSMV is well structured and most of the time well documented, some code-specific challenges had to be faced during the development phase. The first one stems from the fact that NuSMV was not designed to be a reusable *library* but rather to be a *tool*. Therefore, some of the most important data structures are stored as global variables and this choice had to somehow be translated in the code of PyNuSMV. The most prominent example of this choice being the *boolean encoded FSM* which is made accessible through the function `master_be_fsm()` in the module `pynusmv.bmc.glob`.

Besides that, defining a proper delimitation of the set of NuSMV features and data structures to expose through the upper interface of PyNuSMV proved to be harder than expected. Indeed, it wasn't always obvious to determine whether exposing a data structure was relevant or not. In those border cases, the approach that has been adopted was to expose as much services as possible via the upper interface in order to make the framework as feature rich as it could. However, in some rare cases, it was decided otherwise. For instance, the module C-module `rbc` wasn't exposed although it defines the structures that are used to *internally* represent a boolean expression as a *reduced boolean circuit*[13]. Because the `rbc` structure is never used for itself but is instead always used as an opaque `Be_ptr = void*`, it was decided not to expose the *rbc* APIs via the upper interface of PyNuSMV.

Conversely, some of the functions that are considered desirable from the upper interface perspective have been implemented as `static` functions in the C code and are there-

[12] See appendix C and in particular Figure-C for more details about the test coverage of the developed code.

[13] That is to say, a *directed acyclic graph* used to reduce the size of the boolean expression encoded this way.

fore not accessible to the upper layers. In particular, this is the case for the function `Bmc_Tableau_GetLoopCondition(enc, k, l)` responsible of generating a constraint stating that the path k, l, is a (k,l)-loop. In those cases where the functionality couldn't be immediately reused from NuSMV the decision was taken to re-implement the necessary functions in order to make them accessible through the upper interface without tampering with the code base of NuSMV.

To conclude this paragraph, let us also mention the fact that the boolean encoder as it has been programmed in NuSMV is only able to shift boolean variables from the *untimed block* to some other timed block. This is, in the author's opinion, a major limitation since it prevents a developer to encode two parallel traces one after the other as will be explained in paragraph 4.3.4. In order to compensate that limitation, and because this ability was considered to be a key feature of the framework, it was decided to implement a series of function `*_at_offset` that generate the desired boolean expression starting at the right encoder offset.

3.4 Summary of the development

In this chapter we have seen that the development realized in the scope of this master's thesis was conceived as an extension to PyNuSMV and consequently adopted its three layer architecture. We then mentioned that SWIG has been used in that context to bridge the gap between the bottom layer written in C language and the higher levels: the lower and upper interface. After that we gave a tour of the elements in each of the three sets of modules: *the verification set*, *the core set* and the *general purpose set* and explained their roles in order to show how they help to raise the level of abstraction in *the upper interface* (and hence the developer productivity). Then a word was given about the iterative-style methodology that was used throughout the whole development process and finally, some of the trade-offs that were made and challenges that have been encountered were explained.

Because the length of this dissertation is limited, the role of all the important data structures and functions that were developed could not be presented in the previous pages. Meanwhile, the reader willing to get a more detailed overview of the additions that were made to PyNuSMV is invited to consult Appendix-D.

Chapter 4

Experimental validation

This chapter, illustrates the usability and performance of the SAT-BMC addition to PyNuSMV through two distinct case studies. The first one demonstrates how easily one can use the *core* set of modules to develop SAT-BMC tools to perform the verification of new logical formalisms. This illustration takes the form of a new LTL verification tool implementing the reduction of bounded model checking to SAT exactly as explained in the section 2.3.3. The second one serves the purpose of illustrating that the work which has been done in the scope of this master's thesis does not only enables the creation of verification tools for linear time logics but also opens the door to more complex setups. Concretely, this second example consists in the implementation of a diagnosability test tool and shows how the classes and high level functions that are exposed through the upper interface can be used to implement verification techniques that were not initially foreseen in NuSMV.

4.1 LTL verification tool

Before entering in the details of *how* the LTL verification tool has been implemented, let us first use the upcoming lines to detail *what* has been realized. The program that has been written comes in the form of a simple command line tool which uses an SMV model as input and accepts LTL formulas to verify either on the command line or from the standard input[1]. As opposed to the approach taken by NuSMV, this tool does not implement any of the *past* operators[2] but it does offer the *weak until* modality which is absent from the standard implementation. Moreover, in order to be consistent with the notations used throughout this dissertation, `()` `x` is used to denote $\bigcirc x$, `[]` `x` for $\Box x$ and `<>` x for $\Diamond x$[3].

4.1.1 High level technical overview

From a more technical point of view, because the objective of the development of the tool is to illustrate how the core modules can be used to implement a BMC technique for a new logic; it was decided to not use any of the high level features defined in `pynusmv.bmc.*`. The only exception to this rule being the function `generate_counter_example` which is defined in the `pynusmv.bmc.utils` module. This choice was simply motivated by the fact that this function does not interfer with the actual verification and is more tied to the structure of the encoder *(BeEnc)* than the LTL logic itself. However, using the

[1] For details about the usage of this tool, use the `-h` flag to display the built-in help.
[2] Previously, since, historically.
[3] This syntax was inspired by that of LTSA [31].

`BeEnc.decode_sat_model(..)` was also a viable option which would only have led to a very small increase in the complexity of the code written down.

Architecture

Practically, the code of the LTL verification tool is structured in five python modules:

- `tools.bmcLTL.main` which, as the name suggests, is the main entry point of the tool. It uses the `argparse` [21] library in order to correctly process the options passed via the command line and to display an helpful message if need be.
- `tools.bmcLTL.parsing` which uses the `pyparsing` [32, 1] to implement a parser able to deal with the recursive grammar of LTL.
- `tools.bmcLTL.check` which implements the high level verification functions `check_ltl` and `check_ltl_onepb` which respectively perform the verification of an LTL formula for all trace sizes until the bound is reached, or the verification of that same LTL formula for all traces having an exactly specified length.
- `tools.bmcLTL.gen` which contains the implementation of all the functions used to generate the SAT problem $[\![K, \neg\phi]\!]_k$ at the core of the BMC technique as well as many variants that may be configured by the end user. For instance, this module contains the code necessary to generate the constraints that force the verification process to only consider *fair traces*[4](enabled by default) or the constraints to enforce/ignore the invariant assumptions declared in the SMV model.
- `tools.bmcLTL.ast` which really constitutes the heart of this tool and implements one class per type of symbol that may possibly appear in an LTL formula alongside with some utility function (for instance the generation of a loop condition of fixed length). A structural view of these classes is reproduced in Figure-4.1.

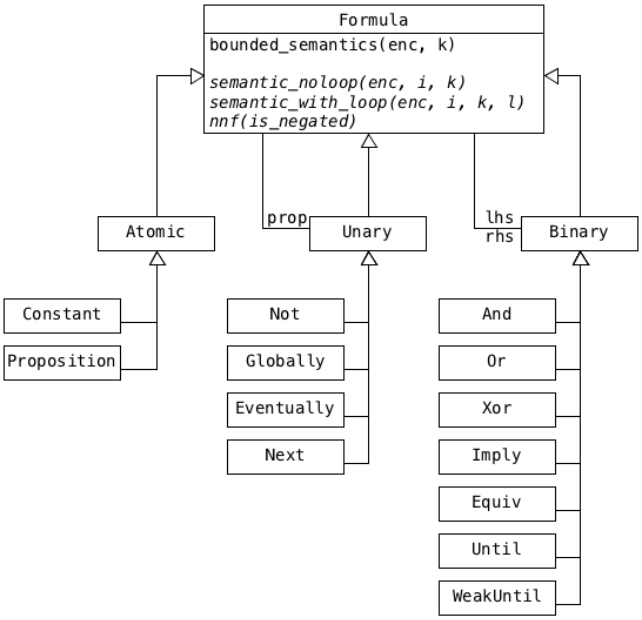

Figure 4.1: Structural overview of the tools.bmcLTL.ast module

[4]The fairness condition must nevertheless be declared in the SMV text in order to be taken into consideration.

4.1.2 Handling formulas with Python code

As illustrated by Figure-4.1, the python code to implement the conversion of a `Formula` object to the corresponding $[\![\phi]\!]_k$ revolves around three methods:

- `semantic_no_loop(enc, i, k)` which serves to produce the expression $[\![\phi]\!]_k^i$ standing for the conservative bounded semantic of the formula ϕ in the case where the considered trace is a straight path.
- `semantic_with_loop(enc, i, k, l)` which serves to produce the expression ${}_l[\![\phi]\!]_k^i$ that can be used to assess the infinite behavior of ϕ provided that the path contains a (k,l)-loop.
- `nnf(is_negated)` which serves to "push" the negation signs inwards so as to yield a *negation normal form* formula equivalent to ϕ.

To make things more concrete, let us consider the code snippet in Figure-4.2 which features the complete implementation of these three methods for the *globally* modality. This snippet was chosen over the other implemented operators because it is both concise, easy to understand while still sufficiently elaborated to illustrate what it takes to implement a real, interesting operator using our extended version of the PyNuSMV framework.

```
class Globally(Unary):
    '''
    Encapsulates the logic associated with the verification of a property of the form [] phi
    '''
    def semantic_no_loop(self, enc, i, k):
        ''' The generation to apply when the path is a straight line '''
        return Be.false(enc.manager)

    @memoize_with_loop
    def semantic_with_loop(self, enc, i, k, l):
        ''' The generation to apply when the path is a (k,l)-loop'''

        # without moving at least one step, it is impossible to go through a loop
        if k == 0:
            return Be.false(enc.manager)

        # internal function to iterate over all times in the loop and make express
        # that the formulaholds iff the subformula on which [] bears is always true
        # on that loop.
        def _semantic(time, cnt):
            if cnt == k:
                return Be.true(enc.manager)
            now = self.prop.semantic_with_loop(enc, time, k, l)
            return now & _semantic(successor(time, k, l), cnt+1)

        return _semantic(i, 0)

    def nnf(self, negated):
        ''' Converts this sub formula to negation normal form '''
        if not negated:
            return Globally(self.prop.nnf(False))
        else:
            return Eventually(self.prop.nnf(True))
```

Figure 4.2: The class responsible for the handling of formulas of the form $\Box\ \phi$

Taking a closer look at the implementation, it is easy to convince oneself that the code effectively corresponds to the bounded semantics of a formula of the form $\Box\ \phi$: indeed, as announced in section 2.3.3, in the absence of a loop the constant *false* is always returned (line 7). Besides that one also observes that the code of `semantic_with_loop(enc, i, k, l)` constitutes a relatively straightforward object-oriented translation of its formal definition:

- **Line 22**: The constant *true* is returned when all the possible steps in the loop have been exhausted because *true* is neutral for the $\wedge$ operation.
- **Line 23**: The computation of the constraint expressing that ϕ must hold at the current time i is delegated to `self.prop` which actually stands for ϕ in a more formal parlance.
- **Line 24**: A recursive call is made to ensure that the formula ϕ not only holds at time i but also in the state succeeding i in the loop.

