UNIVERSITY OF TRENTO - Italy

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Timed NuXmv

Formal verification of synchronous timed transition systems

Supervisors
Roberto Sebastiani
Alessandro Cimatti
Alberto Griggio

Student
Enrico Magnago

Academic year 2017/2018

Acknowledgments

I would like to express my gratitude to the whole Embedded System unit in FBK and its head Alessandro Cimatti. He has given me the possibility to develop this master thesis and acquire a deeper knowledge on the subject of model checking with particular focus on real time systems.
Other than my supervisor, Alberto Griggio, who advised and supported me during each step of this work, I would like to thank also the researchers Marco Roveri and Stefano Tonetta. The first for his contribution in the design of the newly implemented software modules and for lending me his deep knowledge of the code-base. The second one for the encoding of the interval semantic required by MTL and validation of the infinite traces representation, execution and completion.
I thank all my colleagues and friends for their support and different points of view on the many issues faced during this thesis work. Finally a special thanks to my family in the persons of my mother Monica, my father Pierluigi and my brother Valerio.

Contents

Summary

Context

This thesis is placed in the context of formal verification of real-time systems. Most state-of-the-art tools require the real-time system to be modeled as a timed automaton. This is a well known and studied formal representation that restricts the number of handled problem instances to a decidable fragment. The main techniques applied to solve these verification problems are described in chapter 3. This work presents an extension of the nuXmv symbolic model checker developed by the Embedded System unit in Fondazione Bruno Kessler. The tool previously supported the verification of discrete finite and infinite synchronous transition systems and it is now able to handle the verification of complex time properties on real-time systems.

Motivations

Real-time systems can be found in many technological devices and, in particular, in embedded systems. Often they are key components of safety-critical processes and devices. Some application domains are, for example, health-care, transportation and avionics. In recent years their complexity is steadily growing and the need for accurate and reliable systems able to ensure their quality has become even greater. In many safety-critical domains model checking has been adopted. These techniques are capable of fully verifying a system design against some properties. However, most state-of-the-art technologies available to perform this task on real-time systems suffer from limited scalability and expressiveness with respect to properties. In particular not many of them allow to check specifications with complex timing constraints which are fundamental for this kind of systems.

Contribution: formal verification of synchronous timed transition systems with dense time domain

The contribution of this thesis is the development of a new version, called Timed nuXmv, of the nuXmv model checker. The new software components are fully integrated in nuXmv and the new version is completely back-ward compatible. Timed nuXmv supports the same main functionalities supported by nuXmv: model compilation, simulation, counter example generation, trace visualization, trace re-execution and trace completion. The following paragraphs give a high level description of how these features have been designed and implemented in Timed nuXmv.

Input language extension

The nuXmv input language has been extended with a model annotation, an additional type of variables (clock), a type modifier, a new constraint type (urgent) and four new operators are available in Linear Temporal Logic (LTL) specifications. These new constructs allow to describe timed transition systems with continuous time semantic. The model description is parsed and compiled in the internal data structures that have been extended to correctly represent the new elements. A more detailed description of the input language of Timed nuXmv is provided in chapter 5.

Reduction to discrete infinite transition system

This work refers to time-unaware structures as *discrete* or *untimed*, while *untiming* is the process that reduces a time-aware representation into an equivalent untimed one. Chapter 6 describes the untiming procedure for Timed nuXmv models: their are reduced into an equivalent representation of a discrete infinite transition system. Section 7.1 shows the untiming procedure applied to specifications. The obtained untimed model with the associated specifications can be stored to file as a valid discrete SMV model. Therefore it is also possible to handle it using nuXmv without time extension. In Timed nuXmv, to complete the reduction, the trace obtained on the discrete model is transformed into the corresponding execution of the timed transition system that violates the specification. This procedure is described in section 8.5.

Traces

nuXmv traces are able to represent only finite or lazo-shaped executions. They are not expressive enough to represent traces of a timed transition system. In these executions the symbol representing time does not follow these patterns: it is monotonically increasing. For this reason, in chapter 8, a more expressive trace representation, called infinite trace, is introduced. These traces allow to specify a subset of symbols whose value in loops is defined by a recurrence relation. These symbols, in this work, are called diverging. Timed nuXmv can show these infinite traces in all four formats that are available for nuXmv traces. They can also be loaded into the system from an external file.

Timed traces simulation

Timed nuXmv allows to simulate the execution of the input timed transition system. As nuXmv, it allows to choose between two possible modes: automatic and interactive. In the automatic mode an execution of length up to a given constant it built. In the interactive mode at each step the system asks the user to choose between a time elapse or a discrete transition, then the user is required to choose the next state of the execution from a list of possible states. This functionality allows to inspect the possible behaviors of a model. A detailed description is given in section 8.2.

Timed traces execution

It is also possible to check if a given execution is valid with respect to a model description. In the case of infinite traces this implies that the loop has to be validated with respect to the diverging symbols. It is necessary to prove that the described system is allowed to repeat that loop infinitely many times, therefore that the loop of the infinite trace represents a valid infinite execution of the model. This problem is encoded as a unsatisfiability problem of a SMT formula. The formula is such that it is satisfied by a model if and only if there exist an iteration number in which a transition of the loop violates the conditions prescribed by the model. This formulation is shown and explained in section 8.3.

Timed traces completion

Section 8.4 describes how the completion of partial infinite traces is performed in Timed nuXmv. Due to the complexity class of the completion problem, in this version of Timed nuXmv, a more efficient, sound but incomplete procedure has been implemented. The system picks a possible completion and then checks if the completed trace is a valid execution by exploiting the feature described above.

Experimental evaluation

Chapter 9 presents the experimental evaluation of Timed nuXmv. The software is compared to some other state-of-the-art tools on the verification of some invariant and MTL specifications on two different kinds of models. In these benchmarks the newly implemented techniques showed good scalability in terms of time and memory consumption with respect to the model size. In section 10.1 some possible directions to further improve and extend the implemented procedures are highlighted. Particular attention is given to the constraints on the modeling and specification languages and to the kind of infinite behaviors that the system is able to detect and represent. For each direction some preliminary observations are reported.

Part I

Ground work

The first part of this thesis describes the theoretical background and existing tools upon which this work relies. Chapter 1 gives an introduction to the topic of formal verification of real-time systems and provides some motivating examples. In chapter 2 the theoretical background required to understand this work is introduced. First the structures used to represent a model are described, then the syntax and semantic of the formal languages used to express specifications is defined, finally the last section of this chapter describes the main techniques used to perform the verification tasks. Chapter 3 briefly reports the most relevant results on the decidability of some problems on models of real-time systems, then it describes the main verification procedures for such systems used by tools in the state of the art. Chapter 4 describes the relevant features that the symbolic model checker nuXmv supported before this thesis work. Particular attention is given to the input language (4.1) and to the operations on traces (4.3). These chapters do not provide a complete description, but focus only on the aspects that are most relevant to this work. In particular some topics are only cited and some references are provided for further readings.

The second and last part of this work (II) describes the actual contribution of this thesis. The additional features of the modeling and specification languages are described, the reduction and verification procedures are explained, the supported operation on traces are shown and finally the implemented techniques are compared with other state-of-the-art model checkers.

1. Introduction

In engineering solving a problem often involves the design and construction of a system (hardware, software or both) to address specific tasks. An error in such systems might lead to undesirable effects. In some application domains, like space, avionics, transportation and health-care, these effects can cause heavy economic losses and physical injuries. For these reasons the definition of a development process that minimizes the probability of such events is critical. In the past there have been many failures of this kind, two of them are reported here as motivating examples. In 1993 a bug was discovered in the Intel Pentium chip, floating point divisions in a certain range lead incorrect results. $475 million was the cost sustained by Intel to replace the defective chips [1]. Between 1982 and 1985 the Therac25 radiation therapy machine was involved in at least six accidents. These machines contained two software faults that, in some conditions, caused them to deliver a dose of beta radiations approximately 100 times the intended one [28].
A common approach to address this issues is to perform extensive tests on a completely developed product. This technique has two major disadvantages: it requires a fully fledged system to be performed and, assuming the system to be deterministic, it only guarantees that it behaves correctly in that particular cases. In this approach mistakes made in the early stages will be detected only at the end of the development process. From these observations the need for tools that support the early phases of the development becomes evident. Over the years many formalisms to represent different points of view

of a system have been proposed and used in many application scenarios. These more or less formal representations allow the designers to provide a model that is precise enough to highlight possible inconsistencies and ease the review process. The modeling language should provide an abstraction as close as possible to the application scenario in order to reduce the cost of creating such model and decreases the probability of incurring modeling errors.
For these reasons many companies and agencies, like Boeing, Airbus, ESA, NASA, Intel and RFI, integrated in their development process formal verification procedures. Moreover, also design tools like Simulink provide formal verification features.

This thesis work is concerned with the modeling and verification of synchronous real-time systems. Real-time systems are becoming pervasive, often in the form of embedded devices; their application domains range from transportation and automation in industries to economics and health-care. Over the years their complexity has increased and people rely on them to perform critical tasks with absolute precision and reliability.

1.1. Formal verification of real-time systems

In automated formal verification, given a description of the system and a property in some formal language, the machine is able to determine whether the system satisfies the property in all possible executions; if it does not a counter-example is usually provided. There are many software products, called model checkers, that address this issue by exploiting different techniques to solve the verification problem and by providing different languages for describing the system and specify properties. Some examples of such tools are: *ATMOC*, *LTSmin*, *nuXmv* and *Uppaal*.

The correctness of a real-time system depends not only on the logical result of the computation but also on the time required to compute such result. Many safety-critical systems have hard real-time constraints; some examples are: defense and space systems, networked multimedia systems and embedded automative electronics. In this application scenario tools that allow to verify the system design against its real-time requirements are fundamental to achieve a better quality and reliability of the final system. However, as highlighted in chapters 2 and 3, depending on the kind of constraints this class of verification problems quickly becomes undecidable. Most state-of-the-art techniques carefully limit the expressiveness of their modeling languages to restrict the input models only to decidable instances. These restrictions forbid the verification of systems with complex time behaviors that are not expressible in such languages. It is still possible to provide a sound/correct answer in some of these cases, while due to the undecidability issue completeness is impossible to achieve. Increasing the number of behaviors for which the verification procedure is able to provide an answer, broadens the applicability of these techniques to safety-critical domains with more complex dynamics.

This work describes how the nuXmv symbolic model checker has been extended to handle the verification of synchronous real-time systems. The new version of the tool is called Timed nuXmv. Both language and performance of the newly implemented software are compared to other related state-of-the-art model checkers.

2. Background

This chapter provides the definition of the terminology and key concepts that are used throughout this thesis. An extensive presentation of most of the notions introduced in this chapter can be found in the work of Baier and Katoen [5]. The chapter is organized as follows: first the formal structures used to formally represent the systems are introduced. In particular different notations used to represent transition systems are defined. Among these, the notion of *timed transition systems* is central to this thesis. Later both syntax and semantic of the most relevant formal languages used to specify properties on these systems are explained. In the end the three main approaches used to verify such properties on these models are presented.

2.1. Transition systems

The formal description of the model is given as a *transition system* [5]. A transition system is a directed graph in which the nodes represent all possible states of the system and the edges represent all possible transitions from a state to another. A subset of the nodes represents the *initial states*: all possible configuration from which the model can start. Every path on the graph starting from an initial state represents a possible execution of the system and it is uniquely identified by a, possibly infinite, sequence of states. A state s is said to be *reachable* if there exist a path from an initial state to s. States with no outgoing edges are called *dead-lock states*. From an automata-theoretic point of view it is possible to borrow a slightly different terminology: the set of all possible executions of the transition system is also called language of the automata and every execution is a word.

Two transition systems T, T' are said *equivalent* if they accept the same language: $T \equiv T' \iff L(T) = L(T')$, where L is a function that given a transition system it computes its language. The synchronous composition operator given two transition systems T_0, T_1 computes another transition system T such that its language $L(T)$ is the intersection of the languages of T_0 and T_1: $T := T_0 \times T_1 \iff L(T) = L(T_0) \cap L(T_1)$. This operator is defined for each kind of transition system and its usage is shown in the formal verification procedures.

Fair transition system A *fair transition system* is a transition system enriched with a set of one or more *fairness* conditions. These additional constraints define a list of subsets of states. Every valid execution of the system infinitely often reaches at least one state in each one of these sets. The fairness conditions allow to impose progress constraints on the model executions which are not otherwise expressible in a transition system.

There are many different representations of transition systems, e.g. Kripke structures and Büchi automata. They are slightly different notations to represent these objects, each of them can also represent fairness conditions and has a synchronous product operator to compute the language intersection. This work focuses on symbolic transition systems. These objects can be defined as a triple $M := (S, I, T)$ where:

- S is a set of symbols, also called variables.
- I is a propositional logic formula over the variables in S, this formula represents the set of initial states.
- T is a formula that defines the relation between the current and the next state, all transitions that satisfy this formula are valid transitions.

Sometimes they are defined using a 4-tuple $M := (S, I, N, T)$. This is an equivalent notation in which N allows to explicitly state invariants of the transition system. The invariants can be represented in T as $N \wedge N'$, where N' refers to the next state assignments. Thus forbidding any transition starting from or leading to a state in which N is not satisfied.

2.2. Timed transition systems

In discrete (or untimed) transition systems changes happen atomically and the evolution of the model is given by a sequence of discrete steps. Timed transition systems extend discrete systems with the

notion of *time*: a new kind of variables is introduced. These variables, called **clocks**, keep track of time elapses. A transition in a timed system is either a discrete step or a time elapse. During a time elapse all discrete variables retain their values while all clock variables increase of the same amount, equal to the time elapsed.

Timed automata

A timed automaton [4] is a transition system that can be represented by a finite graph in which nodes are called locations and edges represent discrete transitions. All clock variables are initialized to 0 and time can elapse inside locations constrained by some location invariant. Every discrete transition is associated with a **guard** condition and a set of clocks called **reset**. The automaton can perform a transition only if it satisfies the **guards** on such edge and when such transition is performed all clock variables specified in its **reset** are set to 0.
Location invariants impose progress conditions on the automaton, they forbid the system to stay in the same location indefinitely, while guards and resets on the edges constrain its behavior.
Given a set of clocks X, a clock variable $x \in X$ and a rational constant $c \in \mathbb{Q}$, let $\mathcal{B}(X)$ be the language defined by:

$$\varphi :: x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_0 \wedge \varphi_1 \tag{2.1}$$

This language defines the set of clock constraints that can be expressed in a timed automaton. Notice that it contains only conjunctions of comparisons with constants.
Formally a timed automaton M is a 6-tuple $M := (L, L_0, \Sigma, X, I, E)$ where:

- L is a finite set of locations;
- $L_0 \subseteq L$ is the set of initial location;
- Σ is the set of labels;
- X is the set of clock variables;
- $I : N \to \mathcal{B}(X)$ maps every location to its invariant condition;
- $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is the set of edges, each edge has a starting location, a guard, a label, a set of clock to be reset and a target location;

The syntax $l \xrightarrow{\text{g,a,r}} l'$ is equivalent to $(l, g, a, r, l') \in E$, where $l, l' \in L$ are locations, $g \in \mathcal{B}(X)$ is a guard, $a \in \Sigma$ is a label and $r \in 2^X$ is the set of clock reset by the transition.

A *clock interpretation* μ is a function that to every clock associates a value in $\mathbb{R}$, $\mu : X \to \mathbb{R}^{|X|}$. The semantic of a timed automaton is a timed transition system in which states are pairs (l, μ), where $l \in L$ is a location and μ is a clock interpretation. The transitions are defined by the following rules:

- let $d \in \mathbb{R}_+$, then $(l, \mu) \xrightarrow{\text{d}} (l, \mu + d) \iff (\mu \models I(l)) \wedge ((\mu + d) \models I(l))$; it is possible to perform a time elapse of d in location l if the clock interpretation at the beginning and at the end of the transition satisfies the location invariant of l.
- $(l, \mu) \xrightarrow{\text{a}} (l', \mu') \iff \exists (l, g, a, r, l') \in E : (\mu \models g) \wedge (\mu' = [r \mapsto 0]\mu) \wedge (\mu' \models I(l'))$; it is possible to perform a discrete transition labeled with $a \in \Sigma$ from location l to location l' if the current clock interpretation μ satisfies the guard g of the transition and the clock interpretation updated by the reset r ($\mu' = [r \mapsto 0]\mu$) satisfies the location invariant of the target state l'.

An **action** of a timed automaton M is a pair (t, a) where $a \in \Sigma$ is the label of the transition taken by the automaton M after $t \in \mathbb{R}_+$ time units from the start of the system. A **trace** of a timed automaton M is defined as a sequence of actions $\xi := (t_0, a_0), (t_1, a_1), \ldots$ such that $\forall i \geq 0 : t_i \leq t_{i+1}$. A **run** or **execution** of a timed automaton $M := (L, L_0, \Sigma, X, I, E)$ over a trace $\xi := (t_0, a_0), (t_1, a_1), \ldots$ is a sequence of transitions: $(l_0, \mu_0) \xrightarrow{d_1} \xrightarrow{a_1} (l_1, \mu_1) \xrightarrow{d_2} \xrightarrow{a_2} (l_2, \mu_2) \ldots$ where $\forall i \geq 1 : t_i = t_{i-1} + d_i$ and $(l_0, \mu_0) \in L_0$. The language of the timed automaton M, written $L(M)$, is the set of all traces ξ for which there exist a run of M over ξ

2.3. Propositional logic

This section gives the basic definitions of the most relevant terminology used in logic. These concepts are then used to define temporal logics, which are the most commonly used formal notation to express properties on transition systems.

A boolean formula in propositional logic φ is either a constant ($\top$, $\bot$), a propositional atom or a boolean operator applied to boolean formulae.

$$\varphi := \top \mid \bot \mid atom \mid \varphi_0 \wedge \varphi_1 \mid \varphi_0 \vee \varphi_1 \mid \neg\varphi_0 \mid \varphi_0 \rightarrow \varphi_1$$

where $\top$, $\bot$ are the constants representing respectively *true* and *false* and *atom* is an atomic proposition: a symbol whose value is either $\top$ or $\bot$. A *literal* is either a propositional atom or its negation. Let *Atoms* be a function that computes the set of atomic propositions occurring in a formula.

A total truth assignment μ over a formula φ is a function that to every atom in φ associates either *true* or *false*.

$$\mu : Atoms(\varphi) \rightarrow \{\top, \bot\}$$

A partial truth assignment μ over a formula φ is a function that for every atom in a subset of $Atoms(\varphi)$ it associates either *true* or *false*.

$$\mu : A \rightarrow \{\top, \bot\}, A \subseteq Atoms(\varphi)$$

A total truth assignment μ satisfies a formula φ ($\mu \models \varphi$) if and only if:

- $\mu \models A_i \iff \mu(A_i) = \top$ with $A_i \in Atoms(\varphi)$;
- $\mu \models \neg\varphi \iff \mu \not\models \varphi$
- $\mu \models \varphi_o \wedge \varphi_1 \iff (\mu \models \varphi_0) \wedge (\mu \models \varphi_1)$
- $\mu \models \varphi_o \vee \varphi_1 \iff (\mu \models \varphi_0) \vee (\mu \models \varphi_1)$
- $\mu \models \varphi_o \rightarrow \varphi_1 \iff (\mu \models \varphi_0) \rightarrow (\mu \models \varphi_1)$

A partial truth assignment satisfies a formula if all its total extensions satisfy that formula.

Validity, satisfiability, unsatisfiability, equivalence and equi-satisfiability

A formula φ is said:

- **valid** if $\forall\mu : \mu \models \varphi$, every truth assignment satisfies the formula,
- **satisfiable** if $\exists\mu : \mu \models \varphi$, there is at least one truth assignment that satisfies the formula,
- **unsatisfiable** if $\forall\mu : \mu \not\models \varphi$, no truth assignment satisfies the formula.

From these definitions it follows that φ is valid if and only if $\neg\varphi$ is unsatisfiable: $\forall\mu_0 : \mu_0 \models \varphi \iff \forall\mu_1 : \mu_1 \not\models \neg\varphi$. Two formulae φ and ϕ are **equivalent**, written $\varphi \equiv \phi$, if and only if $\forall\mu : (\mu \models \varphi \iff \mu \models \phi)$, while two formulae are **equi-satisfiable** if and only if $\exists\mu_0 : \mu_0 \models \varphi \iff \exists\mu_1 : \mu_1 \models \phi$. Notice that if two fomulae are equivalent then they are also equi-satisfiable, but the converse does not hold.

