

Université  
de Toulouse

# THÈSE

En vue de l'obtention du  
**DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE**

Délivré par :

Discipline ou spécialité :

---

Présentée et soutenue par :

le :

Titre :

---

JURY

---

Ecole doctorale :

Unité de recherche :

Directeur(s) de Thèse :

Rapporteurs :

# Verification of Real Time Properties in Fiacre Language

Nouha ABID



◇□ **Her story all over** ◇□

There will be time, there will be time,  
to prepare a face to meet the faces that you meet,  
there will be time to murder and create  
and time for all the works and days of hands,  
that lift and drop a question on your plate,  
time for you and time for me  
and time yet for a hundred indecisions  
and for a hundred visions and revisions,  
before the taking of a toast and tea.

– **T.S. Eliot**–

## Abstract

The formal verification of critical, reactive systems is a very complicated task, especially for non experts. In this work, we more particularly address the problem of *real time systems*, that is in the situation where the correctness of the system depends upon timing constraints, such as the “timeliness” of some interactions. Many solutions have been proposed to ease the specification and the verification of such systems. An interesting approach—that we follow in this thesis—is based on the definition of *specification patterns*, that is sets of general, reusable templates for commonly occurring classes of properties. However, patterns are rarely implemented, in the sense that the designers of specification languages rarely provide an effective verification method for checking a pattern on a system. The most common technique is to rely on a timed extension of a temporal logic to define the semantics of patterns and then to use a model-checker for this logic. However, this approach may be inadequate, in particular if patterns require the use of a logic associated to an undecidable model-checking problem or to an algorithm with a very high practical complexity.

We make several contributions. We propose a complete theoretical framework to specify and check real time properties on the formal model of a system. First, our framework provides a set of real time specification patterns (that may be viewed as a timed extension of Dwyer’s patterns). We provide a verification technique based on the use of observers that has been implemented in a tool for the Fiacre modelling language. Finally, we provide two methods to check the correctness of our verification approach; a “semantics”—theoretical— method as well as a “graphical”—practical— method.

**[Keywords:]** Verification; Requirement; Specification Patterns; Model Checking; Observers; Real Time Systems

# Contents

<b>List of Figures</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 State of the Art</b>	<b>8</b>
2.1 Introduction . . . . .	8
2.2 Modeling of Real Time Systems . . . . .	10
2.3 Specification of Real Time Systems . . . . .	11
2.3.1 Timed Automata . . . . .	11
2.3.2 Time Petri Nets . . . . .	12
2.3.3 Discussion . . . . .	13
2.3.4 PROMELA Language . . . . .	13
2.3.5 Fiacre Language . . . . .	14
2.4 Verification of Real Time Systems . . . . .	15
2.4.1 Static Analysis . . . . .	15
2.4.2 Theorem Proving . . . . .	16
2.4.3 Model Checking . . . . .	17
2.5 Comparison with Related Works and Contributions . . . . .	27
2.5.1 Contributions on Specifying Properties . . . . .	28
2.5.2 Contributions on Verifying Properties . . . . .	29
2.5.3 Contributions on Checking the Correctness of the Model Checker	30
<b>3 Formal Framework</b>	<b>32</b>
3.1 Fiacre Language . . . . .	33
3.1.1 Processes . . . . .	34
3.1.2 Components . . . . .	34

## CONTENTS

---

3.2	Systems Specification . . . . .	35
3.2.1	Time Transition Systems . . . . .	35
3.2.2	Semantics of Time Transition Systems expressed as Timed Traces	39
3.2.3	Composition of Time Transition Systems and Composition of Timed Traces . . . . .	41
3.3	Formal Framework for Expressing Timed Properties . . . . .	46
3.3.1	Metric Temporal Logic . . . . .	46
3.3.2	First Order Formula over Timed Traces . . . . .	47
3.3.3	Timed Graphical Interval Logic . . . . .	48
3.4	Conclusion . . . . .	54
<b>4</b>	<b>Real Time Specification Patterns</b>	<b>55</b>
4.1	Introduction . . . . .	55
4.2	Observable Events . . . . .	56
4.3	Catalogue of Real Time Patterns . . . . .	57
4.3.1	Existence patterns . . . . .	60
4.3.2	Absence patterns . . . . .	61
4.3.3	Response patterns . . . . .	62
4.3.4	Universality patterns . . . . .	63
4.3.5	Precedence patterns . . . . .	64
4.3.6	Pattern Composition . . . . .	64
4.4	Case Study: Rail car System . . . . .	64
4.5	Meta Model . . . . .	70
4.6	Conclusion . . . . .	70
<b>5</b>	<b>Checking Patterns Using TTS Observers</b>	<b>73</b>
5.1	Observers at the TTS level . . . . .	74
5.1.1	Observers for the Leadsto Pattern . . . . .	75
5.2	Experimentation . . . . .	77
5.2.1	Experiment analysis: why Data observer is the best in practice?	81
5.3	Proving the correctness of TTS observers . . . . .	85
5.4	Catalogue of TTS Observers for Patterns Verification . . . . .	89
5.4.1	Composite Patterns . . . . .	91
5.5	Conclusion . . . . .	92

<b>6</b>	<b>Checking Patterns Using Probes and Fiacre Observers</b>	<b>93</b>
6.1	Introduction . . . . .	93
6.2	Probes and Observers in Fiacre . . . . .	95
6.2.1	Probes Syntax . . . . .	95
6.2.2	Observers in Fiacre . . . . .	96
6.3	Catalogue of Fiacre Observers for Patterns Verification . . . . .	96
6.3.1	Existence Patterns . . . . .	97
6.3.2	Absence Patterns . . . . .	101
6.3.3	Response Patterns . . . . .	105
6.4	Proving the correctness of observers . . . . .	109
6.4.1	Universal Program. . . . .	109
6.4.2	Graphical Verification. . . . .	109
6.5	Experimentation . . . . .	112
6.6	Conclusion . . . . .	114
<b>7</b>	<b>Conclusions</b>	<b>115</b>
<b>A</b>	<b>Catalogue of Real Time Patterns</b>	<b>117</b>
A.1	Existence Patterns . . . . .	117
A.2	Absence patterns . . . . .	122
A.3	Response patterns . . . . .	126
A.4	Universality patterns . . . . .	130
<b>B</b>	<b>pFrac: Frac with Real Time Properties</b>	<b>131</b>
B.1	Technical Description . . . . .	131
B.2	Installation . . . . .	132
B.3	Usage . . . . .	132
<b>C</b>	<b>Résumé en Français</b>	<b>137</b>
	<b>Bibliography</b>	<b>146</b>



# List of Figures

2.1	The process of modeling and verification of (Real Time) Systems . . . . .	9
2.2	Timed Automaton Example . . . . .	11
2.3	Time Petri Nets Example . . . . .	12
2.4	The Observer based Approach . . . . .	25
3.1	A double-click example in Fiacre . . . . .	33
3.2	The double-click example in TTS . . . . .	36
4.1	The approach used to name events in Fiacre . . . . .	58
4.2	The Airlock system . . . . .	60
4.3	Rail car System . . . . .	65
4.4	A Rail car system example in Fiacre: The process Terminal . . . . .	66
4.5	A Rail car system example in Fiacre: The process Control . . . . .	67
4.6	A Rail car system in Fiacre: The process Car Panel, Car Door and Passenger . . . . .	68
4.7	A Rail car system in Fiacre: The process Car Handler . . . . .	69
4.8	The Meta Model of Real Time Specification Patterns . . . . .	72
5.1	The global verification tool chain . . . . .	74
5.2	Transition Observer: $O_t$ . . . . .	76
5.3	Data Observer: $O_d$ . . . . .	76
5.4	Place Observer: $O_p$ . . . . .	77
5.5	Compared complexity of the data and place observers (in percentage of system size growth)—average time for invalid properties (above) and valid properties (below). . . . .	79
5.6	Total verification time for APOTA (in seconds) . . . . .	81

## LIST OF FIGURES

---

5.7	A Protocol example in Fiacre . . . . .	82
5.8	State space graph for Protocol example . . . . .	83
5.9	State space graph of invalid properties for Protocol example with data observer . . . . .	83
5.10	State space graph of valid properties for Protocol example with state observer . . . . .	83
5.11	State space graph of invalid properties for Protocol example with state observer . . . . .	84
6.1	A simple observer example . . . . .	96
6.2	State graph for (Universal    Present) . . . . .	111
6.3	Experimentation using Fiacre observer and data observer . . . . .	113
B.1	pFrac usage syntax. . . . .	133
B.2	pFrac options . . . . .	135

## LIST OF FIGURES

---

# Chapter 1

## Introduction

*“To be or not to be.”*

*-William Shakespeare-*

For the last few decades, formal verification of embedded systems has held an important place among the many research fields in computer science. On the one hand, this is motivated by the fact that embedded systems are increasingly present in our everyday life and that they have a major impact on our societies. On the other hand, it is also explained by the fact that it is a very challenging scientific problem, with still a lot of open, interesting problems to be solved.

Broadly speaking, formal verification is a field of computer science concerned with the problem of checking (or proving) that a system or software is correct with respect to the requirements fixed by its designers. Formal verification has attracted a lot of attention in the context of *safety critical systems*—such as those found in the aeronautic or nuclear domains for example—where the term critical is used to emphasize the fact that a failure, a malfunction or a problem in the design of the system can have catastrophic consequences: such as deaths or serious injuries; environmental harm; or high losses. To take a simple reference, in North America alone, losses related to “system crashes” amounts to about 3 billions dollars a year. (The cost of software failures in the UK alone is conservatively estimated at 900 million of dollars per year.) Likewise, airlines estimate that every minute of down-time for a reservation system costs around 70 k\$.

## 1. INTRODUCTION

---

### Some Motivating Examples for Formal Verification

We present, next, a few examples of major software errors to illustrate the very serious consequences which can arise from software errors, and the value to be attached to verification if it is able to reduce the number of such failures (see (Neu94) for a list of incidents):

- The bank of New York disaster: In November 1985, the bank of New York has installed a new version of treasury bond market's deliveries and payments systems software. There was an error in the new version which has blocked the system and ignored all requests. To surpass this mistake, the bank has borrowed 20 billion of dollars.
- Phobos 1: In September 1988, the control centre of the Russian Phobos mission to Mars has moved. He joins the outskirts of Moscow. A program from 20 to 30 pages was sent to Phobos 1 to make some adjustments for the change. After receiving the program, the probe, instead of taking note of the changes, reoriented its solar panels away from the sun. Before that the control centre receives the acknowledgement message that the probe is reoriented and corrects the error, the probe has exhausted his energy. The entire mistake has cost the equivalent of one billion of dollars.
- Ambulance Dispatch System failure: In November 1992, the computerised dispatch system for the London (England) Ambulance Service, broke down suddenly and could not manage calls and ambulance service. As a solution, requests must be managed manually. Luckily, the breakdown occurred in early morning hours, so that there were no human services for patients. As a consequence, there were 46 deaths that would have been avoided if the requested ambulance arrived on time.
- Ariane 5: In June 1996, the first flight of the Ariane V launcher ended in failure. About 40 seconds after ignition, the rocket has been broken and destructed immediately. This accident has cost 370 billion of dollars.

The market demands for more efficient and automated solutions has pushed the complexity of embedded systems to levels never imagined before. For this reason, we

---

need new languages, methods and tools to understand and analyze systems as early as possible during their conceptions. This problem raises the following questions:

1. How to express the requirements of a system?
2. Which properties should be verified?
3. How to check properties?
4. How to verify that the results obtained after verification are correct (meaningful)?  
Or to rephrase it, who checks that the model checker is correct?

## Formal Specification of Requirements

A convincing answer to the third problem has been proposed in the 1980's by Edmund M. Clarke and Allen Emerson (CE82) and by Joseph Sifakis and Jean-Pierre Queille (QS82) with the work on *model checking*. Model checking can be viewed as a set of automated techniques to check whether a (model of a) system behavior meets its requirements by an exhaustive exploration of the model state space. Model checking has attracted a lot of attention from academia as well as industry; mainly because it offers a “push button” solution to the verification of systems. A drawback of this approach is the *state explosion problem*, an expression meaning that the state space built during the system verification may grow exponentially bigger as the complexity of the system increases.

In the context of model checking, requirements are often expressed using *temporal logics*. If we consider the questions listed at the end of the previous section, temporal logics provide a solution to the first and second questions. Actually, an issue limiting the adoption of model-checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the model checking tools. Indeed, there is often a significant gap between the boilerplates used in requirements statements and the low-level formalisms used by model-checkers. This limitation has motivated the definition of dedicated assertion languages for expressing properties at a higher level. However, only a limited number of assertion languages support the definition of timing constraints and even fewer are associated to an automatic verification tool, such as a model-checker.

## 1. INTRODUCTION

---

A main motivation for the work performed during my PhD thesis is to fill some of the gaps left by the existing solutions. The goal of my work is to develop a complete framework to specify and verify properties in the context of *real time critical systems*, that is in the situation where the correctness of the system depends upon timing constraints, such as the “timeliness” of some interactions. For the implementation part of my study, I have worked with Fiacre (BBF<sup>+</sup>08)—a formal language for the specification of real time systems developed in our team—and the Tina toolbox (BRV04), a model-checker for Timed Petri Nets and their extensions. We had multiple objectives:

- to develop a simple method to specify requirements, which can be used by all users especially non experts one;
- to propose an efficient approach, in practice, for the verification of these requirements;
- to define a method to check the correctness of our proposed verification approach.

During my thesis, I have proposed a complete framework that includes: the definition of timed patterns; an approach for checking timed properties; and methods for proving the correctness of our verification approach. Our first contribution is the definition of a set of patterns that extends the specification language of Dwyer (CDHR00, DAC99) with hard, real time constraints. For example, we define a pattern “Present  $A$  after  $B$  within  $[0, 4]$ ” to express that event  $A$  must occur within 4 units of time (u.t.) of the first occurrence of  $B$ , if any. Our main objective is to propose an alternative to timed extensions of temporal logic during model-checking. Our patterns are designed to express general timing constraints commonly found in the analysis of real time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both clarity and computational complexity. In particular, each pattern should correspond to a decidable model-checking problem.

### Brief (Chronological) Description of the Thesis Work

The contribution of this thesis can be divided into three main axes: (1) properties specification; (2) properties verification; and (3) proving the correctness of the verification method.

---

Chronologically, we started our work by studying the set of observable events that can be taken into account in a Fiacre specification: the *observables* of a Fiacre program. Then, we have studied the existing approaches used to specify requirements, with the goal to provide a simple approach to specify properties, targeting specifically non-experts users, and to surpass the existent problems of temporal logics such as decidability or complexity problem.

Our first contribution to the specification of requirements is the definition of a set of real time patterns that can be viewed as a real time extension of the pattern language of Dwyer et al (AZB12b). We give more details on Dwyer’s work in the chapter on related work. (A Recent study (BGPS12) has shown that Dwyer’s patterns are the most used in practice in industry and academia.) We make several contributions. First, we present a simple way to specify requirements, especially for non-experts, since our approach is based on natural language; more precisely *boilerplates*, or semi-structured, English sentences. Next, we propose a decidable verification method for checking real time patterns based on the use of observers together with the verification of simpler LTL formula. Finally, we have defined a Timed Graphical Interval Logic (TGIL) (ADZ12), that is a graphical method for defining the semantics of patterns. The idea is to provide an alternative formal definition of patterns based on TGIL.

We believe that the approach defined during this PhD thesis may ease the work of system engineers that are not well trained for the use of formal verification techniques. Our patterns present a useful and simple framework to specify qualitative as well as quantitative (real time) properties based on a decidable model checking approach. When compared with temporal logics, our catalog of real time patterns is less expressive, but it is enough to express the most used requirements in practice. Indeed, in the case of untimed patterns, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a small number of patterns. Moreover, we are able to implement an automatic verification method, using a less complex algorithm and with an overall better runtime and space complexity than with a model-checker for a “full-fledged” timed temporal logic.

After defining our list of real time patterns, we have studied the verification approaches used in the literature. Our second contribution is to define a verification method for checking patterns on the formal model of a real time system (AZB12a). This method is based on the use of observers and model checking techniques in order to



## 1. INTRODUCTION

---

transform the verification of timed patterns on a system into the verification of simple LTL formula on the composition of the system with an observer. We have chosen an observer-based technique because it is simpler to use and implement; indeed it does not necessitate the introduction of a new formalism (observers are defined in the same language than the system). While the use of observers for model checking is quite common, our contribution is original in one way. Indeed, we define different classes of observers for each pattern and use a pragmatic approach in order to select the most efficient candidate in practice. That is, we follow a pragmatic approach with the goal to provide the best possible observer in practice in terms of verification time and system size growth.

Finally, we have defined a formal framework to verify that our observers are correct and non-intrusive, meaning that they compute the correct answer and have no impact on the system under observation (ADZB12). The formal framework we have defined is not only useful for proving the validity of formal results but also for checking the soundness of optimisation in the implementation. Our framework is composed of a graphical proof and a formal proof. The two methods are complementary. We use the graphical proof method to reason about the correctness of observers at an early stage (this is rather similar to a bootstrapping technique for checking a compiler), while we use a more heavyweight, theoretical proof method for checking the correctness of our best “candidates”.

## Outline

This section gives a brief summary of the contents of the chapters of this document.

**Chapter 2—Related Work:** In this chapter we present the context of this thesis. We briefly present the steps used to verify real time systems (modelling, specification and verification) and then delves into some previous works that are the most relevant for the context of this thesis. We start by presenting related work for modelling of real time systems. Then, we deal with the most significant works for specifying real time systems. Finally, we present the works related to the verification step. We try to stress out, in these sections, the drawbacks of each approach in order to define the gaps presented in existing works. We conclude this chapter with a detailed presentation of the contributions made in this thesis.

---

**Chapter 3—Formal Framework:** This chapter describes the different formal framework used to check the correctness of our verification method. First, we start with a presentation of the Fiacre language and the Time Transition Systems (TTS) model. TTS is an extension of Time Petri Nets with data (shared) variables and priorities. Then we give formal definitions for some of the most important notions used in our approach, that will be used in subsequent chapters to prove the correctness of our verification method. We also define the different formal frameworks used to define the semantics of patterns.

**Chapter 4—Real Time Specification Patterns:** This chapter describes our catalogue of real time specification patterns. We start by defining the set of observable events in the Fiacre language. Then we define our real time specification patterns. We present next a use case to show the use of our patterns in a realistic case. Before concluding the chapter, we present a meta model of our patterns (that is an implementation of the pattern language that is suitable for the use of model-based approaches, such as the Eclipse Modelling Framework).

**Chapter 5—Checking Patterns Using TTS Observers:** This Chapter describes our verification method based on the use of TTS observers. We focus first on a single pattern; then we define for this pattern a set of observers; and finally we show a methodology to try and select the best possible observer in practice. After that, we prove that our observers are correct. Before concluding this chapter, we list for each pattern in our language its corresponding observer.

**Chapter 6—Checking Patterns Using Probes and Fiacre Observers:** This Chapter describes another verification method based on the use of the Fiacre language. More precisely, a new extension to Fiacre that allows the definition of “probes” and “observers” directly in a Fiacre specification. We define, for each pattern, a Fiacre observer. We present, next, the graphical approach used to check the correctness of the proposed observer. Before concluding, we experiment and compare approach based on the TTS observer, on one hand, and the Fiacre observer, on the other hand.

**Chapter 7—Conclusions:** We conclude this document by defining possible lines for future work.

## Chapter 2

# State of the Art

*“There’s always one more bug, even after that one is removed.”*

*-Weinberg-*

The concept of modeling and verifying systems is defined as the act of describing formally the system and checking that it performs its tasks within given constraints. In this chapter, we start by presenting, in a general way, the main steps followed to model and verify systems, especially real time systems. Then, we will concentrate more deeply on each step by presenting the particular related works relevant for the context of this thesis.

This chapter is organized as follow. Section 2.1 gives a general introduction to the subjects that are the basis of this dissertation. We present, in Section 2.2, the work related to the modeling of systems. Section 2.3 presents the work relative to the systems specification, in which we introduce important approaches used to present formally the systems. We present, in Section 2.4, the work related to the verification of systems. We conclude this chapter in Section 2.5, where we list the contributions of this thesis.

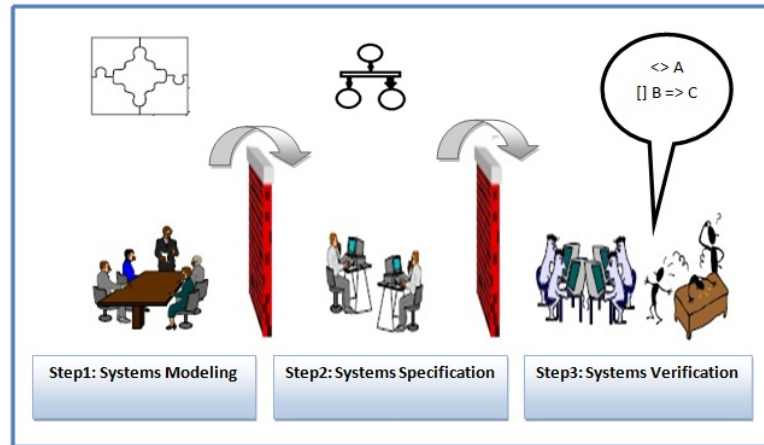
### 2.1 Introduction

Real time systems may be defined as hardware or software systems such that the correct functioning depends on the results produced by the system (its behavior) and the time at which these results are produced (its timeliness). The concept of real time systems

has appeared in the 1970s (NM10). Since then, it has been used in many disciplines, in academia as well as in industry, such as defense and space systems, networked multimedia systems, embedded automotive electronics, ...

Verification of real time systems is mainly concerned with the control and analysis of dynamically evolving systems for which control of timeliness is required. Usually, this is implemented by imposing constraints or deadlines on the termination of certain activities. Since checking real time systems is difficult, it is important to be able to apply formal validation techniques early during the development process and to formally define the requirements that need to be checked.

In this context, many works, see e.g. (MT00, AD90, Pet62, Hol03, BRV<sup>+</sup>03, CE82), have been done to model and verify real time systems. We can define a standard process to model and verify reactive (real time) systems (see Fig. 2.1). The process of modeling and verification of systems is composed of three main steps: *Modeling*, *Specification* and *Verification*.



**Figure 2.1:** The process of modeling and verification of (Real Time) Systems

During the modeling step, the system should be modeled using a *Modeling Language* (Section.2.2). Usually, modeling languages are close to business preoccupations and use a business oriented notation, such as Architecture Descriptions Languages (ADL) or UML. These notations may lack a formal, unambiguous semantics. Hence, in a second step, the system description must generally be specified formally using a *Formal Description Approach* (Section.2.3) such as Time Petri Nets or Timed Automata. The final step is

## 2. STATE OF THE ART

---

the application of the *Verification Method* (Section.2.4) to check properties which are expressed generally using temporal logics or specification patterns (Section.2.4.3.1).

In the following sections, we present in detail the existent works related to the different steps presented above.

### 2.2 Modeling of Real Time Systems

The notion of *Software Architecture* is defined as the composition of the structures of the system (which comprise software elements), the externally visible properties of those elements and the relationships among them. This notion has appeared in the 1990s (NM10) in the purpose of facilitating the representation and the update of systems. Since then, it has been integrated in the process of specification and verification of real time systems. However, the software architecture solutions proposed at this time are not compatible with the domain of real time systems, which exhibit specific issues such as complexity or distributed platforms.

Research work has been done in order to take into account real time systems issues and the notion of *Architecture Description Language (ADL)* has appeared in the 1990s. An ADL is a language used to model systems as a set of software components by showing their connections and their behavior interactions.

Many ADL have been developed during this period: Wright, Darwin, Unicon, Rapide, Aesop, C2 SADL, MetaH, ... (MT00). The main disadvantage of these approaches is that each one operates in isolated manner which makes sharing of architectural description difficult. In order to cope with this problem, the ACME language (GMW97) was developed. However, it only provides a common interchange format for architecture design tools. Nowadays, there are some works aimed at providing a solution to such a problem; for example, the Society of Automotive Engineers (SAE) has recently standardized a “real time ADL” for avionics applications: *AADL (Architecture Analysis and Design Language)*. The AADL standard (FGH06) includes a UML profile useful for modeling systems in avionics, space, automotive, robotics and other real time concurrent processing domains including safety critical applications.

In the context of this thesis, we will sometimes consider real time systems (use cases) obtained from an AADL model. One of the main reason for this choice is that

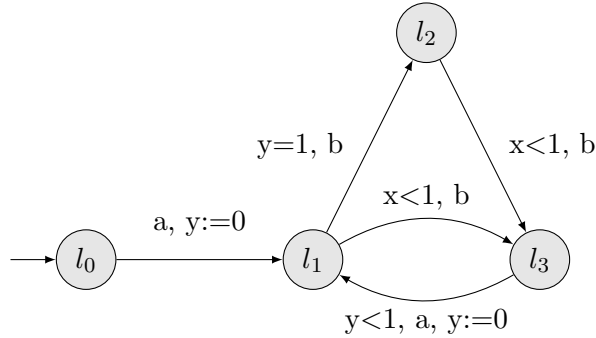
AADL supports the use of various formal approaches—and in particular a plug-in for generating a Fiacre specification from an AADL model (BBDZ<sup>+</sup>10).