Finally, one can also observe at lines 28-33 that the conversion to NNF is performed recursively, pushing the negations signs inwards and rewriting the formula according to the duality rules given in section 1.2.2 if need be.

Meanwhile, the results of `semantic_no_loop()` and `semantic_with_loop()` cannot immediately combined in order to obtain $[\![\phi]\!]_k$. As a matter of fact, because `semantic_with_loop()` does not generate any constraint to enforce the presence of a loop on the path, a little bit of extra code (Figure-4.3) is necessary.

```
1  # part of the Formula abstract base class.
2  def bounded_semantics(self, enc, k):
3      '''
4      Returns a boolean expression corresponding to the bounded semantics of the formula
5      denoted by `self` on a path of length k. This combines both the semantics in case
6      of a loopy path and the case of a non-loopy path.
7      '''
8      noloop = self.semantic_no_loop(enc, 0, k)
9
10     w_loop = Be.false(enc.manager)
11     for l in range(k):    # [0; k-1]
12         w_loop |= (loop_condition(enc, k, l) &
13                    self.semantic_with_loop(enc, 0, k, l))
14
15     return noloop | w_loop
```

Figure 4.3: Generation of the expression $[\![\phi]\!]_k$

Moreover, because NuSMV has no explicit notion of state and does not provide an easy access to the *meaning* of the transition relation; the `loop_condition` constraint is defined as the equality between all the variables in the states s_k and s_l. The correctness of this formula stems from the definition of a state which was given in section 1.1.1. Indeed, in that context, we defined a state as a variable *valuation*. That is to say, an assignment of a value to each of the variables composing the model. Hence is easy to convince oneself that the constraint produced this way (see Figure-4.4) is only satisfiable in the case where the path π is a (k,l)-loop.

```
def loop_condition(enc, k, l):
    '''
    This function generates a Be expression representing the loop condition
    which is necessary to determine that k->l is a backloop.
    '''
    cond = Be.true(enc.manager)
    for v in enc.curr_variables: # for all untimed variable
        vl   = v.at_time[l].boolean_expression
        vk   = v.at_time[k].boolean_expression
        cond = cond & ( vl.iff(vk) )
    return cond
```

Figure 4.4: Generation of the loop condition ${}_lL_k$

4.1.3 The automaton

However, being able to generate a boolean expression $[\![\phi]\!]_k$ to represent the bounded semantics of the (negation of) the property we want to verify is only part of the problem. Indeed, the SAT-BMC technique requires to combine that expression with an other representing the system of interest in order to find potential violation traces.

Conceptually, generating that second expression requires to start from the initial state of the automaton and unroll k steps all the traces belonging to the language of the automaton. As shown hereunder, in PyNuSMV, one can achieve that goal utilizing the general APIs provided by the classes `BeFsm` and `BeEnc`. These classes respectively provide provide ways to interact with a *booleanized version*[5] of the FSM (for instance: `fsm.init` to get the initial state, `fsm.trans` to get an expression encoding the transition relation, etc..) and to shift entire BEs to some desired time (`enc.shift_to_time(..)`).

```
def model_problem(fsm, bound):
    """
    Computes the unrolled automaton expression [[M]]_{k}
    """
    enc    = fsm.encoding
    # initial state
    init0  = enc.shift_to_time(fsm.init, 0)
    # transition relation (unrolled k steps)
    trans  = Be.true(enc.manager)
    for k in range(bound):
        trans = trans & enc.shift_to_time(fsm.trans, k)

    return init0 & trans
```

Figure 4.5: Generation of the $[\![K]\!]_k$ expression

[5] The complete model structure is *flattened* to be entirely represented in terms of the variables composing it; and all variables are reduced to bits which are interpreted as boolean variables.

4.1.4 Detection of counterexamples

To conclude the technical presentation of LTL verifier tool that has been developed to illustrate the use of the core PyNuSMV SAT-BMC functionalities, let us show how all the pieces fit together and how one uses the `sat` module APIs to produce a violation trace.

In the Figure-4.6, the lines 6-8 proceed as explained in the previous sections to generate the SAT problem which will be submitted to the SAT solver. Then this problem is converted to *conjunctive normal form* to produce (line 8) a `BeCnf` object which may then be processed by the solver. In order to make this happen, a solver is created (line 10) and the clauses composing the CNF expression are added to the clause database of the solver (line 11). The next line assigns a *polarity* to the set of clauses that have just been added to the database of the solver. This step simply tells the solver whether the symbols in that set of clause should be considered positive or negative (all literals are negated). Although not strictly required, this operation is greatly helps the solver in being efficient since it enable the use of the *pure symbol heuristic* implemented in most modern solvers. Finally the `solve()` method of the solver is invoked (line 14) to determine the existence of violation traces in the model.

```
1  def check_ltl_onepb(fml, length, no_fairness=False, no_invar=False):
2      """
3      This function verifies that the given FSM satisfies the given property
4      for paths with an exact length of `length`.
5      """
6      fsm    = master_be_fsm()
7      pb     = generate_problem(fml, fsm, length, no_fairness, no_invar)
8      cnf    = pb.to_cnf()
9
10     solver = SatSolverFactory.create()
11     solver += cnf
12     solver.polarity(cnf, Polarity.POSITIVE)
13
14     if solver.solve() == SatSolverResult.SATISFIABLE:
15         cnt_ex = generate_counter_example(fsm, pb, solver, length, str(fml))
16         return ("Violation", cnt_ex)
17     else:
18         return ("Ok", None)
```

Figure 4.6: Utilization of the SAT solver related APIs

4.2 Performance evaluation

The performance of the BMC extension of PyNuSMV was evaluated through a benchmark comparing two distinct implementations of an LTL SAT-based bounded model checker. The two implementations that have been used are the following:

- **The regular NuSMV** implemented in C language, configured to use the MiniSat solver as it was observed to deliver faster results.
- **The tool implemented in section 4.1**.

4.2.1 Methodology

In order to get a clear idea of the performance of both of the different implementations and deliver results which are as fair and representative of the capabilities of each tool as possible, a set of test scenarios have been devised (called *test bench* as of now). For each scenario of the test bench, the execution time of each implementation was measured using the system time of the test machine. Moreover, in order to dampen the effects of potential impediments not directly related to the test being run, each test case has been reproduced five times and the results were averaged on a per implementation basis.

The test bench itself develops along four dimensions characterizing the factors that can possibly limit the scalability of the SAT-BMC technique:

- **The size of the Kripke structure** because it determines the number of boolean variables encoded in each time block of the global BeEnc instance.
- **The problem bound** (maximum number of steps allowed in a trace) because longer traces mean larger boolean expressions to generate and hence, larger SAT problem to solve.
- **The formula depth**, that is to say: the number of nesting levels of the temporal operators. For instance $\Box\ \phi$ has a depth of one whereas $\Diamond\ \Box\ \phi$ has a depth of two.
- **The difficulty of the problem**. Indeed, not all problem instances are equally difficult to solve for the various tools and in general, because of the pruning heuristics implemented in the modern SAT solvers; the test cases for which the verifier must find a *long* counterexample take much more time to solve than those for which no counterexample or a short one must be found.

Concretely, in order to make the size of the KS representing the system vary, an SMV model for the well known *dining philosophers problem*[6] was used and adapted to make the number of *philosophers* and *forks* range from two to fifity nine. For each version of the model, the different tools were asked to verify the following two LTL properties: $\Box\ (p1.waiting \Rightarrow \Diamond\ \neg p1.waiting)$ and $\Diamond\ \Box\ (p1.waiting \Rightarrow \Diamond\ \neg p1.waiting)$ and the tests were repeated and measured for problem bounds ranging from 10 to 30. The first property, $\Box\ (p1.waiting \Rightarrow \Diamond\ \neg p1.waiting)$, stipulates that the first philosopher is never stuck in a deadlock. The second property is somewhat more contrived: it stipulates that after a (potentially long but) finite amount of time, the first philosopher can no longer be stuck in a deadlock. Even though the usefulness of verifying this property is debatable, this property was arbitrarily chosen as an example of property of depth three.

Finally, in order to make the problem difficulty vary, a different fairness assumption was used for the set of *easy* instances than for the *hard* ones. Concretely, the *easy* instances used the assumption `!waiting` which stipulates that in the traces considered, no philosopher must be waiting for ever whereas the *hard* instances used the assumption `running` which stipulates that no philosopher *process* must be inactive forever. Although this difference might seem a little bit subtle, it is nonetheless remarkably important: `waiting` forces the different philosophers processes to make progress whereas `running` only makes them active.

The details of the measurements can be found on the Github repository of the project.

[6] This model is a variation of the one proposed by D. Hoffman from East Carolina University which was detailed in `https://youtu.be/JBESxiZ5Wao` .

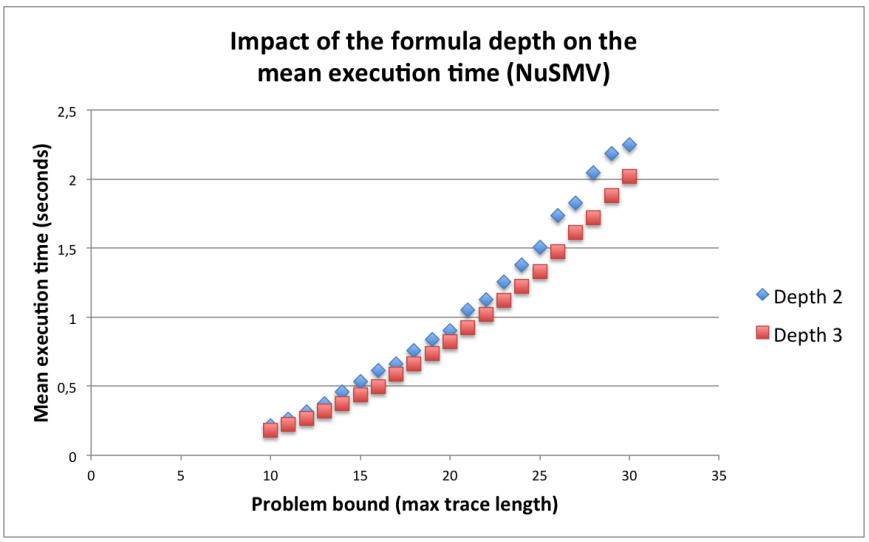

Figure 4.7: Impact of the formula depth on the mean execution time. This graph shows the measurements done while running NuSMV on an easy instance with ten philosophers for formulas of depth two and three with problem bounds ranging from 1 to 30.