Moreover, by induction on the structure of the formula, it can be shown that the following equivalence relationships hold for every propositional formulae φ_0 and φ_1:

$$\begin{aligned}
\varphi_0 &\equiv \neg\neg\varphi_0 \\
\varphi_0 \vee \varphi_1 &\equiv \neg(\neg\varphi_0 \wedge \neg\varphi_1) \\
\varphi_0 \wedge \varphi_1 &\equiv \neg(\neg\varphi_0 \vee \neg\varphi_1) \\
\varphi_0 \rightarrow \varphi_1 &\equiv \neg\varphi_0 \vee \varphi_1
\end{aligned}$$

Complexity

Given a formula φ with N propositional atoms ($|Atoms(\varphi)| = N$) there are 2^N distinct truth assignments. The problem of deciding whether a formula is satisfiable (is there an assignment among the 2^N that satisfies φ?) is a well known and studied **NP-complete** problem and it is usually referred to as the **SAT** problem. Deciding validity and unsatisfiability of a formula are **coNP-complete** problems since their language is the complementary of **SAT**. It is possible to check if a formula is valid by deciding whether the negated is unsatisfiable, and a formula φ is unsatisfiable if and only if $\varphi \notin$ **SAT**.

2.4. Temporal logics

This section provides some background on temporal logics. They extend the syntax and semantic of propositional logic (2.3) with temporal operators. Many different temporal logics have been defined, some examples are: **LTL**[30], **CTL**[20], **CTL***[21], **MTL**[26][29] and **TCTL**[12][13]. A survey on temporal logics can be found in [12] and [13]. This section introduces the syntax and semantic of **LTL** in 2.4.1 and **MTL** in 2.4.2.

Temporal logic is any system of propositions qualified in terms of time, they can be categorized in linear temporal logics and branching logics. Linear temporal logics, like **LTL** and **MTL**, consider every execution as a single time line. Every state of a run has a single well defined successor and consists of a path with no branches starting from an initial state. On the other hand branching logics, like **CTL**, **CTL*** and **TCTL**, consider multiple time lines at a time. At every step there are multiple next states that represent all the possible choices. In this kind of logics each run consists in a directed graph rooted at an initial state, where the children of a node represent all possible next states of the system.

LTL

Linear Temporal Logic (**LTL**)[30] is a temporal logic that reasons on model executions as single time lines. It extends propositional logic with temporal operators that allow to predicate over previous and next states of these executions. In this work the version of **LTL** extended with past operators is considered.

Syntax

An atomic proposition is a LTL formula, a boolean operator applied to LTL formulae is a LTL formula, a temporal operator applied to LTL formulae is a LTL formula.

$$\begin{aligned}\varphi :: \top \mid \bot \mid atom \mid \varphi_0 \wedge \varphi_1 \mid \varphi_0 \vee \varphi_1 \mid \neg\varphi_0 \mid \varphi_0 \rightarrow \varphi_1 \mid \\ X\varphi \mid G\varphi \mid F\varphi \mid \varphi_0 U \varphi_1 \mid Y\varphi \mid H\varphi \mid O\varphi\end{aligned}$$

where *atom* is an atomic propositional formula and $\top$, $\bot$ are the constants representing respectively *true* and *false*. Notice that with respect to propositional logic (2.3) the only difference are the temporal operators. X (next), G (globally), F (finally) and U (until) are the operators that predicate over future states, while Y (yesterday or previous), H (historically), O (once) and S (since) are the operators that allow to predicate over past states.

Semantic

The semantic of propositional boolean operators remains unchanged, the semantic of temporal operators is defined on a path $\pi := s_0, s_1, \ldots$ as follows:

- $X\varphi$ holds at the current state s_i iff φ holds at the next step:

$$\pi, s_i \models X\varphi \iff \pi, s_{i+1} \models \varphi$$

- $G\varphi$ holds at the current step s_i iff in all future states φ will always hold:

$$\pi, s_i \models G\varphi \iff \forall j \geq i : \pi, s_j \models \varphi$$

- $F\varphi$ holds in the current state s_i iff after finitely many steps the path reaches a state in which φ holds:

$$\pi, s_i \models F\varphi \iff \exists j \geq i : \pi, s_j \models \varphi$$

- $\varphi_0 U \varphi_1$ holds in s_i iff at some point in the future φ_1 holds and φ_0 holds in all states from the current step to that point in the future:

$$\pi, s_i \models \varphi_o U \varphi_1 \iff \exists j \geq i : \pi, s_j \models \varphi_1 \wedge \forall i \leq k < j : \pi, s_k \models \varphi_0$$

- $Y\varphi$ holds in the current state s_i iff there exist at least one previous state and at the step φ held:

$$\pi, s_i \models Y\varphi \iff i > 0 \wedge \ \pi, s_{i-1} \models \varphi$$

- $H\varphi$ holds at state s_i iff in all previous steps φ held:

$$\pi, s_i \models H\varphi \iff \forall j \leq i : \pi, s_j \models \varphi$$

- $O\varphi$ is satisfied in the current state s_i iff there exists at least one state in the past in which φ held:

$$\pi, s_i \models O\varphi \iff \exists j \leq i : \pi, s_j \models \varphi$$

- $\varphi_0 S \varphi_1$ holds in s_i iff at some point in the past φ_1 held and φ_0 held in all states from that step excluded up to the current step:

$$\pi, s_i \models \varphi_o S \varphi_1 \iff \exists j \leq i : \pi, s_j \models \varphi_1 \wedge \forall j < k \leq i : \pi, s_k \models \varphi_0$$

Notice that the following equivalence relationships between temporal operators hold:

$$\begin{aligned} \neg X\varphi &\equiv X\neg\varphi \\ \neg G\varphi &\equiv F\neg\varphi \\ F\varphi &\equiv \top U\varphi \end{aligned}$$

similarly for past operators it can be shown that:

$$\begin{aligned} \neg Y\varphi &\equiv Y\neg\varphi \\ \neg H\varphi &\equiv O\neg\varphi \\ O\varphi &\equiv \top S\varphi \end{aligned}$$

A path $\pi := s_0, s_1, \ldots$ satisfies property φ if and only if the property holds in its initial state: $\pi \models \varphi \iff \pi, s_0 \models \varphi$. A transition system M satisfies a LTL property φ, written $M \models \varphi$ if and only if $\forall \pi \in M$, executions of the transition system, $\pi \models \varphi$.

MTL

Metric Temporal Logic (**MTL**)[26][29] extends **LTL** (2.4.1) with bounded versions of time operators. In LTL it is possible to predicate about relative time relationships between events. Given two events, LTL allows to check whether one happens before, together or after the other. Its not possible to quantify the amount of time elapsed between them. MTL adds this possibility: it allows to quantitatively predicate about time by placing explicit bounds on time operators. MTL is defined on continuous time semantic. Each execution becomes a dense sequence of configuration, therefore the semantic of the LTL operators X (next) and Y (yesterday) becomes ambiguous.

Syntax

An atomic proposition is a MTL formula, a boolean operator applied to MTL formulae is a MTL formula, a temporal operator applied to MTL formulae is a MTL formula.

$$\begin{aligned} \varphi :: \ & \top \mid \bot \mid atom \mid \varphi_0 \wedge \varphi_1 \mid \varphi_0 \vee \varphi_1 \mid \neg\varphi_0 \mid \varphi_0 \rightarrow \varphi_1 \mid \\ & G\varphi \mid F\varphi \mid \varphi_0 U \varphi_1 \mid H\varphi \mid O\varphi \mid \\ & \varphi_0 \ U_I \ \varphi_1 \mid G_I\varphi \mid F_I\varphi \mid H_I\varphi \mid O_I\varphi \\ I :: \ & [l, u] \mid (l, u] \mid [l, u) \mid (l, u) \mid [l, +\infty) \mid (l, +\infty) \end{aligned}$$

where l and u are integer constants, $+\infty$ represent positive infinite, *atom* is an atomic propositional formula, $\top$, $\bot$ are the constants representing respectively *true* and *false*.

Notice that with respect to LTL (2.4.1) the only difference are the bounded temporal operators: U_I, G_I, F_I, H_I and O_I and the absence of X and Y.

Semantic

The semantic of LTL and propositional operators is unchanged; the semantic of bounded temporal operators is defined on a path π, where $\pi(t)$ represents the total assignment over all symbols of the model defined by such execution at time $t \in \mathbb{R}_+$.

For some time interval I, the bounded time operators are defined as follows:

- $G_I\varphi$ holds at the current state $\pi(t)$ iff φ holds in all configurations in the time interval I:

$$\pi(t) \models G[l,u]\varphi \iff \forall k \in I : \pi(t+k) \models \varphi$$

- $F_I\varphi$ holds at the current state $\pi(t)$ iff φ holds in a configuration in the time interval I:

$$\pi(t) \models F_I\varphi \iff \exists k \in I : \pi(t+k) \models \varphi$$

- $\varphi_0\ U_I\ \varphi_1$ holds at the current state $\pi(t)$ iff φ_1 holds at some point in the interval I and until that time φ_0 holds:

$$\pi(t) \models \varphi_0\ U_I\ \varphi_1 \iff \exists j \in I, j > 0 : \pi(t+j) \models \varphi_1 \wedge \forall t < k < j\ \pi : s(t+k) \models \varphi_0$$

- $H_I\varphi$ holds at the current state $\pi(t)$ iff φ held in all configurations between in I:

$$\pi(t) \models G_I\varphi \iff \forall l \in I : \pi(t-k) \models \varphi$$

- $O_I\varphi$ holds at the current state $\pi(t)$ iff φ held in a configuration in interval I:

$$\pi(t) \models F_I\varphi \iff \exists l \in I : \pi(t-k) \models \varphi$$

Notice that all time intervals are interpreted relatively to the time t of the current state.

2.5. Formal verification of properties

This section provides a brief introduction of the main techniques applied to solve the model checking problem: $M \models \varphi$. This thesis focuses on the verification of properties expressed in LTL and a fragment of MTL. Three main approaches to model checking are shown: explicit state, symbolic and SAT/SMT based. This work is mostly interested in the symbolic and SMT based techniques. The last approach is a category of techniques that involves procedures like: bounded model checking, K-induction and IC3.

Formal verification is concerned with proving or disproving the correctness of some system with respect to certain specifications. A prominent approach to formal verification is model checking. Given a formal description of the system and a specification, usually expressed in logic, it performs an exhaustive exploration of the model in order to check whether the specification holds. When the specification is violated these techniques are able to provide a representation of the model execution that does not satisfy the property, this trace is called counter-example.

LTL model checking

This section shows the main result used to perform LTL model checking, both in explicit state and symbolic approaches. The verification problem is reduced to checking the language emptiness of a transition system.

Let M be a transition system and φ an LTL specification to be verified on M. The model checking problem $M \models \varphi$ can be stated in terms of language inclusion as $L(M) \subseteq L(\varphi)$. This implies that $L(M) \cap \overline{L(\varphi)} = \emptyset$, where $\overline{L(\varphi)}$ is the set of all execution that do not satisfy φ (set complement). From the definition $\overline{L(\varphi)} = L(\neg\varphi)$, substituting this in the previous formulation it is possible to

obtain $L(M) \cap L(\neg\varphi) = \emptyset$. Let $M_{\neg\varphi}$ be a transition system such that $L(M_{\neg\varphi}) = L(\neg\varphi)$, then $L(M) \cap L(M_{\neg\varphi}) = \emptyset$. Since the language intersection of two transition system is equal to the language of the automaton given by their synchronous composition: $L(M \times M_{\neg\varphi}) = \emptyset$.

In explicit state model checking these operations are performed on the graph that represents the transition system, while in the symbolic technique SAT/SMT formulae that represent sets of states are manipulated. This allows to avoid the creation of the explicit graph. The language emptiness in explicit state techniques can be easily decided by performing a visit on the graph and searching for a looping path starting from an initial state and such that the loop involves at least one fair state. In symbolic techniques the same condition can be checked by resorting to symbolic CTL model checking.

Many details have been omitted in this brief explanation, in [5] it is possible to find a complete description of both explicit state and symbolic LTL model checking. The following section provides a more detailed description of the symbolic approach; it also highlights the main differences with respect to explicit state techniques.

Symbolic model checking

The explicit state techniques require to explicitly create and store the transition system. The number of configurations may be exponentially large in the number of propositional atoms of the model. In these cases this kind of techniques quickly become too expensive to be performed. Symbolic model checking [5] tries to avoid this issue by manipulating sets of states at a time. This is achieved by representing them as formulae in propositional logic. In a similar way, these techniques are able to manipulate sets of transitions represented by propositional formulae expressed in terms of current and next state variables. It is important to notice that all logically equivalent formulae represent the same set of states or transitions. This allows to perform all the simplifications available in propositional logic to obtain a more compact expression. This also implies that the size of a formula is not directly related to the cardinality of the set it represents.

More formally, each state of the transition system is represented by an array of boolean state variables $V := (x_0, x_1, \ldots, x_k)$. Let $\xi(s)$ be the symbolic representation of the state s, which is a total assignment over V. A subset of states $Q \subseteq S$ is represented by $\bigvee_{s \in Q} \xi(s)$ and all equivalent formulae. Set operations are mapped by boolean operators, in particular for $P, Q \subseteq S$:

$$\xi(P \cap Q) = \xi(P) \wedge \xi(Q)$$

$$\xi(P \cup Q) = \xi(P) \vee \xi(Q)$$

$$\xi(S \setminus P) = \neg\xi(P)$$

Transitions in $R \subseteq (S \times S)$ are represented by propositional formulae over V and V', where V' is the array of boolean variables representing their values after the transition. R can be symbolically represented by all formulae equivalent to:

$$\bigvee_{(s,s') \in R} \xi(s, s') \equiv \bigvee_{(s,s') \in R} \xi(s) \wedge \xi(s')$$

where $\xi(s)$ is a total assignment over V and $\xi(s')$ is a total assignment over V'.

SAT/SMT based model checking

SAT/SMT based techniques are symbolic techniques that reduce the verification problem to a satisfiability problem. They try to exploit the advancements in SAT and SMT fields to achieve better scalability.

The SAT problem has already been introduced in section 2.3.2. Satisfiability Modulo Theory (SMT) is the problem of deciding whether a boolean formula has at least one model with respect to combinations of background theories (e.g. reals, uninterpreted functions, arrays and bit vectors). In an SMT instance predicates consist of boolean expressions on some underlying theories: $x - 3 < 3.5$ is a SMT predicate over the reals. Some examples of SMT solvers are CVC4, MathSAT, SMT-RAT, Yices and Z3.

Bounded model checking

In *bounded model checking* [10] the execution of a model M is unrolled up to k steps. A single formula representing all executions of length k violating the specification φ is built. The formal verification task is solved by checking if there exists a model for such formula. The assignments prescribed by this model represent a run of length k of M that violates the specification: a counter-example. This procedure is iterated for increasing values of k until a counter-example is found. BMC is only capable of falsifying properties: if a property holds this procedure will never halt. For this reason it is often used in combination with *K-induction* (2.5.3).

Let $M := (S, I, T)$ be a transition system and φ be a LTL formula. Let $[[\varphi]]_t^f$ be the BMC expansion of φ from step $f \in \mathbb{N}$ to step $t \in \mathbb{N}$ without loop-backs and ${}_l[[\varphi]]_t^f$ the encoding with loop-back from t to l. This encoding is straightforward for standard boolean operators, while more attention is required for temporal operators.

φ	$[[\varphi]]_t^f$	${}_l[[\varphi]]_t^f$
p	p	p
$\neg p$	$\neg p$	$\neg p$
$\varphi_0 \wedge \varphi_1$	$[[\varphi_0]]_t^f \wedge [[\varphi_1]]_t^f$	${}_l[[\varphi_0]]_t^f \wedge {}_l[[\varphi_1]]_t^f$
$\varphi_0 \vee \varphi_1$	$[[\varphi_0]]_t^f \vee [[\varphi_1]]_t^f$	${}_l[[\varphi_0]]_t^f \vee {}_l[[\varphi_1]]_t^f$
$X\varphi_0$	$if\ i \geq k:\ \bot,$ $else:\ [[\varphi_0]]_t^{f+1}$	$if\ i \geq k:\ {}_l[[\varphi_0]]_t^l,$ $else:\ {}_l[[\varphi_0]]_t^{f+1}$
$G\varphi_0$	$\bot$	$\bigwedge_{j=min(l,f)}^{t} {}_l[[\varphi_0]]_t^j$
$F\varphi_0$	$\bigvee_{j=f}^{t} [[\varphi_0]]_t^j$	$\bigvee_{j=min(l,f)}^{t} {}_l[[\varphi_0]]_t^j$

Table 2.1: BMC encoding

$X\varphi$ corresponds to evaluating φ in the next state, if such state does not exists its false. $G\varphi$ is true only if there is a loop and in each state of the loop φ holds. $F\varphi$ holds if there exists a state from the beginning to the k-th that satisfies φ. The until temporal operators has been omitted for brevity.

The BMC procedure at every step k checks if the propositional formula

$$I(s_0) \wedge \bigwedge_{i=0}^{n-1} (T(s_i, s_{i+1} \wedge \varphi(s_i)) \wedge \neg\varphi(s_n)$$

is satisfiable. If a model for such formula exists, then its assignments are a counter-example for property φ and it is possible to conclude that the specification is violated. Notice that the encoding generates a symbolic representation of all paths of length k starting from the initial states such that at every step but the last one φ holds. As already stated, the satisfiability check is performed by calling SAT or SMT procedures. The BMC technique performs one SAT/SMT call for every k. Subsequent encodings share most of the clauses, this can be exploited to improve scalability and decrease the running time.

K-induction

K-induction [31] complements BMC in the sense that it is able to verify properties. This technique symbolically identifies all states that violate an invariant property (*bad states*) and tries to prove that none of them is reachable. In order to do this it tries to show that there exist no path of length k that ends in a bad state. As in the BMC case this is repeated for increasing values of k. The paths are created by going backward from the bad states, if at step k no such path exists then the specification holds, otherwise the procedure is repeated for $k + 1$. Notice that checking whether one of the paths starts from an initial state leads exactly to the BMC formulation.

As for the BMC case, let $M := (S, I, T)$ be a transition system and φ be a LTL formula. In the K-induction encoding there are two main components: one rules out all loops

$$\bigwedge_{0 \leq i < j \leq k} \neg(s_i = s_j)$$

and the other one builds a symbolical representation of paths of length k ending in a bad state

$$\bigwedge_{i=0}^{k} (T(s_i, s_{i+1}) \wedge \varphi(s_i)) \wedge \neg\varphi(s_{k+1}).$$

As in the BMC case subsequent encodings share most of the formula, moreover the formula that creates a path of length k is shared between BMC and K-induction. For this reason these two techniques are often applied together exploiting the sharing and incrementality of these formulae to improve performance.

IC3

Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) is a model checking algorithm invented by Aaron Bradley in 2010. In its original form it was meant to solve reachability problems expressed in SAT. Later works extended it to deal also with liveness, incremental reasoning and SMT formulae. Experimentally IC3 appears to be superior to any other single solver used in the hardware model checking competition [2]. This work first provides a high level description of the algorithm in the propositional finite state case, as described by Aaron R. Bradley in [14], then two relevant extensions to SMT infinite state are presented.

Propositional finite state

Let S be a transition system described by $I(X)$, $T(Y, X, X')$ that represent respectively the set of initial states and the transition relation; X and X' represent current and next state variables, while Y represents primary input variables to the system. Let $P(X)$ describe a set of *good states*. The objective of the IC3 algorithm is to prove that all states reachable by S are good. This is achieved by finding an inductive invariant $F(X)$ which proves that S satisfies P [17]. $F(X)$ must be such that:

(a) $I(X) \models F(X)$, the initial states satisfy the invariant;

(b) $F(X) \wedge T(Y, X, X') \models F(X')$, every state reached in one step from a state that satisfies the invariant also satisfies it;

(c) $F(X) \models P(X)$, from the invariant its possible to conclude P.