## 2.3 Specification of Real Time Systems

Specification of systems is defined as the step of describing formally the required behaviour of the system. Among the techniques proposed to specify systems in which time appears as a parameter, there are two prominent ones: Timed Automata 2.3.1 and Time Petri Nets 2.3.2. The following sections explain in more detail both of these techniques.

### 2.3.1 Timed Automata

Timed automata have been proposed by Alur et Dill in the 1990s (AD90). A timed automaton is a classical finite automaton which can manipulate clocks, evolving continuously and synchronously with absolute time. In a timed automaton, each transition has a guard (a constrain over clock value) which indicates when such transition can be fired and a set of clocks to be reset when the transition is fired (MN08). An example of timed automaton is given in Fig.2.2.



**Figure 2.2:** Timed Automaton Example

Figure 2.2 illustrates the specification of a system that starts from the initial state ( $l_0$ ) with all clocks ( $x$  and  $y$ ) set to zero. From the initial state, there are two possible actions : (1) an action transition (' $a$ ' or ' $b$ ' in our case) that takes into account if the current clock value satisfies the guard; or (2) a delay transition which increase all clocks by the same amount of time.

## 2. STATE OF THE ART

---

A possible execution of the timed automaton in Fig.2.2 is :

$$(l_0, (0, 0)) \xrightarrow{2.67} (l_0, (2.67, 2.67)) \xrightarrow{a} (l_1, (2.67, 0)) \xrightarrow{1} (l_1, (3.67, 1)) \xrightarrow{b} (l_2, (3.67, 1)) \dots$$

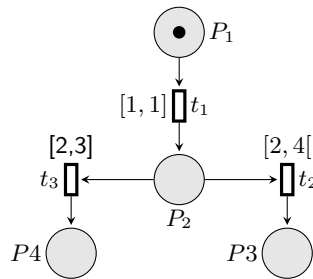
where the pair (3.67,1) represents the value of clocks 'x' and 'y' respectively.

Timed Automata are a good technique to model the behavior of real time systems. There are many system verification tools that are based on timed automata theory (see Sect.2.4.3), but they suffer from a high complexity related to the decidability and accessibility problems (AD94, Rey07).

### 2.3.2 Time Petri Nets

A Petri Net, defined in the 1960s (Pet62), is a graph composed by places, transitions, and arcs. The transitions present events that may occur and places present conditions. The arcs describe which places are preconditions and/or postconditions of which transitions. Among the motivations behind their proposition, we found that they can present easier the behavior of systems than automata approach. Many works have extended Petri Nets to take into account the concept of time.

In (Mer74), the author extend Petri Nets by associating two dates with each transition  $min(t)$  and  $max(t)$ . The difference between these two dates represents the interval in which the transition  $t$  can fire. For example, we suppose that transition  $t$  was fired at  $\gamma$  in the last time then  $t$  can not be enabled before  $\gamma + min(t)$  and can not fire after  $\gamma + max(t)$ . The firing of a transition is immediate. An example of Time Petri Nets is presented in Fig.2.3.



**Figure 2.3:** Time Petri Nets Example

$P_1$  is the initial state of the net presented in Fig.2.3. After 1 u.t,  $t_1$  can be fired and sets a token in place  $P_2$ . After that,  $t_2$  and  $t_3$  are enabled such that  $t_2$  can be fired

between 2 and 4 (opened) u.t, and  $t_3$  can be fired between 2 and 3 u.t.

Time Petri Nets are very useful to model real time systems in which the time of firing a transition is not fixed. Moreover, they allow to represent time out in an easy way.

### 2.3.3 Discussion

We have presented two approaches used to specify real time systems from High Level Models: Timed Automata (Sect.2.3.1) and Time Petri Nets (Sect.2.3.2). In practice, each high level model is transformed into a formal description model using *Timed Automata* or *Time Petri Nets*. However, the process of transformation becomes more and more complicated due to the evolution of high level languages. Some works (BBF<sup>+</sup>08, Hol03) proposed to add intermediate languages between high level models and systems formal models because of two main reasons. First, the use of a formal intermediate modeling language helps to reduce the semantic gap between high-level models and low level models that usually rely on Petri Nets or Timed Automata. Second, the use of a formal language makes it possible to precisely define the semantics of the input language “only once” and to share this work among different verification tool chains. We present next a brief description of existent intermediate languages.

### 2.3.4 PROMELA Language

PROMELA (PROcess MEta LAnguage) is a verification modeling language used to model distributed systems. The design of PROMELA language started in 1979 (Hol03). First, the language was used to support the specification of verification models for PAN (Hol81) (an earliest predecessor of SPIN tool (Hol03)). After that, it has been used as an intermediate language between specification Languages such as RSML and the model checker SPIN (Hol03).

PROMELA was inspired by Dijkstra work (Dij75), Hoare’s CSP language (Hoa78) and the programming language C as first described in Kernighan and Ritchie (KR78).

A specification in PROMELA language represents a set of processes which communicate together. The communication can be synchronous using message channels or asynchronous using variables. Message channels and variables can be declared either globally or locally within a process. An example of PROMELA language is presented in Listing.2.1. The example describes “Hello Word” program modeled in PROMELA



## 2. STATE OF THE ART

---

language. The process is specified using the key word *proctype*. An instance of that proctype will be active in the initial system state due to the presence of *active* keyword. The keyword *printf* is used to print a message.

```
active proctype main()
{
    printf("Hello world")
}
```

**Listing 2.1:** *Hello Word* example in PROMELA language

In the basic PROMELA language there is no mechanism for expressing timed requirements. However, there are some solutions which associate time, presented as *clocks*, to the specification of PROMELA language when verified. Most of the existing solutions assume a concurrency model, which inadvertently excludes the more common method of concurrent process execution by time sharing. Moreover, little can be firmly known about the real time performance of an implementation. It is generally unwise to rely on speculative information, when attempting to establish a system's critical correctness properties.

### 2.3.5 Fiacre Language

Fiacre <sup>1</sup> is a french acronym for an Intermediate Format for Embedded Distributed Components Architectures (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués). It has been developed in the 2000s. Fiacre is a formal specification language to represent both the behavioral and timing aspects of real time systems.

Fiacre was directly inspired by two works, namely V-Cotre (BRV<sup>+</sup>03) and Ntif (GL02), and indirectly by a number of research works in concurrency theory and real time systems theory over the last two decades.

The Fiacre language (BBF<sup>+</sup>08) has been designed in the context of the TOPCASED project (FGC<sup>+</sup>06) to serve as an intermediate format between high level description languages such as UML or AADL and the verification toolboxes such as TINA <sup>2</sup> and CADP <sup>3</sup>.

---

<sup>1</sup><http://projects.laas.fr/fiacre>

<sup>2</sup><http://projects.laas.fr/tina>

<sup>3</sup><http://www.inrialpes.fr/vasy/cadp>

A Fiacre language is transformed into Time Transition System (TTS) model after compilation using the Frac tool (Fiacre language compiler). TTS is an extension of Time Petri Nets taking into account shared variables and priorities. The syntax of Fiacre language and the semantic of TTS will be presented in Chapter 3.

## 2.4 Verification of Real Time Systems

In order to verify a system, we should first define the requirements to be checked. We can classify properties into two main categories:

- *Safety properties* that express that nothing bad ever happens;
- *Liveness properties* that assert that something good eventually happens.

Typically, properties are expressed using temporal logics or patterns. Still to verify these properties!

In the literature, there are three main approaches commonly used to check properties: Static Analysis, Theorem Proving and Model Checking. The difference between these approaches is the level of abstraction used to present the system to be verified. Each one of the approach cited above (Static Analysis, Theorem Proving and Model Checking) is used to verify a certain kind of properties in a system. For example, static Analysis is used to verify Assembler code or Source code, however Model Checking and Theorem Proving are used to verify the Specification.

### 2.4.1 Static Analysis

Static analysis (CC77, Kil73) is the set of techniques allowing to deduce algorithmically a set of properties from the analysis of the source code and/or the assembly code of a software.

Typically, we use static analysis at the compilation step in order to detect common bugs and to optimize the code obtained without influencing the program behavior. The compiler builds an abstract syntax tree from source code representing all possible executions. This structure is easily converted into assembler code later. The compiler checks the properties on this tree and changes it in order to optimize the Assembler code without changing the behavior of the system (the original program and the optimized program must be similar).

## 2. STATE OF THE ART

---

Static analysis allows the verification of this kind of properties: “Is variable *var*, here declared, used later?”, “Does the program tends to access to ‘p+1’ element of a table containing ‘p’ elements?”, “Is the value of variable ‘a’ corrupted by bad memory access?”.

However, this technique has its limitations. As its name indicates, it is a static analysis and not a dynamic one. It means that we can verify only static variables (variables initialized statically in the program). This eliminates the verification of dynamic variables because the static analysis explores all possible executions, and in the case of dynamic variables, the number of possible executions is infinite.

Static analysis can only verify concrete properties on assembler code but it makes inaccessible certain properties representing the algorithm behavior. Currently, there are approaches that combine static analysis and abstract interpretation in order to include some of its dynamic variables in the verification (CC76). But this solution is an approximation which indicates probable problems that must be checked by the programmer himself.

To conclude, only techniques related to high abstraction level like Theorem Proving or Model Checking can verify general properties on the system.

### 2.4.2 Theorem Proving

Theorem Proving (Rus00b) is a set of techniques to deduce, through the use of a proof assistant (PVS (COR<sup>+</sup>95), Coq (HKPM97)), the properties on the behavior of software, algorithm or protocol from the analysis of its mathematical model.

The approach consists, first, of expressing the program and its environment as a mathematical model. This model is obtained either by an automatic processing of source code, or by user interpretation when the program is too complex to be automatically handled. In both cases, it is important to prove that the properties to be verified behave identically in the program as well as in the obtained mathematical model.

The next step is to translate the desired properties into the same formalism as the the program, and then verify them with the help of a proof assistant. The properties are considered as statements of a theorem that we try to prove using the mathematical model of the program and its environment as axiomatics.

$$\textit{Theorem} : \textit{environment} + \textit{program} \models \textit{properties}$$

The proof assistant then provides a number of intermediate lemmas, once they are proved, it certifies that the model is valid. The human operator, usually with the help of the proof assistant, makes the proofs of these lemmas.

Theorem proving approach allows the verification of a large number of properties due to the fact that it is based on human intervention. Trivial steps of this approach are deducted automatically by the proof assistant.

Most mathematical theories used to represent programming languages include inevitably arithmetic. However *Gödel's* theorem states that any mathematical theory containing arithmetic is undecidable. For this reason, theorem proving has two major problems related to undecidability: (1) we are not sure to obtain the result due to the undecidability problem; (2) human intervention is necessary in all most of the cases due to the fact that proof assistant is partially automatic.

Actually, theorem proving is used in parallel with Model checking by automatically generating a finite abstraction of the system to be verified. This method allows to decrease the complexity of the system and to resolve the undecidability problem (Rus00a).

### 2.4.3 Model Checking

Model checking is the set of automated techniques which check whether a system behavior (software, algorithm or protocol) meets its requirements by exploring its representation model (Timed Automaton, Time Petri Nets, etc).

Model checking techniques were first proposed by Edmund M. Clarke and Allen Emerson in (CE82) and Joseph Sifakis and Jean-Pierre Queille in (QS82). Since then, this approach attracts, more and more, the attention of academia as well as industries; mainly because they offer a “push button” solution for the verification of finite systems.

Model checking approach is based on three steps. The first step is aimed at defining a *formal model* of the system to be verified. The structures commonly used to represent the formal model are *Kripke structures* or *Labelled Transitions Systems* (CGP99). A Kripke structure and a Labelled Transitions Systems are composed of a set of states, transitions between states and a function. The only difference between them is that the function associates a set of verified properties to the corresponding state in the case of Kripke structures, however, the function associates actions to each state in the case of Labelled Transitions Systems.

## 2. STATE OF THE ART

---

The second step defines the property to be verified, which is generally expressed using temporal logics.

Finally, the third step is aimed at showing that the defined property is verified by the model. In general, this step is done automatically by the software that combines the model with the property to verify. The requirement satisfies the system if there is a path between the initial state of the system and the set of states which verifies the property. Otherwise, a counter example may be provided in order to reproduce a path leading to a given error.

The advantage of using model checking techniques is their “push button” automatic approach, which does not require hand constructed proofs (Cla08), that can be quite tedious and hard to scale. Moreover, a counter example is provided when the property is not verified and this may help to find the origin of the error.

One disadvantage of model checking is that it suffers from the *state explosion problem*. That is, the number of states that should be inspected can grow exponentially in function of the system complexity and exceed the capacity of available computing resources. However, it is possible to reduce the number of states by choosing an abstraction that preserves the properties to be verified, while reducing the number of states to check.

There are several model checking tools and algorithms that have been developed based on the temporal logics which is used to specify properties, and the approach used to specify the model (Time Petri Nets or Timed Automata). We present in Section 2.4.3.2 the main model checking approaches that have been developed in the literature; but before that, we present an overview of the works related to the definition of properties.

### 2.4.3.1 How to specify properties?

Properties are expressed using two main methods: *temporal logics* and *specification patterns*.

**Temporal Logics.** Since its first definition, given in the 1950s by Arthur Prior (Pri57), temporal logic has been a good candidate to express the behavior of reactive systems. Temporal logic is a special case of modal logics that refers to the succession of events along time using time operators such as “eventually” (“globally”) or “always”. Modal

logic extends propositional logic (logics based on propositions) to include operators that define modality.

In the context of this thesis, we are interested in temporal logics as a mean for specifying properties. In the following paragraphs, we present the most important temporal logics used to specify requirements. In order to compare the presented temporal logics, we utilize two criteria: *expressiveness* and *decidability*. Both of them help us to measure the accuracy of a temporal logic when it is used as a specification formalism :

- **Expressiveness** : *Which properties can be specified?* The expressiveness of a logic is measured as the set of real time properties that can be defined by the temporal logic. We compared if the temporal logic takes into account the metric of time, which determines the capability of expressing temporal constraints in a quantitative form such as the duration between two events. We have named quantitative properties to the properties that support such expressiveness. We have also compared if the logic takes into account qualitative properties in order to express temporal-order relations between events.
- **Decidability** : *How difficult is to verify the defined properties?* The decidability of a logic is related to the existence of an effective method to determine if the property is valid.

The order in which the temporal logics are presented is quite close to its chronological apparition, from the earliest to the latest.

**Linear Temporal Logic (LTL)** was first introduced by Pnueli in 1977 (Pnu77). It is a propositional logic. Formulas in propositional logic are built on the basis of a set of elementary facts (i.e., atomic formulas) by using a set of logic operators: negation ( $\neg$ ), conjunction ( $\wedge$ ) and union ( $\vee$ ).

LTL is composed of five basic temporal operators used to describe properties of a path through the tree; the  $X$  operator means “next time” and requires that the property holds in the second state of the path; the  $F$  ( $G$ ) means “eventually” (“always”) and requires that the property holds at some (at every) state on the path; the  $U$  means “until” and combines two properties, it holds if there is a state on the path where the

## 2. STATE OF THE ART

---

second property is true and the first property holds for every preceding state on the path; the  $R$  means “release” and it is the logical dual of the “ $U$ ” operator.

Linear Temporal Logic is used to express properties in which we can not specify temporal constraints (qualitative properties). It has been proved that LTL is decidable.

**Interval Logic (IL)** was presented by Schwartz et al. (SMSV83) in 1983. The logic is based on time interval and propositional logic. A typical Interval Logic formula  $\phi$  is in the following form  $[I]\phi$  where  $I$  is the interval that is the context in which formula  $\phi$  has to be verified. This formula is satisfied if the next time the interval  $I$  can be built, formula  $\phi$  will hold in it or if  $I$  can not be found.

Interval logic is based on three operators: eventually ( $\Diamond$ ), globally ( $\Box$ ) and negation ( $\neg$ ); The  $[I]\Diamond\phi$  means that  $\phi$  can be true in  $I$ ; The  $[I]\Box\phi$  means that  $\phi$  is always true in  $I$ ; The  $[I]\neg\phi$  means that  $\phi$  is not verified in  $I$ .

Time intervals are bounded by events and by the changes in the system state, described by the formulas. The interval bounds can be defined by the occurrence of events. Finally, in order to describe system behavior, operators *at*, *in*, and *after* have been defined. These specify the truth *at the start*, *during*, and *at the end* of the interval, respectively.

Interval Logic does not present an explicit metric for time, which means that there is no possibility of expressing temporal constraints. Thus, we can only reason about order relations between events (qualitative properties). The decidability of Interval Logic is not studied because the logic was introduced as a specification language and is verified by means of automatic instruments, without taking into consideration the possibility of simulating or executing the specifications (BMN00).

**Computational Tree Logic (CTL)** was presented by Clarke et al. (CES86). CTL is a propositional branching time temporal logic. CTL formulas have the form  $QL$  where  $Q$  stands for one of the path quantifiers  $A$  or  $E$  and  $L$  for the linear-time operators (the same supported by LTL). CTL formulas are constructed as follows:

$$\phi ::= p | \neg\phi | \phi_1 \wedge \phi_2 | X(\phi) | AU(\phi, \psi) | EU(\phi, \psi) | AF(\phi) | EF(\phi) | AG(\phi) | EG(\phi)$$

With CTL, we can only reason about the order-relations between propositions (qualitative properties). CTL is decidable.

**Real Time Logic (RTL)** was presented by Jahanian and Mok (JM86). RTL is a logic that extends first-order logic with a set of elements for the specification of real time systems requirements. RTL proposes a logic approach for the specification of real time systems, but it is not a temporal logic in the classical sense. real time logic is based on an absolute clock to measure time progression. The value of this clock can be referenced in the formulas. Function “@” permits one to assign a temporal value (execution instant) to an event occurrence.

We can specify ordering and quantitative time constraints with RTL since it is possible to control time with the clock variable, however, the problem with RTL is that formulas are complicated when verifying systems related to absolute time. Moreover, It has been proved that RTL is undecidable (AH90).

**Timed Proposition Temporal Logic (TPTL)**. Alur and Henzinger (AD94) presented Timed Propositional Temporal Logic (TPTL), an extension of Propositional Temporal Logic (PTL) (Pnu77).

TPTL is based on both *next* and *until* operators of LTL. Nonetheless, TPTL includes some other operators : the *eventually* and *always* operators which assert that a proposition may be satisfied and always satisfied respectively. The interesting point of TPTL is the use of *freeze* operator. A variable  $x$  can be bounded by a freeze quantifier “ $x$ ”, which “freezes”  $x$  to the time of the local temporal context. For example, formula  $\Box x.(p \rightarrow x \leq 10)$  means that whenever  $p$  is satisfied, the time should be less than 10. The adoption of *freeze* operator can be very interesting to model systems in which more than a real time clock is present.

With TPTL , we can specify quantitative temporal constraints. In (AH90), the authors proved that the choice of natural numbers for temporal domains is essential to obtain a temporal logic for which the satisfiability problem is decidable. However, when using complex temporal domains, different from natural numbers, the satisfiability problem is undecidable.

**Metric Temporal Logic (MTL)** was presented by Koymans (Koy90). MTL extends FOL with temporal operators from modal logic: G “It will always be the case that ...”, F “It will at some time be the case that ...”, H “It has always been the case that ...”, P “It has at some time been the case that ...”. MTL includes a metric for time, for this reason we can specify quantitative and qualitative temporal constraints. MTL



## 2. STATE OF THE ART

---

is undecidable but a deductive system is available. We give the definition of MTL in details in Chapter 3 Section 3.3.

To conclude, we present in Table 2.1 a summary of the presented logics based on the two characteristics presented above.

Logics	Expressiveness (qualitative or quantitative properties)	Decidability problem
LTL logic	qualitative properties	decidable
IL logic	qualitative properties	decidable
CTL logic	qualitative properties	decidable
RTL logic	qualitative and quantitative properties	undecidable
TPTL logic	qualitative and quantitative properties	decidable in the general cases
MTL logic	qualitative and quantitative properties	undecidable in the general cases

**Table 2.1:** Comparison between existent temporal logics

The other approach for specifying properties is using specification patterns. The following paragraph presents in more detail such approach.

**Specification Patterns.** Based on software engineering, a pattern is a general reusable solution to a commonly occurring problem within a given context.

Specification patterns are considered as an alternative way to specify properties. The advantage of using this kind of approach is its simplicity, especially for non-experts users, since it is based on natural language.

Dwyer et al. (DAC99) have defined a set of specification patterns. In this seminal work, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a small number of “pattern formulas”. Then, he classifies his patterns into categories like absence and existence. For each category, every pattern can be specified using a scope (before, after, etc). The authors give a definition of their patterns using different logical frameworks (LTL, CTL, Quantified Regular Expressions, etc.). As a consequence, they do not need to consider the problem of checking requirements as they can readily rely on existing model checkers. Dwyer’s patterns language is still supported, with several tools, an on line repository of examples (ksu) and the definition of the Bandera Specification Language (CDHR00) that provides a structured-English language front-end. A recent study by Bianculli

et al. (BGPS12) shows the relevance of this pattern-based approach in an industrial context. An example of Dwyer's patterns is the pattern "*present evt1*" such that *evt1* and *evt2* are the observed events. This pattern is used to describe a portion of a system's execution that contains an instance of event *evt1*. This pattern can be used to verify that on all the executions of the system we eventually reach a terminal state ("*present TerminalState*").

Dhaussy et al. (DPC<sup>+</sup>09) extend Dwyer's patterns in order to take into account real time constraints. They classify their patterns into five categories and enrich them using some options like *Immediacy*, *Precedence*, *Repeatability*. Guards based on the occurrences of events can be added in the patterns definition. They define also the possible order of events (ordered, not ordered).

Another related work based on Dwyer's patterns is (GL06), where the authors present their patterns and a verification approach based on timed automata. They are based only on safety properties. Their systems give the possibility to define composed events. As for *scopes*, the authors support the possibility to express that a property holds before, after and until a certain duration since the last occurrence of a certain events.

Konrad et al. (KC05) also extend Dwyer's patterns and define three classes of patterns (duration, periodic and real time order) collected from the analysis of 80 real time system requirements. They give a mapping from timed pattern to TCTL and MTL logic. Moreover, to facilitate the understanding of their patterns, they offer a structured english grammar that includes support for real time properties.

In (Bit01), the authors have proposed patterns that are also based on Dwyer's patterns but they are focused on safety properties. Their patterns are classified into two categories. Each category is refined by subclasses.

The work presented in (CP99) extends Dwyer's patterns by supporting events in LTL formula. They define the notion of *edge* used to define events in LTL logic and add this notion to patterns. In (SACO02), the PROPEL approach is based on a "disciplined" natural language and finite state automata and it is used to represent fine-tuned Dwyer's patterns. In (Gru08), the authors present patterns related to probabilistic quality property. The satisfaction of patterns in this case is related to a given probability. They are based also on structured english grammar to define patterns.

## 2. STATE OF THE ART

---

We have presented the approaches used to specify properties and next we present the work related to the properties verification.

### 2.4.3.2 Existing Model Checking Approaches

Many formalisms have been developed to verify properties. In general, they repose on temporal logics to specify properties.

In (BRV04), the authors present TINA (Time Petri Net Analyzer) toolbox. TINA is a software environment allowing the manipulation and analysis of Petri Nets and Time Petri Nets. Tina offers various abstract state space constructions that preserve specific classes of properties of the concrete state spaces of the nets. These classes of properties may be general properties (reachability properties, deadlock freeness, liveness) or specific properties. TINA is based on SE-LTL logic (CCO<sup>+</sup>04) to express specific properties on the system. SE-LTL extends LTL logic to manipulate proposition based on states and transitions. To verify properties expressed in SE-LTL logic, TINA uses SELT, (SE-LTL) model checker. First, SELT generates a Büchi automata accepting the words which do not satisfy the property and then it composes the state space of the system (generated by TINA) with the automata and verifies if there is a strongly connected component including an accepting state of büchi automata. If such component is not found then the property is satisfied, otherwise SELT generates a counter-example with the infinite sequence states containing that state.

While TINA is based on Time Petri Nets, UPPAAL (BDL04) is based on timed automata. UPPAAL is a toolbox for verification of real time systems. The tool is designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronisation. Properties in UPPAAL are specified using a subset of TCTL (Timed Computation Tree Logic) (Kat99). The query language consists of state formula or path formula. State formula describe individual states, whereas path formula quantify over paths or traces of the model. Path formula can be classified into reachability, safety and liveness. Each formula to be verified is transformed into a timed automata and composed with the system. If the property is not verified a counter-example is generated.

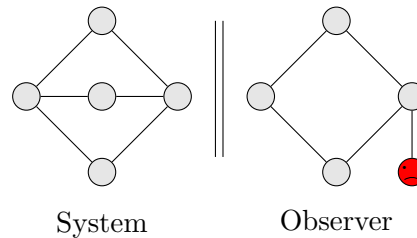
KRONOS (Yov97) is a verification tool for real time systems. Like UPPAAL, KRONOS is based on timed automata and TCTL logic. The model checking algorithm

is based on a symbolic representation of the infinite state space by sets of linear constraints.

SPIN (Hol03) is a verification system for models of distributed software systems. The input of SPIN is modeled using PROMELA language. SPIN can verify properties expressed using LTL logic supporting all correctness requirements expressible in Linear Time Temporal Logic but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as LTL requirements, as formal Büchi automata, or more broadly as general omega-regular properties in the syntax of “never claims”, which are used in the context of SPIN tool to specify behavior that should never happen using a series of propositions or boolean expressions on the system state.