4.2.2 Observations

There are essentially four observations to be made about the results of this benchmark:

1. As illustrated in Figures-4.7 and 4.8, in all the scenarios and for all the implementations that were considered, the depth of the formula has only had a marginal impact on the total execution time of the tool.
2. As is illustrated by the comparison of Figures-4.9 and 4.10, and of Figures-4.11 and 4.12, the mean execution time of the scenarios dramatically changes depending on whether it is an easy or a hard instance.
3. For *easy instances*, the python implementation of section 4.1 seems to run slower than the C backed implementation. For *hard instances*, both implementations seem to feature similar levels of performance.
4. As shown on Figure-4.9, for the easy instances, the maximal trace length has an higher impact on the Python implementation of the section 4.1 than on the other implementation.

4.2.3 Analysis of the observed results

Depth of the formula

The first observation regarding the marginal impact of the formula depth on the total execution time can be easily explained by the fact that both implementations use a similar memoization mechanism to cache the results of previous invocations of the functions used to generate $[\![\cdot]\!]^i_k$ and ${}_l[\![\cdot]\!]^i_k$. As explained at the end of the theoretical part of this master's thesis, the use of such a caching mechanism makes the reduction of BMC to propositional SAT solving linear in time and space. Therefore, it seems only logical that the depth of the formula only has a very small impact on the overall run time of the verification tools.

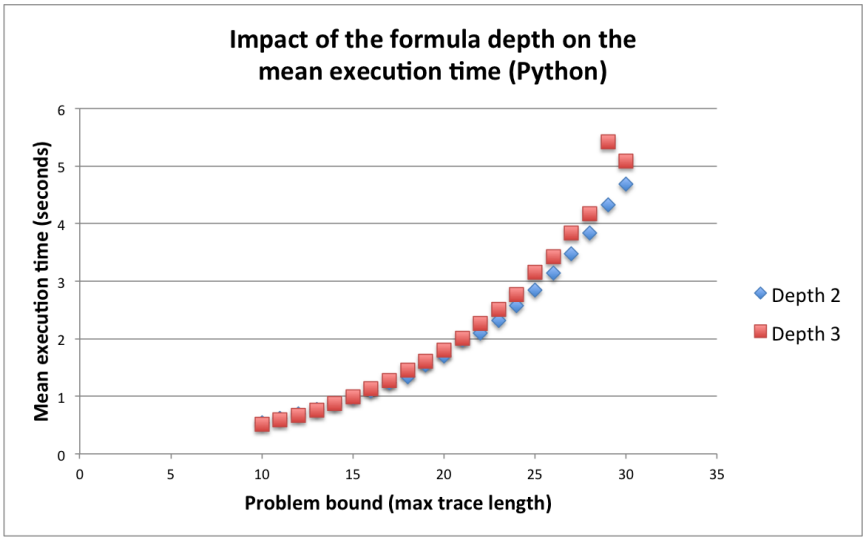

Figure 4.8: Impact of the formula depth on the mean execution time. This graph shows the measurements done while running the Python tool on an easy instance with ten philosophers for formulas of depth two and three with problem bounds ranging from 1 to 30.

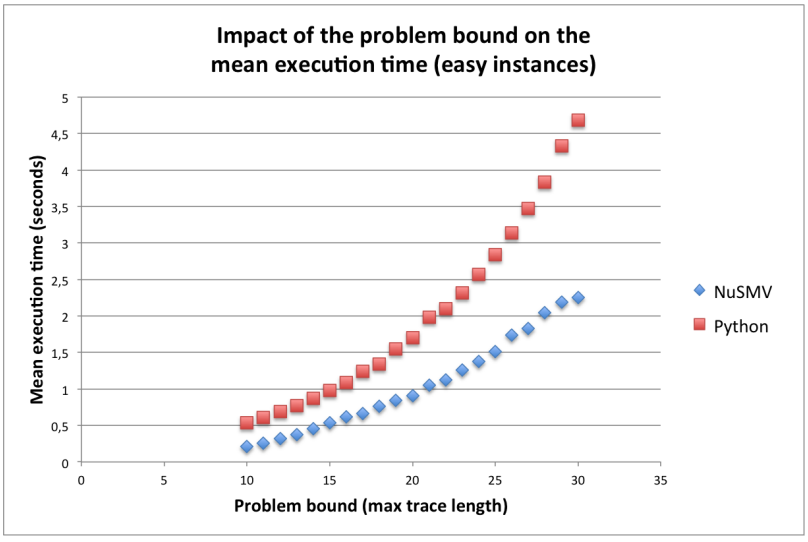

Figure 4.9: Impact of the problem bound on the mean execution time. This graph compares the mean execution times of NuSMV and the Python tool explained in the previous pages on an easy instance with ten philosopher for a problem bound ranging from 1 to 30.

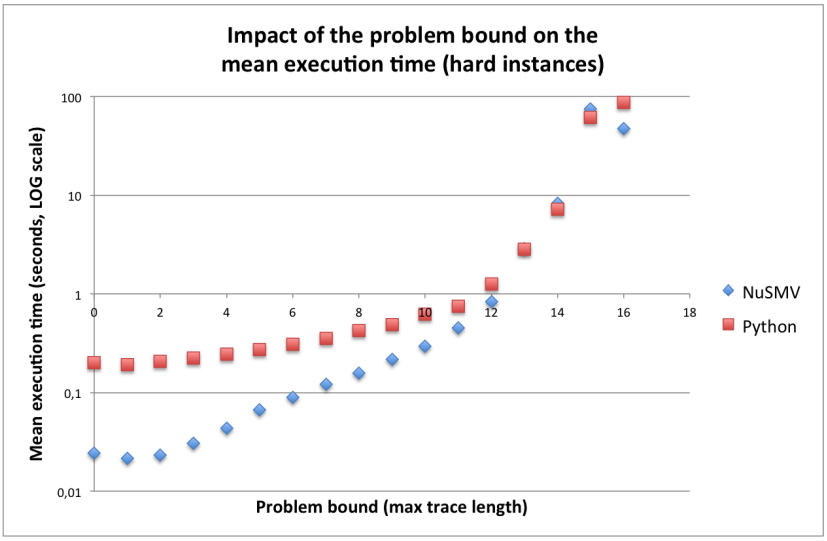

Figure 4.10: Impact of the problem bound on the mean execution time. This graph compares the mean execution times of NuSMV and the Python tool explained in the previous pages on a hard instance with ten philosophers for a problem bound ranging from 1 to 16. This bound was the minimal bound for both tools to find a counterexample violating the property being verified. Note: For better readability, the y-axis is in LOG scale.

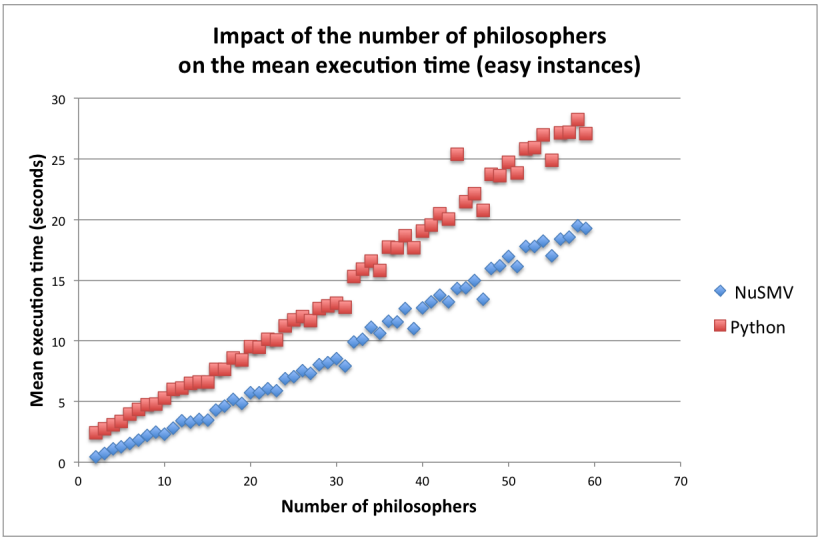

Figure 4.11: Impact of the number of philosophers on the mean execution time. This graph compares the mean execution times of NuSMV and the Python tool explained in the previous pages on easy instances with two to fifty-nine philosophers for a problem bound of 30.

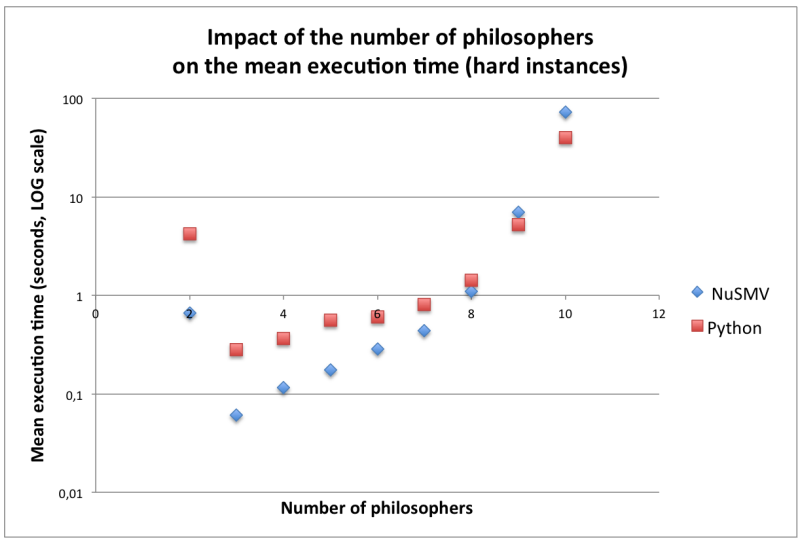

Figure 4.12: Impact of the number of philosophers on the mean execution time. This graph compares the mean execution times of NuSMV and the Python tool explained in the previous pages on hard instances with two to ten philosophers with a maximal problem bound of 30. Note: For better readability, the y-axis is in LOG scale.

Difference between the run times of easy and hard instances

To explain the large difference in the observed execution times of the verification tools when handling easy instance versus hard instances, the hypothesis was made that the run time for the former was essentially driven by the time needed to generate the boolean expressions representing the problem. For the hard instances, the hypothesis suggested that the run time was driven by the execution time of the SAT solver which is intrinsically exponential.