Its easy to notice that the conjunction of these 3 requirements on F is sufficient to conclude that P holds in every reachable configuration.

F is built by keeping a sequence, called *trace*, of formulae, called *frames*, $F_0(X), F_1(X), \ldots, F_k(X)$ such that:

- $F_0 = I$, the first formula of the trace represents the initial states of the transition system;
- $\forall i > 0$ F_i is a set of clauses: conjunction of disjunctions;
- $\forall 0 < i < k : F_{i+1} \subseteq F_i$, which implies $F_i \models F_{i+1}$;
- $\forall 0 < i < k : F_i(X) \wedge T(Y, X, X') \models F_{i+1}(X')$, the symbolic representation obtained by performing a single step from a given frame is a model for the following frame;
- $\forall 0 \leq i < k : F_i \models P$.

Each F_i symbolically represents a superset, or over-approximation, of the set of states that the machine M can reach in i steps. Therefore the trace $F_0(X), F_1(X), \ldots, F_k(X)$ represents an over-approximation of all possible executions of length k.

The IC3 procedure can be split into two main phases: a *blocking* and a *propagation* phase. During the *blocking* phase the approximation is refined by adding additional clauses. If during this operation some frame F_i becomes such that $F_i \wedge \neg P \not\models \bot$: it has a non empty intersection between the states

it represents and the bad states, then its possible to reconstruct a counter-example using the frames $F_0, \ldots, F_i$. The *propagation* phase tries to extend the trace with an additional frame F_{k+1}. This frame is generated by moving forward the clauses of the previous F_i. If during this process for some i $F_i = F_{i+1}$ holds, then a fix point has been reached and F_i is the inductive invariant that proves the property.

The following section gives a more detailed description on how the approximation is refined. IC3 maintains a set of *proof obligations*, which are pairs (s, i) such that i is the step index and s is a *counter-example to induction* (CTI). A CTI is symbolic representation of a set of states that might not be reachable from the initial states and have at least one successor that either is or can reach a bad state. The unreachability of s from F_{i-1} is recursively proved by checking whether $\neg s$ is relative inductive to F_{i-1}: $F_{i-1} \wedge \neg s \wedge T \models \neg s'$. This can be decided by the unsatisfiability of:

$$F_{i-1} \wedge \neg s \wedge T \wedge s' \tag{2.2}$$

If 2.2 is unsatisfiable and s does not contain any initial state, then $\neg s$ can be used to strengthen F_i. In particular $\neg s$ is generalized into a formula g which is added to F_i. This generalized formula g must be such that $g \models \neg s$ and it is relative inductive to F_{i-1}: it also makes 2.2 unsatisfiable. Since the objective of this section is to provide a high level intuition on how IC3 works, the description of the generalization procedure is omitted. If 2.2 is satisfiable, then either s is reachable or the over-approximation F_{i-1} is too coarse grained to prove its unreachability. Let c be the symbolic representation of the set of states such that $(F_{i-1} \wedge \neg s \models c) \wedge (\exists Y : c(X) \wedge T(Y, X, X') \models s(X'))$. The states in c are a subset of the intersection of F_{i-1} and $\neg s$, this intersection can not be empty since 2.2 is satisfiable by hypothesis. Moreover each state in c can reach a state in s' in one step for some assignment over the input variables Y: they are in the preimage of s. To decide whether $\neg s$ is reachable or not it is sufficient to check if c is reachable from F_{i-2}. This corresponds to trying to block the *proof obligation* $(p, i-1)$. This is repeated recursively until either the unreachability is proved or the first frame $(i = 0)$ is reached. In the latter case the generated sequence of *proof obligation* represents a counter example for the property P on S.

SMT infinite state generalization

IC3 has been generalized to be applicable to SMT problems and infinite state transition systems. While in the previous case the problem was decidable, in this case it is not, therefore completeness is impossible to achieve.

In the SMT case if formula 2.2 is satisfiable a total assignment c in the preimage of s is obtained. Depending on the theory used, c might represent a single point in an infinite space and as a consequence lead to a high chance of divergence in the blocking phase. For this reason it is necessary to generalize c into a set of predecessors of s.

In the literature it is possible to find different techniques to solve this problem: some rely on quantifier elimination [17], others try to obtain better performance by embedding in the generalization procedure theory specific procedures [22] and finally implicit abstraction techniques are less dependent on the specific underlying theory [19]. The following paragraphs provide a high level overview of the IC3 SMT generalization of the last two approaches.

Theory dependent generalization Hoder and Bjørner [22] in 2012 proposed an extension of IC3 to the SMT case by embedding in the generalization of the CTI a theory dependent component. The propositional and theory specific procedures are combined together with the objective of obtaining a SMT formula g such that $g \models \neg s$ and $F_{i-1} \wedge g \wedge T \wedge \neg g'$ is satisfiable (2.2). In the case of linear rational arithmetic (LRA) after the boolean generalization procedure the algorithm tries to weaken inequalities of the form $t \leq c$ to $t \leq c + c'$ with $c' > 0$ such that the SMT formula 2.2 still holds [22].

The main drawback of this approach is the requirement of a specialized generalization procedure for each theory.

Implicit abstraction Cimatti et alii [19] in 2016 proposed IC3-IA: an abstraction based extension to the infinite state SMT case of IC3 that does not require theory specific generalization

procedures. $\hat{S}$ is an abstraction of the transition system S if and only if every state reachable in S has a corresponding abstract state reachable in $\hat{S}$. This procedure allows to reduce the size of the state space and improve the performance of the verification procedures. The abstract state space is induced by a set of predicates $\mathbb{P}$ and generated by the abstraction relation $H_{\mathbb{P}}(X, X_{\mathbb{P}}) \coloneqq \bigwedge_{p \in \mathbb{P}} x_p \leftrightarrow p(X)$, where x_p is a boolean variable of the abstract space. The predicate abstraction of a formula φ with respect to $\mathbb{P}$, written $\hat{\varphi}_{\mathbb{P}}$ is computed as:

$$\begin{aligned} \hat{\varphi}_{\mathbb{P}}(X_{\mathbb{P}}) &\coloneqq \exists X \quad : \varphi(X) \quad \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \\ \hat{\varphi}_{\mathbb{P}}(X_{\mathbb{P}}, X'_{\mathbb{P}}) &\coloneqq \exists X, X' : \varphi(X, X') \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X', X'_{\mathbb{P}}) \end{aligned} \tag{2.3}$$

The IC3 procedure is modified to learn clauses over predicates in the abstract space. The predicate abstraction (2.3) is used to obtain an abstract transition relation $\hat{T}(X, Y')$ from $T(X, X')$. This relation is replaced in 2.2 leading to:

$$F(X) \wedge s(X) \wedge \hat{T}(X, Y') \wedge \neg s(X') \wedge \bigwedge_{p \in \mathbb{P}} (p(X') \leftrightarrow p(Y')) \tag{2.4}$$

$s(X)$ is relative inductive to $F(X)$ if formula 2.4 is unsatisfiable. If $s(X)$ is relative inductive the inductive strengthening procedure described for the boolean case is applied, otherwise the abstract predecessor c can be computed from the satisfying SMT model μ as:

$$c \coloneqq \{p(X) \mid p \in \mathbb{P} \wedge \mu \models p(X)\} \cup \{\neg p(X) \mid \mu \not\models p(X)\}$$

Notice that it is not necessary to compute the abstract transition relation $\hat{T}(X, Y')$ explicitly. It can be done implicitly by adding the abstraction procedure to the SMT formulation.

If property $\hat{P}$ holds in the abstract transition system $\hat{S}$, then the original property P holds in the original transition system S. Otherwise an abstract counter-example $\hat{\pi} \coloneqq \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_k$ is generated. This execution of $\hat{S}$ might not correspond to any execution on S, it might be spurious. This can be verified by considering the SMT formula representing all paths of the same length of $\hat{\pi}$ on the original model and imposing the abstraction of each step to be the abstract state prescribed by $\hat{\pi}$. If this formula is unsatisfiable then $\hat{\pi}$ is spurious, otherwise the satisfying model provides an execution $\pi \coloneqq s_0, \ldots, s_k$ such that every $\hat{s}_i \in \hat{\pi}$ is the abstract representation of $s_i \in \pi$ and $\pi \not\models P$. If a spurious counter-example is identified, the set of predicates $\mathbb{P}$ is updated so that at least $\hat{\pi}$ is removed from the language of $\hat{S}$. In the literature many possible implementations of this refinement procedure have been developed. However in this work they are not discussed, a more detailed explanation of this topic can be found in [19].

It is relevant to notice that the selection of the predicates to be considered in $\mathbb{P}$ greatly affects the overall performance.

3. State of the Art

After some general considerations about the decidability of some problem on timed automata, this chapter provides a description of the main state-of-the-art approaches in the field of formal verification of timed transition systems. Tools implementing these techniques are used in chapter 9 to evaluate the performance of *Timed nuXmv* for invariant and MTL checking.

3.1. Timed automata decidability

Alur and Dill in [3] and [4] show some interesting results about the complexity of some problems on timed automata. The reachability problem and checking language emptiness of timed automata are decidable, as in the discrete case, and they belong to the PSPACE-complete class. However, in the same publications, they show that given two timed automata M and M', the language inclusion problem $L(M) \subseteq L(M')$ is undecidable. Moreover, as shown in [15], it is enough to allow additive clock constraints to the timed automata language $\mathcal{B}(X)$ (2.1) to make also the emptiness checking undecidable.

3.2. Region automata

Timed automata have real valued clocks, this implies that the state space of their transition system is infinite. Most state-of-the-art approaches build a finite state abstraction of a timed automata. The time behavior of the system is represented by a finite set of equivalence classes called **regions**. This section describes the main procedure used to create this finite state abstraction of the infinite timed transition system [5].
The infinite state space is split into finitely many partitions by considering each of these partitions as an abstract state. This allows to build a finite abstract state space. Every partition represents an equivalence class over the states of the timed transition system, these states are such that they all exhibit the same behavior in the abstract space. Notice that, from the definition of $\mathcal{B}(X)$ (2.1), clocks are compared only with constants and, since the model description is finite, then for each clock x there are finitely many of such values in $\mathbb{Q}$. Let c_x be the maximum constant to which the clock x is compared to. Given two clock interpretations $\mu, \mu' : X \to \mathbb{R}^{|X|}$, let $\lfloor \mu(x) \rfloor$ be the integral part of the value associated to clock x by the interpretation μ, and let $frac(\mu(x)) := \mu(x) - \lfloor \mu(x) \rfloor$ be its fractional part. This notation allows to define the equivalence relationship over clock interpretations. μ is equivalent to μ', written $\mu \equiv \mu'$, if and only if the following conditions hold:

1. $\forall x \in X : \lfloor \mu(x) \rfloor = \lfloor \mu'(x) \rfloor \vee (\mu(x) > c_x \wedge \mu'(x) > c_x)$, for every clock the interpretations μ and μ' either agree on the integral part or the assigned values are greater than the maximum constant corresponding to that clock variable.
2. $\forall x, y \in X \ \mu(x) \leq c_x \wedge \mu(y) \leq c_y : frac(\mu(x)) \leq frac(\mu(y)) \iff frac(\mu'(x)) \leq frac(\mu'(y))$, for all pairs of clocks that are mapped to values smaller than their respective maximum constants, the two interpretations have the same ordering of the clocks based on their fractional part. Notice that from the previous condition these two clocks share the same integral part.
3. $\forall x \in X \mu(x) \leq c_x : frac(\mu(x)) = 0 \iff frac(\mu'(x)) = 0$, one interpretation assigns a value with fractional part 0 (an integer) smaller than the maximum constant associated to a clock if and only if also the other interpretation assign an integer value to the same clock. Notice that, together with the first condition, this implies that they assign the same value.

These three conditions partition the infinite clock space $\mathbb{R}^{|X|}$ into finitely many equivalence classes called regions. Their number is finite since in each model there can be only a finite number of constants.

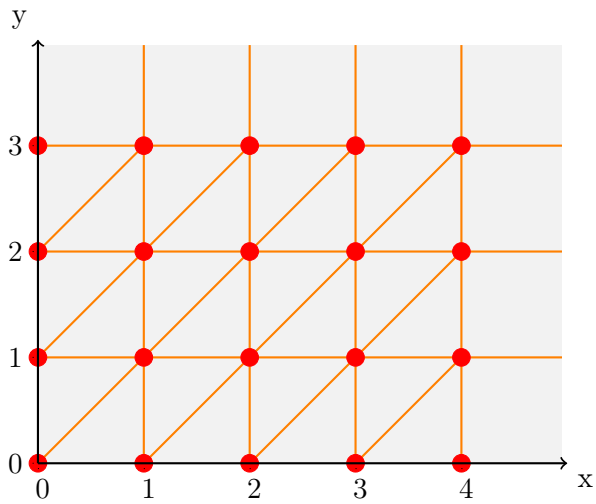


Figure 3.1: Regions of a simple timed automaton with two clock variables bounded respectively by 4 and 3

Figure 3.1 shows the regions created by a timed automaton with two clocks x, y, $c_x = 4$ and $c_y = 3$. The regions are highlighted by the colors: each light gray area, red dot and orange segment identifies a different region.

The region automaton is defined as the transition system that has a node for each region and there exists an edge between two regions r_0, r_1 with label $a \in \Sigma$ if and only if there exist two locations l_0, l_1 of the timed automaton such that $l_0 \in r_0$ and $l_1 \in r_1$ and there is a transition with label a from l_0 to l_1. Regions preserve the behavior of the timed automaton in the sense that for every transition of the timed automaton there is a corresponding transition of the region automata. More formally there is a **bisimulation** relationship between the timed automaton and its corresponding region automaton.

- For every action transition of the timed automaton if μ_0, μ_1 are two clock interpretations such that $\mu_0 \equiv \mu_1$ and $(l, \mu_0) \xrightarrow{a} (l', \mu'_0)$ for some μ'_0 then $\exists \mu'_1 : \mu'_1 \equiv \mu'_0 \wedge (l, \mu_1) \xrightarrow{a} (l', \mu'_1)$.
- For every delay transition of the timed automaton if μ_0, μ_1 are two clock interpretations such that $\mu_0 \equiv \mu_1$, then $\forall d \in \mathbb{R}\ \exists d' \in \mathbb{R} : \mu_0 + d \equiv \mu_1 + d'$.

Tools like Atmoc (9.1.2) symbolically encode the region equivalence over clock variables and solve the model checking problem on the region abstracted transition system.

3.3. Zones automata and DBM

Another technique symbolically handles multiple regions at a time using **zones**. A **zone** ψ is the conjunction of a set of clock constraints of the form $x - y \leq c$, $x - y < c$, $x < c$, $x \leq c$, $x = c$, $x > c$ or $x \geq c$ where x, y are clocks and c is a constant. ψ can be represented in the clock space as a $|X|$ dimensional convex shape built by the union of a subset of regions, where $|X|$ is the cardinality of the set of clock variables. There are three main operations that can be performed on zones: intersection, delay and reset.

The **intersection** of two zones ψ_0, ψ_1 is a zone $\psi_2 := \psi_0 \wedge \psi_1$ that is the conjunction of the constraints of ψ_0 and ψ_1; ψ_2 is visually represented by the intersection of the two areas that describe ψ_0 and ψ_1. This operator allows to compute the logical conjunction of the clock constraints represented by the two zones.

The **delay** of a zone $\psi := r_0, r_1, \ldots, r_k$, where the $r_0, r_1, \ldots, r_k$ are regions, is defined as $\psi \Uparrow := \{\mu + d \mid \exists r_i \in \psi : \mu \in r_i \wedge d \in \mathbb{R}_+\}$, which represents the zone made by all regions that are reachable from ψ with delay transitions; $\psi \Uparrow$ is graphically obtained by stretching ψ along the diagonal direction. This operator allows to compute all states reachable from a zone by performing only time elapse transitions.

The **reset** of a zone ψ given a subset of clocks $r \subseteq X$ is obtained by projecting its area along the axes corresponding to those clock variables: $\psi[r := 0]$. This last operator allows to compute the resulting

zone after the resets of a transition.

The **zone automaton** is defined as a transition system in which states are pairs (l, ψ), where l is a location and ψ a zone. Its transition relation is defined by:

- $(l, \psi) \rightsquigarrow (l, \psi \Uparrow \wedge I(l))$, where $I(l)$ is the location invariant of l;
- $(l, \psi) \rightsquigarrow (l', \psi[r := 0] \wedge g \wedge I(l'))$ if $l \xrightarrow{g,a,r} l'$

This definition of the zone automaton, corresponding to a timed automaton with initial state (l_0, μ_0), guarantees that:

- *soundness*: $((l_0, \mu_0) \rightsquigarrow^* (l_f, \psi_f)) \Longrightarrow \exists \mu_f \in \psi_f : (l_0, \mu_0) \rightarrow^* (l_f, \mu_f)$, if a configuration is reachable in the zone automaton then there exists a corresponding reachable state in the timed automaton.
- *completeness*: $(l_0, \mu_0) \rightarrow^* (l_f, \mu_f) \Longrightarrow \exists \psi_f : \mu_f \in \psi_f \wedge ((l_0, \mu_0) \rightsquigarrow^* (l_f, \psi_f))$, if a configuration is reachable in the timed automaton then the corresponding zone is also reachable in the zone automaton.

Zones can be stored using Difference Bound Matrices (**DBM**). They efficiently support zones operations and have a canonical representation. Let $X_0 := X \cup \mathbf{0}$, where $\mathbf{0}$ is a reference clock. Every clock constraint in a zone ψ is rewritten in the form $x - y \circ n$ where $x, y \in X_0 \wedge \circ \in \{<, \leq\}$. ψ can be represented in a $|X_0| \times |X_0|$ matrix $M(\psi)$, where the first row and first column represent the reference clock $\mathbf{0}$ and the others represent the other clocks in some order. Each cell of the matrix keeps track of whether the bound is strict or not and the value of the bound itself: $M(\psi)_{ij} = (n, \circ)$ implies that $c_i - c_j \circ n$ for some $\circ \in \{<, \leq\}$.

Tools like Uppaal (9.1.1) and Opaal (9.1.3) use this kind of representation to handle to behavior over time of the model, while the discrete development is handled using explicit state techniques. This implies that the zone representation is done for each possible discrete location. For this reason these kind of techniques usually scale well with respect to increasing time constraints. However, they are unable to manage complex discrete behavior since the number of states increases exponentially and the explicit state representation of such systems becomes very large.

4. nuXmv

This work is grounded on the nuXmv model checker. nuXmv is actively maintained and used by the Embedded System unit in Fondazione Bruno Kessler. It lies at the core of a set of other tools like HyComp, Ocra and xSAP. It allows to analyze synchronous finite and infinite state systems. In this chapter the relevant fragment of the input language of nuXmv will be described along with its main functionalities. In particular the model simulation and trace re-execution features will be presented. For a complete description please refer to the nuXmv user manual, [16] and [9].

4.1. Input language

In this section the main constructs of the nuXmv input language will be described. For compactness reasons this description will focus on the subset of most relevant features that are required for a better understanding of the extensions and reduction technique presented in chapters 5, 6 and 7.

Supported types

Boolean Like most formal languages, nuXmv supports the `boolean` type which comprises the symbolic values `TRUE` and `FALSE`.

Enumeration Enumeration types are sets of values, in particular three different types of enumerations are supported:

- **symbolic enumeration**: the elements of the set are all symbolic constants,
  e.g. `{a, z, e, k, ya}`;
- **integer enumeration**: the elements of the set are all integer constants,
  e.g. `{2, 6, 20, 17, -3}`;
- **integer and symbolic enumeration**: the elements of the set are either integer constants or symbolic constants,
  e.g. `{2, h, 20, k, -3}`.

Word `unsigned word[N]` and `signed word[N]` are used to represent bit vectors of fixed length `N`. Signed vectors allow signed operations while unsigned vector allow unsigned operations. Signed words are represented using the usual *two's complement*.

Integer The domain of a variable of type `integer` is the whole set of integer numbers: $\mathbb{Z}$. Therefore is has an infinite discrete domain. However, in the input model it is possible to specify integer constants only in the range: $[-2^{32}+1; 2^{32}-1]$.