Roméo (GLMR05) is a verification software for Time Petri Nets (TPN). The input of Roméo is a TPN model ou an extension of TPN. The tool can verify properties using TCTL logic.

**Observers Approaches.** Usually, properties are expressed using temporal logics (see Section. 2.4.3.1) and verified with the suitable model checker (see Section 2.4.3.2). However, some properties require the use of complex temporal logics. For this reason, many works are based on the programming of a suitable *observer*. Intuitively, an observer is a synchronous program, representing the property to be verified and composed with the system to check. The role of an observer is to “observe” the behaviour of the system during its execution and produce an output which remains true if the property is satisfied and false otherwise (see Fig.2.4).



**Figure 2.4:** The Observer based Approach

## 2. STATE OF THE ART

---

An observer can define safety properties and liveness properties. The advantage of using observers is that they can be programmed in the same language than the system to be checked. Thus, it is not necessary to learn new formalisms. Moreover, approaches based on observers require just the definition of the observers and the check the reachability to obtain the result. These arguments are very important for the diffusion of formal methods in the industry. However, the negative side of using observers is their reachability problems and their complexity to generate in the case of complicated requirements.

We can classify existant observers approaches into two main categories: one related to automaton and the other is not related to automaton. In the first category, observers are presented as an automaton. We release two approaches: Büchi automata and Test automata.

- Büchi automata is a type of  $\omega$ -automaton ( $\omega$ -automaton is an automata that takes as input an infinit set of strings), which extends a finite automata to infinite inputs. Verifying properties using Büchi automata is based on the following principle: the property is transformed into a Büchi automaton which contains all the executions  $E_\phi$  that do not satisfy the property. To prove that the system  $S$  satisfies the property  $\phi$  ( $S \models \phi$ ), it returns to verify that all the executions of  $S$  are not of the forms described in  $E_\phi$ .

In (VW86), the authors use büchi automata to verify properties. They transform each PTL formula into a büchi automata. An other work also based on büchi automata is (GPV<sup>+</sup>95). The authors present an algorithm to generate during the execution, a büchi automata from a formula. The model checker of LTL logic (LP85) is considered as a case of using Büchi automata observer in the verification approach.

- Test automata are used to verify that a reject state is never reached. A test automata  $T_\phi$  is constructed for each property  $\phi$ . For a temporised system A, the principle is :

$$A \text{ satisfy } \phi \Leftrightarrow A || T_\phi \text{ can not reach a reject state in } T_\phi$$

In (ABBL03, ABL98), test automata are used to check properties of reactive systems. The goal is to identify properties on timed automata for which model

## 2.5 Comparison with Related Works and Contributions

---

checking can be reduced to reachability checking. In this framework, verification is limited to safety and bounded liveness properties. In the context of Time Petri Net, a similar approach has been experimented by (TSLT97) but they propose a less general model for observers.

The second category of observers do not rely on timed automata, however, this class provides fundamental concepts to observers. In (HPUB), the authors use GOAL (a specification language) observers to verify properties. In this case, observers are defined as processes which contain reject and acceptance states. The work presented in (JMG88) defines observers in the context of distributed systems. The observers and the systems are specified using Estelle language (Jr.85). In (DJC94), the authors present observers in the context of self-checking distributed systems. In self-checking systems, the system to be verified is composed with the observer. The goal of the observer is to signal the dysfunction of the system.

## 2.5 Comparison with Related Works and Contributions

Before presenting our contributions, it is important to introduce the context of the thesis and the main guidelines of our work. My work during this PhD thesis has taken place in the context of the project *QUARTEFT*<sup>1</sup>, funded by the FNRAE (Fondation de Recherche pour l'Aéronautique et l'Espace).

In the context of our work, systems are modeled using modeling Languages such as AADL and then transformed into Fiacre programs. Each Fiacre program is transformed into a Time Transition Systems (TTS) after compilation. TTS is an extension of TPN taking into account shared variables and priorities. We use Tina toolbox and SELT model checker to check requirements.

This framework is useful to specify and verify real time systems. The use of Fiacre language helps, first, to reduce the semantic gap between high-level models and the input format of verification tools and, second, makes it possible to define precisely the semantics of the input language “only once” and to share this work among different verification toolchains. This is particularly helpful when we try to address emergent system modeling language, whose semantic is still maturing. Since Tina toolbox and SELT model checker are related to Fiacre language, we rely on these tools to verify

---

<sup>1</sup><http://quarteft.loria.fr>

## 2. STATE OF THE ART

---

systems. We verify properties expressed using LTL logic. However, it is hard to verify real time properties or complicated properties using LTL logic. This requires the addition of some variables or some transitions. This task is, in general, complicated and difficult, especially for non-experts, hence the motivation of our work.

In this thesis, we address the problem of specifying and verifying (real time) properties in the context of critical, real time systems modeled using Fiacre language. We have defined a global framework including an approach to specify properties, methods to their verification and an approach to check the correctness of the verification approaches. We have implemented our framework in an extension of the Frac compiler (Fiacre language compiler) <sup>1</sup>, called pFrac <sup>2</sup>.

Section 2.5.1 presents our contributions for specifying properties. We are based on patterns approach. The goal of our patterns is to define, first, a rich set of properties including the most used requirements in practice and, second, to provide a simple way to define properties which will facilitate the task of specifying properties for non-experienced users.

Section 2.5.2 presents our contributions for the verification of requirements. We are based on observers. The goal of our method is to associate each pattern with an automated verification approach to check its correctness. We want to provide an efficient verification technique in terms of execution time and system states growth.

Section 2.5.3 presents our contributions for checking the correctness of the verification approach. Our goal is to prove that our model checking method is correct—that it provides an answer that is sound with respect to the semantics of patterns— and non intrusive—meaning that observer cannot interfere with the observed system.

Finally, we want to stress that the contributions of the thesis are not limited to TINA toolbox and SELT model checker. Since Fiacre language is the input of other tools such as *Construction and Analysis of Distributed Processes* (CADP), our work, which is integrated in the Fiacre language, can be used in the CADP verification framework.

### 2.5.1 Contributions on Specifying Properties

We define a catalog of real time specification patterns that can be viewed as a real time extension of Dywer et al patterns.

---

<sup>1</sup><http://projects.laas.fr/fiacre/home.php>

<sup>2</sup><http://homepages.laas.fr/nabid/pfrac.html>

---

## 2.5 Comparison with Related Works and Contributions

Concerning specification patterns approaches, we have presented in section 2.3 the related works relatives to the definition of patterns. Dwyer’s patterns (DAC99) are a good candidate to specify properties. However, it is not possible to define timed constraints using these patterns. Gruhn and Laue (GL06) present a set of patterns based on Dwyer’s approach, but they consider a less expressive set of patterns (without some modifier) and they have not integrated their language inside a tool chain or proved the correctness of their verification approach. The work presented in (KC05) define also real time patterns. Nonetheless, they do not consider the complexity of the verification problem (the implementability of their approach). The work presented in (Bit01) define a set of patterns but they are concentrated only on safety properties. In (CP99), the authors extend patterns by including new notion of event, however, their notion is limited to some kind of verified events.

We make several contributions. First, we present a simple way to specify timed requirements, especially for non-experts, since our approach is based on natural language (one of the advantages of using patterns approach). Moreover, we extend Dwyer’s patterns to take into account real time concepts. Recent study (BGPS12) has shown that Dwyer’s patterns are the most used in practice in industry and academia. Based on this analysis, we are confident about the utility of our patterns. Finally, we transform our patterns into LTL formula to be verified. We use a decidable model checking, since we can reduce our verification problem to model-checking of LTL formulas (see Table 2.1).

### 2.5.2 Contributions on Verifying Properties

We have defined several methods to check patterns on a system. Our methods are based on a classical observer-based approach. Our goal is to provide an efficient verification approach in terms of verification time and systems size growth.

Concerning observers approaches, we have presented in section 2.4.3 the works related to the definition of observers. We can cite the work of (ABBL03, ABL98) where test automata are used to check properties of reactive systems. The goal is to identify properties on timed automata for which model checking can be reduced to reachability checking. In this framework, verification is limited to safety and bounded liveness properties. A similar approach has been experimented by (TSLT97), but they propose



## 2. STATE OF THE ART

---

a less general model for observers and consider only two verification techniques over four kinds of time constraints.

In contrast to the related works presented in section 2.4.3, we have defined, for each pattern, a set of verification methods to check its correctness. The methods are based on the use of observers and model checking techniques in order to transform the verification of patterns into the verification of simpler LTL formula. An observer-based approach is easy to use since it is not necessary to introduce another formalism in order to define observers (they are defined in the same language than the system). While the use of observers for model checking is quite common, our contribution is original in two ways. First, we define different classes of observers for each pattern and use a pragmatic approach in order to select the most efficient candidate in practice. Our goal is to provide the best observer in practice in terms of verification time and system size growth. Second, we propose a formal framework to verify the correctness of observers (see Section.2.5.3).

### 2.5.3 Contributions on Checking the Correctness of the Model Checker

We have defined a formal framework to verify the correctness and the non-intrusiveness of our verification approach.

We have presented in section 2.4.3 the related works relatives to existent verification approaches. Few works consider the verification of model-checking tools. Indeed, most of the existing approaches concentrate on the verification of the model-checking algorithm, rather than on the verification of the tool itself. For example, Smaus et al. (SMS09) provide a formal proof of an algorithm for generating Büchi automata from a LTL formula using the Isabelle interactive theorem prover. This algorithm is at the heart of many LTL model-checker based on an automata-theoretic approach. The problem of verifying verification tools also appears in conjunction with certification issues. In particular, many certification norms, such as the DO-178B, requires that any tool used for the development of a critical equipment be qualified at the same level of criticality than the equipment. (Of course, certification does not necessarily mean formal proof!) In this context, we can cite the work done on the certification of the SCADE compiler <sup>1</sup>, a tool-suite based on the synchronous language Lustre that integrates a model-checking engine. Nonetheless, only the code-generation part of the compiler is certified and not

---

<sup>1</sup><http://www.esterel-technologies.com/products/scade-suite/>

## 2.5 Comparison with Related Works and Contributions

---

the verification part. Concerning works presented in 2.4.3, all the presented works do not reason about their verification methods. In all most the cases, they provide verification approaches without checking them.

In contrast to the related works, we have defined a formal framework to verify that our observers are correct and non-intrusive, meaning that they compute the correct answer and have no impact on the system under observation. The formal framework we have defined is not only useful for proving the validity of formal results but also for checking the soundness of optimization in the implementation. Our framework is composed of a graphical proof and a formal proof. The two methods are complementary. We use graphical proof to reason about the correctness of observers at an early step before doing formal proof. We add an observer to our catalog after reasoning about its correctness.

## Chapter 3

# Formal Framework

*“An expert is a man who has made all the mistakes which can be made in a very narrow field.”*  
*-Niels Bohr-*

This chapter is devoted to the definition of the different *formal frameworks* used in the context of this work. The chapter has two main parts. The first part is dedicated to the definition of the mathematical tools (theories) needed to reason about the correctness of our verification approach. The second part is dedicated to the methods used to define the semantics of our patterns.

The chapter is organized as follows. We present the Fiacre language in Section 3.1. Fiacre is a modeling language that has been designed in the context of the TOPCASED project (FGC<sup>+</sup>06) and it is the modelling language used in this thesis.

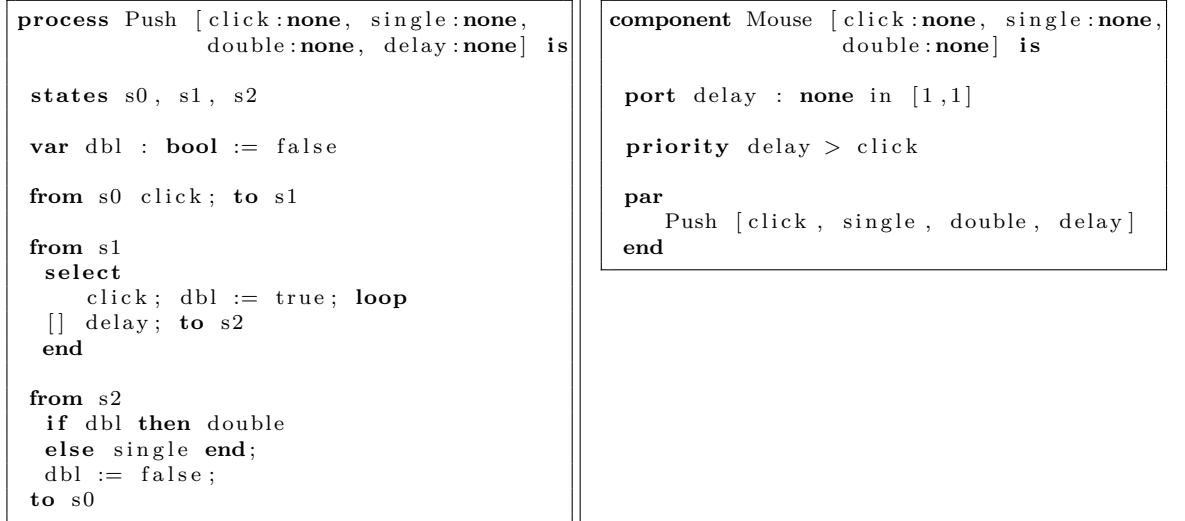
In the context of our work, each Fiacre program is transformed into a Time Transition Systems (TTS). We present, in Section 3.2, a definition of TTS model and its semantics expressed as Timed Traces.

Finally, Section 3.3 introduces three different methods that can be used to define timed requirements: Metric Temporal Logic (MTL); First-Order formulas over Timed Traces (FOTT); and a graphical notation called TGIL. Our experience, gathered when explaining patterns in front of different audiences, has shown that it is good to have several contrasting approaches to teach the meaning of patterns to new users.

### 3.1 Fiacre Language

We consider systems modelled using Fiacre language. Fiacre<sup>1</sup> is a formal specification language designed to represent both the behavioural and timing aspects of reactive systems. The design of the language is inspired by Time Petri Nets for its timing primitives, while the integration of time constraints and priorities into the language can be traced to the BIP framework (BBS06). A formal definition of the language is given in (BBFH<sup>+</sup>12, BBF<sup>+</sup>08). Fiacre supports two of the most common communication paradigms: communication through shared variable and synchronisation through (synchronous) communication ports. In the latter case, it is possible to associate time and priority constraints to communication over ports.

Fiacre programs are stratified in two main notions: *processes*, which are well-suited for modeling structured activities, and *components*, which describes a system as a composition of processes, possibly in a hierarchical manner. We give in Fig. 3.1 a simple example of Fiacre specification for a mouse button with double-click. The behaviour, in this case, is to emit the event `double` if there are more than two click events in strictly less than one unit of time (u.t.).



**Figure 3.1:** A double-click example in Fiacre

<sup>1</sup><http://projects.laas.fr/fiacre/>

### 3. FORMAL FRAMEWORK

---

#### 3.1.1 Processes

A process is defined by a set of parameters and control states, each associated with a set of *complex transitions* (introduced by the keyword **from**). The initial state of a process is the state corresponding to the first **from** declaration. Complex transitions are expressions that declares how variables are updated and which transitions may fire. They are built from deterministic constructs available in classical programming languages (assignments, conditionals, sequential composition, ...); non-deterministic constructs (such as external choice, with the **select** operator); communication on ports; and jump to a state (with the **to** or **loop** operators). For example, in Fig. 3.1, we declare a process named **Push** with four communication ports (**click** to **delay**) and one local boolean variable, **dbl**. Ports may send and receive typed data. The port type **none** means that no data is exchanged; these ports simply act as synchronisation events. Regarding complex transitions, the expression for **s1**, for instance, declares two possible behaviours when in state **s1**: first, on a **click** event, set **dbl** to **true** and stay in state **s1**; second, on a **delay** event, change to state **s2**.

#### 3.1.2 Components

A component is built from the parallel composition of processes and/or other components, expressed with the operator **par**  $P_0 \parallel \dots \parallel P_n$  **end**. In a composition, processes can interact both through synchronisation (message-passing) and access to shared variables (shared memory).

Components are the unit for process instantiation and for declaring ports and shared variables. The syntax of components allows to associate timing constraints with communications and to define priority between communication events. The ability to express directly timing constraints in programs is a distinguishing feature of Fiacre. For example, in the declaration of component **Mouse** (see Fig. 3.1), the **port** statement declares a local event **delay** and asserts that a transition from **s1** to **s2** should take exactly one unit of time. Additionally, the **priority** statement asserts that a transition on event **click** cannot occur if a transition on **delay** is also possible.

## 3.2 Systems Specification

In the context of our work, systems are modelled using Fiacre. Each Fiacre program is transformed into a Time Transition Systems (TTS) after compilation with a tool named *frac*. A TTS is an extension of Time Petri Nets (TPN) (Mer74) with shared variables and priorities. The behaviour of a Time Transition Systems is abstracted as a set of traces, called Timed Traces. We present in Section 3.2.1 a formal and informal definition of Time Transition Systems model and in Section 3.2.2 a definition of Timed Traces. Section 3.2.3 presents the notion of composition. The notion of composition is important in our work since we use TTS models for both the system and the observer and, for verification, we use TTS composition to graft the system with the observer.

### 3.2.1 Time Transition Systems

Time Transition Systems (TTS) is an internal format used in our model-checking tool. It is the output of the *Frac* compiler<sup>1</sup>, the Fiacre language compiler realised in our team. *Frac* transforms a Fiacre program into TTS model. TTS is an extension of Time Petri Nets taking into account shared variables and priorities.

#### 3.2.1.1 Informal Definition

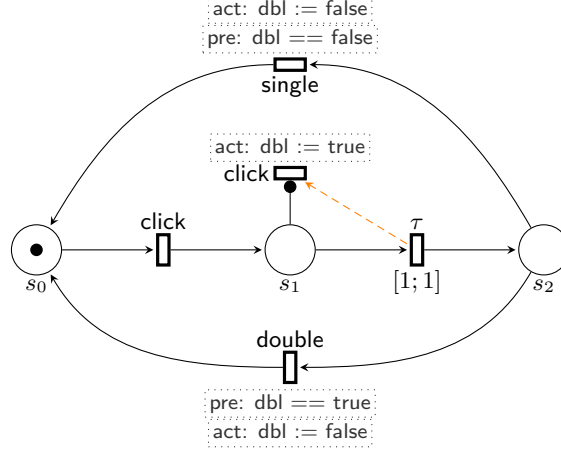
We introduce next a graphical syntax of TTS model. We present in Fig. 3.2 the corresponding TTS model of the example presented in Fig. 3.1.

Ignoring at first side conditions and side effects (the *pre* and *act* expressions inside dotted rectangles), the TTS in Fig. 3.2 can be viewed as a TPN with one token in place  $s_0$  as its initial marking. From this “state”, a click transition may occur and move the token from  $s_0$  to  $s_1$ . With this marking, the internal transition  $\tau$  is enabled and will fire after exactly one unit of time, since the token in  $s_1$  is not consumed by any other transition. Meanwhile, the transition labeled click may fire one or more times without removing the token from  $s_1$ , as indicated by the *read arc* (arcs ending with a black dot). After exactly one unit of time, because of the priority arc (a dashed arrow between transitions), the click transition is disabled until the token moves from  $s_1$  to  $s_2$ .

Data is managed within the *act* and *pre* expressions that may be associated to each transition. These expressions may refer to a fixed set of variables that form the *store*

<sup>1</sup><http://projects.laas.fr/fiacre/download.php>

### 3. FORMAL FRAMEWORK



**Figure 3.2:** The double-click example in TTS

of the TTS. Assume  $t$  is a transition with guards  $act_t$  and  $pre_t$ . In comparison with a TPN, a transition  $t$  in a TTS is enabled if there is both: (1) enough tokens in the places of its pre-condition; and (2) the predicate  $pre_t$  is true. With respect to the firing of  $t$ , the main difference is that we modify the store by executing the action guard  $act_t$ . For example, when the token reaches the place  $s_2$  in the TTS of Fig. 3.2, we use the value of the variable `dbl` to test whether we should signal a double click or not.

#### 3.2.1.2 Formal Definition

Since TTS model is an extension of TPN, we present first a formal definition of TPN and then extend this definition to take into account TTS.

Labeled Time Petri Nets (or TPN) extends Time Petri Nets (Mer74) with an action alphabet and a function labelling the transitions with those actions.

Notation : Let  $I^+$  be the set of nonempty real intervals with non negative rational endpoints. For  $i \in I^+$ , the symbol  $\downarrow i$  denotes the left end-point of the interval  $i$  and  $\uparrow i$  its right end-point, if  $i$  is bounded, or  $\infty$  otherwise. We use  $\mathbb{N}$  to denote the set of non negative integers.

**Definition 1.** A labeled Time Petri Net (or TPN) is a 8-tuple  $(P, T, B, F, M_0, I_s, \Sigma, L)$  in which:

- $P$  is a finite set of places  $p_i$ ;

- $T$  is a finite set of transitions  $t_i$ ;
- $B$  is the backward incidence function  
 $B : T \rightarrow P \rightarrow \mathbb{N}$ ;
- $F$  is the forward incidence function  
 $F : T \rightarrow P \rightarrow \mathbb{N}$ ;
- $M_0$  is the initial marking function  
 $M_0 : P \rightarrow \mathbb{N}$ ;

- $I_s$  is a function called the static interval function

$$I_s : T \rightarrow I^+;$$

Function  $I_s$  associates a temporal interval  $I_s(t) \in I^+$  with every transition of the system.  $\downarrow I_s(t)$  and  $\uparrow I_s(t)$  are called the static earliest and latest firing times of  $t$ , respectively. Assuming that a transition  $t$  became enabled at time  $\tau$ , then  $t$  cannot fire before  $(\tau + \downarrow I_s(t))$  and no later than  $(\tau + \uparrow I_s(t))$  unless disabled by firing some other transition.

- $\Sigma$  is a finite set of actions, or labels, not containing the silent action  $\varepsilon$ ;
- $L : T \rightarrow \Sigma \cup \{\varepsilon\}$  is a transition labelling function.

A marking is a function  $M : P \rightarrow \mathbb{N}$  that records the current (dynamic) value of the places in the net, as transitions are fired. The transition  $t \in T$  is enabled by  $M$  iff  $(M \geq B(t))$ . The dynamic interval function  $I : T \rightarrow I^+$  is a mapping from transitions to time intervals. The dynamic interval function is used to record the current timing constraints associated to each transition, as time passes.

A transition  $t$  can fire from  $(M, I)$  if  $t$  is enabled at  $M$  and instantly fireable, that is  $0 \in I(t)$ . In the target state, the transitions that remained enabled while  $t$  is fired ( $t$  excluded) keep their time interval, the intervals of the others (newly enabled) transitions are set to their respective static intervals. Together with those “discrete” transitions, a Time Petri Nets adds the ability to model the flowing of time. A continuous transition of amount  $d$  (i.e. taking  $d$  time units) is possible iff  $d$  is less than  $\uparrow I(t)$  for all enabled transitions  $t$ .

The definition of TTS is a natural extension of TPN that takes variables and priorities into account. We present, next, a definition of TTS based on Definition. 1.



### 3. FORMAL FRAMEWORK

---

**Definition 2.** *[Time Transition Systems] A Time Transition Systems (or TTS) is a 10-tuple  $(P, T, S, <, B, F, M_0, I_s, \Sigma, L)$  where:*

- *$P$  is a finite set of places  $p_i$ ;*
- *$T$  is a finite set of transitions  $t_i$ ;*
- *$S$  is a finite set of stores  $s_i$ ;*
- *$<$  is a binary, transitive relation over  $T$  which encodes the (static) priority relation between transitions;*
- *$B$  is the backward incidence function*  

$$B : T \rightarrow P \rightarrow \{0, 1\};$$
- *$F$  is the forward incidence function*  

$$F : T \rightarrow P \rightarrow \{0, 1\};$$
- *$M_0$  is the initial marking function*  

$$M_0 : P \rightarrow \{0, 1\};$$
- *$I_s$  is a function called the static interval function*  

$$I_s : T \rightarrow I^+;$$
- *$\Sigma$  is a finite set of actions, or labels, not containing the silent action  $\varepsilon$ ;*
- *$L : T \rightarrow \Sigma \cup \{\varepsilon\}$  is a transition labelling function.*

*Like in Definition. 1, we use the function  $M : P \rightarrow \{0, 1\}$  to record the current (dynamic) value of the places in the net, as transitions are fired and  $I : T \rightarrow I^+$  to record the current timing constraints associated to each transition, as time passes. The transition  $t \in T$  is enabled by  $M$  and  $S$  iff  $(M \geq B(t))$  and  $t$  is enabled under store  $s$ , it means that before firing  $t$  the predicate of the transition must be verified and the action of the transition must be executed after firing  $t$  (the predication and the action are represented by the store of  $t$ ).*

We note by the triple  $(m, s, I)$  the *state* of a TTS and by  $(m^{\text{init}}, s^{\text{init}}, I_s)$  its initial state, which requires the dynamic time constraint of every transition to be equal to its static time constraint.

The set  $\{0, 1\}$  used to define  $B$ ,  $F$  and  $M$  implies that the underlying Time Petri Nets is one-safe.