This hypothesis has been empirically verified through the use of a profiler for the implementation written in Python. Concretely, the Python `cProfile` module has been used to gather the data and `snakeviz` to analyze them. This analysis revealed that a rough 97.5% of the time spent during the execution of a hard instance was used in the `solve` method of the SAT solver. The same analysis revealed that for the easy set of instances, this fraction dropped to about $15 - 19\%$ of the total execution time.

Similar performance of both tools for hard instances

The similar performance showcased by the two the tools when executing their verification algorithm on hard instances is understood as a plain consequence of what was explained above. Indeed, because the vast majority of the execution time for these instances is devoted to the SAT-solving of the boolean problem and because the SAT-solver is the exact same in both implementations, it is normal for this fraction of the code to require similar amount of time to complete. Given that in those cases, the rest of the code only accounts for $2 - 3\%$ of the total execution time, it is easy to understand that its efficiency can only have a marginal impact on overall run time of the tool.

Python implementation of section 4.1 runs slower than NuSMV on easy instances

In order to understand the reason why the Python implementation described in section 4.1 runs slower than NuSMV on the *easy instances*, the same profiler as before has been used. This analysis highlighted that the generation of the loop condition ${}_lL_k$ and the SWIG overhead induced by calls to the method `__and__` from the `Be` class were largely responsible for that difference.

Higher impact of the trace length on the Python implementation for easy instances

This observation is explained in the exact same way as the previous one. Indeed, generating SAT problems to identify longer counterexamples necessitates to generate larger SAT-problems. Hence, given that the generation of the SAT problem is precisely the cause of the lower performance of the Python implementation, it is normal to observe such a correlation.

With that in mind, one easily understand that the execution time *trends* illustrated on Figure-4.9 are really the same for both implementations. The only difference being that the *slope* of that trend is higher in the case of the Python implementation. Meanwhile, it is important to remember that this observation is only valid for the *easy* instances and must therefore not be generalized to all execution scenarios.

4.2.4 Partial conclusion

In section 4.1 and subsequent, we have seen how one can use the BMC addition to the PyNuSMV framework to develop prototypes of SAT-BMC based verification tools. Then we have compared the performance of a tool written in such a way to the standard NuSMV implementation written in C language.

Based on the benchmark that was realized and the analysis that was made to understand the behavior of the different implementations, it clearly appeared that this extension of the framework enables the easy development of fairly efficient prototypes. Indeed, for a large fraction of the scenarios that can be encountered, a prototype developed with that framework behaves similarly to an equivalent implementation in C language. The only real performance limitation of this kind of prototype being the handling of *easy* instances where the generation of the SAT problem lasts longer than the satisfiablity test itself.

Meanwhile, the author would like to emphasize on the fact that developing a verification tool prototype using PyNuSMV greatly helps in the creation of a high performance implementation of such a verification tool. Indeed, a third implementation was also realized using the `*_at_offset` series of APIs explained in section 3.3 and compared to the other two using the same test bench. This comparison revealed that this third implementation had a performance similar to NuSMV for all the tests that have been realized. It is thus worth nothing that this last implementation was first written in pure Python using PyNuSMV and then only, the most time consuming functions were translated back to C language. Thanks to this *middle step* done with PyNuSMV, the translation to C language became an almost no-brainer and was realized in about an hour since it mostly consisted in copy-pasting the code and altering the syntax of the program.

4.3 Diagnosability verification tool

As announced in the introduction of this chapter, the second experiment that was devised in the context of the experimental validation of PyNuSMV SAT-BMC features aims at illustrating the potential of this framework on the floor of the verification of partially observable models and non-singly linear logics. To this end, a second tool has been implemented which is able to perform a diagnosability test.

The coming paragraphs start by giving the intuition behind diagnosability. Then, following [39], the intuitive concepts are formalized in order to explain the principle of the BMC refutation performed in this context. After that, the paragraph 4.3.4 shows how these concepts translate in PyNuSMV. Finally, the last section provides an evaluation of the performance of the tool when used against real models originating from researches done by NASA in the years 2000.

4.3.1 Intuition

Informally, given a dynamic partially observable system, the diagnosability problem consists in being able to "verify at design time if some [external observer called] diagnosis system will be able to infer at runtime the required information about the hidden part of the dynamic state [of the observed system]" [39]. This informal definition features two important elements that need to be highlighted: first, the system under consideration is partially observable and hence, for each observed state of the system there may exist more than one *belief state*. Such belief state represents the set of actual system states (including the hidden part) the automaton might possibly be in with respect to the observations that were made.

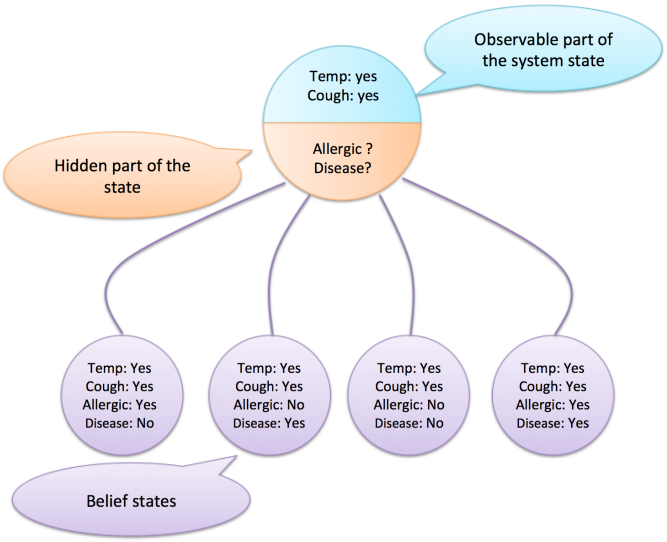

Figure 4.13: Example of partially observable state and associated belief states. The sickness status of a patient.

The second element of importance in the previous definition stems from the fact that the verification process does not try to assess whether intrinsic properties hold for the evolution of the model but instead try to verify the claims that an *external* diagnosis system which can only see the evolution of the *observable* state of the system may make without ever being wrong. In addition to this, it is also important to precise that, as opposed to what is

done in the case of POMDP[7], the verification carried by the kind of diagnosability test which has been implemented does not account for the likelihood of the possible belief states.

A real life example of a disagnosability test would consist in deciding whether a doctor would be able to know what makes a patient sick based only on the symptoms he can observe. For instance, as illustrated in Figure-4.13, the doctor might observe that his patient is coughing and has an high temperature but cannot be absolutely certain of the cure that must be used because the patient might be allergic to some substance or suffer from an other disease. Moreover, this kind of verification allows the diagnosis system to use its knowledge of the past observations of the system in order to rule out some of the belief states that would otherwise have been considered. Still using the doctor metaphor, that would mean that in addition to just observing the patient's symptoms, the doctor would also be allowed to consult the patient's file since his birth before making a decision.

4.3.2 Formalization

Formally, the kind of partially observable system discussed above was defined in [39] as "a structure $P = \langle X, U, Y, \delta, \lambda \rangle$ where X, U, Y are finite sets respectively called the state space, input space and output space, $\delta \subset X \times U \times X$ is the transition relation, and $\lambda \subset X \times Y$ is the observation relation". In that context, $x \xrightarrow{u} x'$ denotes $(x, u, x') \in \delta$ for all $x, x' \in X, u \in U$ and x/y (read y is visible in state x) is used to denote $(x, y) \in \lambda$, for $x \in X$ and $y \in Y$.

Furthermore, because of its partial observability, the dynamics of such a system is described in terms of *observable traces* (a sequence of observable states and stimuli applied to the system) and *feasible traces* which represent the possible traces (full states) that could have produced the observable trace seen from the outside.

More formally, if we denote $\forall i : x_i \in X, u_i \in U, y_i \in Y$: a *feasible trace* is a sequence $\sigma : x_0, u_0, y_0, x_1, u_1, y_1, \cdots, x_k, u_k, y_k$ such that $x_{i-1} \xrightarrow{u_i} x_i$ for all $1 \leq i \leq k$ and x_i/y_i for all $0 \leq i \leq k$. The observable trace w of σ is the sequence $y_0, u_1, y_1, \cdots, u_k, y_k$ and the notation $\sigma : x_0 \xrightarrow{w} x_k$ is used to express that when the system is observed from the outside, σ is seen as w. By convention, we call Σ_P the set of feasible traces of P.

Meanwhile, because it would be highly unlikely for the diagnosis system to be able to infer the *complete state* of the system at all times and under all circumstances, it is necessary to define a *diagnosis condition* expressing the kind of assessment to be made by the diagnosis system. In our formal framework, a diagnosis condition is a pair of state conditions c_1, c_2 describing non empty subsets of X and is written $c_1 \perp c_2$ ($\perp$ is only used as a separator in this context). Informally, $c_1 \perp c_2$ expresses the ability for the diagnosis system to decide which of the state condition c_1, c_2 holds for the current state of the observed system. For example: assuming an electric device, the diagnosability condition $on \perp off$ is true iff an external observer can decide whether that device is turned on or off.

In addition to that, it is also useful to define the *context* in which the diagnosability test should be realized. Such a context takes the form of a pair $C = \langle \theta, \Sigma_{12} \rangle$ where θ defines the state condition to be enforced on the first states of all the considered feasible traces while $\Sigma_{12} \subset \Sigma_P \times \Sigma_P$ characterizes the pairs of relevant executions. The context C is said to be satisfied (written $\langle (x_{01}, x_{02}), w \rangle \models C$) by a pair (x_{01}, x_{02}) of initial states and an observable trace w if and only if $x_{01} \models \theta$, $x_{02} \models \theta$ and $\exists\ \sigma_1,\ \sigma_2$ st $\sigma_1 : x_{01} \xrightarrow{w} x_1$ and $\sigma_2 : x_{02} \xrightarrow{w} x_2$.

[7] Partially observable Markov decision processes.

4.3.3 Refutation

In [39] Pecheur et al. explain that a diagnosis condition $c_1 \perp c_2$ is diagnosable for a partially observable system P over a given context C if an only if no critical pair can be found. A critical pair is defined for P and an observable trace w as a pair of feasible executions $\sigma_1 : x_{01} \xrightarrow{w} x_1$ and $\sigma_2 : x_{02} \xrightarrow{w} x_2$ such that $x_1 \models c_1$ and $x_1 \models c_2$.