Real The domain of a variable of type `real` is the whole set of rational numbers: $\mathbb{Q}$. A rational constant can be expressed in the input model using the following syntax: `f'3/4`

Array Array types allow to model sequences of elements of other types. In particular it is possible to specify the lower and upper bound for the index of the array and the type the elements in it. The element type can be an array type itself.

Variables declarations

Variables define the set of possible states of the model. A total assignment over the variables of a model uniquely identifies a configuration of the system. The set of all possible configurations is the **state space**. Every variable is associated to one of the types described above.

State variables State variables keep the system's configurations, their values might change as the system evolves through transitions.

Input variables Input variables allow to model external inputs that are not under the control of the system. In particular these kind of variables allow to put labels on the transitions.

Frozen variables Frozen variables are variables that retain the same value for the whole model execution. Their initial value can be constrained in the same way of state variables. The reader may think of fronzen variables as constant or immutable variables of other languages.

Define declarations

Defines allow to create symbols that are equivalent to another expressions. They are macros, since whenever a define identifier is found, it is syntactically replaced by the expression it is associated with.

Constants declarations

In the nuXmv language it is possible to declare a set of symbolic constant that can be used in the model description.

Constraints

In the following the main syntactical elements to describe a system behavior are presented.

INIT constraints `INIT` allows to specify the possible initial configurations of the system. This construct allows to specify symbolically the set of initial states: every state such that all the `INIT` constraints evaluate to `TRUE` is an initial state.

INVAR constraints `INVAR` allows to specify conditions that must be satisfied by every state of the system. Every `INVAR` constraints identifies a subset of valid states in the state space. The intersection of all these sets is the set of valid states of the model.

TRANS constraints `TRANS` constraints allow to specify how the system evolves at each step. The language allows to access the value of a symbol `x` after the next transition using `next(x)`. `TRANS` constraints are predicates over current and next variables. The possible transitions of the systems are all pairs of current and next state that satisfy such predicates. If at a given state there are multiple transitions the system can choose any of them. Otherwise, if at a given state there are no valid transitions, such state is called **deadlock state**.

MODULE declarations

A `MODULE` allows to encapsulate a collection of declarations, constraints and specifications in a reusable component that has a new identifier scope. Modules can be instantiated as normal variables and each of these instances refers to different data structures. A module has a set of formal parameters that are matched with a set of actual parameters for each module instance. `MODULE` resemble the concept of classes of Object oriented programming languages.

Specifications

This section provides a brief description of the three main kinds of properties which nuXmv is able to verify. For a formal description of the semantic of the languages described in this section please refer to 2.4.

Invariant specification

Invariants are propositional formulas built using standard logical and mathematical operators over current and next states. An invariant specification is verified by a model if the initial states satisfy such formula and all states that the system can reach starting from them also satisfy the formula. If an invariant specification does not hold nuXmv shows a trace of the model that starting from an initial state reaches a configuration in which the invariant is falsified. This counter-example may or may not belong to an infinite run of the system.

LTL specification

nuXmv can also check specifications expressed using Linear Temporal Logic (LTL) extended with past operators. A LTL property is verified by a system if such property holds in all the initial states of the model. In LTL a single trace is considered at a time, therefore previous and next state are uniquely

determined. The verification procedure proves that in every such trace the specification holds. In particular the main LTL operators are:

- `X ltl_expr` : holds iff `ltl_expr` holds in the next state.
- `G ltl_expr` : holds iff for all following states `ltl_expr` holds.
- `F ltl_expr` : holds iff there exists a following state in which `ltl_expr` holds.
- $\texttt{ltl\_expr}_1$ `U` $\texttt{ltl\_expr}_2$: holds if `F` $\texttt{ltl\_expr}_2$ holds in the current state and $\texttt{ltl\_expr}_1$ holds in all states until the state in which $\texttt{ltl\_expr}_2$ holds is reached.

For the formal definition of the semantic of the LTL operators see 2.4.1.

4.2. Model example

In the following a simple example of nuXmv model is presented.

```
MODULE main
  VAR
  p0 : Philosopher(p1);
  p1 : Philosopher(p2);
  p2 : Philosopher(p3);
  p3 : Philosopher(p4);
  p4 : Philosopher(p0);

-- invariant: never happens that three or more eat together: TRUE
INVARSPEC count(p0.state = eating, p1.state = eating, p2.state = eating,
                p3.state = eating, p4.state = eating) < 3;

-- infinitely often if someone wants to eat then someone eventually eats.
LTLSPEC G ((F (p0.state = trying | p1.state = trying | p2.state = trying |
               p3.state = trying | p4.state = trying)) ->
           F (p0.state = eating | p1.state = eating | p2.state = eating |
              p3.state = eating | p4.state = eating));

MODULE Philosopher(neighbor)
  VAR
    fork : {down, mine, other};
    state : {thinking, trying, eating};

  DEFINE eat := (state=eating)? 1 : 0;

  INVAR
    (state = eating <-> (fork = mine & neighbor.fork = other)) &
    (state = thinking <-> (fork != mine & neighbor.fork != other)) &
    (state = trying <-> ((fork = mine & neighbor.fork != other) |
    (fork != mine & neighbor.fork != other))) &
    (neighbor.fork = other -> fork = mine)

  TRANS
    (state = trying -> (next(state) = trying | next(state) = eating)) &
    (state = eating -> (next(state) = eating | next(state) = thinking)) &
    (state = thinking -> (next(state) = thinking | next(state) = trying)) &
    ((state = trying & fork=down) -> (next(state) = trying & next(fork)=mine)) &
    ((state = trying & fork=mine & neighbor.fork = down) ->
       (next(state) = eating | (next(neighbor.fork) = mine & next(fork) = mine)));
```

This simple example describes the popular dining philosophers problem. There are four philosophers on a circular table, each of them has a shared fork at his right and one at his left. In order to eat they need to take both forks. In the model there are two properties. The invariant allows to verify that it never happens that three or more philosophers eat at the same time. The LTL specification does

not hold and shows that there exists a sequence of actions that makes the philosophers starve: each of them picks one fork and waits for the neighbor to release his own.

4.3. Trace simulation, execution and completion

nuXmv provides a set of commands to generate, validate and complete traces. This section describes and explains the SAT/SMT formulations used to perform these operations.

Simulation

Given a model description, nuXmv is able to simulate possible executions of such model. These executions, or traces, can be generated either automatically or built step by step with the user interaction. The `pick_state` command allows to select the initial state of the trace from the set of all possible initial states of the model. This set can be filtered by specifying additional constraints. The `simulate` command allows to extend existing traces either automatically or interactively. When performed in interactive mode the tool performs a sequence of SAT/SMT calls to generate the list of possible next states.

Given a transition system $M := (S, I, N, T)$ the ith possible initial state $s_i \in S$ is generated by identifying a total assignment that satisfies:

$$I \wedge N \wedge \bigwedge_{j=0}^{i-1} \neg s_j$$

the first part requires an assignment that satisfies the initial conditions I and the invariants N, then the conjunction of the negation of the previously detected assignment rules out all possible initial states that the procedure has already found. If this formula becomes unsatisfiable then all possible initial states have been found.

Similarly when a trace is extended of one step, the new state is selected from a list of possible ones. The ith possible successors of state s is built by identifing a total assignment satisfying:

$$s \wedge T \wedge N' \wedge \bigwedge_{j=0}^{i-1} \neg s'_j$$

where N' and s' refer to the next variables. This formulation requires to find an assignment over the next variables that is reachable in one step from s and satisfies the invariants N. The list of assignments that have already been detected is excluded as in the previous case. If this formula becomes unsatisfiable then all possible next states have been found.

Execution

Given a model description and a trace nuXmv is able to check if such trace belongs to the language of the model.

Let the model be represented as a transition system $M := (S, I, N, T)$ and the trace be a sequence of $k \in \mathbb{N}$ total assignments over S: $\pi := s_1, s_2, \ldots, s_k$. $\pi \in L(M)$ if and only if $s_1 \models I \wedge N$ and $\forall 1 \leq i < k : s_i \wedge s'_{i+1} \models T \wedge N'$, where N' are the invariant conditions expressed over next state variables and similarly s'_i is the ith total assignment over next variables. These conditions encode that the trace must start from a initial state that satisfies the system invariants N and every step of the trace must satisfy the transition constraints T. Moreover the destination configuration of every transition has to satisfy the invariants N. These conditions can be verified by checking the satisfiability of the following formulae:

$$s_1 \wedge I \wedge N$$

and for every $1 \leq i < k$:

$$s_i \wedge s'_{i+1} \wedge T \wedge N'$$

If all these formulae are satisfiable then π is a possible execution of M, otherwise π does not belong the the language of the transition system.

Completion

A partial trace is a trace that does not provide a total assignment for all state: there exist at least one state such that the value of some variable is missing. In this case nuXmv it is able to complete these partial assignments so that the trace represents a valid execution of the model.

Let $M := (S, I, N, T)$ be a transition system representing the model and $\pi := s_1, s_2, \ldots, s_k$ be a partial trace: a sequence of $k \in \mathbb{N}$ partial assignments over S. The objective of the completion procedure is to find a trace $\pi' := s'_1, s'_2, \ldots, s'_k$ such that $\pi' \in L(M)$, each assignment is total and $\forall 1 \leq i \leq k : s'_i \models s_i$. A basic approach to perform this task is to encode the unrolling of $k-1$ transitions and at each step impose the values prescribed by the partial trace. Every model satisfying this formula provides a valid completion of π, if no such model exists then the partial trace can not be completed. Let $s_i[j]$ be the assignment s_i over the variables at step j, similarly $I[j]$, $N[j]$ be the initial and invariant constraint over the variables at step j and $T[j]$ be the transition over the variables at step j and $j+1$.

$$I[0] \wedge \bigwedge_{i=0}^{k-2} (T[i]) \wedge \bigwedge_{i=0}^{k-1} (s_{i+1}[i] \wedge N[i])$$

The first part of this formula encodes every possible path of the transition system of length k, the last part requires the assignment at each step to agree with the ones found in the partial trace. If such formula is satisfied by some model, then this model prescribes total assignments over all variables, from step 0 to step $k-1$. These assignments represent the trace π', which is the completion of π.

Part II

Contribution

The second part of this work describes the actual contributions of this thesis. First, in chapter 5, the new input language is described and its expressiveness is compared to the language of timed automata. Chapter 6 explains the reduction procedure from timed to untimed synchronous infinite transition systems. Chapter 7 provides a more in depth explanation of the newly introduced operators and describes how properties are verified on the timed model. Chapter 8 first shows how traces for the timed model are represented, then describes the three operations that can be performed on such traces: simulation, execution and completion, finally this chapter explains how counter-examples are lifted from the discrete to the timed model. In chapter 9 the work described in this thesis is compared to some tools that implement the main state-of-the-art techniques presented in chapter 3. The first part of this chapter describes the main features of the software used, then each benchmark is described and performance of the tools on that benchmark are presented. The last chapter of this work (10) briefly summarizes the most relevant contributions of this thesis also in comparison with other state-of-the-art software and then highlights some possible directions for further development of what has been presented.

All elements described in this part are completely integrated into the nuXmv flow. The new version of nuXmv, Timed nuXmv, is completely backward compatible with respect to all previously supported elements. Additionally, it enables all the main features available for discrete models also for timed models. Each section explains one of these components in greater detail.

5. Input language extension

The input language of nuXmv, described in 4.1, has been extended to allow the definition of timed transition systems (2.2). This chapter describes in detail the newly supported constructs and the changes made to existing features of the nuXmv language syntax and semantic. Afterwards, an explanation of the syntax and semantic of LTL and a fragment of MTL is provided. The end of the chapter reports some considerations about the expressiveness differences between the Timed nuXmv language and the language of timed automata. This extended language provides full backward compatibility with the one of the previous version of nuXmv.

5.1. Definition of timed transition system

The language allows to specify a timed transition system in which each transition is either **discrete** or a **time elapse**. In discrete transitions the amount of time elapsed is always 0, while in time elapse (or **delta**) transition the time elapse is strictly greater than 0.

Time domain

Timed nuXmv supports dense time semantic: the time domain is in bijection with $\mathbb{R}$. This implies that between two time instants there is an infinite number of steps. At the beginning of a model description

the annotation `@TIME_DOMAIN` allows to enable or disable the timed fragment of the language. The possible values are `continuous` and `none`. This annotation is not mandatory and the default value is `none` which disables all constructs specific to the timed language.

Clocks

nuXmv state variables can be declared of type `clock`. The domain of this new type is $\mathbb{R}$ and variables of this type can be declared only if the time domain is different from `none`.
In delta transitions all clock variables increase their value of the same amount, equal to the amount of time elapsed. Their value in the initial states and in discrete transitions, by default, is unconstrained. This behavior can be overridden by specifying some constraints in the `INIT`, `INVAR`, `TRANS` and `ASSIGN` sections, just like all other nuXmv variables types. The language provides a built-in `clock` variable, accessible through the reserved keyword `time`. It represents the amount of time elapsed from the initial state till now. `time` is initialized to 0 and its value does not change in discrete transitions. While all other `clock` variables can be used in any expression in the model definition, `time` can be used only in comparison with constants. In this work all variables of type different from `clock` will be called discrete variables.

non-continuous variables

In time elapses, by default, all discrete variables keep their assigned values. If the declaration of a state variable of type different from `clock` is followed by the type modifier `noncontinuous` then the value assigned to this variable during delta transitions will be constrained only by its invariants.

Constraints

The following sections explain the changes made on the semantic of `TRANS`, how clock invariants can be expressed and the newly introduced `URGENT` constraints. `INIT` and `ASSIGN` constraints are unchanged with respect to the previous version of the language.

TRANS The syntax of the `TRANS` constraints is unchanged, however in Timed nuXmv their are not evaluated for all transitions but only for the discrete ones. This construct allows to specify "arbitrary" clock constraints. A clock can be reset to any value (not necessarily a constant) or can keep its own value. Like all other nuXmv state variables, if a clock is not constrained during a discrete transition, its next value can be chosen undeterministically among the whole time domain.

***timed* INVAR** Clock variables can be used in an `INVAR` only in a particular structure of the formula: $\varphi \rightarrow \phi$. φ is a formula built using only the discrete variables and ϕ is convex over the clock variables. This closely maps the concept of location invariant described for timed automata (2.2.1): all locations satisfying φ have invariant ϕ.

An `INVAR` must always hold. As in timed automata, since the constraint on clock variables is convex it is sufficient to check that the invariant is satisfied in the starting and ending state of a transition.

URGENT `URGENT` allows to specify a set of locations in which time can not elapse. In particular the keyword `URGENT` is followed by a predicate over the discrete variables. In a model there can be multiple instances of this constructs, they are combined by disjunctions

5.2. Specifications

Timed nuXmv supports invariant, LTL and a MTL fragment checking using BMC or an IC3 extension to deal with LTL specification on timed systems described in [18] based on IC3-IA (described in section 2.5.3). The only clock variable allowed in specifications is the built-in `time`. In both timed and untimed models two new LTL operators have been introduced: `@F~` (at next) and `@O~` (at last). Moreover, in timed models it is possible to use four additional LTL operators: `time_since`, `time_until`, `X~` (timed next) and `Y~` (timed previous).

timed next, timed previous operators

In the Timed nuXmv language next and previous operators come in two possible versions. `X~` and `Y~` allow to predicate about the evolution over time of the system. They are always `FALSE` in discrete steps and hold in time elapses if the argument holds in the open interval immediately after/before the current step. The operators `X` and `Y` can hold only in discrete steps, they retain the semantic they had in the untimed language with respect to the discrete behavior of the model. The disjunction of these operators $X(\varphi) \vee \tilde{X}(\varphi)$ allows to check if the argument φ holds after the current state without distinction between time or discrete evolution.

at next, at last operators

The operators **at next** and **at last**, written $@\tilde{F}$ and $@\tilde{O}$ are binary operators allowed in LTL specifications. Their operands must be formulae without temporal operators over current and next variables. Their second operand must evaluate to either $\top$ or $\bot$ and their overall type is the type of the first operand.

In models with `none` time domain their semantic is defined as the evaluation of their first operand the next [last] time their second operand will hold [held]. If this last condition will never happen [happened] the operator evaluates to a default value.

Formally over a path $\pi := s_0, s_1, \ldots$ $@\tilde{F}$ and $@\tilde{O}$ are defined as:

$$s_i(u@\tilde{F}\varphi) := \begin{cases} \psi \text{ if } \exists j > i : s_j(u) = \psi \wedge (\pi, s_j \models \varphi) \wedge (\forall i < h < j : \pi, s_h \not\models \varphi) \\ def_{u@\tilde{F}\varphi} \text{ otherwise} \end{cases}$$

$$s_i(u@\tilde{O}\varphi_1) := \begin{cases} \psi \text{ if } \exists j < i : s_j(u) = \psi \wedge (\pi, s_j \models \varphi) \wedge (\forall i > h > j : \pi, s_h \not\models \varphi) \\ def_{u@\tilde{O}\varphi} \text{ otherwise} \end{cases}$$

where $s_i(u)$ is the evaluation of the expression u at state $s_i \in \pi$ and $def_{u@\tilde{F}\varphi}$, $def_{u@\tilde{O}\varphi}$ are symbols representing their respective default values.

Their semantic in dense time domain (`continuous`) must be updated since there might not be a single point after which an expression holds: φ might evaluate to $\top$ in an open time interval (a, b) for some $a, b \in \mathbb{R}$. To match more closely their intuitive definition in a dense time domain their semantic on a path π becomes:

$$\pi(t)(u@\tilde{F}\varphi) := \begin{cases} \pi(t')(u) \text{ if } \exists t' > t : \pi, t' \models \varphi \wedge \forall t < t'' < t' \ \pi, t'' \not\models \varphi \\ \pi(t')(u) \text{ if } \exists t' \geq t : \pi, t' \models \tilde{X}\varphi \wedge \forall t < t'' \leq t' \ \pi, t'' \not\models \varphi \\ def_{u@\tilde{F}\varphi} \text{ otherwise} \end{cases}$$

$$\pi(t)(u@\tilde{O}\varphi) := \begin{cases} \pi(t')(u) \text{ if } \exists t' < t : \pi, t' \models \varphi \wedge \forall t' < t'' < t \ \pi, t'' \not\models \varphi \\ \pi(t')(u) \text{ if } \exists t' \leq t : \pi, t' \models \tilde{Y}\varphi \wedge \forall t' < t'' \leq t \ \pi, t'' \not\models \varphi \\ def_{u@\tilde{O}\varphi} \text{ otherwise} \end{cases}$$

where $\pi, t \models \varphi$ means that in path π at time instant t φ holds, $\pi(t)(u)$ is the evaluation of expression u at time instant t on path π and $def_{u@\tilde{F}\varphi}$, $def_{u@\tilde{O}\varphi}$ are symbols representing their respective default values. Notice that the the only difference with respect to the previous formulation is the additional case in which the timed next and previous operators are used. The first case is identical to the untimed case, a different syntax is used since it is no more possible to talk about a discrete sequence of states. In words the new semantic requires the existence of either a point or a left/right open interval in which φ holds. The left open interval is required for the $@\tilde{X}$ operator, the right open interval for the $@\tilde{Y}$ operator. Recall that in discrete transitions the timed next and previous operators are always evaluated to $\bot$.

time since, time until operators

`time_until` and `time_since` are two additional unary operators that can be used in LTL specifications of timed models. Their argument must be a boolean predicate over current and next variables, no temporal operator is allowed. $time\_until(\varphi)$ evaluates to the amount of time elapse required to

reach the next state in which φ holds, while $time\_since(\varphi)$ evaluates to the amount of time elapsed from the last state in which φ held. As for the $@\tilde{F}$ and $@\tilde{O}$ operators if no such state exists they are assigned to a default value.

$$\pi(t)(time\_until(\varphi)) := \begin{cases} \pi(t')(time) - \pi(t)(time) \text{ if } \exists t' > t : \pi, t' \models \varphi \wedge \forall t < t'' < t' \ \pi, t'' \not\models \varphi \\ def_{time\_until(\varphi)} - \pi(t)(time) \text{ otherwise} \end{cases}$$

$$\pi(t)(time\_since(\varphi)) := \begin{cases} \pi(t)(time) - \pi(t')(time) \text{ if } \exists t' < t : \pi, t' \models \varphi \wedge \forall t' < t'' < t \ \pi, t'' \not\models \varphi \\ \pi(t)(time) - def_{time\_since(\varphi)} \text{ otherwise} \end{cases}$$

where $\pi(t)(time)$ is the value assigned to the built-in clock variable $time$ on path π at point t and $def_{time\_until(\varphi)}$, $def_{time\_since(\varphi)}$ are symbols representing the default values.