### 3.2.2 Semantics of Time Transition Systems expressed as Timed Traces

The behaviour of a TTS is abstracted as a set of traces, called *Timed Traces*. In contrast, the behaviour expected by the user is expressed with some properties (some invariants) to be checked against this set of timed traces. For instance, one may check the invariant that variable `dbl` is never true when `s0` is marked (using an accessibility check), or that transition `click` is followed by either `single` or `double` within one time unit (see Fig. 3.2). As a consequence, traces must contain information about fired transitions (e.g. `single`), markings (e.g.  $m(s_0)$ ), store (e.g. current value of `dbl`), and elapsing of time (e.g. to detect the one-time-unit deadline).

Formally, we define an event  $\omega$  as a triple  $(t, m, s)$  recording the marking and store immediately after the transition  $t$  has been fired. We denote  $\Omega$  the set  $T \times \mathcal{M} \times S$  of possible events.

**Definition 3** (Timed Trace). *A Timed Trace  $\sigma$  is a possibly infinite sequence of events  $\omega \in \Omega$  and durations  $d(\delta)$  with  $\delta \in I^+$ . Formally,  $\sigma$  is a partial mapping from  $\mathbb{N}$  to  $\Omega^* = \Omega \cup \{d(\delta) \mid \delta \in I^+\}$  such that  $\sigma(i)$  is defined whenever  $\sigma(j)$  is defined and  $i \leq j$ . The domain of  $\sigma$  is written  $\text{dom } \sigma$ .*

Using classic notations for sequences, the empty sequence is denoted  $\epsilon$ ; given a finite sequence  $\sigma$  and a—possibly infinite—sequence  $\sigma'$ , we denote  $\sigma.\sigma'$  the *concatenation* of  $\sigma$  and  $\sigma'$ . The concatenation operation is associative.

**Definition 4** (Duration). *Given a finite trace  $\sigma$ , we define its duration,  $\Delta(\sigma)$ , using the following inductive rules:*

$$\Delta(\epsilon) = 0 \qquad \Delta(\sigma.d(\delta)) = \Delta(\sigma) + \delta \qquad \Delta(\sigma.\omega) = \Delta(\sigma)$$

*We extend  $\Delta$  to infinite traces, by defining  $\Delta(\sigma)$  as the limit of  $\Delta(\sigma_i)$  where  $\sigma_i$  are growing prefixes of  $\sigma$ .*

Infinite traces are expected to have an infinite duration. Indeed, to rule out Zeno behaviours, which mean that the system includes an infinite number of discrete steps in a finite amount of time, we only consider traces that let time elapse. Hence, the following definition:

**Definition 5** (Well-formed Traces). *A trace  $\sigma$  is well-formed if and only if  $\text{dom}(\sigma)$  is finite or  $\Delta(\sigma) = \infty$ .*

### 3. FORMAL FRAMEWORK

---

The following definition provides an equivalence relation over timed traces. This relation guarantees that a well-formed trace (not exhibiting a Zeno behaviour) is only equivalent to well-formed traces. One way to achieve this would be to require that two equivalent traces may only differ by a finite number of differences. However, we also want to consider equivalent some traces that have an infinite number of differences, such as for example the infinite traces  $(d(1).d(1).\omega)^*$  and  $(d(2).\omega)^*$  (where  $X^*$  is the infinite repetition of  $X$ ). Our solution is to require that, within a finite time interval  $[0, \delta]$ , equivalent traces must contain a finite number of differences.

**Definition 6** (Equivalence over Timed Traces). *For each  $\delta > 0$ , we define  $\equiv_\delta$  as the smallest equivalence relation over timed traces satisfying :*

- $\sigma.d(0).\sigma' \equiv_\delta \sigma.\sigma'$ ;
- $\sigma.d(\delta_1).d(\delta_2).\sigma' \equiv_\delta \sigma.d(\delta_1 + \delta_2).\sigma'$ ;
- $\sigma.\sigma' \equiv_\delta \sigma.\sigma''$  whenever  $\Delta(\sigma) > \delta$

*The relation  $\equiv$  is the intersection of  $\equiv_\delta$  for all  $\delta > 0$ .*

By construction,  $\equiv$  is an equivalence relation. Moreover,  $\sigma_1 \equiv \sigma_2$  implies  $\Delta(\sigma_1) = \Delta(\sigma_2)$ . Our notion of timed trace is quite expressive. In particular, we are able to describe events which happen at the same date (with no delay in between) while keeping a causality relation (one event is before another).

We now consider briefly the dynamic semantics of TTS, which is similar to the semantics of Time Petri Nets (Mer74). It is expressed as a binary relation between states labeled by elements of  $\Omega^*$ , and written  $(m, s, I) \xrightarrow{l} (m', s', I')$ , where  $l$  is either a delay  $d(\delta)$  with  $\delta \in I^+$  or an event  $\omega \in \Omega$ . We say that transition  $t$  is *enabled* if  $(M \geq B(t))$  and  $t$  is enabled through store  $s$ . A transition  $t$  is *fireable* if it is enabled, *time-enabled* (that is  $0 \in I(t)$ ) and there is no fireable transition  $t'$  that has priority over  $t$  (that is  $t < t'$ ).

Given these definitions, a TTS with state  $(m, s, I)$  may progress in two ways:

- *Time elapses* by an amount  $\delta$  in  $I^+$ , provided  $\delta \in I(t)$  for all enabled transitions, meaning that no transition  $t$  is urgent. In that case, we define  $I'(t) = I(t) - \delta$  for all enabled transitions  $t$  and  $I'(t) = I_s(t)$  for disabled transitions. Under these hypotheses, we have

$$(m, s, I) \xrightarrow{d(\delta)} (m, s, I')$$

- A fireable transition  $t$  fires:

$$(m, s, I) \longrightarrow^t (m', s', I')$$

such that  $m'$ ,  $s'$  and  $I'$  are respectively the newly marking, store and interval, obtained after firing  $t$ .

We inductively define :

- $st \longrightarrow^\sigma st'$  where  $\sigma$  is a finite trace which transform the system from state  $st$  to state  $st'$ ;
- $st \longrightarrow^\epsilon st'$  where  $\epsilon$  is defined as the identity relation over states  $st$  and  $st'$ ;
- $st \longrightarrow^{\sigma.\omega} st'$  is defined as the composition of  $\longrightarrow^\sigma$  and  $\longrightarrow^\omega$  and transform the system from state  $st$  to state  $st'$  (we omit details).

We write  $(m, s, I) \longrightarrow^\sigma$  whenever there exist a state  $(m', s', I')$  such that  $(m, s, I) \longrightarrow^\sigma (m', s', I')$ .

Given an infinite trace  $\sigma$ , we write  $(m, s, I) \longrightarrow^\sigma$  if and only if  $(m, s, I) \longrightarrow^{\sigma'}$  holds for all  $\sigma'$  finite prefixes of  $\sigma$ . Finally, the set of traces of a TTS  $N$  is the set of well-formed traces  $\sigma$  such that  $(m^{\text{init}}, s^{\text{init}}, I_s) \longrightarrow^\sigma$  holds. This set is written  $\Sigma(N)$ .

The TTS model is a good target for compiling the Fiacre language: a process  $P$  is compiled to a TTS with one place for every state in  $P$ , while parallel composition of processes is modelled by composition of TTS (see Section. 3.2.3). The reference semantics of Fiacre (BBFH<sup>+</sup>12) is defined using a structural approach to operational semantics. This semantics has been implemented in a tool called *frac*, that compiles a Fiacre program into a Time Transition Systems.

### 3.2.3 Composition of Time Transition Systems and Composition of Timed Traces

We study the composition of two TTS and consider the relation between traces of the composed system and traces of both components. This operation is particularly significant in the context of this work, since both the system and the observer are TTS and we use composition to graft the latter to the former. In particular, we are interested

### 3. FORMAL FRAMEWORK

---

in conditions ensuring that the behaviour of the observer does not interfere with the behaviour of the observed system.

The “parallel composition” of Labeled Time Petri Nets is a fundamental operation that is used to model large systems by incrementally combining smaller nets. Basically, the composition of two Labeled TPN  $N_1$  and  $N_2$  is a Labeled net  $N \stackrel{\text{def}}{=} (N_1 \parallel N_2)$  such that: the places of  $N$  is the cartesian product of the places of  $N_1$  and  $N_2$ , and the transitions of  $N$  is the fusion of the transitions in  $N_1$  and  $N_2$  that have the same label. A formal definition for the composition of two TPN is given in (PBV11).

Composition of TTS is basically the same, with the noticeable restriction that transitions which have priority over other transitions may not be synchronised across components. This is required to ensure the composition theorem, which we introduce in Definition 8.

**Definition 7** (Composable TTS, synchronized transitions). *We consider two TTS, namely  $N_1$  and  $N_2$ , defined as  $(P_i, T_i, S_i, <_i, B_i, F_i, M_0^i, I_s^i, \Sigma^i, L^i)$  for  $i \in \{1, 2\}$ , respectively. The set of synchronised transitions of  $N_1$  is  $\{t_1 \in T_1 \mid \exists t_2 \in T_2 . L(t_1) = L(t_2)\}$ . We define the set of synchronised transitions of  $N_2$  similarly. Then,  $N_1$  and  $N_2$  are composable if the following conditions hold:*

1.  $P_1 \cup T_1 \cup S_1$  is disjoint from  $P_2 \cup T_2 \cup S_2$ .
2. for  $i = 1, 2$ , every synchronised transition  $t_i$  of  $N_i$  is such that  $I_i(t_i) = [0, +\infty[$ , and there is no transition  $t' \in T_i$  with  $t' < t_i$ .

The first condition ensures that  $N_1$  and  $N_2$  are disjoint, in particular they must use disjoint stores. As a consequence, no information can be exchanged through shared variables. Thus, synchronisation occurs through transitions only. As stated by the second condition, in every pair of synchronised transitions, both transitions must have a trivial time constraint  $[0, +\infty[$ . This condition as well as the condition on priorities is necessary to ensure composition, as stated by Property 1 below.

**Definition 8** (Composition of two TTS). *Assuming  $N_1$  and  $N_2$  are defined as above, let  $N$  be the TTS corresponding to their composition, which we write  $N = N_1 \parallel N_2$ . It is defined by the 10-tuple  $(P, T, S, <, B, F, M_0, I_s, \Sigma, L)$  where:*

1.  $P = P_1 \cup P_2$

2. Let  $\perp$  be an element not in  $T_1 \cup T_2$ . Let  $T_1^\perp$  be  $T_1 \cup \{\perp\}$  and  $T_2^\perp$  be  $T_2 \cup \{\perp\}$ . We define  $T$  as the following subset of  $T_1^\perp \times T_2^\perp$ :

$$\begin{aligned} T = & \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, L(t_1) = L(t_2)\} \\ & \cup \{(t_1, \perp) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised}\} \\ & \cup \{(\perp, t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised}\} \end{aligned}$$

We remark that  $(\perp, \perp) \notin T$ .

3.  $S = S_1 \times S_2$

4.  $<_1$  is a binary relation on  $T_1$ . We may freely consider it as a binary relation on  $T_1^\perp$  (and so there is no  $t \in T_1^\perp$  with  $t < \perp$  or  $\perp < t$ ). Similarly,  $<_2$  is considered as a binary relation on  $T_2^\perp$ . Then,  $<$  is defined by: for all  $(t_1, t'_1, t_2, t'_2) \in (T_1^\perp)^2 \times (T_2^\perp)^2$ , we have  $(t_1, t_2) < (t'_1, t'_2)$  if and only if  $t_1 < t'_1$  or  $t_2 < t'_2$ . As required,  $<$  is transitive (we omit the proof).

5.  $B$  is the backward incidence function

$$B : T \rightarrow P \rightarrow \{0, 1\};$$

where  $T$  and  $P$  are respectively the set of transitions and places of TTS  $N$ .

The function  $B$  is defined as follow:

$$\begin{aligned} B(t) = & B(t_1) + B(t_2) \mid t_1 \in T_1, t_2 \in T_2, L(t_1) = L(t_2) \\ & \cup B(t_1) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised} \\ & \cup B(t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised} \end{aligned}$$

6.  $F$  is the forward incidence function

$$F : T \rightarrow P \rightarrow \{0, 1\};$$

where  $T$  and  $P$  are respectively the set of transitions and places of TTS  $N$ .

The function  $F$  is defined as follow:

$$\begin{aligned} F(t) = & F(t_1) + F(t_2) \mid t_1 \in T_1, t_2 \in T_2, L(t_1) = L(t_2) \\ & \cup F(t_1) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised} \\ & \cup F(t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised} \end{aligned}$$

7.  $M_0$  is the initial marking function

$$M_0 : P \rightarrow \{0, 1\};$$

where  $P$  is respectively the set of places of TTS  $N$ .

8.  $I_s$  is a function called the static interval function

$$I_s : T \rightarrow I^+;$$

The function  $I_s$  is defined as follow:

### 3. FORMAL FRAMEWORK

---

$$\begin{aligned}
I_s(t) &= I_s(t_1) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised} \\
&\cup I_s(t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised} \\
&\cup [0, +\infty[ \text{ otherwise}
\end{aligned}$$

$$9. \Sigma = \Sigma_1 \cup \Sigma_2$$

10.  $L : T \rightarrow \Sigma \cup \{\varepsilon\}$  such that:

$$\begin{aligned}
L(t) &= L(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, L(t_1) = L(t_2) \\
&\cup L(t_1) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronised} \\
&\cup L(t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronised}
\end{aligned}$$

Additionally, the initial state is defined as  $((m_1^{init}, m_2^{init}), (s_1^{init}, s_2^{init}), (I_s^1, I_s^2))$ .

We now define composition of timed traces, and then show, in Property 1, that traces generated by  $N$  correspond to composition of traces from  $N_1$  and traces from  $N_2$ .

We extend  $L$  to events and durations by defining  $L(d(\delta)) = d(\delta)$ . Additionally, we say that an event  $\omega$  is synchronised if and only if  $\omega$  is a delay  $d(\delta)$  or  $\omega$  is  $(t, m, s)$  and  $t$  is synchronised (as defined in Definition 7).

In the same way that systems can be composed, it is possible to compose a timed trace of a TTS  $N_1$  with the trace of another TTS  $N_2$  when some conditions are met. Basically, events with the same label must occur synchronously, time elapses synchronously in both systems, and unsynchronised events—events that are not shared between  $N_1$  and  $N_2$ —can only be composed with  $d(0)$  (meaning they are not synchronised with an observable event).

**Definition 9** (Composable Traces). *Let  $\Omega_1$  and  $\Omega_2$  be the set of events of two TTS  $N_1$  and  $N_2$ , respectively. We define the relation  $\bowtie$  between  $\Omega_1^*$  and  $\Omega_2^*$  as the smallest relation satisfying the following inference system:*

$$\begin{array}{ccc}
\frac{\omega_1 \bowtie \omega_2}{L(\omega_1) = L(\omega_2)} & \frac{d(0) \bowtie \omega_2}{\omega_2 \text{ not synchronised}} & \frac{\omega_1 \bowtie d(0)}{\omega_1 \text{ not synchronised}}
\end{array}$$

*This relation can be extended to pairs of traces  $(\sigma_1, \sigma_2)$  of  $N_1 \times N_2$  as follows. We say that  $\sigma_1$  and  $\sigma_2$  are composable, which we write  $\sigma_1 \bowtie \sigma_2$ , if and only if  $\text{dom } \sigma_1 = \text{dom } \sigma_2$  and  $\sigma_1(i) \bowtie \sigma_2(i)$  holds for all  $i \in \text{dom } \sigma_1$ . Notice that  $\sigma_1 \bowtie \sigma_2$  implies  $\Delta(\sigma_1) = \Delta(\sigma_2)$ .*

We are now able to state the composition property. Remind that  $\Sigma(N)$  is the set of traces of  $N$ .

**Property 1** (Compositionality). *Assume  $N_1$  and  $N_2$  are composable systems with events in  $\Omega_1$  and  $\Omega_2$  respectively. Let  $N$  be  $N_1 \parallel N_2$ ; we write  $\Omega$  its set of events. Then there exists a bijection  $f$  between  $\Omega$  and a subset of  $\Omega_1^* \times \Omega_2^*$  such that:  $\sigma \in \Sigma(N_1 \parallel N_2)$  if and only if there exists a pair of traces  $(\sigma_1, \sigma_2)$  of  $\Sigma(N_1) \times \Sigma(N_2)$  with  $\sigma_1 \bowtie \sigma_2$ ,  $\text{dom } \sigma_1 = \text{dom } \sigma_2$ , and  $\forall i \in \text{dom } \sigma$   $\left\{ \begin{array}{l} \sigma(i) = d(\delta) \text{ if } \sigma_1(i) = \sigma_2(i) = d(\delta) \\ \sigma(i) = f^{-1}(\sigma_1(i), \sigma_2(i)) \text{ otherwise} \end{array} \right.$ .*

In other words, given a trace  $\sigma \in \Sigma(N)$ , one may extract two composable traces  $\sigma_1 \in \Sigma(N_1)$  and  $\sigma_2 \in \Sigma(N_2)$ . Conversely, given two composable traces  $\sigma_1 \in \Sigma(N_1)$  and  $\sigma_2 \in \Sigma(N_2)$ , one may build a corresponding trace in  $\Sigma(N)$ , which we will write  $\sigma_1 \parallel \sigma_2$ . Thus, this property characterises the set of traces of  $N$  in terms of traces of  $N_1$  and  $N_2$ .

*Proof.* Let  $E$  be the set  $(\Omega_1 \cup \{d(0)\}) \times (\Omega_2 \cup \{d(0)\}) \setminus (d(0), d(0))$ . Let  $f$  be the bijection between  $\Omega$  and  $E$  defined as:

- if  $t_1 \neq \perp \wedge t_2 \neq \perp$ , then  $f((t_1, t_2), (m_1, m_2), (s_1, s_2))$  is  $((t_1, m_1, s_1), (t_2, m_2, s_2))$ .
- if  $t_1 \neq \perp$ , then  $f((t_1, \perp), (m_1, m_2), (s_1, s_2))$  is  $((t_1, m_1, s_1), d(0))$ .
- if  $t_2 \neq \perp$ , then  $f((\perp, t_2), (m_1, m_2), (s_1, s_2))$  is  $(d(0), (t_2, m_2, s_2))$ .

We now show that  $f$  satisfies the given property. We prove each way of the equivalence independently.

Assume  $\sigma$  is a trace of  $N$ . Let us define  $\sigma_1$  and  $\sigma_2$  by  $\text{dom } \sigma_1 = \text{dom } \sigma_2 = \text{dom } \sigma$  and for all  $i \in \text{dom } \sigma$ , if  $\sigma(i)$  is  $d(\delta)$  (for some  $\delta$ ), then  $\sigma_1(i) = \sigma_2(i) = d(\delta)$ , otherwise  $\sigma(i)$  is an event  $\omega$ , then let  $(\sigma_1(i), \sigma_2(i))$  be  $f(\omega)$ . It is straightforward to check that  $\sigma_1 \bowtie \sigma_2$  holds. It remains to be shown that  $\sigma_1$  (resp.  $\sigma_2$ ) is a trace of  $N_1$  (resp.  $N_2$ ). We show only the result for  $\sigma_1$ , the proof for  $\sigma_2$  being similar. This is a consequence of the two following implications, where  $I_1$  is the function  $I$  restricted to the domain  $T_1$  (and similarly for  $I'_1$ ), and where  $f_1(\omega)$  is the first projection of  $f(\omega)$ . To ease the reading, we display these implications as inference rules:

$$\frac{(m_1, s_1, I_1) \xrightarrow{d(\delta)} (m_1, s_1, I'_1)}{((m_1, m_2), (s_1, s_2), I) \xrightarrow{d(\delta)} ((m_1, m_2), (s_1, s_2), I')}$$

$$\frac{(m_1, s_1, I_1) \xrightarrow{f_1(\omega)} (m'_1, s'_1, I'_1)}{((m_1, m_2), (s_1, s_2), I) \xrightarrow{\omega} ((m'_1, m'_2), (s'_1, s'_2), I')}$$



### 3. FORMAL FRAMEWORK

---

We only sketch the proof of these rules. The delicate part concerns  $I$  (dealing with  $m$  and  $s$  is straightforward, by applying the definition of composition). More precisely, the (first projection of) enabled transitions of  $N$  is only a subset of enabled transitions of  $N_1$ , as a consequence of the definition of enabled transition. To state it otherwise, not all enabled transitions of  $N_1$  are enabled in  $N$ . Thus, there may be transitions  $t \in T_1$  such that  $I'_1(t_1) = I(t_1)$  whereas the expected value (according to the semantics of  $N_1$ ) would be  $I'_1(t_1) - \delta$  (in the first rule) or  $I_s^1(t_1)$  (in the second rule). Fortunately, these transitions  $t_1$  must be synchronised transition, therefore their time interval is always unconstrained, that is  $I_1(t_1) = I_s^1(t_1) = [0; +\infty[$ , as required by Definition. 7, and by remarking that  $[0; +\infty[ - \delta = [0; +\infty[$  for any  $\delta$  in  $\mathbb{R}^+$ . Additionally, in the second rule, if  $f_1(\omega)$  is  $(t_1, m_1, s_1)$ , we must ensure that no other transition in  $N_1$  has priority over  $t_1$ , so that  $t_1$  is fireable. This is a consequence of the definition of priorities in Definition. 8 and of the condition on priorities in Definition.7.

Conversely, we assume given two composable traces  $\sigma_1$  and  $\sigma_2$  of  $N_1$  and  $N_2$ , respectively. The trace  $\sigma$  is defined as stated in the property. It remains to be shown that  $\sigma$  is indeed a trace of  $N$ . The proof is actually very similar to the previous case (with the small difference that the condition on priorities is not used).

□

This result is used to show the innocuousness of observers in Chapter 5 Section 5.3.

## 3.3 Formal Framework for Expressing Timed Properties

We will use different methods to define the semantics of patterns (that is, essentially, to define the set of timed traces for which the pattern holds). Our experience shows that being able to confront different definitions for the same pattern, using contrasting approaches, is useful for teaching patterns.

### 3.3.1 Metric Temporal Logic

Metric Temporal Logic (MTL) (OW06, Koy90) is an extension of LTL where temporal modalities can be constrained by a time interval. For instance, the MTL formula  $A \mathbf{U}_{[1,3[} B$  states that in every execution of the system (in every trace), the event  $B$  must occur at a time  $t_0 \in [1, 3[$  and that  $A$  holds everywhere in the interval  $[0, t_0[$ . A

### 3.3 Formal Framework for Expressing Timed Properties

---

MTL formula is defined as follow:

$$\phi ::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc_I \phi \mid \phi_1 U_I \phi_2 \mid \Diamond_I \phi \mid \Box_I \phi$$

where  $p$  is a proposition,  $I \subseteq \mathbb{R}^+$  is an open, closed, or half-open interval with end points in  $\mathbb{N} \cup \infty$ ,  $\bigcirc$  represents the *next* operator,  $U_I$  represents the *until* operator and  $\Diamond$  and  $\Box$  represent the *eventually* and *globally* operator respectively. We will use also a weak version of the “until modality”, denoted  $A W B$ , that does not require  $B$  to eventually occur.

An advantage of using MTL is that it provides a sound and unambiguous framework for defining the meaning of patterns. On the negative side, however, it is known that the model-checking problem for full MTL is undecidable (see Chapter. 2). Some works, see e.g. (OW06), have been done to find suitable decidable fragments of MTL. One such subset, called MITL, is obtained by disallowing punctual time intervals in formulas, that is interval of the form  $[d, d]$ .

#### 3.3.2 First Order Formula over Timed Traces

We have presented in previous section, the MTL logic used to define our patterns. Nonetheless, this partially defeats one of the original goal of patterns, that is to circumvent the use of temporal logic in the first place. For this reason, we propose an alternative way for defining the semantics of patterns that relies on first-order formulas over timed traces (FOTT).

Our approach is based on the notion of Timed Traces, presented in Definition 3. FOTT is the first-order theory over well-formed timed traces  $(\sigma_1, \sigma_2, \dots)$  with trace concatenation, trace duration  $(\Delta(\sigma))$ , and modulo trace equivalence  $(\equiv)$ . (In the remainder of the text, for readability, we will often use the notation  $=$  instead of  $\equiv$ ).

Assume  $A$  is an event and  $\sigma$  is a timed trace. For brevity, we use the notation  $A \in \sigma$  for the formula  $\exists \sigma_1, \sigma_2. (\sigma = \sigma_1.A.\sigma_2)$ . With FOTT, it is also possible to express timing constraints over a trace. For example, the formula  $\exists \sigma_1, \sigma_2. (\sigma = \sigma_1.\sigma_2) \wedge (\Delta(\sigma_1) \geq 1)$  states that  $\sigma$  has a prefix of duration more than 1 u.t.

We can give more useful examples of FOTT formulas that corresponds to the notion of “scope” found in Dwyer’s patterns. We consider a timed trace  $\sigma$  and events  $A$  and  $B$ .