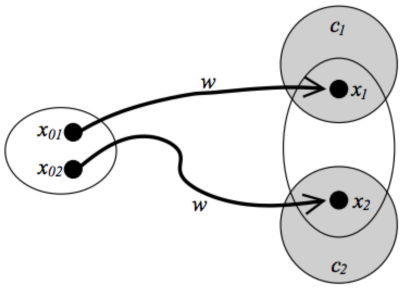

Figure 4.14: Critical pair (source [39])

Therefore, in order to prove that $c_1 \perp c_2$ is *not* diagnosable in P for the context C, it suffices to find two feasible traces σ_1, σ_2 having the same observable trace w such that the first states of these two traces satisfy the condition θ. Moreover, assuming that Σ_{12} can be expressed in as a pair (Σ_1, Σ_2) of LTL formulas, the following two conditions must hold as well: $\sigma_1 \models \Sigma_1$ and $\sigma_2 \models \Sigma_2$. Finally, in order to make (σ_1, σ_2) a critical pair we must also have: $\Diamond\ (\sigma_1 \models c_1 \wedge \sigma_2 \models c_2)$.

4.3.4 Implementation

In order to stay consistent with the definition of the system states which was given in the first part of this work, and in order to be able to leverage the rich NuSMV modeling language, the tool that has been developed does not explicitly maintain the three sets of states X, U, Y nor the two relations λ, δ. Instead, the partial observability of the system is modeled by letting the user partition the state variables[8] in an *observable* set and an *hidden* one using the many foreseen command line options.

For instance, by using the `--observable-regex (-or)` the user can specify one or several regular expressions to match the name of the variables that must be considered visible. Or, because the author believes that it will often be the case that all `IVAR` will need to be observable, the tool foresees the `--observable-inputs (-oi)` to fulfill that exact use case.

Similarly, the diagnosis context and condition are also configured by the end user through command line arguments. For instance, `--sigma1` and `--sigma2` may be used to specify the set of relevant pairs of traces (Σ_{12}). However, because the author envisions that it might be desirable to perform diagnosability tests for several diagnosis conditions using one same diagnosis context, the tool foresees the possibility for the user to specify the diagnosis condition in an interactive fashion.

Although the problem of specifying the different parts (partially observable model, context, and diagnosis condition) has been evacuated through the use of command line options, several other problems had to be tackled before being able to perform actual diagnosability tests. The first and most important among these was the need to generate and encode two parallel executions in terms of boolean expressions. This was solved as shown on

[8] Regardless of their NuSMV variable 'type': VAR, IVAR, FROZENVAR.

Figure-4.15 by using one single boolean encoder (BeEnc) and encoding the two traces one after the other. The only downside of this choice is the impossibility to use incremental SAT solving to perform the verification since adding one time step to the first trace imposes to shift by one all the time blocks of the second trace.

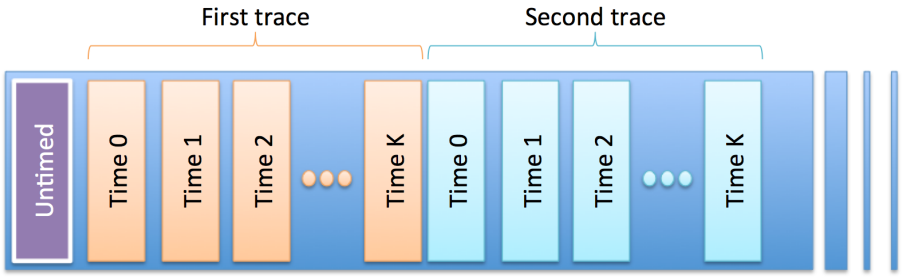

Figure 4.15: Encoding of two parallel traces with one single encoder

Concretely, as shown in Figure-4.16, the generation of these two parallel traces was simply achieved by using the `init` and `unrolling` methods of the `BmcModel` class.

```
1 def generate_path(offset, length):
2     """
3     Returns a boolean expression representing a path of length 'length' in the
4     fsm described by the loaded model.
5     """
6     model = BmcModel()
7     path = model.init[offset] & model.unrolling(offset, offset + length)
8     return path
```

Figure 4.16: Parallel traces generation.

However, in order to constitute a valid proof of non-diagnosability, the two traces must form a critical pair and may consequently not be totally independent from one another. Indeed, they need to be constrained to have identical observable traces and to end up in states satisfying c_1 (resp. c_2). To this end, two constraints have been generated as shown in Figure-4.17). The constraint to force the observable traces to be identical uses a similar approach to the one which has been used to generate the loop condition explained earlier whereas the constraint to force incompatible final state uses a simplification of the algorithm used to generate $[\![\Diamond\ \phi]\!]_k$. Indeed, because c_1 and c_2 are *state conditions*, the shape of the trace (straight or (k,l)-loop) is irrelevant to the evaluation of their truth value.

Furthermore, some additional constraints are required to restrict the diagnosability analysis to the context $C = \langle \theta, \Sigma_{12} \rangle$ specified by the user and consider only the set of critical pairs satisfying that context. In both the case of the constraints enforcing θ on the initial states (Figure-4.18) and the case of the constraints to enforce Σ_{12} on the parallel traces (Figure-4.19 lines 14-15) very little effort was required as the desired API's were readily available in the `BeEnc` class and `ltlspec` module[9].

Finally, the last problem that needed to be addressed in order to turn this tool into something which can actually be used by someone to preform real diagnosability analysis was the display of counterexamples when a critical pair is identified. Unfortunately, using `pynusmv.bmc.utils.generate_counter_example(..)` was not an option because the returned trace does not contain a complete translation of the SAT model (the valuation

[9]Note however that although bounded_semantics_at_offset was foreseen during the development phase of the framework, it does not map to any *native* API from NuSMV. Therefore it has been implemented in a way similar to what was explained in section 4.1.

```
def constraint_same_observations(observable_vars, offset_path1, offset_path2, length):
    """
    Generates a boolean expression stating that the observable state of both
    paths should be the same (all input vars are equivalent).
    """
    fsm = master_be_fsm()
    constraint = Be.true(fsm.encoding.manager)
    for time_ in range(length+1):
        for v in observable_vars:
            ep1 = v.at_time[time_ + offset_path1].boolean_expression
            ep2 = v.at_time[time_ + offset_path2].boolean_expression
            constraint &= ep1.iff(ep2)
    return constraint

def constraint_eventually_critical_pair (formula_nodes, offset_path1, offset_path2, length):
    """
    Generates a boolean expression representing the critical pair condition.
    That is to say, it generates a condition that verifies if it is possible that
    the two belief states are inconsistent wrt 'formula'.
    """
    enc = master_be_fsm().encoding
    c1 = make_nnf_boolean_wff(formula_nodes[0]).to_be(enc)
    c2 = make_nnf_boolean_wff(formula_nodes[1]).to_be(enc)

    constraint = Be.false(enc.manager)
    for time_ in range(length+1):
        constraint |= ( enc.shift_to_time(c1, time_ + offset_path1)
                      & enc.shift_to_time(c2 , time_ + offset_path2) )
    return constraint
```

Figure 4.17: Critical pair enforcement constraints generation

```
def constraint_context_theta_initial ( initial_condition , offset_path1, offset_path2):
    """
    Generates the theta part of the context constraint which imposes a certain
    condition to be satisfied in the initial belief state.
    """
    enc = master_be_fsm().encoding
    # enforce cond on trace 1
    w01 = enc.shift_to_time( initial_condition .to_be(enc), offset_path1)
    # enforce cond on trace 2
    w02 = enc.shift_to_time( initial_condition .to_be(enc), offset_path2)
    return w01 & w02
```

Figure 4.18: θ enforcement constraints generation

that makes the SAT problem true) but instead strips off much of the useful information. However, as illustrated in Figure-4.20 this issue was easily solved thanks to the `decode_sat_model(..)` method from the boolean encoder. Indeed, this method returns a nested hashed structure organized as follows: $time\_block \rightarrow symbol\_name \rightarrow decoded\_value$ which is then easily processed to produce a counterexample.

```
def generate_sat_problem(observable_vars, formula_nodes, length, theta, sigma1, sigma2):
    """
    Generates a SAT problem which is satisfiable iff the given 'formula' is
    *NOT* diagnosable for the loaded model for traces of length 'length'.
    """
    fsm = master_be_fsm()
    offset_1 = 0
    offset_2 = length +1

    problem = generate_path(offset_1, length) & generate_path(offset_2, length) \
            & constraint_same_observations(
                                    observable_vars, offset_1, offset_2, length)\
            & constraint_context_theta_initial (theta, offset_1, offset_2)      \
            & bounded_semantics_at_offset(fsm, sigma1, length, offset_1)        \
            & bounded_semantics_at_offset(fsm, sigma2, length, offset_2)        \
            & constraint_eventually_critical_pair (
                                    formula_nodes, offset_1, offset_2, length)

    return problem
```

Figure 4.19: Generation of the complete SAT problem to be sumbmitted to a solver to perform a diagnosability test

```
def diagnosability_violation (observable_names, solver, k):
    """
    Interprets the result (model of the sat solver) and prints the parallel
    traces (having the same observations) that lead to some critical pair.
    """
    lexicographically = lambda x: str(x)
    be_enc = master_be_fsm().encoding
    decoded= be_enc.decode_sat_model(solver.model)
    # create the trace that will actually get returned
    counter_ex = "### DIAGNOSABILITY VIOLATION ###\n"

    for time in range(k):
        counter_ex+= "*** TIME {:03} ***\n".format(time)
        counter_ex+= "--- OBSERVABLE STATE ---\n"
        # add all observable values.
        for symbol in sorted(decoded[time].keys(), key= lexicographically ):
            if str(symbol) in observable_names:
                counter_ex+="{} = {}\n".format(symbol, decoded[time][symbol])
        # add all non-observable values of the belief STATE A____
        counter_ex+= "--- BELIEF STATE A ---\n"
        for symbol in sorted(decoded[time].keys(), key= lexicographically ):
            if str(symbol) not in observable_names:
                counter_ex+="{} = {}\n".format(symbol, decoded[time][symbol])
        # add all non-observable values of the belief STATE B____
        counter_ex+= "--- BELIEF STATE B ---\n"
        for symbol in sorted(decoded[k+1+time].keys(), key= lexicographically ):
            if str(symbol) not in observable_names:
                counter_ex+="{} = {}\n".format(symbol, decoded[1+k+time][symbol])

    return counter_ex
```

Figure 4.20: Processing of the decoded SAT model to display an understandable counterexample

4.3.5 Performance evaluation

Similarly to what was done in section 4.2, the performance of the diagnosability test tool described in the previous section was evaluated using a benchmarking approach. To this end, a test bench was created using examples originating from research done at NASA in the years 2000. These examples are described in [20, 40, 43] and respectively describe the behavior of a Martian propellant production plant imagined to prepare a potential human exploration of the red planet, a valve with multiple flow levels and failure modes, and a cascade of circuit breakers and LEDs.