5.3. Comparison with timed automata

In this section the main differences in the expressiveness of the Timed nuXmv and timed automata languages are highlighted.

From the provided description it should be evident that the language of timed automata is a subset of the Timed nuXmv language. The following considerations show that this inclusion relation is strict. In the language just described it is possible to express any kind of constraint over clock variables in discrete transitions, while in timed automata it is only possible to reset them to 0 in transitions or compare them to constants in guards. This additional expressiveness allows to describe much more complex behaviors with respect to time. However, it makes impossible to build the region abstracted automaton as described in 3.2: clocks might be compared to other not constant symbols. Moreover, the discrete variables of a timed automaton always have finite domain, thus by creating the region abstracted machine it is possible to obtain a finite abstraction of the clock variables and therefore an overall finite abstract transition system. In Timed nuXmv this is not the case since also the discrete variables might have an infinite domain.

From the computability perspective the restrictions on the timed automata ensured the decidability of some key properties of their language. In particular checking whether the language of a timed automaton is empty is a decidable problem [8], however this is not the case for the Timed nuXmv language since it describes an infinite transition systems. The language inclusion problem $L(A) \subseteq L(B)$ is undecidable for non-deterministic timed automata [4]. Therefore this decision problem is undecidable also for the timed nuXmv language, since its language is a superset of the language of timed automata.

6. Timed to Untimed system

The formal verification of a property φ_t on a timed infinite transition system M_t described through the input language of Timed nuXmv is reduced to the verification of a property φ on an infinite state transition system M such that $M_t \models \varphi_t \iff M \models \varphi$. This chapter describes how M is generated from M_t. The Timed nuXmv description of M_f is reduced to a description of model M in the nuXmv input language described in 4.1. Chapter 7 explains how φ is built from φ_t.

6.1. variables

All discrete variables are declared in M and they retain their original type. All clock variables, including `time`, are declared as variables of type `real`. An additional input variable (`__delta`) of type `real` is declared. This input variable defines the amount of time elapse in every delta transition. The symbol `__is_timed_trans` is declared in a `DEFINE` section and it is a flag that allows to decide whether the transition starting from the current state is a time elapse or not. A boolean variable `__iota` is declared, this variables is used to distinguish between singular and open time intervals. States in which `__iota` holds represent singular time intervals, $[a; a], a \in \mathbb{R}_+$, the states in which such variable is assigned to $\perp$ represent open time intervals $(a, b), a, b \in \mathbb{R}_+$. This semantic allows the correct encoding of the LTL temporal operators $\tilde{X}$, $\tilde{Y}$, $@\tilde{F}$, $@\tilde{O}$, *time_since* and *time_until* in the `continuous` time domain.

6.2. INIT

All `INIT` constraints of M_t are copied in M. In the untimed model description the built-in clock variable `time` is initialized to 0 and, since the first time interval must be singular to include 0, `__iota` is initialized to $\top$. Notice that if the initial values of clock variables are not constrained in M_t they will also be unconstrained in M.

6.3. TRANS

The constraints on the transitions of M_t restrict the discrete behavior of the timed transition system. Therefore, in M, they must be considered only on transitions labeled as discrete. This label is provided by the `__is_timed_trans` symbol, in particular all discrete transition have as source a state that assigns this symbol to $\perp$. Let φ be the conjunction of all `TRANS` in M_t, then the transition constraint

$$\texttt{\_\_is\_timed\_trans} \rightarrow (\varphi \wedge \texttt{\_\_delta} = 0 \ \wedge \texttt{\_\_iota} \wedge \texttt{next}(\texttt{\_\_iota}))$$

is added to M. Notice that the predicates over `__iota` forbid M to perform a discrete transition from a state representing an open time interval.

In M the time elapse transitions must be constrained so that all clock variables increase of the same value `__delta` and all discrete variables which are not declared as `noncontinuous` keep the same assignment. This is encoded by the following formula:

$$\begin{aligned}(\neg\texttt{\_\_is\_timed\_trans}) \rightarrow (&\texttt{\_\_delta} > 0 \wedge \\ &(\forall c \in C : \texttt{next}(c) = c + \texttt{\_\_delta}) \wedge \\ &(\forall x \in X \setminus X_{noncontinuous} : \texttt{next}(x) = x))\end{aligned}$$

where C is the set of variables of type `clock`, X is the set of discrete variables and $X_{noncontinuous} \subseteq X$ is the set of discrete variables declared with the `noncontinuous` modifier.

6.4. INVAR

Invariants and "timed" invariants of M_t are added in M as invariants. There is no need to distinguish between time elapses and discrete transitions since in the first case all discrete variables keep the same assignments and in the second case clock updates can not violate the location invariant.

6.5. URGENT

`URGENT` constraints in M_t define a list of sets of states in which time can not elapse. Let φ be the formula representing their union, computed as the disjunction of all such constraints. Since this formula can contain both current and next variables it is encoded in M as a `TRANS`: $\varphi \rightarrow$ `__delta = 0`. This forbids any time elapse transition from states in which φ holds.

7. Timed properties verification

Timed nuXmv supports the verification of invariant, LTL and a MTL fragment. It is possible to use either an implementation of an IC3 version for timed systems, described in [18], or bounded model checking (2.5.3). The property specified on the timed model is transformed into a property on the untimed model such that the first one holds on the timed system if and only if the second one holds on the corresponding discrete model.

7.1. Property rewriting

The formal verification of a property φ_t on a timed infinite transition system M_t described through the input language of Timed nuXmv is reduced to the verification of a property φ on an infinite state transition system M such that $M_t \models \varphi_t \iff M \models \varphi$. This section describes how φ is generated from φ_t. φ is created such that $\varphi := \psi \rightarrow \phi$, where $\phi := \mathcal{D}(\varphi_t)$ and ψ restricts the paths to the ones that follow a specific behavior with respect to `time` and ι.

First the definition of $\mathcal{D}$ for every operator is provided and explained, then the definition of ψ is given.

The function $\mathcal{D}$ is recursively defined on the subformulae of boolean operators:

- $\mathcal{D}(\neg\varphi) = \neg\mathcal{D}(\varphi)$,
- $\mathcal{D}(\varphi_0 \wedge \varphi_1) = \mathcal{D}(\varphi_0) \wedge \mathcal{D}(\varphi_1)$,
- $\mathcal{D}(\varphi_0 \vee \varphi_1) = \mathcal{D}(\varphi_0) \vee \mathcal{D}(\varphi_1)$

next and previous operators

In timed models there is a discrete and a timed version of the operators next and previous. This allows to distinguish between a step representing an evolution with respect to time or a discrete transition.

The LTL operators next and previous in timed domains refer only to the discrete evolution of the model and do not predicate over the temporal behavior.

$$\mathcal{D}(X\varphi) = \iota \wedge X(\iota \wedge \mathcal{D}(\varphi))$$
$$\mathcal{D}(Y\varphi) = \iota \wedge Y(\iota \wedge \mathcal{D}(\varphi))$$

In all time elapses they evaluate to $\bot$, while in discrete steps they evaluate to $\top$ if and only if φ holds in the following/previous state.

If the time domain is different from `none`, it is possible to use a timed version of these operators. They allow to predicate over the behavior of the system with respect to time elapses.

$$\mathcal{D}(\tilde{X}\varphi) = (\neg\iota \wedge \mathcal{D}(\varphi)) \vee X(\neg\iota \wedge \mathcal{D}(\varphi))$$

$$\mathcal{D}(\tilde{Y}\varphi) = (\neg\iota \wedge \mathcal{D}(\varphi)) \vee Y(\neg\iota \wedge \mathcal{D}(\varphi))$$

$\tilde{X}\varphi$ holds at time t if there exists an open time interval with lower bound t, in which φ holds. This semantic definition is encoded into the model by splitting each time elapse into a sequence of singular, open, singular intervals. Singular intervals are closed intervals of the form $[a; a], a \in \mathbb{R}$, while open intervals are $(a, b),\ a, b \in \mathbb{R}$. The intersection of all pairs of intervals is always $\emptyset$ and the union of all of them is the whole time domain ($\mathbb{R}$). In closed intervals ι is assigned to $\top$, while in open intervals $\neg\iota$ holds.

until and since operators

The definitions of U and S LTL operators need to be updated to take into account the possibility of having formulas that hold in an open interval. The until operator is satisfied if the second operand holds in a left open interval of a state in which the first operand holds. Similarly the operator since is satisfied if the second operand holds in a right open interval of a state in which the first operand holds. This can be achieved by exploiting the timed versions of next and previous.

$$\mathcal{D}(\varphi_0 U \varphi_1) = \mathcal{D}(\varphi_0) U (\mathcal{D}(\varphi_1) \vee (\mathcal{D}(\varphi_0) \wedge \mathcal{D}(\tilde{X}\varphi_1)))$$

$$\mathcal{D}(\varphi_0 S \varphi_1) = \mathcal{D}(\varphi_0) S (\mathcal{D}(\varphi_1) \vee (\mathcal{D}(\varphi_0) \wedge \mathcal{D}(\tilde{Y}\varphi_1)))$$

All other LTL temporal operators can be expressed as a combination of U, S and $\neg$.

at next and at last

The operators $@\tilde{F}$ and $@\tilde{O}$ in continuous timed systems must take into account that φ might hold in an open interval and therefore there might not be $t \in \mathbb{Q} : \pi, t \models \varphi$ even if $F\varphi$ holds.

$$\mathcal{D}(u@\tilde{F}\varphi) = \begin{cases} \mathcal{D}(u) & \text{if } \mathcal{D}(\tilde{X}\varphi) \\ \mathcal{D}(u)@\tilde{F}(D(\varphi) \vee X(\neg\iota \wedge \mathcal{D}(\varphi)) & \text{otherwise} \end{cases}$$

$$\mathcal{D}(u@\tilde{O}\varphi) = \begin{cases} \mathcal{D}(u) & \text{if } \mathcal{D}(\tilde{Y}\varphi) \\ \mathcal{D}(u)@\tilde{Y}(D(\varphi) \vee Y(\neg\iota \wedge \mathcal{D}(\varphi)) & \text{otherwise} \end{cases}$$

Notice that these operators in the timed case are rewritten using their semantic in the discrete model. $u@\tilde{F}\varphi$ evaluates to the current value of u if in the open interval following the current state φ holds. Otherwise its value is the one assigned to u the first time that φ holds in either an open or closed interval. The $u@\tilde{O}\varphi$ case is symmetric, it refers to the past instead of the future.

at next, at last in untimed models. In discrete models for each distinct occurrence of these operators a new monitor variable is introduced to represent their value. Let m be a new symbol and $u@\tilde{F}\varphi$ a LTL subformula. The following constraints are added for m:

1. $\varphi \rightarrow m = u$, when the formula holds the monitor variable must correctly represent the value of u.
2. $next(m) \neq m \rightarrow \varphi$, the value assigned to the monitor variable can change only right after φ held.

These two constraints force m to assume the value that u will have the next time that φ will hold, since it can only change after a state that satisfies φ and in that state its value should match the one of u.

Similarly for $u@\tilde{O}\varphi$, it is sufficient to replace the second constraint with $next(m) \neq m \rightarrow next(\varphi)$. This forces the monitor to change value only before a state in which φ holds.

time since and time until

In the input language the built-in variable `time` can only be used in comparison operations. The operators `time_since` and `time_until` allow to express an additional type of formulae.

$$\mathcal{D}(\texttt{time\_since}(\varphi)) = \mathcal{D}(\texttt{time}) - \mathcal{D}(\texttt{time}@\tilde{O}\varphi)$$

$$\mathcal{D}(\texttt{time\_until}(\varphi)) = \mathcal{D}(\texttt{time}@\tilde{F}\varphi) - \mathcal{D}(\texttt{time})$$

These operators allow to predicate over the amount of time that passes between occurrences of φ. Notice that the formulae on the right side are not directly expressible in the input language.

time and iota constraints

This section provides the additional constraints of `time` and ι. $\psi := \psi_\iota \wedge \psi_{\texttt{time}}$, where $\psi_{\texttt{time}}$ forces the uniformity of `time` in open intervals for all predicates and ψ_ι defines the alternation between singular and open intervals.

Let $Sub_{time}(\varphi)$ be the set of atomic predicates containing `time`. The input language restricts such predicates to be in the form $\texttt{time} \bowtie u, u \in \mathbb{R} \wedge \ \bowtie \in \{<, \leq, >, \geq\}$. Then $\psi_{\texttt{time}}$ is defined as:

$$\psi_{\texttt{time}} := \bigwedge_{\texttt{time} \bowtie u \in Sub_{time}(\varphi)} G(\neg\iota \rightarrow (\mathcal{D}(time \leq u) \rightarrow X\mathcal{D}(time \leq u)) \wedge (\mathcal{D}(time \geq u) \rightarrow X\mathcal{D}(time \geq u)))$$

The definition ψ_ι depends on the type of the property, if the property is an invariant (`INVARSPEC`), then it can not contain any temporal operator. This allows to simplify the model behavior by avoiding

all open intervals:

$$\psi_\iota := G\iota$$

If the property is a LTL specification it is defined as follows:

$$\psi_\iota := \iota \wedge G((\iota \wedge \delta = 0 \wedge X(\iota)) \vee (\iota \wedge \delta > 0 \wedge X(\neg\iota)) \vee (\neg\iota \wedge \delta > 0 \wedge X(\iota)))$$

where $\delta := next(\texttt{time}) - \texttt{time}$. This constraint forces the initial value of ι to $\top$: each execution starts from a singular interval. The value assigned to ι can develop in two ways: in discrete steps its value is always $\top$, in time elapses it starts from $\top$ and goes to $\bot$, at this point the system is forced to perform another time elapse to restore the value of ι to $\top$. This formula forces the alternation between singular and open intervals when a LTL specification is checked.

Diverging time

In nuXmv checking a formula φ as `INVARSPEC` has a different semantic from checking $G\varphi$ as `LTLSPEC`. In the latter the tool searches for an infinite lazo-shaped execution in which there exists at least one state where φ does not hold. In the first case, instead, it only checks if a state that satisfies $\neg\varphi$ is reachable from the initial states. nuXmv is able to manage infinite executions that can be represented as lazo-shaped path. Each of these paths can be split into a prefix and a sequence of states that are in a loop. A loop is detected when the system finds two reachable states with the same assignments over all variables. In the models generated by the untiming procedure there is always at least one variable which is monotonically increasing: `time`. This variable in many cases prevents the system from finding such loops. These observations highlight the need of a more general definition of loop. The IC3 extension presented in [18] allows to specify a single symbol that represents `time`. Its value is ignored in the loop detection. The system is constrained to search for loop-backs such that every state in the loop has `time` above the maximum constant to which this variable is compared. This in conjunction with the fact that `time` is monotonically increasing guarantees the loop validity.

Theorem Let π be an finite execution of a timed model M with n transitions and total time elapse t_{last}. Assume that there exists a step $t_{loop} \in \pi$ that agrees with t_{last} on all variable assignments but `time` and this symbol is used only in comparisons with other constants. Let max_{time} be the greatest constant to which `time` is compared and $\forall t_{loop} \leq t \leq t_{last} : \pi(t)(\texttt{time}) > max_{time}$. Then π can be extended to an infinite execution in which `time` diverges.

proof $\pi(t_{loop})$ and $\pi(t_{last})$ are total assignments over all variables (discrete and clocks) that differ only for the value of `time`. Since, by hypothesis, $\pi(t_{last})(\texttt{time}) \geq \pi(t_{loop})(\texttt{time}) > max_{time}$, they agree on the truth assignment of every predicate. For this reason from both states it is possible to perform the same transitions: they satisfy the same conditions. The same reasoning can be repeated for every state after t_{loop}. Therefore the sequence of states from t_{loop} to t_{last} can be repeated infinitely many times for increasing values of `time` without changing the truth value of any predicate. □

time since, time until This proof assumes that `time` is used only in comparison with constants, however the input language of Timed nuXmv also allows to implicitly express variations of time thanks to `time_since` and `time_until`. In order to solve this issue it is possible to consider only the traces in which loop-back states also agree on the value assigned to these operators. This can be achieved by declaring a monitor symbol for each expression made by these operators. This symbol represents the evaluation of that expression. The assignment of this new variable will have to repeat itself in loops. The rewriting of these operators, in fact, introduces a monitor variable, hence the system will detect only loops in which the value assigned to `time_since` and `time_until` does not diverge. This additional assumption restricts the set of executions the system is able to identify. In the following a more precise characterization of the behavior of the values assigned to $time\_since(\varphi)$ and $time\_until(\varphi)$ in lazo-shaped executions is provided. If in the loop states there exists at least one in which φ holds, then the values of these expressions in every configuration of the loop are well-defined and equal to the distance in time from that state. Otherwise the value of $time\_until(\varphi)$ is the default one and for $time\_since(\varphi)$ two additional cases has to be considered. If in the trace prefix there is a state in which φ holds $time\_since(\varphi)$ diverges, otherwise it is evaluated to the default value.

8. Timed traces

This chapter describes how nuXmv traces have been extended to better represent executions of timed transition systems. The first section describes how the expressiveness of the trace representation has been improved. The following sections show how the simulation, execution and completion operations are achieved for the new trace representation. Finally, the procedure to lift the discrete counter-example to a timed one is described.

8.1. Representation

nuXmv traces allow to represent finite or lazo-shaped executions of an infinite discrete transition system. It is not possible to represent any kind of infinite execution that is not lazo-shaped. In these traces the loop-back state is a configuration that agrees with the last state of the trace on the assignment of all variables. This trace definition is not expressive enough to represent infinite executions of timed transition system. Every infinite timed trace, by construction, has at least one diverging variable which is `time`. From these observations the need to devise a more expressive representation becomes evident. The main objective of this section is the description of such new representation.

This issue is a particular instance of a more general problem, which is the representation of non lazo-shaped behaviors in infinite transition systems. For this reason the problem is addressed in the discrete case and then the solution is also adopted in the timed environment. This work proposes a solution which is expressive enough to represent timed traces, but also useful to address some specific cases of the more general instance. The proposed technique does not allow to represent arbitrary infinite behaviors, but only a very specific subset of them.

Discrete Infinite Trace

Infinite traces enrich nuXmv traces with a set of diverging variables. The symbols in this set are allowed to diverge in loops. The notion of loop is redefined so that the assignment of the last and loop-back state must agree on the assignment of all symbols not in the diverging set. Every state in the trace loop associates every diverging symbol to an expression which is function of the previous value of the same symbol. This defines a recurrence relation for each diverging variable, where the base condition is given by the value assigned to these variables in the last state outside of the loop. In this way, given a position inside the loop and an iteration number, the value assigned to all symbols is well-defined. More formally, an infinite trace over a set of non-diverging symbols X and diverging variables D is a finite sequence of assignments $\pi := s_0, s_1, \ldots, s_k$ over $X \cup D$ such that $\exists 0 < l < k \ \forall x \in X : s_l(x) = s_k(x)$. s_l, called loop-back state, agrees with s_k on the assignment of all non diverging variables in X. For all states not in a loop there is no distinction between variables in X and D: all of them are associated to a concrete constant value. Each state in the loop, instead, associates to every diverging variable a function that given the previous value of that variable returns its current value:

$$\forall d \in D, l < j \leq k \ \exists f \in \mathit{type}(d)^2 : s_j(d) = f(d)$$

where $\mathit{type}(d)$ is the domain of the symbol d and $f(x) = y \iff (x, y) \in f$. Notice that these recurrence relations can refer to non-diverging symbols, upon evaluation they are simply replaced with the value assigned to them in the previous step.

It is possible to iterate over infinite traces in two different modes. One mode allows to unroll the infinite execution and builds the concrete assignments for every state by evaluating the recurrence relation of the diverging symbols. The other mode is more complex: it tries to solve the recurrence relation of every diverging symbol to obtain a closed form solution as a function of the iteration number. The closed form of the recurrence relations is exploited to perform operation on infinite traces: simulation, execution and completion. Currently only the very simple case in which the diverging variable is updated by adding a constant value is handled. It could be possible to rely on external procedures to compute the closed form of such recurrence relations, however this solution has not been explored yet. Notice that this component is the only one that makes assumptions on the shape of the recurrence relation.