### 3. FORMAL FRAMEWORK

---

The “part” (or sub-trace) of  $\sigma$  that is after the first occurrence of  $A$ —or simply  $\sigma$  after  $A$ —can be defined as the trace  $\sigma_2$  such that:

$$\exists \sigma_1. \sigma = \sigma_1.A.\sigma_2 \wedge A \notin \sigma_1$$

Likewise, the scope  $\sigma$  before  $A$ —which determines the part of  $\sigma$  located before the first occurrence of  $A$ —can be defined as the trace  $\sigma_1$  such that:

$$\exists \sigma_2. \sigma = \sigma_1.A.\sigma_2 \wedge A \notin \sigma_1$$

Finally, we can define constraints such as: the duration “between  $A$  and  $B$ ” is equal to  $d$ —which determines the trace  $\sigma$  located between the first occurrence of  $A$  and the first occurrence of  $B$ —using the FOTT formula:

$$\exists \sigma_1. \sigma = A.\sigma_1.B \wedge \Delta(\sigma_1) = d$$

We believe that the use of FOTT may ease the work of engineers that are not trained with formal verification techniques but that have a background on mathematical analysis.

#### 3.3.3 Timed Graphical Interval Logic

In this section, we define a Timed Graphical Interval Logic (TGIL), that is a formal graphical notation for expressing the timing constraints and behavioural properties of a reactive system.

An issue limiting the adoption of model-checking technologies by the industry is the ability, for non-experts, to express their requirements. Engineers frequently use diagrams to explain the behaviour of a system or to describe desired scenarios. Nonetheless, such drawings usually suffer from the same drawbacks than requirements described using natural language: they can be ambiguous or misleading; they are not precise enough (do not cover all the cases); they are not amenable to automated transformation; ...

Many works have been done to solve this problem. In (DKM<sup>+</sup>94), the authors present Graphical Interval Logic (GIL). GIL is based on a graphical presentation to describe systems. A real time extension of the Graphical Interval Logic, called RTGIL, has been proposed by Dillon et al. (MMSR<sup>+</sup>96). RTGIL extends GIL by adding a “time length” search operator. In comparison, TGIL is more expressive since it provides an operator for time constrained search,  $(\setminus_I \phi)$ , that is not derivable in RTGIL (see Section 3.3.3.1).

### 3.3 Formal Framework for Expressing Timed Properties

---

For example, the timed pattern **present**  $A$  **after**  $B$  **within**  $I$  can be expressed in TGIL, but not in RTGIL. Other works propose graphical notations for expressing behavioural properties. Most of these proposals are based on informal diagrammatic notations, such as UML, or are not concerned with verification. Apart from the work on GIL, that we mentioned extensively, Alfonso et al. (ABK04) define Visual Timed event Scenarios (VTS), a graphical language to define complex requirements using annotations on a partial order of event. It is possible to express timing constraints using VTS but some simple requirements cannot be expressed, such as the fact that a given event, say  $A$ , should be true for a duration  $d$ . Concerning tooling, another reference is the TimeEdit tool (SHE01), that is based on timeline diagrams. TimeEdit specifications can be compiled into Büchi automata—just like LTL—and used with the Spin model-checker. Nonetheless, time line diagrams do not directly support the definition of timing constraints.

In contrast to these existent works, we define TGIL, a graphical logic to define timed requirements. We take our inspiration from the Graphical Interval Logic (GIL) of Dillon et al (DKM<sup>+</sup>94), and extends it with two operators for expressing timing constraints.

The usefulness of TGIL goes beyond the definition of timed patterns. It is also a good candidate to replace timed extensions of temporal logic and study their decidable fragments. For example, the timed search operator of TGIL is reminiscent of the  $\triangleright_I$  operators defined in the State Clock Logic (SCL) of Raskin and Schobbens (RyS96), a decidable, real time extension of PTL.

Our main motivation in the design of TGIL was to define the semantics of a set of real time specification patterns using a graphical notation. The idea is to provide an alternative formal definition based on TGIL. We believe that this new approach may ease the work of engineers that are not trained with formal verification techniques.

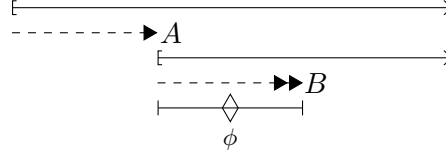
#### 3.3.3.1 TGIL Notations

We use letters  $A, B, \dots$  to denote a predicate on events. Events should be understood as instantaneous actions involved in the evolution of the system (see Chapter 4 Section 4.2 for events definition).

TGIL can be viewed as a real time extension of the Graphical Interval Logic (GIL) of Dillon et al. (DKM<sup>+</sup>94). An example of GIL diagram—that is also in TGIL—is given below.

### 3. FORMAL FRAMEWORK

---



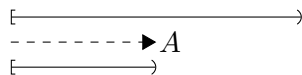
A TGIL specification, or formula, is a diagram that should be read from top to bottom and from left to right. Our example depicts three main notions used in TGIL. For each notion, we describe its graphical notation and propose an equivalent textual syntax.

**Execution Context** every formula is expressed with respect to an execution context, displayed with a straight line  $\vdash \longrightarrow$ , that represents a portion of an execution trace—a time interval—where the property is evaluated. The initial, top-most context symbolises the whole execution trace, that is the time interval  $[0, +\infty[$ .

When a starting point is specified, as in  $\vdash \xrightarrow{A} \longrightarrow$  (for instance  $A$  can be the result of a search primitive), then this execution context represents the part of the trace occurring strictly after  $A$ . The closed version, including the occurrence of  $A$ , is drawn as  $\xrightarrow{A} \vdash \longrightarrow$ .

**Search** formulas and sub-contexts are built from searches, that define instants matching a given constraint in the current context; searches are displayed with a dashed arrow  $---\blacktriangleright$  and are decorated with (a predicate on) events. In our example, the first search starts from the beginning of the initial context (thus at time 0) and define the first time instant in the context where an event  $A$  occurs (say  $t_A$ ). The second execution context is defined by the result of this search; it starts at time  $t_A$ .

A search can be combined with a context in order to define a sub-context: from a context and a search point, say  $S$ , we can define the context located after  $S$  (as we do in our example) or the context located before it (as shown with the diagram below). We use the notation  $[\rightarrow A | \phi\rangle$  for describing the first kind of context and the notation  $\langle \phi | \rightarrow A]$  for the other kind, where  $\phi$  is a TGIL specification.



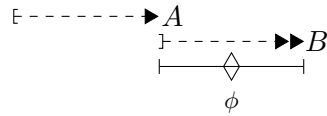
There are two kind of searches in our graphical notation: a weak search ( $---\blacktriangleright$ ) and a strong search ( $---\blacktriangleright\blacktriangleright$ ) version. With a strong search,  $[\rightarrow A | \phi\rangle$ , the formula is false if we fail to find an event matching  $A$  in the current context (if the search fails). At the

### 3.3 Formal Framework for Expressing Timed Properties

opposite, with a weak search, the formula is true if the search fails. We show how to define the weak search version as a derived operator in our logic.

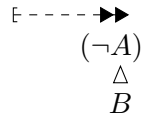
**Formulas** a TGIL specification associates properties to contexts and search points in the diagram. For instance, the last (bottom) element in our example states that the formula  $\phi$  should be true somewhere/sometimes in the context  $[t_A, t_B[$ , where  $t_A$  and  $t_B$  are the dates associated to the pair of events  $(A, B)$  matched in the previous searches. In our case,  $\phi$  can be a formula—expressed using another TGIL diagram—or simply a predicate on events. (The instant where  $\phi$  is true is materialised by the diamond, like in a search.)

As a consequence, our running example can be interpreted as follows: look for the first occurrence of an event  $A$ . If there is any then find the following occurrence of  $B$ . If no such occurrence exists the property is false. Finally, find an event, “in-between”, where  $\phi$  holds. For concision, we omit intermediate contexts when they can be inferred from the diagram. Thus, our example can be equivalently drawn:



The textual equivalent of this requirement could be written  $[\rightarrow A | \langle (\diamond \phi) | \rightarrow B \rangle]$ .

TGIL also provides a construction for expressing “punctual properties”—depicted using a triangle under a search point—that is a property relevant at this given instant in a context. For example, the first part of the TGIL diagram below states that, at the instant where  $A$  is false,  $B$  is true (since it is a strong search, it also states that  $B$  must eventually happens). The textual equivalent of this formula is  $[\rightarrow (\neg A) | B]$ .

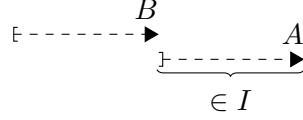


**Timing Constraints** finally, TGIL provides two operators for adding timing constraints on formulas: an operator that bounds the delay between two instants; and an operator that restrict a context to a given time interval. In the following, we use the symbol  $I$  as a shorthand for the time interval  $[d_1, d_2]$ .

### 3. FORMAL FRAMEWORK

---

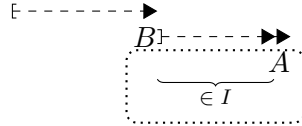
The first operator, called *time length*, uses a “curly braces” notation, illustrated below. It states that the delay between the two instants materialised by the end of the brace is in  $I$ .



We can use this operator to constrain the length (time duration) of a context. In particular, when the two instants are the boundaries of the same search—as it is the case in our example—we use the textual notation  $[\rightarrow_I A | \dots]$  to state that the length of the search  $\rightarrow A$  is in  $I$ . This restricted operator, denoted  $Len(I)$  in (MMSR<sup>+</sup>96), is the only timing operator that was considered in a previous timed extension of GIL.

The second operator, *time restriction*, limits an execution context to a given time interval  $I$  (relatively to the starting point of the context). This operator is drawn using a combination of dotted and solid line.

**Grouping** The dotted rectangle groups together predicate  $A$  with the time constraint. As a consequence, the search primitive looks for the first occurrence of  $A$  such that  $(t(A) - t(B) \in I)$  (other occurrences of  $A$  can exist after  $B$  and before this one).



To the best of our knowledge, the grouping and time restriction operators are totally new in the context of GIL. The time restriction operator is necessary to define the pattern *Present A after B within I*. Indeed, equipped with the *Len* operator alone, we can only detect if the first  $A$  following a  $B$  is within  $I$ . The grouping operator is necessary to define the pattern *A leadsto B within I* (see Chapter 4).

#### 3.3.3.2 Formal Semantics of TGIL

The semantics of a TGIL formula,  $\phi$ , is defined as the set of *timed traces* that holds for  $\phi$ . We consider a finite set of propositional variables,  $A, B, \dots$ , that denote “atomic properties” of events  $\omega \in \Omega$ . We use the expression  $\omega \in A$  to denote that the proposition

### 3.3 Formal Framework for Expressing Timed Properties

$A$  is true for  $\omega$ . (By extension, we should also use  $A$  to denote a predicate over propositional variables.)

We define the semantics of TGIL using our textual syntax. Besides the propositional fragment, the main operators are the punctual formula  $(A)$ , the left and right searches; the sometimes modality  $(\Diamond)$  and time restriction  $(\setminus_I)$ .

$$\begin{aligned} \phi &::= \neg\phi \mid \phi_1 \vee \phi_2 \mid A \\ &\mid [\rightarrow_I A \mid \phi] \mid \langle \phi \mid \rightarrow_I A \rangle \mid (\Diamond \phi) \mid (\setminus_I \phi) \end{aligned}$$

We use the satisfaction relation  $\sigma \models \phi$  to denote that the formula  $\phi$  holds for  $\sigma$ . In this definition, we use  $\sigma_I$  to denote the sub-trace of  $\sigma$  restricted to the time interval  $I$  and the notation  $\sigma \equiv A.\sigma$  as a shorthand for the condition  $(\sigma \equiv \omega.\sigma') \wedge (\omega \in A)$ .

$$\begin{aligned} \sigma \models \neg\phi &\quad \text{iff} \quad \text{not } \sigma \models \phi \\ \sigma \models \phi_1 \vee \phi_2 &\quad \text{iff} \quad (\sigma \models \phi_1) \vee (\sigma \models \phi_2) \\ \sigma \models A &\quad \text{iff} \quad \sigma \equiv A.\sigma' \\ \sigma \models [\rightarrow_I A \mid \phi] &\quad \text{iff} \quad \sigma \equiv \sigma_1.A.\sigma_2 \wedge A \notin \sigma_1 \\ &\quad \wedge \Delta(\sigma_1) \in I \wedge \sigma_2 \models \phi \\ \sigma \models \langle \phi \mid \rightarrow_I A \rangle &\quad \text{iff} \quad \sigma \equiv \sigma_1.A.\sigma_2 \wedge A \notin \sigma_1 \\ &\quad \wedge \Delta(\sigma_1) \in I \wedge \sigma_1 \models \phi \\ \sigma \models (\Diamond \phi) &\quad \text{iff} \quad \exists \sigma_1, \sigma_2 . \sigma \equiv \sigma_1.\sigma_2 \wedge \sigma_2 \models \phi \\ \sigma \models (\setminus_I \phi) &\quad \text{iff} \quad \sigma_I \models \phi \end{aligned}$$

This definition is quite similar to the satisfaction relation for Linear Temporal Logic (LTL). In particular, TGIL is an instance of a linear time logic, in the sense that it cannot be used to reason on “several possible time lines” simultaneously.

We can define additional logical operators that are useful for defining properties. The “true” formula,  $\top$ , can be encoded by any tautology, such as  $A \vee \neg A$ . Another example of derived formula is  $(\setminus_I (\Diamond \phi))$ , which defines a property that is satisfied if  $\phi$  holds sometimes in  $I$ . We use the notation  $(\Diamond_I \phi)$ , for this derived operator, to stress the direct relationship with MTL. The dual of the sometimes operator,  $(\Box_I \phi) \stackrel{\text{def}}{=} \neg(\Diamond_I (\neg\phi))$ , holds for the traces such that  $\phi$  is always true in  $I$ .

It is possible to derive the weak search operator from the strong search version. Indeed, the formula  $[\rightarrow_I A \mid \phi]$  is true if and only if  $[\rightarrow_I A \mid \phi]$  is true or the search for  $A$  fails in the context  $\setminus_I$ . Put another way, if we can find  $A$  in  $\setminus_I$  then  $[\rightarrow_I A \mid \phi]$  should be true:  $[\rightarrow_I A \mid \phi] \stackrel{\text{def}}{=} (\Diamond_I A) \Rightarrow [\rightarrow_I A \mid \phi]$ .

To conclude this section, we have defined a timed extension of Dillon’s et al Graphical Interval Logic (TGIL) that is more expressive than previous proposals. The semantics



### 3. FORMAL FRAMEWORK

---

of TGIL can be easily defined using an equivalent “textual notation” that may facilitate (such as e.g.) proving the consistency of partial proof systems. We will show in Chapter 4 how to apply TGIL for the definition of a real time specification patterns.

#### 3.4 Conclusion

In this chapter, we describe the formal notations that will be used in the rest of this document; in particular, to define the semantics of patterns and to check the correctness of our verification approaches. We have defined formally TTS model which is an extension of TPN taking into account shared variables and priorities. Each Fiacre program is transformed into TTS model after compilation. TTS can be interpreted as a set of timed traces. We define also the notion of composition between TTS models and timed traces.

We have presented also the methods used to define our patterns: MTL logic, First order formula over timed traces (FOTT) and TGIL logic. Our experience shows that being able to confront different definitions for the same pattern, using contrasting approaches, is useful for teaching patterns. While MTL logic is an existent approach to define requirements, TGIL is a new one. It was defined in the context of this thesis. We formally define the semantics of TGIL and we show, in the next chapters, the usefulness of this notation. TGIL is a good technique to define patterns. However, a limitation of this logic is that TGIL is essentially a linear time logic (it expresses constraints on traces), whereas some properties may require a branching time extension.

## Chapter 4

# Real Time Specification Patterns

*“To be conscious that you are ignorant is a great step to knowledge.”*

*-Benjamin Disraeli-*

In this Chapter, we describe a catalogue of real time specification patterns. This Chapter is organised as follows. After a short introduction, we present in Section 4.2 the notion of events (or *observables*) in the context of the Fiacre language. Section 4.3 presents our catalogue of real time patterns. We present a use case in Section 4.4 and we show how we can use our patterns in a real case. Then we present the meta model of our patterns in Section 4.5. Finally we conclude the Chapter in Section 4.6 by presenting the advantages and drawbacks of our approach.

### 4.1 Introduction

An issue limiting the adoption of model-checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the verification tools. Indeed, there is often a significant gap between the boilerplates used in requirements statements and the low-level formalisms used by model-checking tools; the latter usually relying on temporal logic. This limitation has motivated the definition of dedicated assertion languages for expressing properties at a higher level. However, only a limited number of assertion languages support the definition of timing constraints and even fewer are associated to an automatic verification tool, such as a model-checker.

We propose a set of real time specification patterns aimed at the verification of reactive systems with hard real time constraints. Our main objective is to propose an

## 4. REAL TIME SPECIFICATION PATTERNS

---

alternative to timed extensions of temporal logic during model-checking. Our patterns are designed to express general timing constraints commonly found in the analysis of real time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both clarity and computational complexity. In particular, each pattern should correspond to a decidable model-checking problem.

Our patterns can be viewed as a real time extension of Dwyer’s specification patterns (DAC99). In his seminal work, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a small number of “pattern formulas”. We follow a similar philosophy and define a list of patterns that takes into account timing constraints. At the syntactic level, this is mostly obtained by extending Dwyer’s patterns with two kind of *timing modifiers*: (1) *P within I*, which states that the delay between two events declared in the pattern *P* must fit in the time interval *I*; and (2) *P lasting D*, which states that a given condition in *P* must hold for at least duration *D*. For example, we define a pattern *Present A after B within [0, 4]* to express that the event *A* must occur within 4 u.t. of the first occurrence of event *B*, if any, and not simultaneously with it. Although seemingly innocuous, the addition of these two modifiers has a great impact on the semantics of patterns and on the verification techniques that are involved.

### 4.2 Observable Events

Properties checked in the formal verification of systems typically express constraints on the occurrence of *observable events*—and essentially restrict the order and the date of occurrence of these events.

In the context of this work, systems are modelled using Fiacre. Hence the observable events are Fiacre events. In a Fiacre specification, an event is an *instantaneous action* involved in the evolution of the system. It can be:

- the move of a process in a new state. In the specification, we use the key word *state* to refer to this event;
- a communication through shared variables, it means the change of the value of variable *x* in the program. In the specification, we use the key word *value* to refer to this event;

- a communication through ports. In the specification, we use the key word *event* to refer to this event;
- the execution of a transition. In the specification, we use the key word *tag* to refer to this event.

Moreover, we can associate each event with a scope. If  $A$  is an event corresponding to one of the events cited above then we can associate  $A$  with the following scopes : *enter A*, *leave A* and *not A* which correspond to the fact of entering into, leaving and different from event  $A$  respectively.

We extend Fiacre Language with an unambiguous naming convention for the events. Each event  $a$ —defined in a process  $P$  in a Fiacre program—may correspond to several “event instances”, where each event instance corresponds to a different process instantiation. Indeed, each invocation of a process (respectively a component) in a component leads to the generation of a process (resp. component) instance.

A valid Fiacre specification must declare the main component. Each process/-component instance may be connected to this initial component in a hierarchical way. To disambiguate two instances of the same process/component, declared in the same context, we use indices. We show using an example (see Fig. 4.1) how events are named with our approach. Our example is composed of a process  $A$  and a process  $B$ . Process  $A$  uses a port  $p$  and a shared variable  $X$ . From the state *start*, process  $A$  communicates through port  $p$  and sets the value of  $X$  to 1. Process  $B$  waits for 1 unit of times and returns to its initial state. The component  $D$  synchronises processes  $A$  and  $B$ , however component  $C$  synchronises process  $A$  and component  $D$ . The component *main*, which is the main component of the program, is the synchronisation of components  $D$  and  $C$  and process  $A$ . We have added in front of each identifier, between brackets, the processes/components indices computed by Frac, the Fiacre language compiler, when generating the TTS file (right side of Fig. 4.1).

### 4.3 Catalogue of Real Time Patterns

A pattern is defined as a description or a template of how to specify a property that occurs commonly in the specification of concurrent and reactive systems; We describe, in this

#### 4. REAL TIME SPECIFICATION PATTERNS

---

```

process A [p:none] is
  states start, stop
  var x:int:=0
  from start
    p;
    x:=1;
  to stop

process B is
  states start
  from start
    wait[1,1];
  to start

component D[p:none] is
  par A[p] || B end

component C [p:none] is
  par A[p] || D || D end

component Main is
  port p: none

  par C[p] || D[p] || A[p]
end

Main

```

Main- $C_{(1)}-A_{(1)}$   
 |       $-D_{(1)}-A_{(2)}$   
 |      |       $-B_{(1)}$   
 |       $-D_{(2)}-A_{(3)}$   
 |      |       $-B_{(2)}$   
 |       $-D_{(3)}-A_{(4)}$   
 |      |       $-B_{(3)}$   
 |       $-A_{(5)}$

- Event Main/1/1/value ( $x=1$ )—with our notation—corresponds to the transition which sets the value of X to 1 of the process  $A_{(1)}$  in the TTS generated from Frac.
- Event Main/2/2/state start corresponds to the state “start” of the process  $B_{(3)}$  in TTS.

**Figure 4.1:** The approach used to name events in Fiacre

section, our patterns using a hierarchical classification borrowed from Dwyer (DAC99) but adding the notion of “timing modifiers”.

Our patterns are built from five categories, listed below, or from the composition of several patterns:

- **Existence Patterns (Present):** for conditions that must eventually occur;
- **Absence Patterns (Absent):** for conditions that should not occur;
- **Universality Patterns :** for conditions that must occur throughout the whole execution;
- **Response Patterns (Response):** for (trigger) conditions that must always be followed by a given (response) condition;
- **Precedence Patterns :** for (signal) conditions that must always be preceded by a given (trigger) condition.

In each class, generic patterns may be specialised using one of five *scope modifiers* that limit the range of the execution trace over which the pattern must hold:

- **Global** : the default scope modifier, that does not limit the range of the pattern. The pattern must hold over the whole timed trace;
- **Before R** : limit the pattern to the beginning of a timed trace, up to the first occurrence of R;
- **After Q** : limit the pattern to the events following the first R;
- **Between Q and R** : limit the pattern to the events occurring between an event Q and the following occurrence of an event R;
- **After Q Until R** : similar to the previous scope modifier, except that we do not require that R must necessarily occur after a Q.

Finally, timed patterns are obtained using one of four possible kinds of *timing modifiers* that limit the possible dates of events referred in the pattern:

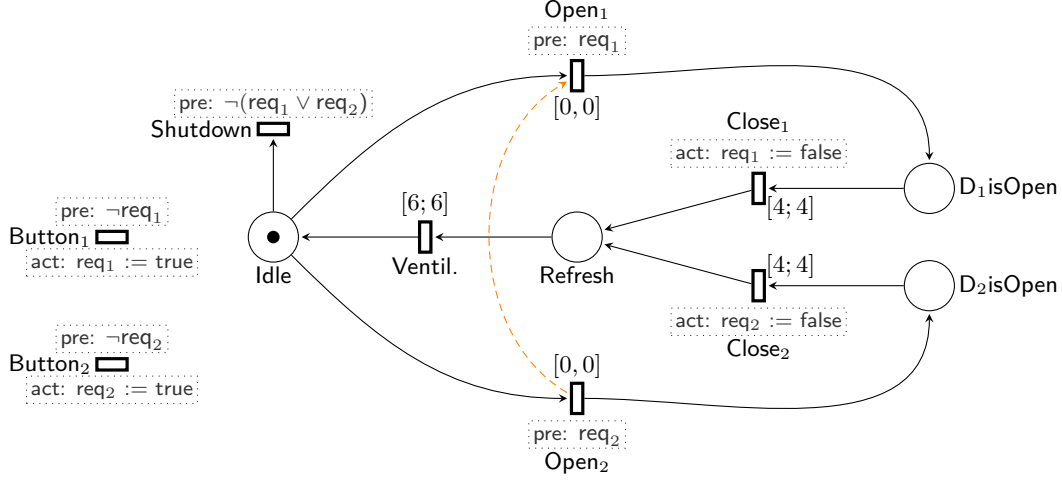
- **Within I, For interval I** : to constraint the delay between two given events to belong to the time interval  $I$ ;
- **Lasting D, For duration D** : to constraint the length of time during which a given condition holds (without interruption) to be greater than  $D$ .

For each pattern, we present a textual description and an example inspired from Fig. 4.2. The example describes the TTS system of an airlock containing two doors ( $D_1$  and  $D_2$ ) and two buttons. At any time, at most one door can be open.

Our model includes two boolean variables,  $\text{req}_1$  and  $\text{req}_2$ , indicating whether a request to open door  $D_1$  (resp.  $D_2$ ) is pending. Those variables are read by pre-conditions on transitions  $\text{Open}_i$ ,  $\text{Button}_i$ , and  $\text{Shutdown}$  and are modified by post-actions on transitions  $\text{Button}_i$  and  $\text{Close}_i$ . For instance, the pre-condition  $\neg \text{req}_2$  on  $\text{Button}_2$  is used to disable the transition when the door is already open. This implies that pressing the button while the door is open has no further effect.

Moreover, we give, for each pattern, its denotational interpretation based on First-Order formulas over Timed Traces (denoted FOTT in the following), a logical definition based on MTL and a graphical definition based on TGIL. When defining patterns, the

## 4. REAL TIME SPECIFICATION PATTERNS



**Figure 4.2:** The Airlock system

symbols  $A, B, \dots$  stand for events of Fiacre. Next, we use the symbol  $I$  as a shorthand for the time interval  $[d_1, d_2]$ . ( Note that we take into account other kind of intervals but we will limit, in this thesis, to present the closed interval due to many particular sub cases).