However, because diagnosability is none of the use cases tackled 'natively' by NuSMV, the comparison could not be as immediate as was the case for LTL. Therefore, following the approach proposed in [39], two versions of the models were realized:

- A simple one, containing only the description of the system subject to the diagnosability test. This is the version of the SMV model meant to be used by the diagnosability tool.
- A *coupled twin* one containing two copies of the same model constrained to have the same observable traces. This is is the version which was submitted to NuSMV in order to let it perform the diagnosability test.

For each example, the diagnosability of a fault condition was tested and the execution times of both tools were measured using the system time of the test machine. Moreover, in order to evaluate the impact of the diagnosability context on the overall execution time of the system, a variation of the valve model was tested with a more realistic context (a Σ_{12} constraint was used to restrict the set of allowed transitions). Finally, in order to smooth out the potential impact of events unrelated to the tests, each test combination was repeated five times and the measured execution times were averaged.

Observations and analysis

As can be seen from the table in Figure-4.21, verifying the diagnosability of a fault condition was extremely fast for all the NASA test cases. Hence, although the PyNuSMV based implementation seems to performs slightly slower than NuSMV for these cases, no firm conclusion can be made based on these results. Indeed, for all the cases that have been tested, the difference between the mean execution time of the two tools is no larger than a fraction of a second and can therefore not be considered significant.

	RWGS	VALVE	CBR	VALVE2
NuSMV	0,4884	0,0104	0,04	0,011
PyNuSMV	1,241	0,1742	0,2068	0,1754

Figure 4.21: Mean execution time to complete a diagnosability test

In order to get over the apparent BMC-easiness of the previous scenarios an additional test case was designed with the explicit goal of being hard to solve for the tools. For that reason, this test case is rather artificial and consists of a series of monotonically increasing counters having a random initial value in the range [0..5]. Moreover, each counter is hidden behind an other component which only exposes the last bit of the mod. 50 value of the counter. With that setup, the diagnosis condition consists in deciding whether the values of the counters sum to some arbitrarily chosen number or equals zero.

As shown in Figure-4.22, for the specific hard scenario explained above, it appears that the tool developed with PyNuSMV performs significantly better than what NuSMV does

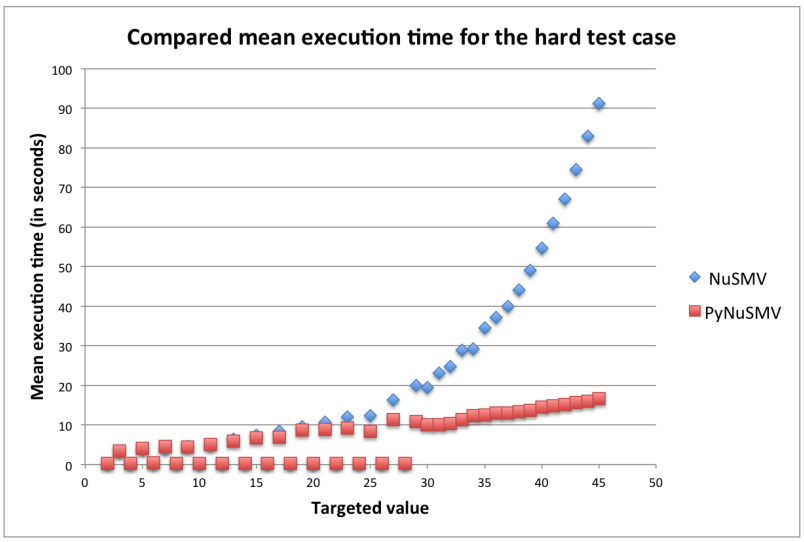

Figure 4.22: Mean execution time to diagnose $sum = x \perp sum = 0$. Note, the marks on the x-axis represent the cases where a trivial counterexample was found.

with an equivalent twin model. Although this cannot be proven with factual evidences, the author hypothesizes that this phenomenon can be explained as such: NuSMV uses a *twin model* and hence two copies of the same FSM (here the collection of counters + modulus) which are combined to create one larger FSM. Conversely, the PyNuSMV based tool works with only one copy. Because of this, the size of the boolean expression used to represent the transition relation is much larger in the case of the twin model than in the case of the simple model. As a consequence of this, the SAT problem submitted by NuSMV to its SAT solver is much larger than the one of the PyNuSMV tool. Moreover, because the submitted problems are *almost* satisfiable, the SAT solvers spend a lot of time before returning their answer. In particular, because the SAT problem issued by NuSMV is larger than the other; the SAT solver of NuSMV has to spend (up to exponentially) more time solving its problem than the SAT solver used by the PyNuSMV tool.

4.3.6 Partial conclusion

The case of the diagnosability verification tool has given an example of the new kind of possibilities that are enabled by the SAT-BMC extension of the PyNuSMV framework. Indeed, it has shown that using this extension, it is possible to implement verification algorithms that work under a partial observability assumption, and to perform verifications that consider multiple paths in parallel very efficiently. Doing so, it paved the way for the development of SAT-BMC verification tools for branching time logics with PyNuSMV. For example, it would be interesting to translate the algorithm proposed by Penczek in [42] to perform SAT-BMC verification of the universal fragment of the computation tree logic (CTL).

Conclusion

Because in the world of today people are confronted every day with systems of ever increasing complexity and because our society is willing to depend on technology to deal with that complexity, it becomes of crucial importance to design software and hardware that can be depended upon. Especially because some of the tasks we delegate to these technologies are dealing with life critical matters such as brain surgery or space exploration. In all these domains failure is not an option and therefore it is essential to be able to avoid *at design time* a large number of flaws that could jeopardize the overall system dependability.

In that context, formal verification methods have been developed to help automate the process of ensuring the correctness of the systems being developed. Throughout this dissertation, we approached this topic with the objective of extending the PyNuSMV framework with SAT-BMC capabilities. To this end, the first chapters laid the necessary formal methods background to clearly understand the principles of SAT-based bounded model checking. Concretely, we started by defining the notion of Kripke Structure which serves as basis for the modeling of real system. Then we explained the syntax and semantics of the Linear Temporal Logic and defined precisely the the model checking problem.

After that, a word was given to explain the traditional approaches to model checking: the automata-theoretic approach and symbolic model checking. Afterwards, the bounded model checking technique was detailed. In particular, the reduction from BMC to propositional SAT solving at the heart of SAT-BMC was explained.

In the second part of this work, the architecture of PyNuSMV was described and the organization of the most important SAT-BMC related modules of the upper interface was depicted. Then, we illustrated the new possibilities offered by the SAT-BMC extension of PyNuSMV through two case studies. The first one consisted in the development of a new LTL verification tool implemented with PyNuSMV *as if it were a brand new logic*. The second one aimed at showing that the BMC addition to PyNuSMV is flexible enough to develop verification tools for new kind of formalisms. To this end, we explored the *diagnosability* problem and, doing so, showed how to implement a verification tool that works on parallel traces of a partially observable model. At the end of each case study, an evaluation of the performance of the implemented tool was realized.

From these evaluations we observed that most of the time, the overhead caused by the use of PyNuSMV to implement tools using SAT-BMC techniques is relatively limited or, to say the least, remains within reasonable bounds. Moreover, we also learned that once a prototype of a tool was realized in Python, it could easily be turned in a high-performance tool given that its migration to C language is a fairly straightforward process. Finally, we learned that the basic building blocks required to build efficient SAT-BMC verification tools for branching time logics such as CTL were made readily available as a part of the framework.

Future work

At the end of this thesis, it appears that the work that has been realized was useful and could be reused in a variety of situations in order to leverage the versatility of the Python language and use the efficiency of NuSMV to help improve the design of many other applications. Besides that, it would be interesting to try to develop an alternative strategy for the encoding of parallel traces as the current approach prevents the use of an incremental SAT solver in those cases. Finally, it would definitely also be interesting to address the SAT-based bounded verification of richer formalisms. For instance, being able to perform an efficient BMC verification for the alternating time logic (ATL) could pave the way for the implementation of a controller synthesis that leverages the fast SAT solvers implementations to scale better than the existing solutions to this problem.

List of acronyms and symbols

Acronyms

Acronym	Description
API	Abstract Programming Interface.
BDD	Binary Decision Diagram, a data structure that compactly encodes large boolean formulas.
BMC	Bounded Model Checking, a formal verification technique that approximates the result of traditional model checking by considering only the k first steps in the execution traces of the analyzed model.
CTL	Computation Tree Logic, a temporal logic with a branching time structure (evaluates on trees of executions rather than computations).
CNF	Conjunctive Normal Form.
FSM	Finite State Machine, a formalism to represent automata.
KS	Kripke Structure, a kind of finite state machine equipped with an interpretation function [38, s. 2].
LTL	Linear Temporal Logic, a temporal logic evaluated on *traces*.
LTS	Labeled Transition System, a kind of finite state machine which hides the internal structure of the states of the represented system and focuses on the transitions that exist between any two such states.
NNF	Negation Normal Form.
POMDP	Partially observable Markov decision process.
SAT	The propositional *satisfiability problem*. The problem of determining whether there exists a coherent valuation of the variables composing a propositional (boolean) formula that makes it evaluate true.
SAT-BMC	A *bounded model checking* technique that relies on boolean propositional satisfiability in order to find counterexamples that falsify the property being verified.
WFF	Well Formed Formula.

Mathematical symbols and notations

Symbol	Description
AP	Stands for the set of atomic propositions that are used to interpret a Kripke Structure as a model of another system.
K	Is used to denote a *Kripke Structure* encoded by the tuple $\langle AP, S, S_0, T, V \rangle$.
S	Denotes the set of *states* of a Kripke Structure.
S_0	Denotes the set of *initial states* of a Kripke Structure.
T	Stands for the *transition relation* of some Kripke Structure.
U	The input space of a partially observable system.
V	Stands for the *interpretation function* which assigns a truth value to all the propositions in AP based on the KS state.
X	The state space of a partially observable system.
Y	The output space of a partially observable system.
k	Is used to denote the maximal length of a trace in a bounded verification.
l	Is used to denote the time index at which a loop 'starts' in a (k,l)-loop.
w	Is used to denote an observable trace of a partially observable system.
$\mathcal{V}$	The set of variables composing the system state
$\mathcal{D}$	The domain of the variables composing the system state
δ	The transition relation of a partially observable system.
λ	The observation relation of a partially observable system.
Σ_K	The set of traces of a Kripke structure K.
$\Sigma_{K_{prefix}}$	The set of all the prefixes of all the traces of a Kripke structure K.
$\Sigma_{K_{suffix}}$	The set of all the suffixes of all the traces of a Kripke structure K.
Σ_P	The set of all the traces of partially observable system P.
π	A trace of the Kripke structure.
π^i	The suffix of the trace π which starts at state π_i.
s_i	The ith state of a trace.
ϕ	Is used as a place holder to denote any logical formula.
ψ	Is used as a place holder to denote any logical formula when there is a need to distinguish from ϕ.
$\top$	The boolean constant TRUE.
$\bot$	The boolean constant FALSE.