Timed trace

A timed trace is an infinite trace where the built-in clock variable `time` is always in the set of diverging variables: `time` $\in D$. The recurrence relation for this symbol is identical in every state of the loop and is given by $time_{curr} = time_{prev} + \delta_{prev}$, where δ is the variable representing the amount of time elapse and it is associated to a, possibly different, actual value in every state. Similarly all clock variables that, inside the loop, are never reset are in the diverging set and they are assigned to the same recurrence relation as `time`.

Notice that model checking on infinite transition systems and therefore also on timed transition systems is undecidable in its general formulation. For this reason, while soundness remains a necessary condition, completeness of the proposed techniques can not be achieved. This observation motivates the restrictions made on the traces that can be represented in the system.

8.2. Simulation

This section explains how the formulation presented in 4.3.1 has been modified to implement the simulation functionality for timed models.
The simulation can be performed either automatically or interactively on a timed transition system $M := (S, I, N, T)$. In automatic mode the system tries to build a finite, non-looping, trace of the model of a specified length. This is achieved by appending one configuration at a time. The initial state of the trace is chosen among the ones that satisfy:

$$I \wedge N$$

Every non-initial configuration is retrieved by a SMT call to check the satisfiability of:

$$s \wedge N \wedge T$$

where s encodes the total assignment of the last configuration of the current trace. This is repeated until either a trace of the desired length has been built or the formula becomes unsatisfiable, in which case the simulation has reached a dead-lock state.

In interactive mode first the user has to choose the initial state from a list. The list is computed in the very same way as in nuXmv trance (4.3.1); the ith configuration is computed as the assignment satisfying:

$$I \wedge N \wedge \bigwedge_{j=0}^{i-1} \neg s_j$$

where $s_0, \ldots, s_{i-1}$ are the total assignments that have been already found. Then the user can choose if the system should perform a discrete transition or a time elapse. The configuration to be added to the trace among the states that satisfy:

$$s \wedge \delta > 0 \wedge T \wedge N' \wedge \bigwedge_{j=0}^{i-1} \neg s'_j$$

if a time elapse has been requested,

$$s \wedge \delta = 0 \wedge T \wedge N' \wedge \bigwedge_{j=0}^{i-1} \neg s'_j$$

if the user asked for a discrete transition.

8.3. Execution

This section explains how the formulation presented in 4.3.2 has been modified to implement the execution functionality for infinite, possibly timed, traces.
Traces without loop-back state and configurations that do not belong to the trace loop can be handled

in the same way explained in 4.3.2. In these cases a state and its successor are total assignments that associate to every symbol a concrete constant value. The consistency of the trace initial state can be easily verified by checking the satisfiability of the following formula:

$$s_0 \wedge I \wedge N$$

where s_0 is the first state of the trace. For every configuration $s_i : 0 \leq i < l$, where s_l is the loop-back state or the last configuration of the trace if no loop exists, the formula:

$$s_i \wedge s'_{i+1} \wedge T \wedge N'$$

must be satisfiable.

At this point the transitions between states in the loop and the loop itself must be checked for consistency with the infinite [timed] transition system. The procedure showed in 4.3.3 it is not applicable in this case. The new representation requires to prove that the recurrence relations assigned to the diverging symbols in the loop states allow the transition system to perform infinitely many iterations. Therefore, it is necessary to prove that the loop of the infinite trace is a valid representation of an infinite execution of the infinite [timed] transition system.

Let $\pi := s_0, \ldots, s_l, \ldots, s_k$ be a trace of length k with loop-back state s_l. Assume that each assignment s_j for $l < j \leq k$ associates to every diverging symbol the closed form solution of its recurrence relation as a function of i_{loop}. i_{loop} is a new symbol that represents the loop iteration number. The objective is to prove that for every $i_{loop} \in \mathbb{N}$ each transition in the loop is valid and that it is possible to perform a transition from $s_k^{i_{loop}}$ to $s_{l+1}^{i_{loop}+1}$: from the last configuration of the trace at iteration i_{loop} it is possible to reach the first loop state and begin iteration $i_{loop} + 1$ for every i_{loop}.

$$\forall i_{loop} \in \mathbb{N}\ [(\forall l \leq j < k : (s_j^{i_{loop}} \wedge s_{j+1}^{i_{loop}}) \rightarrow (T \wedge N_{j+1})) \wedge ((s_k^{i_{loop}} \wedge s_{l+1}^{i_{loop}+1}) \rightarrow (T \wedge N_{l+1}))] \tag{8.1}$$

To prove the validity of this formula it is possible to check the unsatisfiability of its negation using an SMT solver. Notice that each conjunct encodes the validity of one of the transitions in the loop, the last cube verifies that it is always possible to begin the next iteration. By negating 8.1 it is possible to obtain:

$$\exists i_{loop} \in \mathbb{N}\ [\exists l \leq j < k : (s_j^{i_{loop}} \wedge s_{j+1}^{i_{loop}} \vee \neg T \vee \neg N_{j+1}) \vee (s_k^{i_{loop}} \wedge s_{l+1}^{i_{loop}+1} \vee \neg T \vee \neg N_{l+1})] \tag{8.2}$$

the loop defined by states $s_l, \ldots, s_k$ is valid in the transition system M if this last SMT formula (8.2) is unsatisfiable. In the case in which a satisfying model is found, such model assigns to i_{loop} a loop iteration number for which it is not possible to perform one of the transitions. Moreover, by asking the SMT solver to prove the unsatisfiability of each disjunct one at a time it is also possible to precisely identify the transition of the loop that violates the behavior prescribed by the transition system.

8.4. Completion

This section explains how the formulation presented in 4.3.3 has been modified to implement the completion functionality for infinite, possibly timed, traces.
The completion is supported only with respect to non diverging symbols; the complete information about the loop and the set of diverging symbols and all their recurrence relations must be provided. As in the execution case (8.3), the same procedure explained in 4.3.3 is applied to perform the completion of the partial assignments that do not belong to the loop.

Consider the partial assignments in the trace loop: $s_l, ldots, s_k$. It is necessary to complete their assignment so that the loop can be repeated infinitely many times. Let $completes(s, s')$ hold if and only if the total assignment s is a completion of the partial assignment s': they agree on all values of non-diverging symbols and assign the same recurrence relations to diverging symbols. Then it is

necessary to find a trace such that:

$$\begin{aligned}\exists s'_l, \ldots, s'_k : \bigwedge_{j=l}^{k} completes(s'_j, s_j) \wedge \\ \forall i_{loop} \in \mathbb{N} \; [\; (\forall l \leq j < k : (s_j'^{i_{loop}} \wedge s_{j+1}'^{i_{loop}}) \rightarrow (T \wedge N_{j+1})) \wedge \\ ((s_k'^{i_{loop}} \wedge s_{l+1}'^{i_{loop}+1}) \rightarrow (T \wedge N_{l+1}))\;]\end{aligned} \tag{8.3}$$

Notice that the second part is exactly the formula 8.1 on the existentially quantified completed assignments. The nesting of the different quantifiers (existential and universal) makes this formulation much more complex with respect to the one obtained in the execution case. In more detail, considering the propositional case, the formulation showed in 8.1 belongs to the complexity class Π_1 = co-NP while formula 8.3 is an instance of Σ_2. In the general SMT formulation their decidability is conditioned to the decidability of the underlying theories.

Given this observation the decision taken in this work is to avoid the complexity of a complete approach in favor of a more tractable, sound but incomplete procedure. In order to do this it is necessary to remove the outer existential quantification. One possibility is to replace the existential quantification with a universal one. This formulation would check that every possible completion of the trace loop is valid, therefore the completion would succeed only in such cases. Another approach is to pick one among the possible completions and check if the resulting completed trace is valid. This can be achieved using the execution procedure described in 8.3. This second procedure has the advantage of being much simpler and correctly handles all cases handled by the first one.

8.5. From discrete to timed counter-example

As described in chapters 6 and 7, in this work the model checking problem on timed transition systems is reduced to the one on discrete infinite transition systems. This reduction implies that the result obtained in the discrete context has to be mapped back into the timed environment. If the property has been verified, then the timed property also holds on the timed transition system. If the specification is violated in the discrete system, the model checking procedure provides a counter-example: a trace of the discrete model in which the property does not hold. This trace has to be mapped into a corresponding execution of the timed system that proves that the timed specification does not hold.

Given an total assignment in the discrete trace, the corresponding total assignment of the timed trace is built by coping the value assigned to all symbols that belong to both discrete and timed transition system. A transition of the discrete trace is mapped into either a discrete or time elapse based on the value assigned to δ in the source configuration. If $\delta = 0$ then the transition is a discrete one, otherwise it is a time elapse. From the property rewriting (7.1) it is possible to notice that there are three possible kinds of time elapses between two configurations s_0 and s_1:

1. $s_0 \models \iota$ and $s_1 \models \iota$: in this case both states represent singular intervals;
2. $s_0 \models \iota$ and $s_1 \models \neg\iota$: in this case s_0 represents a singular interval, while s_1 represents a time instant in an open interval;
3. $s_0 \models \neg\iota$ and $s_1 \models \iota$: in this case s_0 represents a time instant in an open interval, while s_1 represents a singular interval;

In all cases the states represent total assignments over the variables, therefore they can be represented as total assignments also in the timed trace. In the second and third case one of the states is used to represent an open interval and in the trace it splits the left and right neighborhood of the previous and following configurations. In these neighborhoods temporal operators like $\tilde{X}$, $\tilde{Y}$, $@\tilde{F}$ and $@\tilde{O}$ are evaluated. For these reasons all these transitions are represented in the timed trace as a single time elapses from s_0 to s_1.

9. Experimental evaluation

The techniques described so far have been implemented inside the workflow of the nuXmv symbolic model checker. This implementation is compared to other state-of-the-art model checkers for timed transition system, which implement the main approaches described in chapter 3. This chapter first provides a description of the software to which Timed nuXmv is compared. Their descriptions are not exhaustive, the objective is to provide a high level perspective of the features they support and techniques they implement. Then a brief explanation of the testing environment is provided. Finally the benchmarks models and obtained results are presented and commented.

9.1. Description of the tools

This section briefly describes the model checkers Uppaal (9.1.1), ATMOC (9.1.2) and LTSmin (9.1.3). In 9.1.4 the verification algorithms used to evaluate Timed nuXmv are cited.

Uppaal

Uppaal was released for the first time in 1995 and it is actively developed and maintained by the Uppsala University and Aalborg University. Uppaal supports modeling, simulation and verification of networks of timed automata extended with shared integer variables, urgent channels and committed locations [7]. The formal verification of properties is based on the construction of the zone automaton (3.3) corresponding to the composition of the timed automata in the network. DBMs are used to represent and execute transformations on zones [6]. Uppaal supports only bounded variable types and therefore finite timed asynchronous transition systems. The additional constructs: shared integer variables, urgent channels and committed locations provide some synchronization primitives. In particular: shared integer variables allow for asynchronous communication between processes, urgent channels force the system to perform a certain transition as soon as the transition is enabled and committed locations allow to define a sequence of action to be performed atomically without delays or interference of the other processes. The specification language supported by Uppaal is a subset of TCTL [12][13]:

$$\varphi :: AG\psi \mid EG\psi \mid AF\psi \mid EF\psi \mid \psi \dashrightarrow \phi$$

where $\psi \dashrightarrow \phi$ is equivalent to $AG(\phi \rightarrow AF\psi)$ and ψ and ϕ are propositional logic predicates over state variables, locations, shared variables and clock constraints in $\mathcal{B}(X)$.

Uppaal is split into two main components: a graphical user interface that helps in the definition of the timed automata network and a verification engine that given a xml network description and a specification checks if such network satisfies the property. The verification engine searches for a path in the zone automaton that starts from an initial state and ends in a state such that its location is a final one and its clock assignments satisfy the property [8].

Algorithm 1 Timed automata reachability algorithm

```
procedure REACHABLE((l_0, D_0), (l_f, φ_f), k)        ▷ initial and final states, maximal constant
    passed ← ∅                                        ▷ set of explored states
    wait ← {(l_0, D_0)}                               ▷ states to be explored
    while wait ≠ ∅ do                                 ▷ until there is nothing more to visit
        (l, D) ← wait.pop()                           ▷ extract one element from list
        if l = l_f ∧ D ∩ φ_f ≠ ∅ then                 ▷ final states and clocks satisfy the constraints
            return ⊤
        if ∀(l, D') ∈ passed : D ⊈ D' then            ▷ current state not already included by other visited
            passed ← passed ∪ (l, D)                  ▷ update set of visited states
            for all (l', D') : (l, D) ⇝_{k,G} (l', D') do
                wait ← wait ∪ (l', D')                ▷ add set of states reachable from (l, D)
    return ⊥                                          ▷ explored all states, not reached
```

The algorithm 1 is a simplified version of the reachability algorithm implemented in Uppaal [6].

In particular many optimization are omitted, like the usage of a specialized data structure instead of the two sets *passed* and *wait*. Line 8 checks if among the states that have already been visited with location l, there is one that has a clock interpretation that subsumes the current one; in this case it is not necessary to update the list of visited states, since the current state is already represented. In line 10 the algorithm iterates over all states reachable from the current one by performing a transition with delay at most k and subjected to the difference clock constraints G of the automaton and the property. The termination of procedure 1 is guaranteed by the fact that the regions are finitely many and their power set imposes an upper bound on the number of zones. Therefore, $\rightsquigarrow_{k,G}$ is finite and eventually all possible states will be visited.

The description of the Uppaal GUI is out of the scope of this work, for an introduction on its features and usage refer to [7].

ATMOC

The *Aalto Timed Model Checker* (ATMOC) is a symbolic timed model checker developed in 2012 by Roland Kindermann, Tommi Junttila and Ilkka Niemelä. The tool implements the techniques they describe in [25]: an extension of IC3 (2.5.3) and K-induction (2.5.3) to deal with symbolic timed transition systems (STTS). ATMOC supports the verification of invariant specifications on a model description. From the input model ATMOC implicitly generates the symbolic representation of the region abstracted machine (3.2) to obtain a finite state representation of the STTS.

K-induction ATMOC adds an additional constraint on the K-induction paths: no two states of each path can belong to the same region. This is done symbolically by constraining the integral and fractional part of the clock interpretations. In this way the K-induction technique is forced to look only at paths that visit at most one state for each region [25]. The following formula constrains two states indexed by i, j to belong to different regions:

$$\bigvee_{x \in X} x^{[i]} \neq x^{[j]} \vee \tag{9.1a}$$

$$\bigvee_{c \in C} (c_{int}^{[i]} \neq c_{int}^{[j]} \wedge (\neg max_c^{[i]} \vee \neg max_c^{[j]})) \vee \tag{9.1b}$$

$$\bigvee_{c \in C} (\neg max_c^{[i]} \wedge \neg (c_{fract}^{[i]} = 0 \iff c_{fract}^{[j]} = 0)) \vee \tag{9.1c}$$

$$\bigvee_{c \in C} \bigvee_{d \in C \setminus \{c\}} (\neg max_c^{[i]} \wedge \neg max_d^{[i]} \wedge \neg (c_{fract}^{[i]} \leq d_{fract}^{[i]} \iff c_{fract}^{[j]} \leq d_{fract}^{[j]})) \tag{9.1d}$$

where X is the set of discrete (non clock) variables, C is the set of clocks, $c_{int}^{[i]}$ is the integral part of clock c at state i, $c_{fract}^{[i]}$ is the fractional part of clock c at state i, $max_c^{[i]} := c_{int}^{[i]} > m_c \vee (c_{int}^{[i]} = m_c \wedge c_{fract} > 0)$ and m_c is the maximum constant to which c is compared.

This formulation can be more easily understood by looking at the matrix representation of regions shown in figure 3.1. The clause 9.1a encodes that if the two states disagree on the assignment of at least one discrete variable then they belong to different regions. 9.1b ensures that if in both states c is within its maximum relevant value and the integral part is different then they can not be in the same region. The clause 9.1c distinguishes between the regions that assign to c a value in $\mathbb{Z}$ and the ones that have fractional part different from 0. The subformula 9.1d checks that for every pair of distinct clocks they do not belong to the same side of the "diagonals".

IC3 A similar approach is taken to extend IC3 (2.5.3) to handle STTS [25]. As in the previous case the objective is to lift the procedure to reason on the region abstraction. This is achieved by replacing in every SMT call each concrete state with a representation of its region. Given a state s its region

representation $\hat{s}$ is built so that for every state μ in the same region of s: $\mu \models \hat{s}$.

$$\hat{s} := \bigwedge_{x \in X} x = s(x) \wedge \tag{9.2a}$$

$$\bigwedge_{c \in C : s(c) > m_c} c > m_c \wedge \tag{9.2b}$$

$$\bigwedge_{c \in C : s(c) \leq m_c} \begin{cases} c = s(c) & \text{if } fract(s(c)) = 0 \\ c < \lceil s(c) \rceil \wedge c > \lfloor s(c) \rfloor & \text{if } fract(s(c)) \neq 0 \end{cases} \wedge \tag{9.2c}$$

$$\bigwedge_{c \in C : s(c) \leq m_c} \bigwedge_{d \in C \setminus \{c\} : s(d) \leq m_d} \begin{cases} d = fract(s(c)) + \lfloor s(d) \rfloor, & \text{if } fract(s(c)) = fract(s(d)) \\ d > fract(s(c)) + \lfloor s(d) \rfloor, & \text{if } fract(s(c)) < fract(s(d)) \\ c > fract(s(d)) + \lfloor s(c) \rfloor, & \text{if } fract(s(c)) > fract(s(d)) \end{cases} \tag{9.2d}$$

where X is the set of discrete variables, C is the set of clocks, $s(c)$ is the interpretation of variable c at state s, m_c is the maximum value to which the clock variable c is compared and $fract(k) := k - \lfloor k \rfloor$.

Formula 9.2 is built using the same observations done for 9.1. However, the latter encodes the fact that two states belong to different regions while the first one creates a symbolic representation of a region given one of the states in it. As in the previous case the visualization of regions shown in 3.1 might provide a better intuition. The cube 9.2a encodes the fact that all states in the same region must agree on the assignment over the discrete variables, this is the negation of the clause 9.1a. The constraint 9.2b ensures that all states agree on the set of clock variables assigned to values above their maximum relevant value. The cube 9.2c is the negation of clause 9.1b. It encodes the fact that all states in the same region of s have the same integral part and if $s(c) \in \mathbb{Z}$ they have exactly the same value for every clock variable that does not exceed the maximum relevant value. The subformula 9.2d constrains the relationship between the fractional parts of clock assignments for states in the same region. The formulation is similar to clauses 9.1c and 9.1d: the assignment of every pair of clocks lies on, above or below the "diagonal".

In the IC3 version just described the generalization procedure is applied to $\hat{s}$ instead of s and equation 2.2 becomes:

$$F_{i-1} \wedge \neg \hat{s} \wedge T \wedge \hat{s}' \tag{9.3}$$

Further optimization of these techniques is possible by noticing that if a region is unreachable then also all its time-predecessor regions are. This observation allows ATMOC to exclude more regions at a time [25].

In a later work [24] Kindermann, Junttila and Niemelä extend this approach to support also the verification of a fragment of MTL, called $\text{MITL}_{0,+\infty}$, on timed automata using bounded model checking. Timed nuXmv supports the same fragment, their performance are compared on the benchmark used in [24].

LTSmin

LTSmin is a model checker for labeled transition systems developed by the University of Twente. It is characterized by a highly modular structure; there are three main components: a language front-end, a simplification module and an algorithmic back-end [27]. Each of this components relies on a unifying interface called *Partitioned Next-State Interface* (PINS). The language front-end parses the input model and specification from some formal specification language (like the Uppaal language) and encodes it using the structures provided by PINS. The simplification module consists of a set of functions that perform transformations on the PINS structures to obtain an equivalent but more compact representation. The algorithmic backend can be one of some already provided engines or even an external one. They take as input the PINS structure are perform the verification task [11].