### 4.3.1 Existence patterns

An existence pattern is used to express that, in every trace of the system, some events must occur.

**Present  $A$  after  $B$  within  $I$**

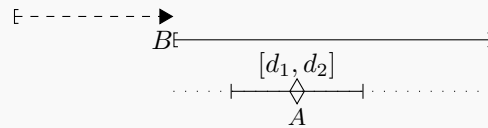
*An occurrence of predicate  $A$  must hold between  $d_1$  and  $d_2$  u.t after the first occurrence of  $B$ . The pattern is also satisfied if  $B$  never holds.*

Example: *present Ventil after  $\text{Open}_1 \vee \text{Open}_2$  within  $[0, 10]$*

MTL def.:  $(\neg B) \mathbf{W} (B \wedge \text{True} \mathbf{U}_I A)$

FOTT def.:  $\forall \sigma_1, \sigma_2 . (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 . \sigma_2 = \sigma_3 A \sigma_4 \wedge \Delta(\sigma_3) \in I$

TGIL def.:



**Present  $A$  before  $B$  within  $I$** 

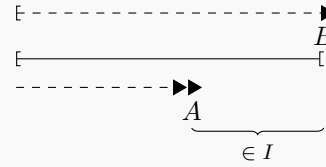
The first occurrence of predicate  $A$  holds between  $d_1$  and  $d_2$  u.t. before the first occurrence of  $B$ . The pattern is also satisfied if  $B$  never holds. (The difference with *Present  $B$  after  $A$  within  $I$*  is that  $B$  should not occur before the first  $A$ .)

Example: *present* Open<sub>1</sub>  $\vee$  Open<sub>2</sub> *before* Ventil. *within* [0, 10]

MTL def.:  $(\Diamond B) \Rightarrow ((\neg A \wedge \neg B) \cup (A \wedge \neg B \wedge (\neg B \cup_I B)))$

FOTT def.:  $\forall \sigma_1, \sigma_2. (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4. \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in I$

TGIL def.:


**Present  $A$  lasting  $D$** 

Starting from the first occurrence when the predicate  $A$  holds, it remains true for at least duration  $D$ .

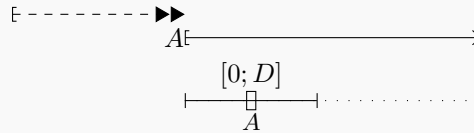
Comment: The pattern makes sense only if  $A$  is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration.

Example: *present* Refresh *lasting* 6

MTL def.:  $(\neg A) \cup (\Box_{[0,D]} A)$

FOTT def.:  $\exists \sigma_1, \sigma_2, \sigma_3. \sigma = \sigma_1 \sigma_2 \sigma_3 \wedge A \notin \sigma_1 \wedge \Delta(\sigma_2) \geq D \wedge A(\sigma_2)$

TGIL def.:


**Present  $A$  within  $I$** 

The pattern holds if there is an occurrence of predicate  $A$  holding in the interval  $I$ . This pattern can be viewed as the pattern “*present  $A$  after init within  $I$* ”.

Comment: We note by *init* the first state of the system. In the context of this work, *init* is defined as the event enter(path/value true).

### 4.3.2 Absence patterns

Absence patterns are used to express that some condition should never occur.

**Absent  $A$  after  $B$  for interval  $I$** 

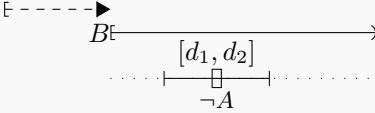
Predicate  $A$  must never hold between  $d_1$ – $d_2$  u.t. after the first occurrence of  $B$ .

Comment: This pattern is dual to *Present  $A$  after  $B$  within  $I$*  (it is not equivalent to its negation because, in both patterns,  $B$  is not required to occur).

Example: *absent* Open<sub>1</sub>  $\vee$  Open<sub>2</sub> *after* Close<sub>1</sub>  $\vee$  Close<sub>2</sub> *for interval* [0, 10]



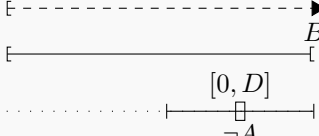
## 4. REAL TIME SPECIFICATION PATTERNS

MTL def.:	$\neg B \mathbf{W} (B \wedge \text{true} \mathbf{U} \Box_I \neg A)$
FOTT def.:	$\forall \sigma_1, \sigma_2, \sigma_3, \omega . (\sigma = \sigma_1 B \sigma_2 \omega \sigma_3 \wedge B \notin \sigma_1 \wedge \Delta(\sigma_2) \in I) \Rightarrow \neg A(\omega)$
TGIL def.:	

### Absent $A$ before $B$ for duration $D$

No  $A$  can occur less than  $D$  u.t. before the first occurrence of  $B$ . The pattern holds if there are no occurrence of  $B$ .

Example: *absent* Open<sub>1</sub> before Close<sub>1</sub> for duration 3

MTL def.:	$\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0,D]} \neg B)) \mathbf{U} B$
FOTT def.:	$\forall \sigma_1, \sigma_2, \sigma_3, \omega . (\sigma = \sigma_1 \omega \sigma_2 B \sigma_3 \wedge B \notin \sigma_1 \omega \sigma_2 \wedge \Delta(\sigma_2) \geq D) \Rightarrow \neg A(\omega)$
TGIL def.:	

### Absent $A$ for interval $I$

This pattern is equivalent to the pattern “absent  $A$  after init for interval  $I$ ”.

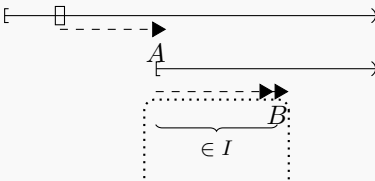
### 4.3.3 Response patterns

Response patterns are used to express “cause–effect” relationship, such as the fact that an occurrence of a first kind of events must be followed by an occurrence of a second kind of events.

### $A$ leadsto $B$ within $I$

Every occurrence of  $A$  must be followed by an occurrence of  $B$  within time interval  $I$  (considering only the first occurrence of  $B$  after  $A$ ).

Example: Button<sub>2</sub> leadsto first Open<sub>2</sub> within  $[0, 10]$

MTL def.:	$\Box(A \Rightarrow (\neg B) \mathbf{U}_I B)$
FOTT def.:	$\forall \sigma_1, \sigma_2 . (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 . \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge B \notin \sigma_3$
TGIL def.:	

**$A$  leadsto  $B$  within  $I$  before  $R$** 

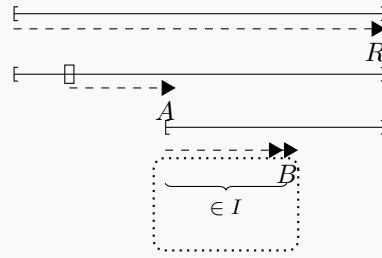
Before the first occurrence of  $R$ , each occurrence of  $A$  is followed by a  $B$ —and these two events occur before  $R$ —in the time interval  $I$ . The pattern holds if  $R$  never occur.

Example: Button<sub>2</sub> leadsto Open<sub>2</sub> within  $[0, 10]$  before Shutdown

MTL def.:  $\Diamond R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R) \mathbf{U}_I B \wedge \neg R) \mathbf{U} R$

FOTT def.:  $\forall \sigma_1, \sigma_2, \sigma_3. (\sigma = \sigma_1 A \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 A \sigma_2 \Rightarrow \exists \sigma_4, \sigma_5. \sigma_2 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in I \wedge B \notin \sigma_4$

TGIL def.:


 **$A$  leadsto  $B$  within  $I$  after  $R$** 

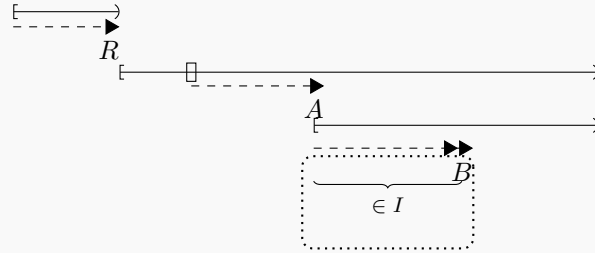
Same than with the pattern “ $A$  leadsto  $B$  within  $I$ ” but only considering occurrences of  $A$  after the first  $R$ .

Example: Button<sub>2</sub> leadsto Open<sub>2</sub> within  $[0, 10]$  after Shutdown

MTL def.:  $\Box(R \Rightarrow (\Box(A \Rightarrow (\neg B) \mathbf{U}_I B)))$

FOTT def.:  $\forall \sigma_1, \sigma_2. (\sigma = \sigma_1 R \sigma_2 A \sigma_3 \wedge R \notin \sigma_1) \Rightarrow \exists \sigma_4, \sigma_5. \sigma_3 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in I \wedge B \notin \sigma_4$

TGIL def.:



## 4.3.4 Universality patterns

Universality patterns are used to express that some condition should always occur. Universality patterns can be viewed as the dual of absence patterns.

**always  $A$  lasting  $D$** 

This pattern is equivalent to “absent  $\neg A$  after init for interval  $I$ ”.

## 4. REAL TIME SPECIFICATION PATTERNS

---

Comment: *init* represents the first state of the system.

**always  $A$  after  $B$  for duration  $D$**

*This pattern is equivalent to “absent  $\neg A$  after  $B$  for interval  $[D, D]$ ”.*

**always  $A$  before  $B$  for duration  $D$**

*This pattern is equivalent to “absent  $\neg A$  before  $B$  for duration  $D$ ”.*

### 4.3.5 Precedence patterns

Precedence patterns are used to describe a relationship between a kind of event  $P$  and an event  $S$  in which the occurrence of  $S$  must be preceded by an occurrence of  $P$  within the same scope. Precedence patterns can be viewed as a dual to Response patterns. Indeed, they describe situations in which some cause precedes each effect whereas, in the case of Response patterns, they describe that some effect follows each cause. The only difference is that Response patterns allow effects to occur without causes, whereas Precedence patterns allow causes to occur without effects.

When looking at the timed requirements found in use cases, we have noticed that Precedence patterns are never used together with a time constraints. For this reason, we have chosen not to provide a timed version of the precedence patterns.

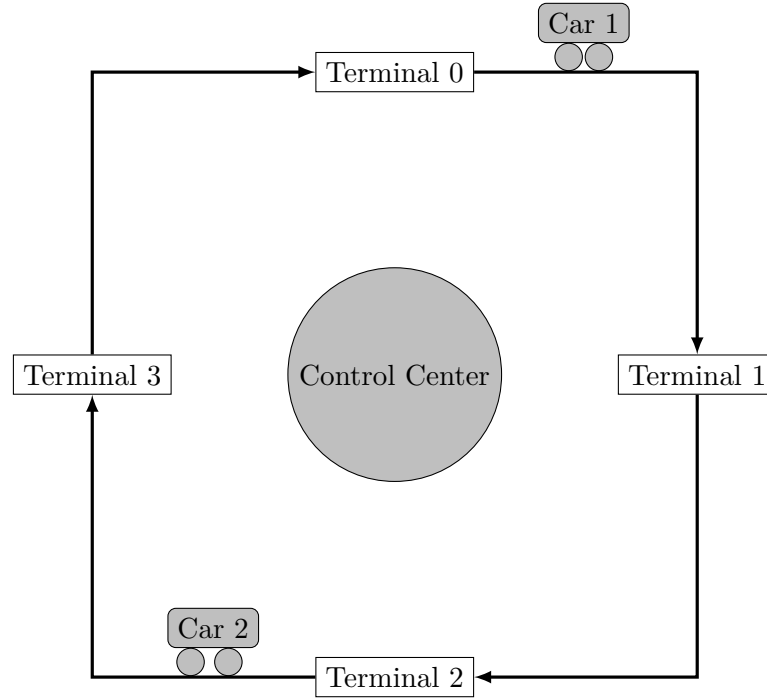
### 4.3.6 Pattern Composition

Finally, patterns can be easily combined together using the usual boolean connectives (*or*, *and*, *imply*). The pattern “ $P_1$  and  $P_2$ ” holds for all the traces where  $P_1$  and  $P_2$  both hold. The pattern “ $P_1$  or  $P_2$ ” holds for all the traces where  $P_1$  or  $P_2$  holds. The pattern “*not*  $P_1$ ” holds for all the traces where  $P_1$  does not hold.

## 4.4 Case Study: Rail car System

We demonstrate the use of our specification patterns to on a realistic use case. We borrowed the example of an automated rail car system from (DHQ<sup>+</sup>08) where it serves as an illustration of the use of timed automata patterns.

The system is composed of four terminals (numbered from 0 to 3) connected by rail tracks in a cyclic network. Several rail cars, operated from a central control center, are available to transport passengers between terminals. Each terminal is equipped with four destination buttons numbered from 0 to 3. Each button corresponds to a terminal number. Each rail car is equipped also with similar buttons representing the destinations inside the car. We illustrate the example on Fig. 4.3.



**Figure 4.3:** Rail car System

At the beginning of an execution, a rail car is immobile and it is located in terminal number 0. A passenger in a terminal who wants to travel to some destination and there is no available car, pushes the corresponding button. The request is called “external request” and it is saved by the *Control center*. When a passenger is inside the car, he can change his destination by selecting the corresponding button. The request, called “internal request”, is saved by the component *Car Panel* of the rail car.

Before moving from a terminal, each car verifies if there is an internal or external request by communicating with the *Control center* and the *Car Panel*. Before leaving a terminal, each car informs the terminal of its departure which will prepare the necessary

## 4. REAL TIME SPECIFICATION PATTERNS

---

conditions to its departure.

When a car approaches its destination, it sends a request to the terminal to signal its arrival. After processing a request, the car sends an acquittement to the responsible component: *Control center* in the case of external request and *Car Panel* in the case of internal request.

This system has several real time constraints: the terminal must be ready to accommodate an incoming car in 5 s; a car arriving in a terminal leaves its door open for exactly 10 s; passengers entering a car have 5 s to choose their destination; etc.

We have modelled this system using Fiacre. The system is composed of four main components: *Terminal*, *Control Center*, *Rail Car* and *Passenger*. The component *Rail Car* is composed of three processes: *Car Handler*, *Car Panel* and *Car Door*. The components are executed together and communicate through ports and shared variables.

```
process Terminal [ service:none, depart_req:num_terminal,depart_ack:num_terminal,
                  apporach_ack:num_terminal, apporach_req:num_terminal,
                  enter_req_term:num_terminal] (id:num_terminal) is

  states idle, prep_depart_terminal, prep_approach_terminal,
          approach_terminal_ack, depart_terminal_ack

  var x:num_terminal, tb:tab:=[false,false,false,false]

  from idle
    select
      enter_req_term?x where x=id; tb[id]:=true; to idle
    [] service?x where x=id; tb[id]:=false; to idle
    [] depart_req?x where x=id; to prep_depart_terminal
    [] apporach_req?x where x=id; to prep_approach_terminal
    end

  from prep_depart_terminal
    wait [5,5];
  to depart_terminal_ack

  from depart_terminal_ack
    depart_ack!id;
  to idle

  from prep_approach_terminal
    wait [5,5];
  to approach_terminal_ack

  from approach_terminal_ack
    apporach_ack!id;
  to idle
```

**Figure 4.4:** A Rail car system example in Fiacre: The process Terminal

We present the behaviour of a terminal in Fig. 4.4. Each Terminal controls the entering and the leaving of a car by preparing the necessary conditions.

```

process Control[enter_req : num_terminal, enter_req_term : num_terminal,
                ext_sched : none, tstop : none, ext_sched_ack : none,
                tstop_ack : none, service : num_terminal]
    (&dest : num_terminal, &curr : num_terminal, &tostop : bool) is

    states idle, ext_schedule, ext_check, send_ack_terminal, send_terminal

    var eb:tab:=[false, false, false, false], x:num_terminal

    from idle
        select
            enter_req?x; eb[x]:=true; to send_terminal
        [] ext_sched; x:=curr; to ext_schedule
        [] tstop; to ext_check
        end

    from send_terminal
        enter_req_term!x;
    to idle

    from ext_schedule
        if eb[x%3]=false then
            if x<3 then
                x:=x+1
            else
                x:=0
            end; to ext_schedule
        else
            if dest<>x then
                dest:=x;
            end
        ext_sched_ack; to idle
        end

    from ext_check
        tostop:=eb[curr];
        eb[curr]:=false;
        tstop_ack;
    to send_ack_terminal

    from send_ack_terminal
        service!curr;
    to idle

```

**Figure 4.5:** A Rail car system example in Fiacre: The process Control

The Control center, presented in Fig. 4.5, manages external requests. However, process Car Panel manages internal requests (see Fig. 4.6). We present also in Fig. 4.6 the behaviour of the process passenger. We present the behaviour of a car in Fig. 4.7 by showing the behaviour of a Car Handler process and a Car Door process (Fig. 4.6).

The key requirements of the rail car system are as follows:

#### 4. REAL TIME SPECIFICATION PATTERNS

---

```

process CarPanel [int_request:num_terminal, int_sched:none, cstop:none,
                  int_sched_ack:none, cstop_ack:none]
                  (&dest:num_terminal,&curr:num_terminal,&tostop:bool) is

  states idle , int_schedule , int_check

  var i:num_terminal:=0,x:num_terminal,b:tab:=[ false , false , false , false ]

  from idle
  select
    int_request?i; b[i]:=true; to idle
  [] int_sched; x:=curr;to int_schedule
  [] cstop; to int_check
  end

  from int_schedule
  if b[x%3]=false then
    if x<3 then
      x:=x+1
    else
      x:=0
    end; to int_schedule
  else
    dest:=x; int_sched_ack; to idle
  end

  from int_check
  tostop:=b[curr];
  b[curr]:=false;
  cstop_ack;
  to idle

process CarDoor [opendoor:none, conf:none, closedoor:none] is

  states close , toOpen , open

  from close opendoor; to toOpen

  from toOpen conf; to open

  from open closedoor; to close

process Passenger [enter_req:num_terminal, int_request:num_terminal]
                  (num:num_terminal) is

  states idle , send_req , send_ext_req , fin

  from idle wait[5,5]; to send_req

  from send_req int_request!num; to send_ext_req

  from send_ext_req enter_req!num; to fin

```

**Figure 4.6:** A Rail car system in Fiacre: The process Car Panel, Car Door and Passenger

- *P1* : when a passenger arrives in a terminal, he must have a car ready to transport him within 15 s. This property can be expressed with a response

## 4.4 Case Study: Rail car System

```

process CarHandler [int_sched:none, int_sched_ack:none, ext_sched:none,
                    ext_sched_ack:none, depart_req:num_terminal, opendoor:none,
                    depart_ack:num_terminal, apporach_req:num_terminal,
                    apporach_ack:num_terminal, tstop:none, tstop_ack:none,
                    cstop:none, cstop_ack:none, conf:none, closedoor:none]
                    (&dest:num_terminal, &curr:num_terminal, &tostop:bool) is

    states idle, carIntSchedule, wait_carIntSchedule_ack, verify_ext_req,
            wait_carExtSchedule_ack, moving, wait_depart_req_ack, car_approach,
            wait_approach_req_ack, carExtCheck, wait_carExtCheck_ack, carIntCheck,
            wait_carIntCheck_ack, wait_conf, wait_close, wait_door

    var i:num_terminal:=0, x:num_terminal

    from idle wait [9,9]; to carIntSchedule

    from carIntSchedule int_sched; to wait_carIntSchedule_ack

    from wait_carIntSchedule_ack int_sched_ack; to verify_ext_req

    from verify_ext_req ext_sched; to wait_carExtSchedule_ack

    from wait_carExtSchedule_ack ext_sched_ack; to moving

    from moving depart_req!curr; to wait_depart_req_ack

    from wait_depart_req_ack
        depart_ack?i where i=curr;
        if curr<3 then curr:=curr+1
        else curr:=0 end;
    to car_approach

    from car_approach apporach_req; to wait_approach_req_ack

    from wait_approach_req_ack apporach_ack?x where x=curr; to carExtCheck

    from carExtCheck tstop; to wait_carExtCheck_ack

    from wait_carExtCheck_ack tstop_ack; to carIntCheck

    from carIntCheck cstop; to wait_carIntCheck_ack

    from wait_carIntCheck_ack cstop_ack; to wait_door

    from wait_door opendoor; to wait_conf

    from wait_conf conf; to wait_close

    from wait_close closedoor; to idle

```

**Figure 4.7:** A Rail car system in Fiacre: The process Car Handler

pattern: Passenger/sendReq *leadsto* Control/ackTerm *within* [0,15], where sndReq is the state where the passenger chooses his destination and Control/ackTerm is the state where it is served.



## 4. REAL TIME SPECIFICATION PATTERNS

---

- *P2* : when the car starts moving, the door must be closed: *present* Car Door/-closeDoor *after* Car Handler/moving *within* [0,10]. This pattern states that when the car prepares for moving (it enters the state **moving**) we must see the event closeDoor within at most 10 s.
- *P3* : when a passenger select a destination (in the car), the signal should stay illuminated until the car is arrived: *absent* Terminal/buttonOff *before* Control/ackTerm *for duration* 10, where Terminal/buttonOff is the state where the signal is turned off and Control/ackTerm is the state where the car reach its destination.

We will use this use case and these properties in the experiments described in Chapter 6.

Next, we describe a possible meta model for encoding our pattern language.

### 4.5 Meta Model

We present in Fig. 4.8 the corresponding Meta Model of our specification patterns. Each specification pattern in our catalog is related to a Fiacre property. We can define atomic or composed pattern using scopes like *or*, *and*, *imply*. We classify the patterns into two main categories: *occurrence patterns* and *order patterns*. Each pattern is composed of Fiacre events and scopes and it is associated to a time requirements.

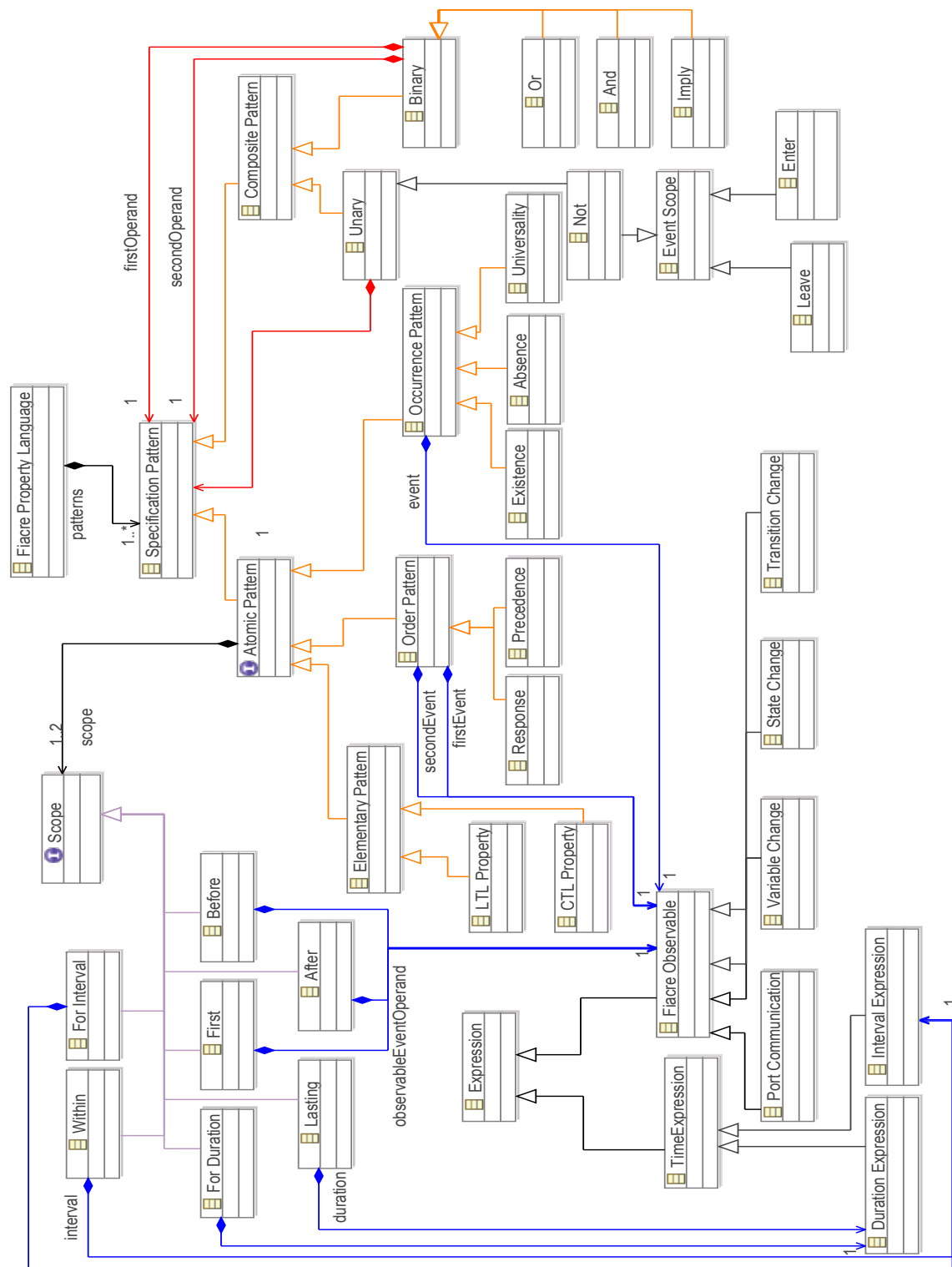
This meta-model—defined using the Eclipse Modeling Framework (EMF)—has been used in order to connect or verification toolchain with the output of language transformations from higher-level modeling language. In particular, in the scope of the Quartet project, we have used this meta-model as a target for the transformation of high-level requirements expressed on a AADL model.