$c_1 \perp c_2$	A diagnosis condition stating that the observer should be able to decide which of the condition c_1 or c_2 apply on some state.
θ	The condition enforced by the context on the initial state of the considered traces in a diagnosability test.
Σ_{12}	The condition enforced by the context on the traces considered during a diagnosability test.
$\models \cdot$	The *entails* relation which is true iff the right hand formula is a consequence of the left term. For instance, $\pi \models \phi$ is says that the formula ϕ is a holds for the trace π.
$\models_k \cdot$	The bounded *entails* relation which is true iff the right hand formula is a consequence of the left term when considered in a verification bounded with k steps.
$\models_k^i \cdot$	The timed and bounded *entails* relation which is true iff the right hand formula is a consequence of the left term when considered at time i of a bounded verification.
$\bigcirc \cdot$	The *next time* modality of LTL.
$\Box \cdot$	The *globally* modality of LTL.
$\Diamond \cdot$	The *eventually* aka *finally* modality of LTL.
$\cdot \mathbf{U} \cdot$	The *until* modality of LTL.
$\cdot \mathbf{W} \cdot$	The *unless* aka *weak until* modality of LTL.
$[\![\cdot]\!]_k$	The boolean formula translating the *bounded semantics* of the enclosed formula.
$[\![\cdot]\!]_k^i$	The boolean formula translating the *bounded semantics* at time i of the enclosed formula (straight path semantics).
${}_l[\![\cdot]\!]_k^i$	The boolean formula translating the *bounded semantics* at time i of the enclosed formula under the assumption of a loop starting at time l on the path ((k,l)-loop semantics).

List of Figures

Bibliography

[1] Pyparsing wiki. `https://pyparsing.wikispaces.com`, 2016. Accessed: May. 11, 2016.

[2] International Atomic Energy Agency. *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama*. International Atomic Energy Agency, 2001.

[3] Krzystof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition, 2010.

[4] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[5] Ned Batchelder et al. Coverage.py, release 4.1b3. `https://media.readthedocs.org/pdf/coverage/latest/coverage.pdf`, 2016. Accessed : May. 10, 2016.

[6] David M Beazley et al. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Tcl/Tk Workshop*, 1996.

[7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[8] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[9] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[10] Simon Busard and Charles Pecheur. Pynusmv: Nusmv as a python library. In *NASA Formal Methods*, pages 453–458. Springer, 2013.

[11] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. Nusmv 2.5 user manual, 2010. *URL http://nusmv. fbk. eu/NuSMV/userman/v25/nusmv. pdf. A ccessed on June*, 24, 2013.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[13] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

[14] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[15] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419. Springer, 1990.

[16] Edsger W Dijkstra. On the reliability of programs. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF`. Accessed : Jan. 19, 2016.

[17] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. Notes on structured programming. `http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf`, 1970. Accessed : Jan. 19, 2016.

[18] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[19] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.

[20] Peter Engrand. Model checking autonomy models for a martian propellant production plant.

[21] Python Sofware Foundation. argparse — parser for command-line options, arguments and sub-commands. `https://docs.python.org/3/library/argparse.html`, 2016. Accessed: May. 10, 2016.

[22] Zhaohui Fu, Yogesh Marhajan, and Sharad Malik. Zchaff sat solver, 2004.

[23] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[24] Daniel Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4):78–88, 2009.

[25] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, revised edition, 2012.

[26] Fundazione Bruno Kessler. Nusmv documentation. `http://nusmv.fbk.eu/NuSMV/progman/v25/html/NuSMV_Pkg_index.html`. Accessed : Dec. 19, 2015.

[27] Fundazione Bruno Kessler. Nusmv syntax. `http://nusmv.fbk.eu/NuSMV/userman/v21/nusmv_3.html`. Accessed : March 7, 2016.

[28] Fundazione Bruno Kessler. Nusmv user manual 2.5. `http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf`. Accessed : March 7, 2016.

[29] Rafe Kettler. A guide to python's magic methods. `https://github.com/RafeKettler/magicmethods/blob/master/magicmethods.pdf`, 2015. Acessed : Feb. 3, 2016.

[30] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[31] Jeff Magee and Jeff Kramer. *State models and java programs*. Wiley, 2nd edition, 2006.

[32] Paul McGuire. *Getting started with pyparsing*. " O'Reilly Media, Inc.", 2007.

[33] Kenneth L McMillan. *Symbolic model checking*. PhD thesis, Carnegie Mellon University, 1993.

[34] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[35] NASA. Java path finder, the swiss army knife of java verification. `http://babelfish.arc.nasa.gov/trac/jpf`. Accessed: Jan. 19, 2016.

[36] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[37] Charles Pecheur. Lingi2143 - lecture 8b: Model checking, November 2015.

[38] Charles Pecheur. Lsinf2224 - lecture 10: State based models, April 2015.

[39] Charles Pecheur, Alessandro Cimatti, and Roberto Cavada. Formal verification of diagnosability via symbolic model checking. In *Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France*. Citeseer, 2002.

[40] Charles Pecheur and Reid Simmons. From livingstone to smv. In *Formal Approaches to Agent-Based Systems*, pages 103–113. Springer, 2000.

[41] Wojciech Penczek and Alessio Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 209–216. ACM, 2003.

[42] Wojciech Penczek, Bożena Woźna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of ctl. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.

[43] Franco Raimondi, Charles Pecheur, Alessio Lomuscio, et al. Applications of model checking for multi-agent systems: verification of diagnosability and recoverability. *Proceedings of Concurrency, Specification & Programming (CS&P), Warsaw University*, pages 433–444, 2005.

[44] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition, 2014.

[45] swig.org. Swig documentation. `http://www.swig.org/doc.html`. Accessed : Dec. 19, 2015.

[46] swig.org. Swig tutorial. `http://www.swig.org/tutorial.html`. Accessed : Dec. 19, 2015.

[47] Moshe Y Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

[48] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, (110–111), 1985.

Appendices

Appendix A

The NuSMV modeling language

There are essentially five kinds of statements[1] in an SMV program: `MODULE`, `VAR`, `ASSIGN`, `DEFINE` and `LTLSPEC`. The first four let us define the Kripke structure of the model against which the verification will be done. The last one let us express the formal *specification* of the properties to be verified. Given that the explanation of this last statement requires some background knowledge about temporal logics, it is deferred until the end of the second part of this chapter.

A.1 MODULE

At the highest conceptual level of the model is the definition of a *MODULE*. In object oriented parlance, the definition of such a module would correspond to a *class* as it is meant to describe a coherent subsystem of the model and introduce a new namespace. These subsystems can be instantiated one or several time and the global system made of all the subsystems can see them either as synchronous or asynchronous (more about this in the VAR section).

In order to be more generic and easier to reuse, the modules definitions can accept parameters which act as variables despite the fact that their declaration is not located in the module.

As in most programming languages, naming a module *main* indicates to the SMV system that this module ought to be considered as the root of the system. In other words, the `main` module is the large Kripke structure obtained from the combination of all the modules that have been instantiated.

A.2 VAR

The VAR section of an SMV module is the part of that module that express the internal state of the (sub-)system. Using the same object oriented metaphor as in the previous paragraph, the variables declared in a VAR section corresponds to the field definition of a class. Note however, that contrary to most programming languages the set of types available for the variable declaration is rather limited. Indeed, since a state of the Kripke

[1] Actually, the input language of NuSMV's is much richer than that and provides many other constructs such as IVAR, FAIRNESS, etc... which prove very useful while modeling a real problem but are not detailed here since they won't improve the reader's understanding of the examples in the rest of this master's thesis. More information can nonetheless be found in [27, 28]

structure is a *valuation* (assignment) of the variables, the finite state nature of the structure imposes the type of all variables to be finite. Therefore, the only types available for assignments are :

- **boolean** represented by 0 and 1.
- **enumerated values** presented in the form $\{value_1, value_2, ..., value_n\}$ or with the shorthand notation $x..y$ for integer values.
- **user modules** which are instantiations of other modules that have been declared. These instantiations come in two flavors: synchronous when the instantiation is not preceded by the keyword *process* and asynchronous when it is. The difference between these two modes is the following:
  - in the *synchronous* mode, both the referencing module and the referenced one progress at the same time. That is to say, whenever a transition is engaged in the referencing module, a transition is engaged in the referenced one.
  - in the *asynchronous* mode, the two subsystems may evolve at any arbitrary speed. Hence, a transition in the composite model can correspond to either a transition in the referencing module or one in the referenced module. The result of this way to consider asynchronism is that concurrent execution of processes is modeled as an interleaving of the executions steps from each process. This has been shown not to harm the generality of the resulting model since the exact ordering of events cannot be measured exactly in a concurrent system[31]. An other way to informally convince oneself of the correctness of such an approach is to consider that concurrency is simulated on a single machine where a scheduler makes decisions regarding the execution of each automaton.

A.3 ASSIGN

The assign section of an SMV module is the place where the initial state and transition relation of the system is being defined. To that end, the notations $init(\cdot)$ and $next(\cdot)$ have been defined. As the name suggests, $init(some\_variable)$ is a way to denote the variable *some_variable* in the initial state whereas $next(some\_variable)$ permits to specify the evolution of the value of *some_variable* over time. This is generally done using a *case* statement which is composed of guarded clauses of the form $< condition >$: *possiblevalues*;. In that case, whenever the left hand side condition of a guarded clause evaluates to *true* then the right hand side value or set of values becomes feasible. There are two peculiarities to such a construct:

1. All the assignments defined in the ASSIGN section are supposed to happen *atomically*, as if all variables changed their values *simultaneously*.
2. In the event where multiple guarded clauses are enabled at the same time, then the value is chosen *non-deterministically*. This is particularly important in SMV since it allows the definition of higher level model (for instance, models of systems in which some parts are implementation dependent).

The Figure-A.1 gives an illustrative example of the way ASSIGN works: the model declares two boolean variables a and b which act as pair of inverted flip flops. The initial state of the transition system is described at the sixth and eleventh lines of the listing where variables a (resp. b) are assigned the initial values $TRUE$ and $FALSE$. Then, the two $next(\cdot)$ clauses that are always triggered together insure that at any given time when the model is observed, $a \Leftrightarrow \neg b$.

```
MODULE main
VAR
  a : boolean;
  b : boolean;
ASSIGN
   init (a) := TRUE;
   next(a) := case
                  a : FALSE;
                 !a : TRUE;
               esac;
    init (b) := FALSE;
    next(b) := case
                  b : FALSE;
                 !b : TRUE;
               esac;
```

Figure A.1: A very simple model which maintains the property $a \Leftrightarrow \neg b$ at any time during the execution of the system.