A model described using PINS is a *Partitioned Transition System* (PTS) $P := (S_P, \rightarrow_P, \mathbf{s}^0, L)$, where:

- $S_P = S_1 \times S_2 \times \ldots \times S_N$ is the set of states, each state $s \in S_p$ is a $N \in \mathbb{N}$ dimensional vector.

- $\rightarrow_P \bigcup_{i=1}^{K} \rightarrow_i$ is the labeled transition relation, it is composed by K transition groups $\rightarrow_i \subseteq S_P \times \mathcal{A} \times S_p$, where $\mathcal{A}$ is the set of action labels.
- $\mathbf{s}^0 \in S_P$ is the initial state.
- $L : S_P \times \mathcal{L} \mapsto \mathbb{N}$, where $\mathcal{L}$ is the set of state labels, is the state labeling function, let $L(\mathbf{s}) := \{l | L(\mathbf{s}, l) \neq 0\}$.

LTSmin allows to specify dependencies between the transition groups and slots of the state vector. These information allow the tool to extract some properties of the transition groups: *read independence* (every transition in the group is independent to the value of a particular slot), *write independence* (every transition in the group does not modify the value of a particular slot) and *label independence* (the same label is assigned to states that differ for the value of a particular slot). Guards are associated to each transition group and not to single transitions [23].

LTSmin supports the language front-end of Uppaal through Opaal. Given a Uppaal model Opaal generates the corresponding C code that implements the PINS. Opaal creates the zone abstraction of the timed automaton which is encoded into a PTS and passed to the simplification module. This module is able to check for subsumption relation between two states on-the-fly. This allows to prune the PTS and reduce the size of the state space.

LTSmin supports LTL model checking through the LTL layer. This layer computes the Büchi automaton corresponding to the negated formula. Then the product of the model and this automaton is computed on the fly. The resulting machine is given to the analysis algorithms that search for an accepting cycle. If such cycle exists then the original property does not hold, otherwise the language of the product machine is empty and the specification holds.

The tool provides different analysis back-ends and allows to specify different search strategies, some of which run multiple parallel visits. NDFS is the well known nested depth first search algorithm, CNDFS run multiple, synchronized depth first searches in parallel, DFSFIFO combines CNDFS with breadth first search to detect livelocks [27].

Timed nuXmv

Timed nuXmv can exploit all verification engines implemented in nuXmv, thanks to the reduction to discrete infinite model checking described in chapters 6 and 7. The following paragraph reports the verification algorithms exposed by Timed nuXmv interface. For invariant verification it is possible to exploit an implementation of the IC3 extensions presented in paragraphs 2.5.3 and 2.5.3. The second one is the default algorithm and the user can request the first one by specifying an additional command line option that disables the implicit abstraction. For LTL and MTL checking Timed nuXmv allows to choose between two algorithms: one is the extension of IC3 with implicit abstraction (2.5.3) and the other one is based on bounded model checking (2.5.3).

9.2. Benchmarks and results

This section describes the benchmarks used to evaluate the different tools and shows the obtained results in terms of time and space required to perform a single model checking task. Unfortunately, in the literature there are not many publicly available models suitable for timed systems and even less for synchronous systems. In this evaluation two popular benchmarks are used. The first one has been taken from the Uppaal test suite, while the second one has been taken from the ATMOC examples. Timed nuXmv has been run using both algorithms for invariant verification: IC3 with and without implicit abstraction.

Fischer mutual exclusion algorithm

The Fischer mutual exclusion algorithm was first proposed by Michael Fischer. A mutual exclusion algorithm is a procedure that allows two or more processes to coordinate so that they access a shared resource (critical section) one at a time. The fischer mutual exclusion is based on real-time assumptions: every process in the system has access to a synchronized clock. Every process that wants to access the critical section sets a shared variable id to 0, then it waits for at most c time units before setting id to its own pid; at this point the process has to wait at least c time units. If, after this amount of time, the shared variable id is still equal to its own pid, then the process can safely enter in the critical

section, otherwise the procedure is repeated. c is a constant known to every process and should be an upper bound of the execution time between successive steps. The key component of this algorithm are the delays. For this reason this procedure is widely used as a benchmark for timed automata model checking.

Algorithm 2 Fischer mutual exclusion

```
procedure FISCHER(pid, c, id)                    ▷ process id, constant c, shared variable
    loop
        while id ≠ 0 do                          ▷ wait if someone else is trying to access CS
            skip
        x ← random(0, c)                         ▷ random delay in the interval
        wait_at_most(c)                          ▷ wait at most c before going on
        id ← pid                                 ▷ process pid wants to access CS
        wait_at_least(c)                         ▷ wait at least c before proceeding
        if id = pid then
            Critical Section
            id ← 0
```

This simple algorithm has been modeled in the input languages of Timed nuXmv, ATMOC and Uppaal using $c = 2$. The language supported by Opaal is the same as Uppaal. It allows to specify a network of timed automata, therefore it allows to directly model the asynchronous composition of the different processes. The languages of Timed nuXmv and ATMOC are very similar, there are some minor differences on the timed fragment of the syntax. Neither of these two languages provides specialized elements to directly model asynchronous systems. For this reason it has been necessary to embed an explicit scheduler in the model. The scheduler is an unconstrained variable that decides which process is allowed to perform a transition. All four model checkers are asked to prove the mutual exclusion invariant property on systems with an increasing number of processes implementing the Fischer protocol.

In [24] four MTL properties are used as benchmarks. In this work the very same properties are used to compare Timed nuXmv with the tool proposed in that publication. The first specification asks to prove that in the unbounded time interval $[0, \infty)$ if a process asks to access the critical section then it eventually enters the waiting state. The second specification tries to prove that in the unbounded interval $[0, \infty)$ there is no execution that visits infinitely often the critical section and the non-critical section. The third specification is the bounded version of the first one: it tries to prove that the process enters the wait state in at most 2 time units. The fourth and last specification is the bounded version of the second one: it tries to prove that a process enters the critical section in at most 3 time units.

Results

This section shows the result obtained by measuring the performance in the verification of the invariant mutual exclusion property and the four MTL specifications on the Fischer protocol.

Invariant

These tests were executed on a machine running Ubuntu 16.04 and equipped with 16 GB RAM and as processor an Intel Xeon with a base frequency of 3.70 GHz. Each model checking task has been executed with a time limit of 1 hour and 20 GB of maximum virtual memory. A job was stopped as soon as the execution exceeded one of these boundaries. Each model checker was asked to prove the mutual exclusion property for an increasing number of processes: from 2 to 40.

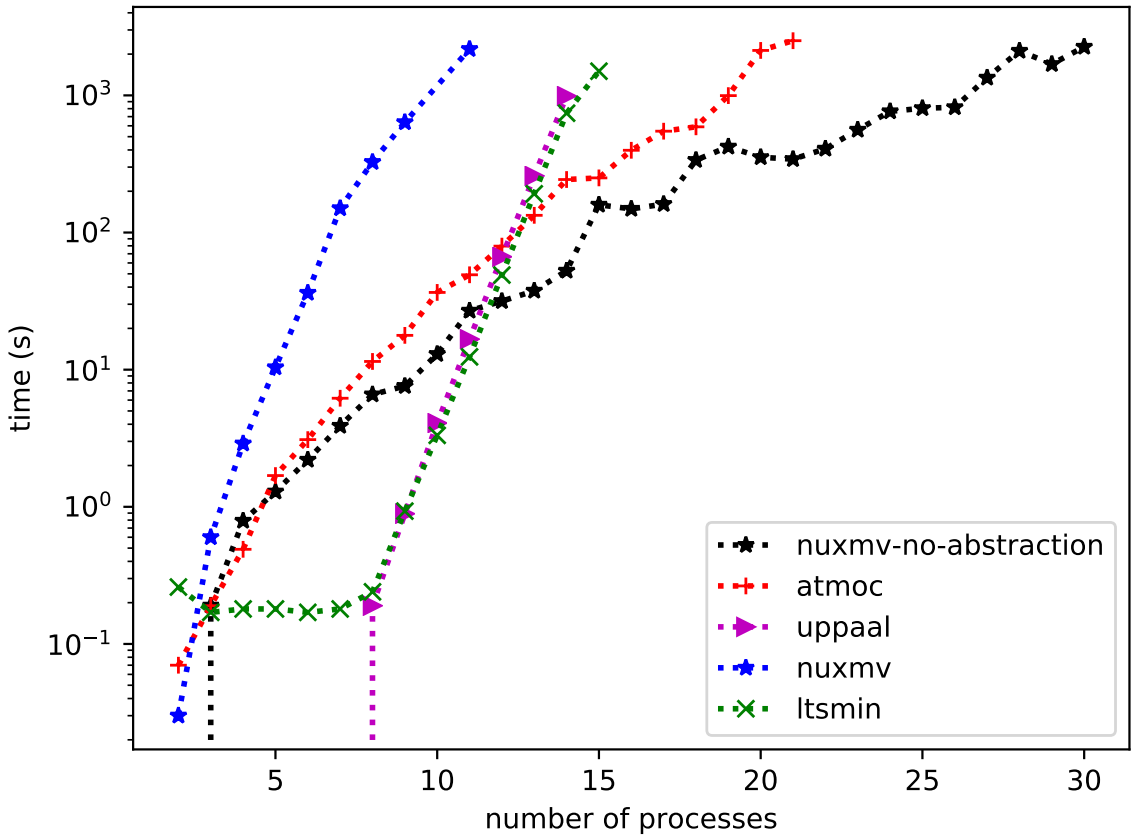


Figure 9.1: Fischer mutual exclusion, runtime

Figure 9.1 shows the CPU time (y-axis) required by the different tools to solve each invariant checking task for increasing number of processes (x-axis). The time is reported in log-scale. Uppaal and LTSmin-Opaal, which are based on zone abstraction, show a similar behavior and they achieve the best results for the smaller models. However, once the size of the model is above 8 processes their running time follows a very steep exponential increment. The default configuration of Timed nuXmv shows poor performance in this benchmark, but with the implicit abstraction disabled it is able to solve the highest number of instances (30 processes) before reaching the time limit. From this plot it is possible to notice that the symbolic techniques require more time with respect to explicit state ones on smaller problem instances, but show a better scalability and eventually achieve better results for large enough models. This behavior might be explained by the exponential growth of the number of states in the explicit approach. In this benchmark the behavior of the discrete component of the model is trivial, therefore one might expect explicit techniques based on zone abstraction to be particularly effective. They seem to require less time up to 13 processes.

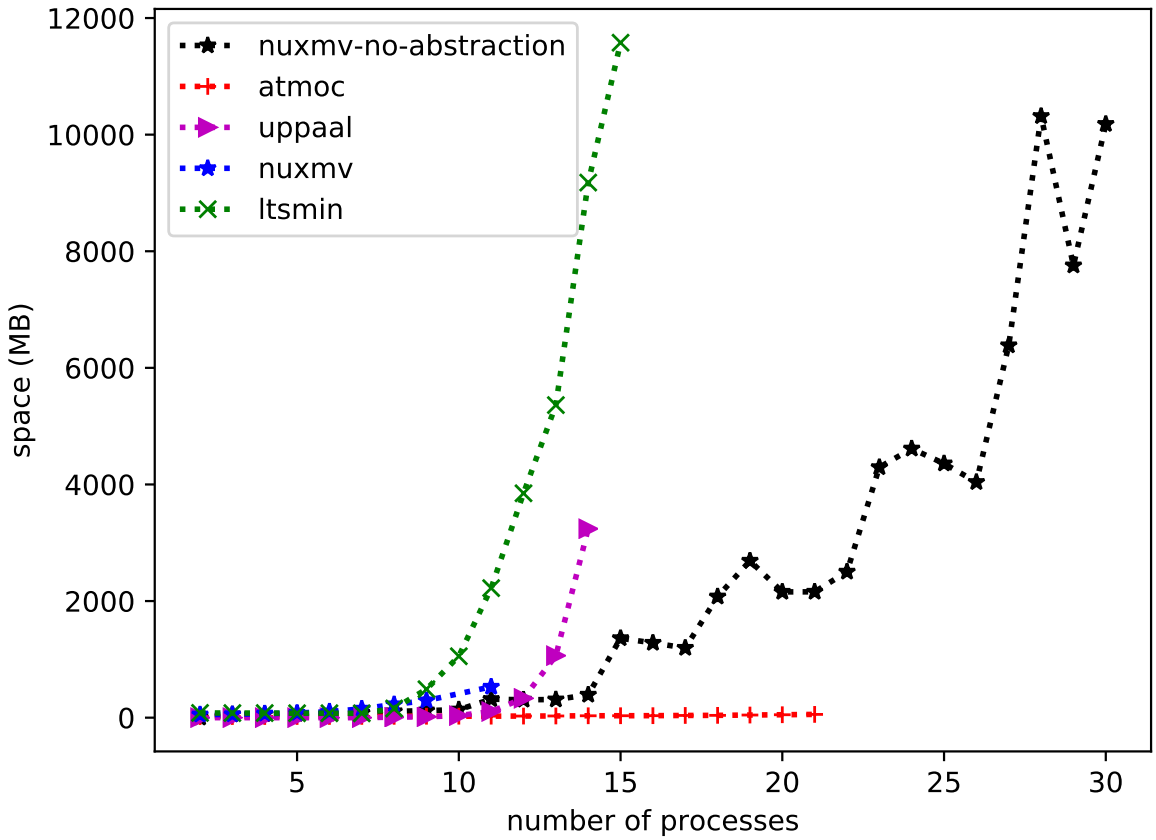


Figure 9.2: Fischer mutual exclusion, memory usage

Figure 9.2 shows the resident memory (y-axis) required by the different tools to solve each invariant checking task for an increasing number of processes (x-axis). All tools require a little amount of memory up to 8 processes. It is relevant to notice that none of the tools gets stopped due to a memory out, but always because of the time limit. As for the runtime, also in this case, the behavior of Uppaal and LTSmin-Opaal are quite similar, with Uppaal being able to handle a few more instances before the exponential increase of memory usage. Timed nuXmv without the implicit abstraction has better memory performance with respect to Uppaal and LTSmin-Opaal for models with more than 12 processes, notice that it also achieves better time performance after 13. ATMOC shows very small memory requirements, with an almost linear increase of about 2 MB for each additional process.

MTL

These tests executed on a machine running Fedora with Scientific Linux 7.3 and equipped with 46 GB RAM and as processor an Intel Xeon with a base frequency of 2.50 GHz. Each model checking task has been executed with a time limit of 1 hour and 40 GB of maximum virtual memory. A job was stopped as soon as the execution exceeded one of these boundaries. Each model checker was asked to prove one MTL property at a time for an increasing number of processes: from 2 to 20.

There are four MTL specifications:

0. The first specification, identified by 0, asks to prove that in the unbounded time interval $[0, \infty)$ if a process asks to access the critical section then it eventually enters the waiting state.

$$G(state = req \rightarrow (F_{[0;\infty)}\ state = wait))$$

1. The second specification, identified by 1, tries to prove that there is no execution in which a

process enters infinitely often the non-critical sections and the critical in the unbounded interval $[0, \infty)$.

$$\neg GF(state = idle \wedge (F_{[0,\infty)}\ state = cs))$$

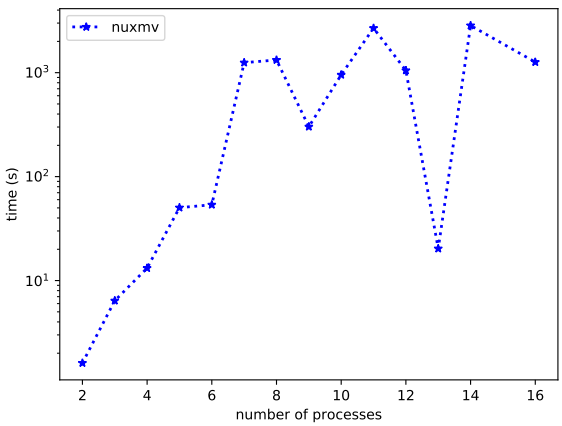
2. The third specification, identified by 2, is the bounded version of the first one: it tries to prove that the process enters the wait state in at most 2 time units.

$$G(state = req \rightarrow (F_{[0;2]}\ state = wait))$$

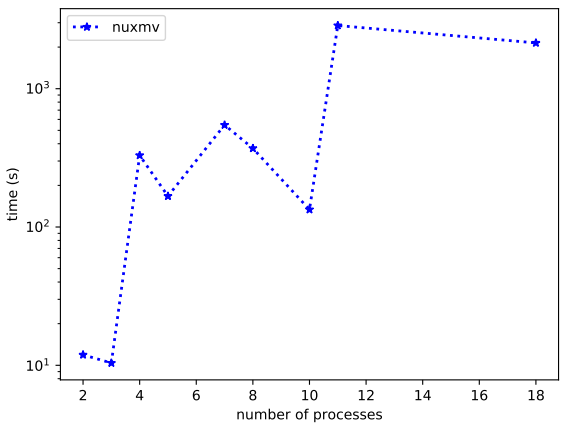
3. The fourth and last specification, identified by 3, is the bounded version of the second one: it tries to prove that a process enters the critical section after at most 3 time units.

$$\neg GF(state = idle \wedge (F_{[0,3]}\ state = cs))$$

Timed nuXmv has been run using both algorithms available for MTL verification: one is based on IC3-IA (2.5.3) and the other based on BMC (2.5.3). Their running time and memory usage are compared with the ones achieved by the software described in [24], called *ATMOC-mtlbmc*.



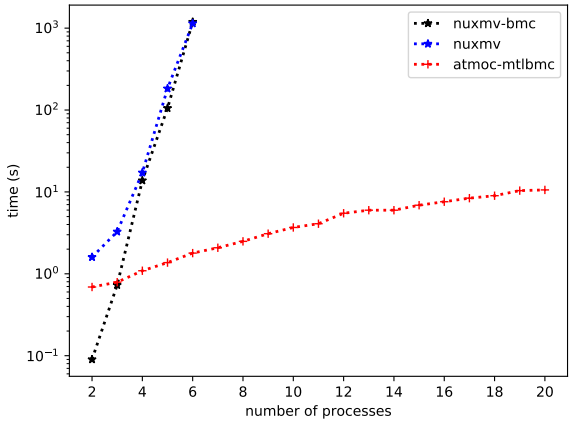
(a) Fischer MTL property 0, runtime



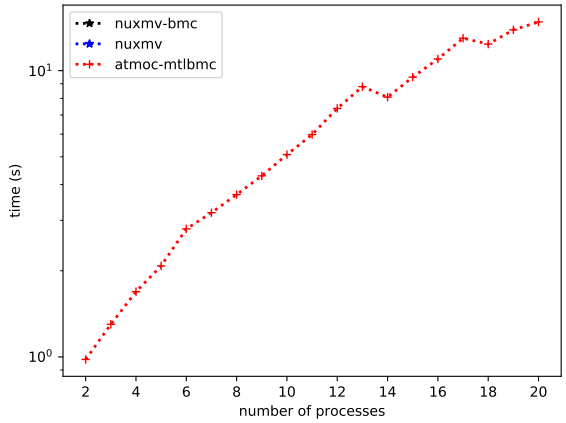
(b) Fischer MTL property 2, runtime

Figure 9.3: Fischer true MTL properties, runtime

The plots in 9.3 show the time required to verify the properties with id 0 and 2. These specifications hold, therefore the BMC (2.5.3) based approaches, nuxmv-bmc and ATMOC-mtlbmc, are unable to terminate. The default algorithm of nuXmv, based on IC3, is able to give an answer. The required time seems to increase as the number of processes increases however their dependency is not as clear as in the invariant checking case. Notice that in figure 9.3b the tool reaches the time limit for models with 6, 9, 13, 14, 15, 16 and 17 processes, while it is able to give an answer within this limit for 18.