### 4.6 Conclusion

We have defined a set of high-level specification patterns for expressing requirements on systems with hard real time constraints. Our patterns are simple to use since they are based on natural language. Moreover, we are based on Dwyer's patterns. A recent study (BGPS12) has shown that Dwyer's patterns are the most used on practice. So we are sure about the utility of our requirements on practice. Compared to temporal logics, our catalog of real time patterns is certainly less expressive, but it permits to

represent requirements that are the most used in practice and to check them using a decidable framework. We believe that our notation is interesting in its own right and can be reused in different contexts.

There are several directions for future works. Most particularly, we plan to define a compositional patterns inspired by the “denotational interpretation” used in the definition of patterns. The idea is to define a lower-level pattern language, with more composition operators, that is amenable to an automatic translation into a verification method.



**Figure 4.8:** The Meta Model of Real Time Specification Patterns

## Chapter 5

# Checking Patterns Using TTS Observers

*“Doing your best is more important than being the best.”*

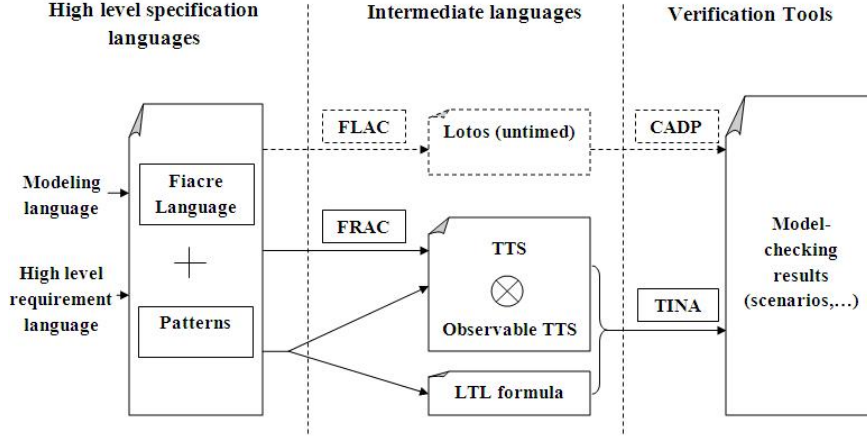
*-Shannon Miller-*

We present a verification method, based on the use of TTS order, for the verification of real time specification patterns. The chapter is organised as follow. In Section 5.1, we present the set of observers defined to verify timed patterns. To compare the proposed verification methods, we present some experimentations in Section 5.2. We prove the correctness of our observers in Section 5.3. We present next the corresponding observer for each pattern of our catalog in Section 5.4. Section 5.5 concludes this Chapter.

While the use of observers in model checking is quite common, our approach is original in several ways. First, we define for each pattern a set of observers. The goal is to provide the best observer in practice in terms of verification time and system space size. We are not based on a method to generate observers rather we look for each pattern to come up with the best candidate in practice.

Second, we propose a decidable verification method for checking real time patterns on Time Transition Systems (TTS). By using observers based approach, we transform the verification of patterns into the verification of simpler LTL formula. Before adding an observer in our framework, we verify its correctness using a formal framework. Our observers are proved correct and non-intrusive, meaning that they compute the correct answer and have no impact on the system under observation. The formal framework we have defined is not only useful for proving the validity of formal results but also to check the soundness of optimisation in the implementation.

## 5. CHECKING PATTERNS USING TTS OBSERVERS



**Figure 5.1:** The global verification tool chain

Finally, we provide a reference implementation for these timed patterns. The complete framework defined in this work has been integrated into a verification tool chain for Fiacre (BBF<sup>+</sup>08), a high-level modelling language that can be compiled to TTS. In our tool chain (see Fig. 5.1) a Fiacre specification is combined with patterns and compiled into a TTS model using the Frac compiler. Then the model can be checked using the TINA toolbox. This is not a toy example. Indeed, Fiacre is the intermediate language used for model verification in Topcased (FGC<sup>+</sup>06), an Eclipse based toolkit for critical systems, where it is used as the target of model transformation engines from various languages, such as SDL, BPEL or AADL (FGH06). Therefore, through the connection with Fiacre, we can check timed patterns on many different modelling languages.

### 5.1 Observers at the TTS level

We define different types of observers at the TTS level that can be used for the verification of patterns. It is important to note that we do not give an automatic method to generate observers. Rather, we define a set of observers for each patterns and, after selecting the “most efficient one”, we prove that it is correct (see the discussion in Sect. 5.3). We make use of the whole expressiveness of the TTS model to build observers: synchronous or asynchronous rendez-vous (through places and transitions); shared memory (through data variables); and priorities. We believe that an automatic method for generating

the observer, while doable, will be detrimental for the performance of our approach. Moreover, when compared to a “temporal logic” approach, we are in a more favourable situation because we only have to deal with a finite number of patterns.

### 5.1.1 Observers for the Leadsto Pattern

We focus on the example of the *leadsto* pattern. We assume that some events of the system are labeled with  $E_1$  and some others with  $E_2$ . In our context, the event of a model can be: a transition that is fired; the system entering or leaving a state; a change in the value of variables; ... (see Section 4.2 Chapter 4 for a description of events). We give three examples of observers for the pattern:  $E_1$  *leadsto*  $E_2$  within  $[0, max[$ . The first observer monitors transitions and uses a single place; the second observer monitors places; the third observer monitors shared, boolean variables injected into the system (by means of composition). We define our TTS observers using a classical graphical notation for Petri Nets, where arcs with a black circle denote *read arcs*, while arcs with a white circle are *inhibitor arcs*. (These extra categories of arcs can be defined in TTS and are supported in our tool chain.) The use of a *data observer* is quite new in the context of TTS systems. The results of our experiments seem to show that, in practice, this is the best choice to implement an observer.

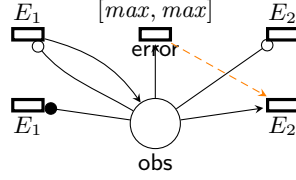
#### 5.1.1.1 Transition Observer

The observer  $O_t$ , see Fig. 5.2, uses a place, *obs*, to record the time since the last transition  $E_1$  occurred. The place *obs* in  $O_t$  is emptied if a transition labeled  $E_2$  is fired, otherwise the transition *error* is fired after *max* unit of time. The priority arc (dashed arrow) between *error* and  $E_2$  is used to observe the transition *error* even in the case where a transition  $E_2$  occurs exactly *max* u.t. after the place *obs* was filled.

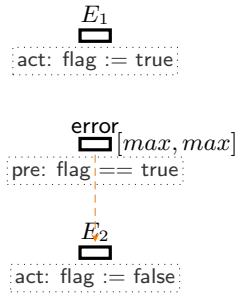
By definition of the TTS composition operator, the composition of the observer  $O_t$  with the system  $N$  duplicates each transitions in  $N$  that is labeled  $E_1$ : one copy can fire if *obs* is empty—as a result of the inhibitor arc—while the other can fire only if the place is full. As a consequence, in the TTS  $N \parallel O_t$ , the transition *error* can fire if and only if the place *obs* stays full—there has been an instance of  $E_1$  but not of  $E_2$ —for a duration of *max*. Then, to prove that  $N$  satisfies the *leadsto* pattern, it is enough to check that the system  $N \parallel O_t$  cannot fire the transition *error*. This can be done by checking the LTL formula  $\Box(\neg \text{error})$  on the system  $N \parallel O_t$ .

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---



**Figure 5.2:** Transition Observer:  $O_t$



**Figure 5.3:** Data Observer:  $O_d$

The observer  $O_t$  given in Fig. 5.2 is *deterministic* and will “react” to the first occurrence of  $E_2$  that miss a deadline. It is also possible to define a non-deterministic observer, such that some occurrences of  $E_1$  or  $E_2$  may be disregarded. This approach is safe since model-checking performs an exhaustive exploration of the states of the system; it considers all possible scenarios. This non-deterministic behaviour is quite close to the treatment obtained when compiling an (untimed) LTL formula “equivalent” to the *leadsto* pattern, namely  $\Box(E_1 \Rightarrow \Diamond E_2)$ , into a Büchi automaton (GO01). We have implemented the deterministic and non-deterministic observers and compared them taking into account their impact on the size of the state graphs that need to be generated and on the verification time. Experiments have shown that the deterministic observer is more efficient, which underlines the benefit of singling out the best possible observer and looking for specific optimisation.

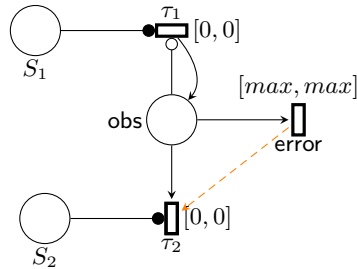
### 5.1.1.2 Data Observer

We define the data observer  $O_d$  in Fig. 5.3. The data observer has a transition *error* conditioned by the value of a boolean variable, *flag*, that “takes the role” of the place

$\text{obs}$  in  $O_t$  (every boolean variable is considered to be initially set to false). Indeed,  $\text{flag}$  is true between an occurrence of  $E_1$  and the following transition  $E_2$ . Therefore, like in the previous case, to check if a system  $N$  satisfies the pattern, it is enough to check the reachability of the event  $\text{error}$ . Notice that the whole state of the data observer is encoded in its store, since the underlying net has no place.

### 5.1.1.3 Place Observer

We define the place observer  $O_p$  in Fig. 5.4. In this section, to simplify the presentation, we assume that the events  $E_1$  and  $E_2$  are associated to the system entering some given states  $S_1$  and  $S_2$ . (But we can easily adapt this net to observe events associated to transitions in the system.) We also rely on a composition operator that composes TTS through their places instead of their transitions (PBV11) and that is available in our tool chain. In  $O_p$ , we use a transition labeled  $\tau_1$  whenever a token is placed in  $S_1$  and a transition  $\tau_2$  for observing that the system is in state  $S_2$  (we assume that the labels  $\tau_1$  and  $\tau_2$  are fresh—private to the observer—and should not be composed with the observed systems). The remaining component of  $O_p$  is just like the transition observer. We consider both a place and a transition observer since, depending on the kind of events that are monitored, one variant may be more efficient than the other.



**Figure 5.4:** Place Observer:  $O_p$

## 5.2 Experimentation

Our verification framework has been integrated into a prototype extension of *frac*, the Fiacre compiler for the TINA toolbox. This extension supports the addition of real time patterns and automatically compose a system with the necessary observers. (Software



## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

and examples are available at <sup>1</sup>.) In case the system does not meet its specification, we obtain a counter-example that can be converted into a timed sequence of events exhibiting a problematic scenario. This sequence can be played back using two programs provided in the TINA tool set, *nd* and *play*. The first program is a graphical animator for Time Petri Net, while the latter is an interactive (text-based) animator for the full TTS model.

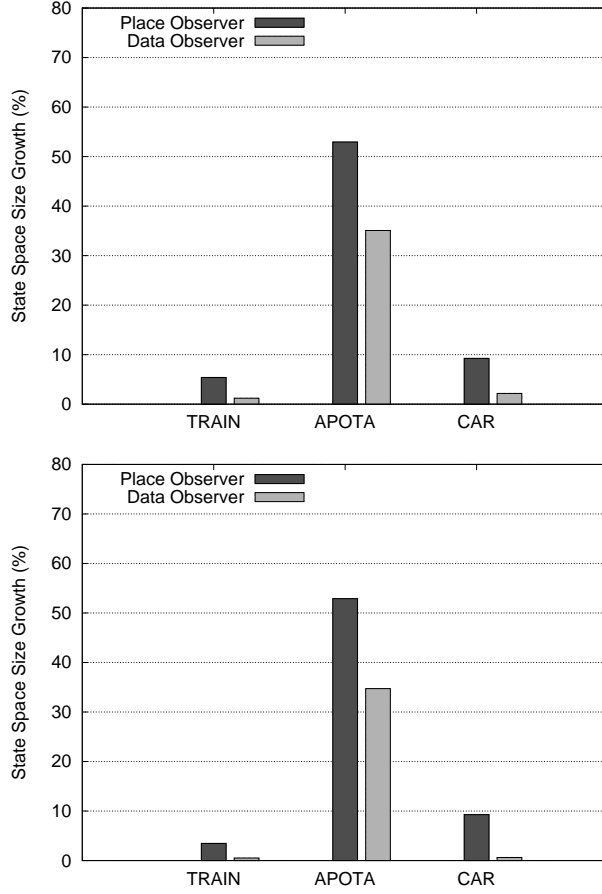
We define the *empirical complexity* of an observer as its impact on the augmentation of the state space size of the observed system. For a system  $S$ , we define  $size(S)$  as the size (in bytes) of the *State Class Graph* (SCG) (BRV04) of  $S$  generated by our verification tools. In TINA, we use SCG as an abstraction of the state space of a TTS. State class graphs exhibit good properties: an SCG preserves the set of discrete traces—and therefore preserves the validation of LTL properties—and the SCG of  $S$  is finite if the Petri Nets associated with  $S$  is bounded and if the set of values generated from  $S$  is finite. We cannot use the “plain” labeled transition system associated to  $S$  to define the size of  $S$ ; indeed, this transition graph maybe infinite since we work with a dense time model and we have to take into account the passing of time.

The size of  $S$  is a good indicator of the memory footprint and the computation time needed for model-checking the system  $S$ : the time and space complexity of the model-checking problem is proportional to  $size(S)$ . Building on this definition, we say that the complexity of an observer  $O$  applied to the system  $S$ , denoted  $C_O(S)$ , is the quotient between the size of  $(S \parallel O)$  and the size of  $S$ .

We resort to an empirical measure for the complexity since we cannot give an analytical definition of  $C_O$  outside of the simplest cases. However, we can give some simple bounds on the function  $C_O$ . First of all, since our observers should be non-intrusive, we can show that the SCG of  $S$  is a sub graph of the SCG of  $S \parallel O$ , and therefore  $C_O(S) \geq 1$ . Also, in the case of the *leadsto* pattern, the transitions and places-based observers add exactly one place to the net associated to  $S$ . In this case, we can show that the complexity of these two observers is always less than 2; we can at most double the size of the system. We can prove a similar upper bound for the *leadsto* observer based on data. While the three observers have the same (theoretical) worst-case complexity, our experiments have shown that one approach was superior to the others. We are not aware of previous work on using experimental criteria to

---

<sup>1</sup><http://homepages.laas.fr/nabid>



**Figure 5.5:** Compared complexity of the data and place observers (in percentage of system size growth)—average time for invalid properties (above) and valid properties (below).

select the best observer for a real time property. In the context of “untimed properties”, this approach may be compared to the problem of optimising the generation of Büchi Automata from LTL formulas, see e.g. (GO01).

We have used our prototype compiler to experiment with different implementations for the observers. The goal is to find the most efficient observer “in practice”, that is the observer with the lowest complexity. To this end, we have compared the complexity of different implementations on a fixed set of representative examples and for a specific set of properties (we consider both valid and invalid properties). The results for the *leadsto* pattern are displayed in Fig. 6.2. For the experiments used in this paper, we will limit to present three examples of systems selected because they exhibit very different features (size of the state space, amount of concurrency and symmetry in the system,

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

...):

- TRAIN is a model of a train gate controller. The example models a system responsible for controlling the barriers protecting a railroad crossing gate. When a train approaches, the barrier must be lowered and then raised after the train's departure. The valid property, for the TRAIN example, states that the delay between raising and lowering a barrier does not exceed 100 unit of time. For the invalid property, we use the same requirement, but shortening the delay to 75.
- APOTA is an industrial use case that models the dynamic architecture for a network protocol in charge of data communications between an air plane and ground stations (BBDZ<sup>+</sup>10). This example has been obtained using a translation from AADL to Fiacre. In this case, timing constraints arise from timeouts between requests and periods of the tasks involved in the protocol implementation. The property, in this case, is related to the worst-case execution time for the main application task.
- CAR is a system modelling an automated rail car system taken from (DHQ<sup>+</sup>08). The system is composed of four terminals connected by rail tracks in a cyclic network. Several rail cars, operated from a central control centre, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to the terminal to signal its arrival. Passengers in the terminal can then book a travel in the car. The CAR example has been presented in Section 4.4 Chapter 4. This example has been modelled using The valid property, for the CAR example, states that a passenger arriving in a terminal, must have a car ready to transport him within 15 unit of time. For the invalid property, we use the same requirement, but shortening the delay to 2 unit of time.

In Fig. 6.2, we compare the growth in the state space size—that is the value of  $C_o(S)$ —for the place and data observers defined in Section. 5.1.1 and our three running examples. We do not consider the transition observer in these results since the events used in the requirements are all related to a system entering a state (and therefore our benchmark favor the place observer over the transition observer). We use one chart to display the result for patterns that are invalid and another for valid patterns.

In Fig. 5.6, we give results on the total verification time for the APOTA example. The value displayed in the table refer to the time spent generating the complete state space of the system and verifying the property. All experimentations have been done using Fedora 11 exploitation system with 2G RAM and an Intel Core 2 Duo CPU.

The row SYSTEM gives the time needed for exploring the complete state space of the system (without adding any observer) while “VALID” and “INVALID” refer to the state space of the system synchronised with data observer and state observer in the case of valid and invalid property respectively.

In our experiments, we have consistently observed that the observer based on data is the best choice; it is the observer giving the minimal execution time in almost all the cases and that seldom gives the worst result. We explain the efficiency of the data observer in Section 5.2.1.

Example	State observer	Data observer
SYSTEM	2.861	2.861
VALID	11.662	10.652
INVALID	11.611	10.179

**Figure 5.6:** Total verification time for APOTA (in seconds)

### 5.2.1 Experiment analysis: why Data observer is the best in practice?

Our experiments allow us to say that, when we compare data, state and transition observer in terms of verification time and size of space system, data observer outperforms state and transition observers. We explain the efficiency of data observer by the fact that it is based on shared variables to verify properties. Hence, it does not add additional state to the state space generated. However, state observer is presented as a “net observer”. Thus, it could be as big as the net if the verification of the property includes markings. Hence, the size of the system may be squared.

We have presented a simple example in order to show how data observer performs better than state observer. We have tested both the case of valid and invalid property as well as controlled the evolution of the state space. We have also tested the case of deterministic—meaning that the observed events involve deterministic transitions in the system—and non deterministic properties—meaning that the observed events may involve non deterministic transitions in the system. Experiments have shown that, in

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

both cases, data observer adds less states to the system state space. In the rest of this section, we present only the experiment corresponding to the deterministic case.

The example describes the behaviour of a simple communication protocol. The system is composed of a sender and a receiver. The sender sends a request to the receiver and must have an acquittement before sending a new request. We present in Fig 5.7 the Fiacre specification of this example.

```

process Emetteur [t1: none, t2: none] is

    states P1, P2

    from P1
    t1;
    to P2

    from P2
    t2;
    to P1

process Receveur [t1: none, t2: none] is

    states P9, P6

    from P9
    t1;
    to P6

    from P6
    t2;
    to P9

component Main is

    port t1: none in [2,2],
          t2: none in [0,1]

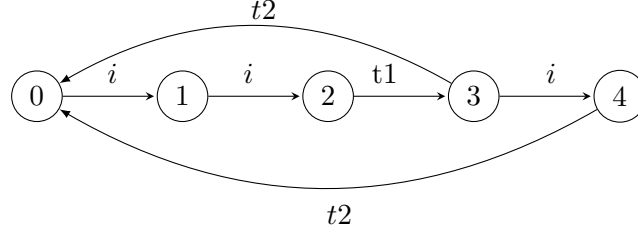
    par * in
        Emetteur [t1,t2]
    ||
        Receveur [t1,t2]

    end
Main

```

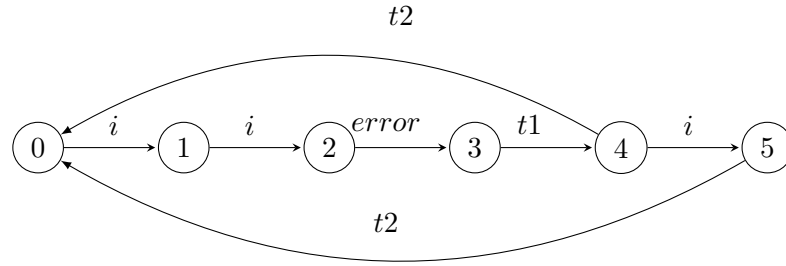
**Figure 5.7:** A Protocol example in Fiacre

Figure 5.8 depicts the state space of the system without properties. From the state 0 the system waits for 2 u.t (presented by the letter *i*) and then moves to state 2. The port *t1* is involved after that and the system moves to state 3. From state 3, the system waits for 1 u.t and then executes the transition corresponding to port *t2* and moves to state 0.

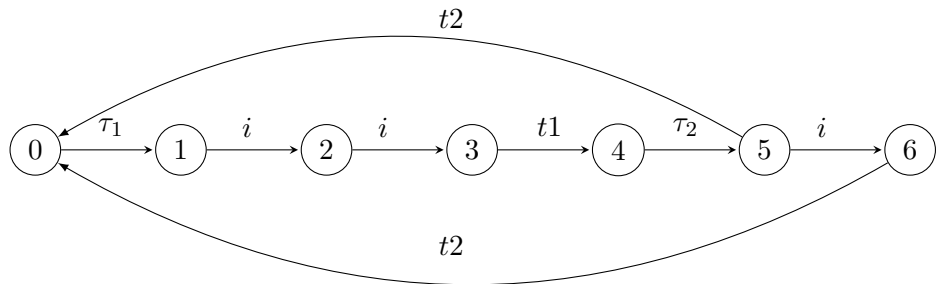


**Figure 5.8:** State space graph for Protocol example

For the verified case, we check that the delay between sending data and receiving an acquitement is less than 7 u.t. The property is defined as “property response is main/1/state P1 leadsto main/1/state P2 within [7,7]”. However, for the invalid case, we have reduced the interval to [2,2].



**Figure 5.9:** State space graph of invalid properties for Protocol example with data observer

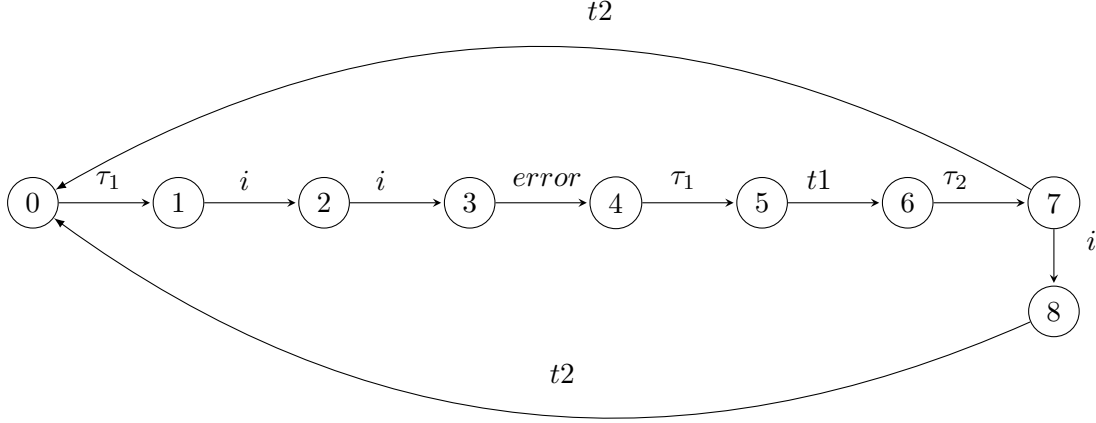


**Figure 5.10:** State space graph of valid properties for Protocol example with state observer

In the case of verified property checked using data observer, we notice that the state

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---



**Figure 5.11:** State space graph of invalid properties for Protocol example with state observer

space is the same than system without property (see Fig 5.8). The only difference is the addition of shared variables which do not increase the state space and do not appear in the figure because they are added to existing states. In the case of invalid property checked using data observer, we notice that the system adds a state corresponding to the “error” transition (see Fig. 5.9). “error” transition is the only transition of the data observer.

In the case of state observer, the state space generated when checking the case of verified property is shown in Fig. 5.10 and the state space generated when checking invalid property is shown in Fig. 5.11. We notice that the system adds a state corresponding to every transition of the state observer.

To conclude, we noticed that the system create a special state for every transition of the observers. Data observer has one transition added when the property is not verified. However, the state observer has three transitions. Hence, if the system involves the invalid property  $n$  time then the state space will increase by a factor of 3 in the case of states observer for each observed events.

### 5.3 Proving the correctness of TTS observers

We start by giving sufficient conditions for an observer  $O$  to be *non-intrusive*, meaning that the observer does not interfere with the observed system. Formally, we show that any trace  $\sigma$  of the observed system  $N$  is preserved in the composed system  $N \parallel O$ : the observer does not obstruct a behaviour of the system (see Lemma 1 below). Conversely, we show that, from any trace of the composition  $N \parallel O$ , we can obtain a trace of  $N$  by erasing the events from  $O$ : the observer does not add new behaviours to the system. This is actually a consequence of Property 1 (see Sect. 3.2.3 Chapter 3).