In order to be complete, let's also mention for completeness that as an alternative to the ASSIGN way to proceed, SMV has foreseen the possibility to define the transition relation explicitly using `INIT` and `TRANS` sections. However, this approach is discouraged because it can lead to the writing of models that cannot be implemented in a real-life system or models that are vacuously true because of logical contradictions. This kind of mistakes in the transition relation definition would render all subsequent analysis useless since the properties could be verified without giving any guarantees about the actual system whose behavior is intended.

A.4 DEFINE

The role of the *DEFINE* section is to associate an expression to a symbol which can then be used to verify properties as if it were a variable. The use of such defined symbols is similar to *methods* in other programming languages as it allows the definition of higher level abstraction whose value depend on the other variables of the state of the system.

A.5 LTL properties in NuSMV

Concretely, NuSMV implements the BMC verification of properties expressed in LTL and requires them to be introduced by the LTLSPEC keyword to that end[2]. Because the ascii charset is used to represent these formulas, the usual boolean operators use a C-style syntax and temporal modalities are denoted using their letter denomination rather than their dedicated mathematical glyph. Besides that, it is also worth noting that the LTL language usable in this scope has been extended with *past* modalities and offers the operator **R** which is an exact dual of **U** rather than the **W** operator detailed above.

[2]NuSMV also implements verification algorithms for some other logics (i.e. CTL) but is unable to carry these verification using *bounded model checking* which is the main focus of this master's thesis.

```
MODULE main
VAR
  breath : {in_out, held, stopped};
DEFINE
  alive := (breath = in_out) | (breath = held);
ASSIGN
  init(breath) := in_out;
  next(breath) := case
                    breath = in_out : {held,    stopped};
                    breath = held   : {in_out, stopped};
                    breath = stopped : stopped;
                  esac;

-- Example of LTL properties that can be verified on the above model.
LTLSPEC
  -- At any time, if it is breathing then it is alive
  G breath = in_out -> alive
LTLSPEC
  -- Eventually, the animal dies. (Note: this property does not hold).
  F ! alive
```

Figure A.2: A summarizing example which represents the same Kripke structure as the one detailed in the example of subsection 1.1.1 but encoded in the SMV language.

```
MODULE main
VAR
  forks : array 0..1 of {1, 2, nobody};
  p1    : process philosopher(1, forks[0], forks[1]);
  p2    : process philosopher(2, forks[0], forks[1]);

ASSIGN
  init(forks[0]) := nobody;
  init(forks[1]) := nobody;

MODULE philosopher(id, left, right)
-- This module models the behavior of one of the philosopher participating to the dinner.
VAR
  status : {thinking, hungry, eating, done};
DEFINE
  gotleft  := (left=id);
  gotright := (right=id);
  waiting  := ((status = hungry) & gotleft & !gotright);

ASSIGN
  init(status) := thinking;

  next(status) := case
    status = thinking : {thinking, hungry};
    status = hungry & gotleft & gotright : eating;
    status = eating : {eating, done};
    status = done & !gotleft & !gotright : thinking;
    TRUE : status;
    esac;

  next(left) := case
    status = hungry & left = nobody : id ;
    status = done & gotleft         : nobody;
    TRUE                            : left;
    esac;

  next(right) := case
    status = hungry & gotleft & right = nobody: id;
    status = done   & gotright      : nobody;
    TRUE                            : right;
    esac;

FAIRNESS
  -- This fairness constraint forces NuSMV to consider traces where the
  -- philosophers processes actually do run.
  -- running is a special variable automatically declared for all processes
  running;

LTLSPEC -- no deadlock expressed in LTL
  G (waiting -> F !waiting)
```

Figure A.3: A summarizing example of SMV model representing the well known *Dining philosophers* problem showcasing examples of properties expressed in LTL

Appendix B

The source code

The complete source code of the project, together with the associated tests and tools that have been developed to experimentally validate the usability and performance of our addition to PyNuSMV can be consulted on the GitHub repository of the project: `https://github.com/xgillard/pynusmv`.

Note however that because another (private) repository was used most of the time, the GitHub commit log does not properly reflect the progression of the development.

Appendix C

Test coverage

The Figure-C.2 reproduces the code coverage report produced by `coverage.py` for the modules developed in the scope of this extension to the PyNuSMV framework. This report was produced using the commands logged in Figure-C.1.

Please note that the low coverage rate of the *diagnosability* module stems from the fact that a large fraction of the code in that module is devoted to the handling of command line arguments.

```
coverage run -m unittest
coverage html --include "pynusmv/bmc/*,
pynusmv/be/*,
pynusmv/sexp/*,
pynusmv/collections*,
pynusmv/sat*,
pynusmv/trace*,
pynusmv/utils*,
pynusmv/wff*,
tools/bmcLTL/*,
tools/ diagnosability *"
```

Figure C.1: Commands used to produce the report of Figure-C.2

Coverage report: 89%

Module	*statements*	*missing*	*excluded*	*coverage*
pynusmv/be/__init__.py	1	0	0	100%
pynusmv/be/encoder.py	277	7	0	97%
pynusmv/be/expression.py	105	2	0	98%
pynusmv/be/fsm.py	64	12	0	81%
pynusmv/be/manager.py	62	6	0	90%
pynusmv/bmc/__init__.py	1	0	0	100%
pynusmv/bmc/glob.py	75	6	0	92%
pynusmv/bmc/invarspec.py	57	6	0	89%
pynusmv/bmc/ltlspec.py	98	13	0	87%
pynusmv/bmc/utils.py	190	32	0	83%
pynusmv/collections.py	253	10	0	96%
pynusmv/sat.py	128	9	0	93%
pynusmv/sexp/__init__.py	1	0	0	100%
pynusmv/sexp/fsm.py	56	12	0	79%
pynusmv/trace.py	137	10	0	93%
pynusmv/utils.py	151	21	0	86%
pynusmv/wff.py	106	11	0	90%
tools/bmcLTL/__init__.py	1	0	0	100%
tools/bmcLTL/ast.py	240	12	0	95%
tools/bmcLTL/check.py	24	1	0	96%
tools/bmcLTL/gen.py	16	0	0	100%
tools/bmcLTL/parsing.py	52	0	0	100%
tools/diagnosability.py	178	89	0	50%
Total	**2273**	**259**	**0**	**89%**

coverage.py v4.0.3, created at 2016-05-27 14:18

Figure C.2: Summary of the test coverage of the modules developed in the scope of this master's thesis

Appendix D

Main functions and data structures

Please note that the following list of data structures and functions is far from being exhaustive and comprises only the elements that have been considered to be the most important. For a complete reference, please refer to the documentation on the git repository of the project.

D.1 Package pynusmv.be and submodules

- `Be`
  A boolean expression used for instance to encode formulas of the form $[\![\cdot]\!]_k$.
- `BeCnf`
  A boolean expression encoded in conjunctive normal form. As opposed to regular Be, the BeCnf cannot be combined to form longer expressions. However, this is the only form that can be treated by the SAT solvers.
- `BeEnc`
  The boolean encoder. This is the entity in charge of maintaining the array-like structure of timed variables.
- `BeFsm`
  The boolean encoded version of the model. It is useful to access the properties of the model (for instance the transition relation) in the form of of boolean expressions.
- `BeVar`
  This class serves the purpose of encapsulating an *abstract* model variable and its manipulation. The introduction of this class greatly helped to simplify the BeEnc API.
- `BeManager`
  Low level object in charge of the life cycle and conversion of Be and BeCnf.

D.2 Module pynusmv.sat

- `SatSolverResult`
  An enumeration describing the possible outcomes of a call to `solve` on a SatSolver (resp. `solve_all_groups`, `solve_groups` or `solve_without_groups` for a SatIncSolver).
- `SatSolverFactory`
  A factory that helps instantiating a SatSolver.
- `SatSolver`
  This class provides a common interface for the supported external SAT solvers (MiniSat, ZChaff).
- `SatIncSolver`
  This class provides a common interface for the supported external *incremental* SAT solvers (MiniSat, ZChaff).

D.3 Package pynusmv.bmc and submodules

- `BmcModel`
  Decorator around the BeFsm that provides higher level functions (ie: all INVARs are encoded in one single Be).
- `BmcSupport`
  Python context manager that initializes and reclaims all system structures associated with the BMC extension of PyNuSMV.
- `master_be_fsm`
  Function that returns the global BeFsm singleton instance.
- `check_ltl`
  Function that performs the end-to-end verification of an LTL property and print the result on standard output.
- `check_ltl_incrementally`
  Function that performs the end-to-end verification of an LTL property incrementally feeding the unrolling of the transition relation to the SAT solver. Result is printed to standard output.
- `generate_ltl_problem`
  Function that generates a Be satisfiable iff the property passed as argument is violated (for the current model and a given trace length). This expression is encoded starting at offset 0 of the encoder.
- `bounded_semantics`
  Function that generates the Be $[\![\phi]\!]_k$ for ϕ, k passed as arguments. This expression is encoded starting at offset 0 of the encoder.
- `bounded_semantics_without_loop`
  Function that generates the Be corresponding to the bounded semantics of the formula passed as argument in the case of a straight path. This expression is encoded starting at offset 0 of the encoder.
- `bounded_semantics_single_loop`
  Function that generates the Be corresponding to the bounded semantics of the formula

passed as argument in the case of a looping path. The loop conditions is included in the returned Be. This expression is encoded starting at offset 0 of the encoder.

- `bounded_semantics_all_loops` Function that generates the Be corresponding to the bounded semantics of the formula passed as argument in the case of a looping path. All possible loops are considered and the loop conditions are included in the returned Be. This expression is encoded starting at offset 0 of the encoder.
- `bounded_semantics_without_loop_at_offset`
  Does the same as `bounded_semantics_without_loop` but starts the encoding of the expression at an arbitrarily chosen encoder offset.
- `bounded_semantics_with_loop_at_offset`
  Does the same as `bounded_semantics_single_loop` but starts the encoding of the expression at an arbitrarily chosen encoder offset and does not incorporate the loop condition.
- `loop_condition`
  Generates a Be constraint stating that the current path is a (k,l)-loop.
- `successor`
  Returns the successor of a given time in a (k,l)-loop or on a straight path.
- `make_nnf_boolean_wff`
  Expresses the given formula in terms of boolean variables then converts the resulting formula to negation normal form.
- `make_negated_nnf_boolean_wff`
  Does the same as `make_nnf_boolean_wff` but negates the formula first.
- `generate_counter_example`
  Decodes the SAT model and returns the result in the form of a Trace.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**