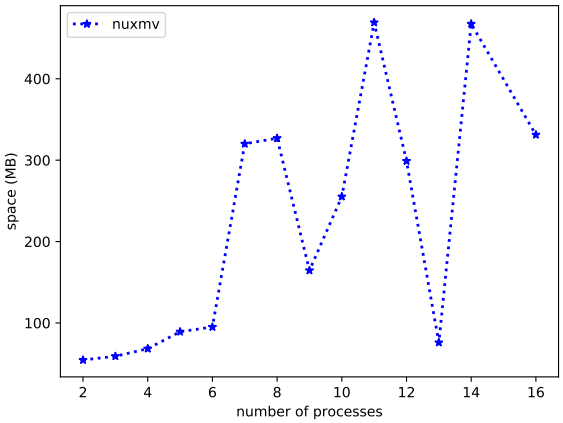
(a) Fischer MTL property 1, runtime



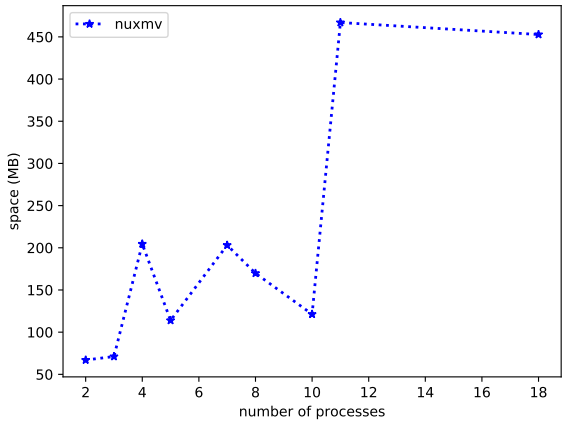
(b) Fischer MTL property 3, runtime

Figure 9.4: Fischer false MTL properties, runtime

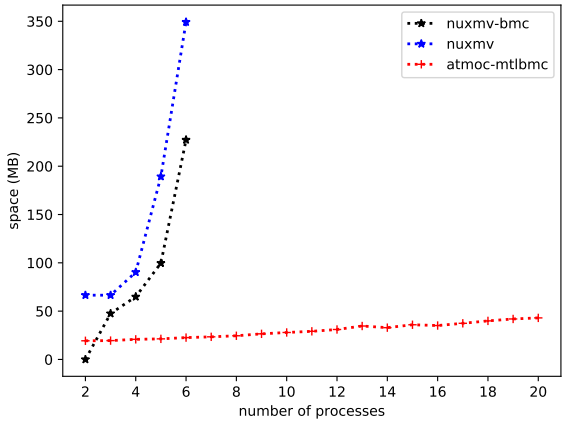
Figure 9.4 shows the time required to find a counter-example for the properties with id 1 and 3. The two algorithms of Timed nuXmv have a very similar behavior, while ATMOC-mtlbmc requires much less time. Timed nuXmv is unable to obtain an answer for property 3 since the rewriting of the property (7.1) introduces some monitors for the `time_since` and `time_until` operators that diverge. This kind of behavior is not recognized at the moment and the procedure never halts (7.1.6). This might also be the cause for the additional time required to identify the counter-example to property 1. Timed nuXmv is unable to consider all that executions, therefore it has to search for longer paths that violate the property. ATMOC-mtlbmc is able to identify these behaviors thanks to the region abstraction, which can not be used by Timed nuXmv due to the additional expressiveness of its input language.



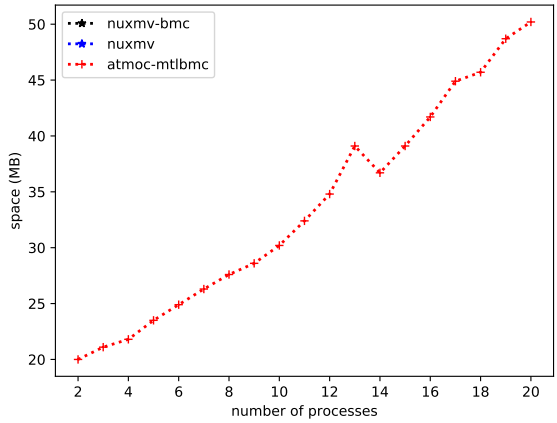
(a) Fischer MTL property 0, memory usage



(b) Fischer MTL property 2, memory usage



(c) Fischer MTL property 1, memory usage

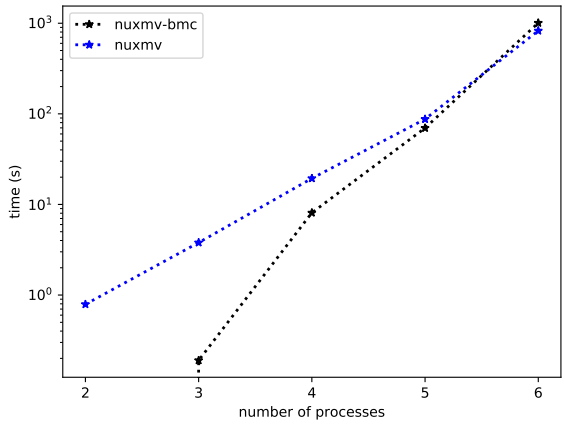


(d) Fischer MTL property 3, memory usage

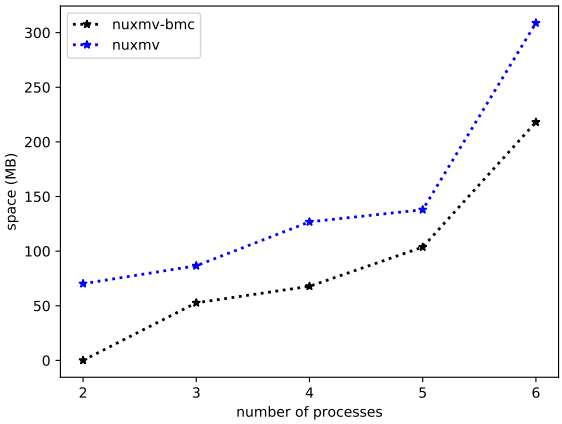
Figure 9.5: Fischer MTL properties, memory usage

Figure 9.5 shows the memory used by the three tools to perform the verification of the four different MTL properties. Only nuXmv is reported for specifications 0 and 2 since it is the only one capable of providing an answer for them. Only ATMOC-mtlbmc is shown for specification 3, since, as explained before, Timed nuXmv is unable to provide an answer in this case. In 9.5c it is possible to see that, as in the invariant case, ATMOC requires much less memory with respect to Timed nuXmv.

Generalized Fischer The Fischer model considered so far embeds a numeric constant (2) that controls how much each process has to wait. However, this protocol should be verified for any such constant greater than 0. This can be encoded in Timed nuXmv models by adding a new infinite state symbol `delay` with type `real` declared as `FROZENVAR` (constant). This symbol is constrained to have value greater than 0. Notice that this simple model can not be expressed as a timed automaton since clocks need to be compared with this new symbol. Therefore ATMOC, ATMOC-bmcmtl, LTSmin and Uppaal input languages are unable to accept such model. However, Timed nuXmv is still able to prove the correctness of the mutual exclusion protocol and perform the MTL checking. The following measurements have been retrieved using 1 hour timeout, a maximum virtual memory of 10 GB on a Ubuntu 16.04 machine with 16 GB of memory and an Intel Xeon with base frequency of 3.70 GHz.



(a) Fischer MTL property 1, time usage

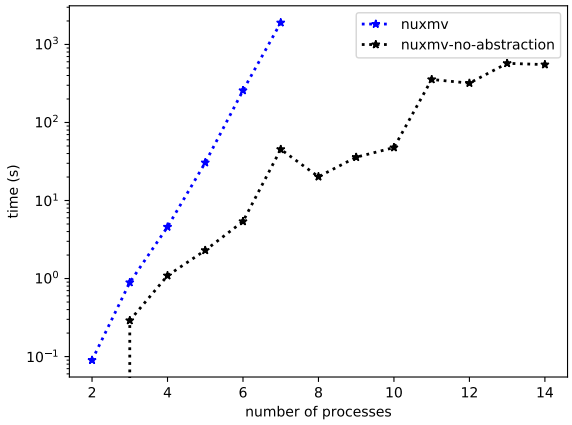


(b) Fischer MTL property 1, memory usage

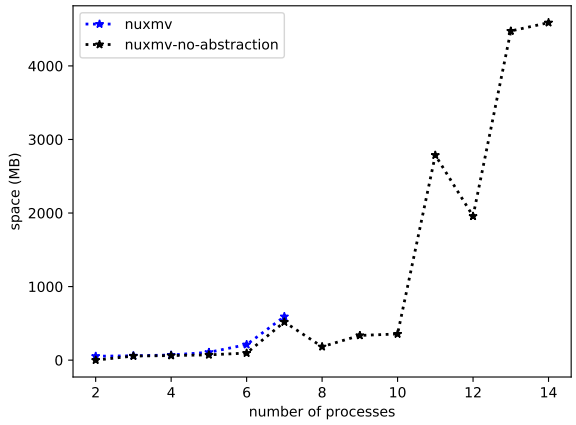
Figure 9.6: Generalized Fischer MTL property

Figure 9.6 shows the time and memory used by the two algorithms available in Timed nuXmv to find the counter-example for property 1. The BMC based procedure seems to perform better for

smaller models and has smaller memory requirements.



(a) Fischer mutual exclusion property, time usage



(b) Fischer mutual exclusion property, memory usage

Figure 9.7: Generalized Fischer mutual exclusion properties

Figure 9.7 shows the time and memory required by Timed nuXmv to verify the mutual exclusion property. It is possible to notice that in this case the time out is reached for 15 processes while in the previous case (9.1, 9.2) Timed nuXmv was able to provide an answer for all model with less than 31 processes.

Diesel generator

The diesel generator benchmark is an *industrial* model of an emergency diesel generator intended for the use in a nuclear power plant. The benchmark comes in three different sizes: small, medium and large. The largest version represents the whole system and has 24 clock variables and 130 state variables. The medium and smaller models represent only a subset of components which are sufficient to prove some of the properties. The medium sized has 7 clocks and 64 variables and the smallest 6 clocks and 36 state variables. These models are not suitable to be represented explicitly, due to the high number of possible states. For this reason only the symbolic techniques have been applied. Each model contains some properties, the tool is asked to verify one property at a time. The specifications are all invariants since ATMOC does not support LTL verification. These models describe synchronous components, therefore no explicit monitor is required.

Results

These tests were executed on a machine running Fedora with Scientific Linux 7.3 16.04 and equipped with 46 GB RAM and as processor an Intel Xeon with a base frequency of 2.50 GHz. Each model checking task has been executed with a time limit of 2 hours and 40 GB of maximum virtual memory. A job was stopped as soon as the execution exceeded one of these boundaries. Each model checker was asked to prove one invariant property at a time for the three different sizes of the model.

property number	ATMOC (s)	nuXmv (s)	nuXmv without abstraction (s)
0	0.5	0.1	0.1
1	0.5	0.0	0.0
2	0.5	0.1	0.1
3	0.5	0.0	0.0
4	0.5	0.2	0.1
5	0.5	0.0	0.0

Table 9.1: diesel generator, small

Table 9.1 shows the time required by ATMOC and the two algorithms implemented in Timed nuXmv to verify 6 different invariant properties on the smallest model of the diesel generator. The time required by both tools to complete all of these task is very small. It might be dominated by bootstrapping tasks rather than the verification procedure.

property number	**ATMOC (s)**	**nuXmv (s)**	**nuXmv without abstraction (s)**
0	1.4	0.4	0.3
1	1.5	0.0	0.0
2	1.4	0.4	0.3
3	1.4	0.0	0.0
4	1.5	0.2	0.2
5	1.5	0.0	0.0
6	1.5	0.7	0.1
7	1.6	0.0	0.0
8	1.5	0.5	0.3
9	1.6	0.0	0.0
10	1.5	0.1	0.1
11	1.6	0.8	0.4
12	1.6	0.0	0.0

Table 9.2: diesel generator, medium

Table 9.2 shows the time required by ATMOC and the two algorithms implemented in Timed nuXmv to verify 13 different invariant properties on the medium sized model of the diesel generator. Timed nuXmv completes this tasks in less than half the time required to ATMOC, moreover by disabling the implicit abstraction it becomes even faster in most of these cases.

property number	**ATMOC (s)**	**nuXmv (s)**	**nuXmv without abstraction (s)**
0	time-out	86.1	34.5
1	4.2	0.1	0.1
2	4.3	1.3	0.8
3	4.2	0.1	0.1
4	time-out	295.9	617.7
5	4.3	0.1	0.1
6	time-out	192.5	168.9
7	4.4	0.1	0.1
8	time-out	592.8	33.4
9	4.3	0.1	0.1
10	4.3	0.1	0.1
11	4.3	2.6	1.2
12	4.4	0.1	0.1
13	3.8	0.1	0.1
14	3.6	0.1	0.1

Table 9.3: diesel generator, large

Table 9.3 shows the time required by ATMOC and the two algorithms implemented in Timed nuXmv to verify 15 different invariant properties on the largest model of the diesel generator. Also in this case Timed nuXmv, in both configurations, appears to be faster than ATMOC. Properties number 4, 6 and 8 seem to be the most difficult to verify. In these cases ATMOC runs out of time and for specifications 4 and 8 there is a huge difference in the running time of the two Timed nuXmv

configurations. The default one is much faster in property 4, while by removing the implicit abstraction the time required to verify property 8 is reduced by one order of magnitude.

These experimental results highlight that the technique designed and implemented in this work is at least competitive with some state-of-the-art tools. It is relevant to notice that most of these comparisons have been performed on systems representable as timed automata, while Timed nuXmv supports the verification of a more general form of timed transition systems. Most of the considered properties were invariant since there is little support for generalized LTL and MTL model checking on timed transition systems. The Fischer benchmarks highlighted that the default configuration of Timed nuXmv sometimes leads to poor performance. In this cases a little exploration of different possible configurations might lead to much better results.

10. Conclusions

This work presented and explained the new features implemented in the nuXmv symbolic model checker. The contribution of this thesis can be summarized in six main different aspects:

1. language definition (chapter 5),
2. model compilation and reduction (chapter 6),
3. verification of properties (chapter 7),
4. system simulation (section 8.2),
5. definition and representation of infinite traces (section 8.1),
6. trace execution and completion (sections 8.3 and 8.4).

Together they led to the development of a new version of the software called Timed nuXmv. This new version is fully backward compatible and extends all high-level functionalities available in nuXmv to the timed case. The tool is now able to handle the formal verification of invariant, LTL and MTL specifications over timed transition systems expressed in an extension of the smv language. The additional expressiveness of the Timed nuXmv input language, with respect to other state-of-the-art tools, does not allow to rely on the region and zone abstraction of the time behavior. However, as shown in chapter 9, the performance in terms of execution time and memory usage are comparable with other model checkers for real-time systems. Among all the tools analyzed, in these benchmarks, Timed nuXmv shows the best scalability in reachability analysis. It has the ability of proving positive results for MTL specifications, however, for negative results, it suffers from the limited capacity to identify and represent diverging variables.

10.1. Future work

This section briefly discusses some possible further developments of the work presented in this thesis: Timed nuXmv. The extensions can be organized in five main possible directions. Each of the following paragraphs analyses one of them.

Language constraints

The language of Timed nuXmv, while more expressive then the one of timed automata, still has some limitations (chapter 5). A possible future development would be to relax some of them. For example: allowing non-convex invariant conditions, the usage of clocks in specification or removing some constraints on the usages of the built-in `time` symbol.

Non-convex invariant conditions would allow to specify more complex relationship between the discrete and time behaviors of the system. Notice that, in this case, showing that the invariant condition holds during the time elapses is not trivial as in the previous case. In fact it is necessary to prove that it holds in every point in time in the continuous interval. In order to relax the constraints over clock variables and time it is necessary to device other ways to restrict the model executions to the ones in which time is allowed to diverge. This might be achievable at level of the problem encoding by modifying the reduction to the discrete case, or at the level of the SMT engine. The first option has the advantage to be potentially independent to the solver used, while in the second case the solver can potentially be more efficient by exploiting the additional knowledge.

Continuous variables

Continuous variables are symbols whose assignment is a function of time. These kind of variables are used to model hybrid systems. An additional layer could be added on top of Timed nuXmv to handle this feature and allow the system to perform model checking on hybrid systems. As for the non-convexity over clock constraints, one of the major issues in this case is to ensure the validity of the invariants during time elapses. The possibility of a unifying approach should be explored.

Timed CTL verification

Timed nuXmv at the moment supports the verification of invariant, LTL and a fragment of MTL specifications. These languages are semantically defined over linear time, adding the support for branching time logics like CTL and TCTL is another interesting direction. These languages would

allow to specify properties that are not otherwise expressible, thanks to the path quantifiers. A similar approach to the one presented in section 7.1 could be used to reduce the CTL verification problem in the timed environment to CTL model checking on an infinite discrete transition system. In the LTL rewriting presented in this work fairness conditions where added to the corresponding discrete specification. This kind of constraints are not expressible in the CTL language, it might be possible to perform a reduction to fair CTL model checking.

Handle more complex infinite traces

The current implementation of Timed nuXmv is able to detect only the simplest infinite behaviors in which the only diverging variable is the built-in `time`. For this reason the user has to pay attention on how the system is described. If the model allows even a single state variable to diverge on at least one execution, then the verification procedures will never halt. As described in section 8.1, this issue can be seen at two different levels. One possibility is to consider the timed transition system level, trying to allow diverging clock variables and `time_until`, `time_since` expressions. Otherwise at the discrete transition system level to handle diverging infinite state variables. Solving the problem in the second context would provide a more general and reusable procedure, however in the first case it could be possible to make additional assumptions on the behavior of these diverging symbols.

Parameter synthesis

Many application domains can be described in terms of parameterized systems, where parameters are variables whose value is invariant. Choosing an appropriate value for these parameters is a widely spread engineering problem, a form of design space exploration where the parameters can represent different designs or deployment decisions. Examples of domains that require the analysis of various solutions include function allocation, automated configuration of communication media, product lines, schedulability analysis, and sensor placement for fault detection and identification. In the context of timed transition systems, parameter synthesis would allow to synthesize guards on clocks such that some property is verified. For example, it could be possible to automatically find the timing constraints of a timed transition system so that it guarantees that two events always happen within a given time interval.

Bibliography

[1] Pentium processors, statistical analysis of floating point flaw. https://web.archive.org/web/20010504180148/http://support.intel.com:80/support/processors/pentium/fdiv/wp/, 1994. [Online; accessed 10-September-2018].

[2] Hardware model checking competition. http://fmv.jku.at/hwmcc/, 2010 - 2014. [Online; accessed 28-August-2018].

[3] Rajeev Alur and David Dill. Automata for modeling real-time systems. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 322–335, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. 01 2008.

[6] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] Armin Biere and Roderick Bloem, editors. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking, 2003.

[11] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 354–359, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Patricia Bouyer. Model-checking timed temporal logics. In Carlos Areces and Stéphane Demri, editors, *Proceedings of the 4th Workshop on Methods for Modalities (M4M-5)*, volume 231 of *Electronic Notes in Theoretical Computer Science*, pages 323–341, Cachan, France, March 2009. Elsevier Science Publishers.

[13] Patricia Bouyer, François Laroussinie, Nicolas Markey, Joël Ouaknine, and James Worrell. Timed temporal logics. In Luca Aceto, Giorgio Bacci, Giovani Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare, editors, *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *Lecture Notes in Computer Science*, pages 211–230. Springer, August 2017.

[14] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[15] B. Bérard and Catherine Dufourd. Timed automata and additive clock constraints. In *Information Processing Letters*, pages 75–1, 2000.

[16] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Biere and Bloem [9], pages 334–342.

[17] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 277–293, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Verifying ltl properties of hybrid systems with k-liveness. In *International Conference on Computer Aided Verification*, pages 424–440. Springer, 2014.

[19] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with ic3 and predicate abstraction. *Formal Methods in System Design*, 49(3):190–218, Dec 2016.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[21] E. Allen Emerson and Joseph Y. Halpern. “sometimes” and “not never” revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[22] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.

[23] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: High-performance language-independent model checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[24] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. Bounded model checking of an mitl fragment for timed automata. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pages 216–225. IEEE, 2013.

[25] Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. Smt-based induction methods for timed systems. *CoRR*, abs/1204.5639, 2012.

[26] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, Nov 1990.

[27] Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-core ltsmin: Marrying modularity and scalability. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 506–511, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[28] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *IEEE computer*, 26(7):18–41, 1993.

[29] Joël Ouaknine and James Worrell. On the decidability of metric temporal logic. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 188–197. IEEE, 2005.

[30] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[31] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.