Let  $\Sigma(N)$  be the set of well-formed traces of the TTS  $N$ . We write  $T_{sync}$  the set of synchronised transitions of the observer, that is, the set of transitions  $t$  of  $O$  such that  $L(t) = L(t')$  for some transitions  $t'$  of  $N$ . Given a state  $\rho$  and a finite trace  $\sigma$ , we write  $\rho \xrightarrow{\sigma}$  to indicate that  $\sigma$  is a valid trace for  $O$  in state  $\rho$ . Then,  $O(\rho, \sigma)$  is the state reached after executing trace  $\sigma$ . We use  $O(\sigma)$  as a shorthand for  $O(\rho_{init}, \sigma)$ , where  $\rho_{init}$  is the initial state. The following lemma states sufficient conditions for the observer to be non-intrusive.

**Lemma 1.** *Assume  $O$  satisfies the following conditions:*

- *For every  $t$  in  $T_{sync}$ ,  $I(t) = [0; +\infty[$*
- *In every reachable state  $\rho$  of  $O$ , and for every  $l$  in  $L(T_{sync})$ , there exists a (possibly empty) finite trace  $\sigma$  not containing transitions in  $T_{sync}$  such that  $\Delta(\sigma) = 0$  and there exists  $t$  with  $L(t) = l$ , which is fireable in  $O(\rho, \sigma)$ .*
- *There exists  $\delta > 0$  such that, in every reachable state  $\rho$  of  $O$ , there exists a (possibly empty) finite trace  $\sigma$  not containing transitions in  $T_{sync}$  with  $\rho \xrightarrow{\sigma D(\delta)}$ .*

*Then for all  $\sigma_1$  in  $\Sigma(N)$  there exists  $\sigma_2$  in  $\Sigma(O)$  such that  $\sigma_1 \bowtie \sigma_2$ .*

As a consequence, by Property 1,  $(\sigma_1, \sigma_2)$  belongs to  $\Sigma(N \parallel O)$ , which means that the original behaviour  $\sigma_1$  is preserved in the composed system.

*Proof.* We assume  $O$  satisfies the given hypotheses. We have to build a trace  $\sigma_2$  in  $\Sigma(O)$  such that  $\sigma_1 \bowtie \sigma_2$ . Since traces are considered up to equivalence, we actually build two traces  $\sigma_2$  and  $\sigma_3$  such that both  $\sigma_2 \bowtie \sigma_3$  and  $\sigma_1 \equiv \sigma_3$  hold:

*For all  $\sigma_1$  in  $\Sigma(N)$ , there exist  $\sigma_2$  in  $\Sigma(O)$  and  $\sigma_3$  in  $\Sigma(N)$  such that  $\sigma_1 \equiv \sigma_3$ , and for all  $t > 0$ , for all  $\sigma_3^a$  finite prefix of  $\sigma_3$  with  $\Delta(\sigma_3^a) < t$ , there exists  $\sigma_2^a$  finite prefix of  $\sigma_2$  such that  $\sigma_2^a \bowtie \sigma_3^a$  holds.*



## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

We provide an algorithm  $f$  that builds  $\sigma_2$  and  $\sigma_3$  incrementally.

- The inputs of  $f$  are  $\sigma_1$ ,  $\sigma_1^a$ ,  $\sigma_2^a$  and  $\sigma_3^a$ . They must satisfy the following:  $\sigma_1 \equiv \sigma_3^a \sigma_1^a$  holds, as well as  $\sigma_2^a \bowtie \sigma_3^a$ . Intuitively,  $\sigma_3^a$  is the part of  $\sigma_1$  (up to equivalence) that has already been done, whereas  $\sigma_1^a$  is the part remaining to be considered.
- The algorithm returns a new triple  $\sigma_1^b$ ,  $\sigma_2^b$  and  $\sigma_3^b$ , which satisfies similar conditions, and such that  $\sigma_2^a$  and  $\sigma_3^a$  are prefixes of  $\sigma_2^b$ , and  $\sigma_3^b$  respectively.
- The algorithm is invoked iteratively with the returned triple. In the finite case (when  $\sigma_1$  is finite), it eventually reaches a point where  $\sigma_1^a$  is empty, in which case  $\sigma_1 \equiv \sigma_3^a$  holds. In the infinite case, we take  $\sigma_2$  as the limit of  $\sigma_2^b$ , and  $\sigma_3$  as the limit of  $\sigma_3^b$ .

We now provide the details of the algorithm, then we consider its soundness, being carefull with respect to well-formedness conditions (checking in particular that the algorithm does not pile up infinite sequences of zero-delay events).

**Algorithm**  $f(\sigma_1, \sigma_1^a, \sigma_2^a, \sigma_3^a)$  : let  $x$  and  $\sigma_1'$  be such that  $\sigma_1^a$  equals  $x.\sigma_1'$ . With no loss of generality, we may freely assume that  $x$  is not  $d(0)$ . We proceed by case on  $x$ :

- If  $x$  is an event  $(t, m, s)$  with  $t \notin T_{sync}$ , then we return  $\sigma_1^b = \sigma_1'$ ,  $\sigma_2^b = \sigma_2^a.d(0)$ , and  $\sigma_3^b = \sigma_3^a.x$ .
- If  $x$  is an event  $(t, m, s)$  with  $t \in T_{sync}$ , then by hypothesis on  $O$ , there exists a finite trace  $\sigma$  not containing transitions in  $T_{sync}$  such that  $\Delta(\sigma) = 0$ ,  $\sigma_2^a \sigma \in \Sigma(O)$  and  $t'$  is fireable in  $O(\sigma_2^a \sigma)$  with  $L(t') = L(t)$ . Let  $\sigma'$  be the sequence with the same length as  $\sigma$  and whose elements are  $d(0)$ . Let  $\omega$  be  $(t', m', s')$  where  $m'$  and  $s'$  are an appropriate marking and state such that  $\sigma_2^a.\sigma.\omega$  is in  $\Sigma(O)$ . Then, we return  $\sigma_1^b = \sigma_1'$ ,  $\sigma_2^b = \sigma_2^a.\sigma.\omega$ , and  $\sigma_3^b = \sigma_3^a.\sigma'.x$ .
- If  $x$  is  $d(\delta)$ , then by hypothesis on  $O$ , there exists  $\delta_2 > 0$  and a finite trace  $\sigma$  not containing transitions in  $T_{sync}$  such that  $\sigma_2^a.\sigma.d(\delta_2)$  is in  $\Sigma(O)$ . Let  $\sigma'$  be the sequence with the same length as  $\sigma$  and whose elements are  $d(0)$  or  $d(\delta'_i)$ , with appropriate  $\delta'_i$  such that  $\sigma' \bowtie \sigma$  holds. We distinguish two sub cases:

### 5.3 Proving the correctness of TTS observers

- (i) either  $\delta \leq \Delta(\sigma) + \delta_2$ , in which case we return  $\sigma_1^b = \sigma'_1$ ,  $\sigma_2^b = \sigma_2^a \cdot \sigma \cdot d(\delta - \Delta(\sigma))$  and  $\sigma_3^b = \sigma_3^a \cdot \sigma' \cdot d(\delta - \Delta(\sigma))$  provided  $\delta - \Delta(\sigma) \geq 0$ . If  $\delta - \Delta(\sigma) < 0$ , we have to truncate both  $\sigma$  and  $\sigma'$  at duration  $\delta$  (omitting the details).
- (ii) either  $\Delta(\sigma) + \delta_2 < \delta$ , in which case we return  $\sigma_1^b = d(\delta - \delta_2) \cdot \sigma'_1$ ,  $\sigma_2^b = \sigma_2^a \cdot \sigma \cdot d(\delta_2)$ , and  $\sigma_3^b = \sigma_3^a \cdot \sigma' \cdot d(\delta_2)$ .

There is no difficulty in checking that, for each case, the returned triple satisfies the output conditions, that is,  $\sigma_1 \equiv \sigma_3^b \cdot \sigma_1^b$  and  $\sigma_2^b \bowtie \sigma_3^b$ , as long as  $\sigma_1^a$ ,  $\sigma_1^b$ , and  $\sigma_1^c$  satisfy the input conditions, that is,  $\sigma_1 \equiv \sigma_3^a \cdot \sigma_1^a$  and  $\sigma_2^a \bowtie \sigma_3^a$ .

This implies that  $\sigma_2 \bowtie \sigma_3$  holds (both in the finite and infinite case). However, care must be taken to show that  $\sigma_1 \equiv \sigma_3$  holds in the infinite case (the finite case being immediate). To prove this last point, we now consider  $\sigma_1$  infinite, which implies  $\Delta(\sigma_1) = \infty$  by well-formedness of  $\sigma_1$ . Thus, we only have to show that  $\Delta(\sigma_3) = \infty$  (that is,  $\sigma_3$  is well-formed), which implies  $\sigma_1 \equiv \sigma_3$  (omitting the details). This is proven by means of contradiction: assume  $\Delta(\sigma_3)$  is finite, although  $\sigma_3$  is infinite. Then, necessarily, at least one case (or sub case) of the algorithm is repeated an infinite number of steps. It cannot be sub case (ii) because  $\delta_2$  is a positive constant, and thus  $d(\delta_2)$  cannot occur an infinite number of times in  $\sigma_3$ , whose duration is finite. As a consequence, after a finite number of iterations, all delays  $d(\delta)$  occurring in  $\sigma_1$  are handled by sub case (i). It cannot be sub case (i) either, because otherwise  $\Delta(\sigma_1)$  would also be finite. It cannot be cases (a) nor (b) either, because otherwise  $\sigma_1$  would end with an infinite sequence of events  $(t, m, s)$ , with no delay, which would imply  $\Delta(\sigma_1)$  is finite.

Finally, it remains to be shown that  $\sigma_2$  and  $\sigma_3$  are well-formed. Since  $\sigma_1 \equiv \sigma_3$  holds, we have  $\Delta(\sigma_1) = \Delta(\sigma_3)$ . Thus,  $\Delta(\sigma_3)$  is infinite if and only if  $\Delta(\sigma_1)$  is infinite, that is,  $\text{dom } \sigma_1 = \mathbb{N}$  (by well-formedness of  $\sigma_1$ ), which is equivalent to  $\text{dom } \sigma_3 = \mathbb{N}$  (by  $\sigma_1 \equiv \sigma_3$ ). Thus,  $\sigma_3$  is well-formed. Additionally,  $\sigma_2 \bowtie \sigma_3$  implies  $\Delta(\sigma_2) = \Delta(\sigma_3)$  and  $\text{dom } \sigma_2 = \text{dom } \sigma_3$ , thus  $\sigma_2$  is also well-formed.

□

**Proof sketch.** Given a trace  $\sigma_1$  in  $\Sigma(N)$ , we build a trace  $\sigma_2$  in  $\Sigma(O)$  that is composable with  $\sigma_1$ . The main difficulty concerns time elapses. Indeed, some time delays  $d(\delta')$  in  $\sigma_1$  are not directly synchronised with  $O$  if  $\delta'$  is too long (intuitively, the observer requests an interruption), so that  $d(\delta')$  must be split in two parts  $d(\delta_1)$  and  $d(\delta' - \delta_1)$ ,

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

where  $\delta_1$  is the delay until the observer's interruption. Additionally, one must show that the observer does not introduce an infinite number of interruptions within a finite time interval (that is,  $\sigma_2$  is well-formed). As for untimed events, they are easily synchronised, introducing dumb events  $\delta(0)$  where necessary.  $\square$

The conditions in Lemma 1 are true for the `leadsto` observer defined in Fig. 5.2. Therefore this observer cannot interfere with the system under observation. Next, we prove that the transition observer is sound, meaning that it reports correctly if its associated pattern is valid or not. We prove the soundness of this observer by showing that, for any TTS  $N$ , the event `error` does not appear in the traces of  $N \parallel O$  if and only if the pattern is valid for  $N$ . We write  $\text{error} \in N \parallel O$  to mean there exists a trace  $(\sigma, \sigma')$  in  $\Sigma(N \parallel O)$  such that  $\text{error} \in \sigma'$ .

**Theorem 1.** *We have  $\text{error} \notin N \parallel O$  if and only if, for all  $\sigma \in \Sigma(N)$  such that  $\sigma = \sigma_1.E_1.\sigma_2$ , there exist  $\sigma_3$  and  $\sigma_4$  with  $\sigma_2 = \sigma_3.E_2.\sigma_4$  and  $\Delta(\sigma_3) \leq \max$ .*

*Proof.* This is a consequence of the two following properties, where we assume that  $\sigma_1 \bowtie \sigma_2$  holds, with  $\sigma_1 \in \Sigma(N)$  and  $\sigma_2 \in \Sigma(O)$ .  $\square$

**Property 2.** *If there exist  $\sigma_1^a$ ,  $\sigma_1^b$ , and  $\sigma_1^c$  such that  $\sigma_1 = \sigma_1^a.E_1.\sigma_1^b.\sigma_1^c \wedge \Delta(\sigma_1^b) \geq \max \wedge E_2 \notin \sigma_1^b$ , then  $\text{error} \in \sigma_2$ .*

*Proof.* First, by Definition 9 in Sect. 3.2.3 Chapter 3 and based on the fact that  $\sigma_1 \bowtie \sigma_2$ ,  $E_1 \in \sigma_1 \implies E_1 \in \sigma_2$  and  $\exists \sigma_2^b$  a prefix in  $\sigma_2$  such that  $\sigma_1^b \bowtie \sigma_2^b$ . By hypothesis,  $\Delta(\sigma_1^b) \geq \max \wedge E_2 \notin \sigma_1^b \implies \Delta(\sigma_2^b) \geq \max \wedge E_2 \notin \sigma_2^b$ .  $\Delta(\sigma_2^b) \geq I(\text{error})$  implies that place `error` is enabled in  $\sigma_2^b$ . We conclude that  $\text{error} \in \sigma_2$ .  $\square$

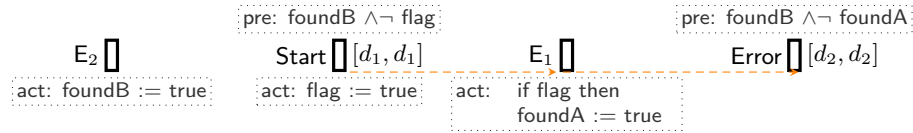
**Property 3.** *If  $\text{error} \in \sigma_2$ , then there exist  $\sigma_1^a$ ,  $\sigma_1^b$ , and  $\sigma_1^c$  such that  $\sigma_1 = \sigma_1^a.E_1.\sigma_1^b.\sigma_1^c \wedge \Delta(\sigma_1^b) \geq \max \wedge E_2 \notin \sigma_1^b$ .*

*Proof.*  $\sigma_2$  is an execution trace such that  $\sigma_2 = \sigma_2^a.\sigma_2^b.\sigma_2^c$ .  $\text{error} \in \sigma_2$  means that `error` is enabled and  $I(\text{error}) \leq \max$ , we deduce that  $\text{error} \in \sigma_2^c$ . Based on the fact that `error` is executed in  $\sigma_2^c$ , the delay of  $\sigma_2^b$  is greater than  $\max$ , which implies that  $E_2 \notin \sigma_2^b$  and  $E_1 \in \sigma_2$ . The  $\sigma_2$  can be defined as  $\sigma_2 = \sigma_2^a.E_1.\sigma_2^b.\sigma_2^c$ . We have  $\sigma_1 \bowtie \sigma_2$  implies that  $\sigma_1 = \sigma_1^a.E_1.\sigma_1^b.\sigma_1^c$ .  $\square$

## 5.4 Catalogue of TTS Observers for Patterns Verification

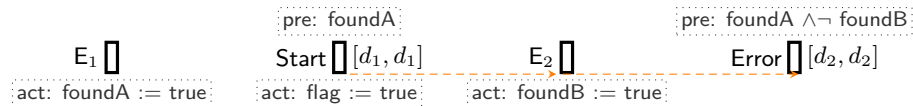
We present next the corresponding TTS observer for each pattern as well as the LTL formula to be check. We define some conventions on observers. In the following, *Error*, *Start*, ... are transitions that belong to the observer, whereas  $E_1$  (resp.  $E_2$ ) represents all transitions of the system that match predicate  $A$  (resp.  $B$ ). We also use the symbol  $I$  as a shorthand for the time interval  $[d_1, d_2]$ . The observers for the pattern obtained with other time intervals—such as  $]d_1, d_2]$ ,  $]d_1, +\infty[$ , or in the case  $d_1 = d_2$ —are essentially the same, except for some priorities between transitions that may change. By convention, the boolean variables used in the definition of an observers are initially set to false.

- **Present  $A$  after  $B$  within  $[d_1, d_2]$**  : The LTL formula to verify is  $\Box \neg \text{Error}$ .



In this observer, transition *Error* is conditioned by the value of the shared boolean variables *foundA* and *foundB*. Variable *foundB* is set to true after transition  $E_2$  and transition *Error* is enabled only if the predicate  $\text{foundB} \wedge \neg \text{foundA}$  is true. Transition *Start* is fired  $d_1$  u.t after an occurrence of  $E_2$  (because it is enabled when *foundB* is true and *flag* is false). Then, after the first occurrence of  $E_1$  and if *flag* is true, *foundA* is set to true. This captures the first occurrence of  $E_1$  after *Start* has been fired. After  $d_2$  u.t., in the absence of  $E_1$ , transition *Error* is fired. Therefore, the verification of the pattern boils down to checking if the event *Error* is reachable. The priority (dashed arrows) between *Start*, *Error*, and  $E_1$  is here necessary to ensure that occurrences of  $E_1$  occurring at  $d_1$  or  $d_2$  are taken into account.

- **Present  $A$  before  $B$  within  $I$**  : The LTL formula to verify is  $(\Diamond B) \Rightarrow \neg \Diamond (\text{Error} \vee (\text{foundB} \wedge \neg \text{flag}))$ .

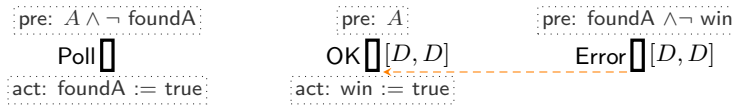


Like in the previous case, variables *foundA* and *foundB* are used to record the occurrence of transitions  $E_1$  and  $E_2$ . Transition *Start* is fired, and variable *flag*

## 5. CHECKING PATTERNS USING TTS OBSERVERS

is set to true,  $d_1$  u.t. after the first  $E_1$ . Then transition **Error** is fired only if its precondition—the predicate  $\text{foundA} \wedge \neg \text{foundB}$ —is true for  $d_2$  u.t. Therefore transition **Error** is fired if and only if there is an occurrence of  $E_2$  before  $E_1$  (because then  $\text{foundB}$  is true) or if the first occurrence of  $E_2$  is not within  $[d_1, d_2]$  of the first occurrence of  $E_1$ .

- **Present  $A$  lasting  $D$**  : The LTL formula to verify is  $\Box \neg \text{Error}$ .

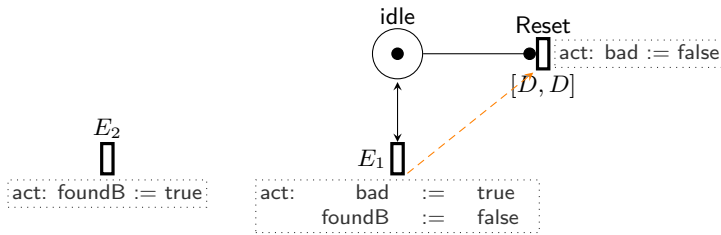


Variable  $\text{foundA}$  is set to true when transition *Poll* is fired, that is when  $A$  becomes true for the first time. Transition *OK* is used to set  $\text{win}$  to true if  $A$  is true for duration  $D$  without interruption (otherwise its timing constraint is resetted). Otherwise, if variable  $\text{win}$  is still false after  $D$  u.t., then transition *Error* is fired. We use a priority between *Error* and *OK* to disambiguate the behaviour  $D$  u.t. after *Poll* is fired.

- **Absent  $A$  after  $B$  for interval  $I$**  : The LTL formula to verify is  $\Diamond B \Rightarrow \Diamond \text{Error}$ .

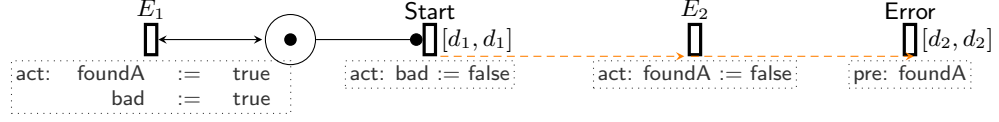
We use the same observer as for *Present  $A$  after  $B$  within  $I$* , but here *Error* is the expected behaviour.

- **Absent  $A$  before  $B$  for duration  $D$**  : The LTL formula to verify is  $\Box \neg (\text{foundB} \wedge \text{bad})$ .



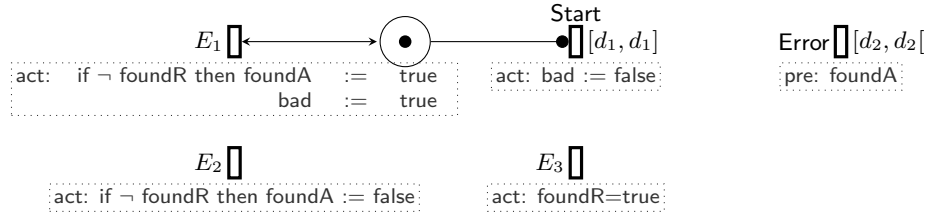
Variable  $\text{foundB}$  is set to true after each occurrence of  $E_2$ . Conversely, we set the variables  $\text{bad}$  to true and  $\text{foundB}$  to false at each occurrence of  $E_1$ . Therefore  $\text{foundB}$  is true on every “time interval” between an  $E_2$  and an  $E_1$ . We use transition *Reset* to set  $\text{bad}$  to false if this interval is longer than  $D$ . As a consequence, the pattern holds if we cannot find an occurrence of  $E_2$  ( $\text{foundB}$  is true) while  $\text{bad}$  is true.

- **A leadsto B within I** : The LTL formula to verify is  $(\Box \neg \text{Error}) \wedge (\Box \neg (B \wedge \text{bad}))$ .



After each occurrence of  $E_1$ , variables  $foundA$  and  $bad$  are set to true and the transition  $Start$  is enabled. Variable  $bad$  is used to control the beginning of the time interval. After each occurrence of  $E_2$  variable  $foundA$  is set to false. Hence  $Error$  is fired if there is an occurrence of  $E_1$  not followed by an occurrence of  $E_2$  after  $d_2$  u.t. We use priorities to avoid errors when  $E_2$  occurs precisely at time  $d_1$  or  $d_2$ .

- **A leadsto B within I before R** : The LTL formula to verify is  $\Diamond R \Rightarrow (\Box \neg \text{Error} \wedge \Box \neg (B \wedge \text{bad}))$ .



Same explanation than for the previous case, but we only take into account transitions  $E_1$  and  $E_2$  occurring before  $E_3$ .

- **A leadsto B within I after R** : The LTL formula to verify is  $\Diamond R \Rightarrow (\Box \neg \text{Error} \wedge \Box \neg (B \wedge \text{bad}))$ .

It is similar to the observer of the pattern *A leadsto first B within I before R* . We should just replace  $\neg \text{foundR}$  in transition  $E_1$  and  $E_2$  by  $\text{foundR}$ .

Same explanation than in the previous case, but we only take into account transitions  $E_1$  and  $E_2$  occurring after an  $E_3$ .

#### 5.4.1 Composite Patterns

Patterns can be easily combined together using the usual boolean connectives. For example, the pattern " $P_1$  and  $P_2$ " holds for all the traces where  $P_1$  and  $P_2$  both hold. To check a composed pattern, we use a combination of the respective observers, as well as a combination of the respective LTL formulas. For instance, if  $(T_1, \phi_1)$  and  $(T_2, \phi_2)$  are the observers and LTL formulas corresponding to the patterns  $P_1$  and  $P_2$ , then the

## 5. CHECKING PATTERNS USING TTS OBSERVERS

---

composite pattern  $P_1$  and  $P_2$  is checked using the LTL formula  $\phi_1 \wedge \phi_2$ . Similarly, if we check the LTL formula  $\phi_1 \Rightarrow \phi_2$  (implication) then we obtain a composite pattern  $P_1 \multimap P_2$  that is satisfied by systems  $T$  such that, for all traces of  $T$ , the pattern  $P_2$  holds whenever  $P_1$  holds.

### 5.5 Conclusion

We give a TTS observers for every real time specification patterns in our catalog. We also provide a formal framework that allows us to check whether a proposed observer is correct and non-intrusive. (We also give a set of sufficient conditions for an observer to be innocuous.)

Our approach has been integrated into a complete verification tool chain for the Fiacre modelling language and can therefore be used in conjunction with Topcased (FGC<sup>+</sup>06). We give several experimental results based on the use of this tool chain in Sect. 5.2. The fact that we implemented our approach has influenced our definition of the observers. Indeed, another contribution of our work is the use of a pragmatic approach for comparing the effectiveness of different observers for the same property. Our experimental results seem to show that data observers look promising.

We are following several directions for future work. A first goal is to define a new low-level language for observers—adapted from the TTS model—equipped with more powerful optimisation techniques and with easier soundness proofs. On the theoretical side, we are currently looking into the use of mechanised theorem proving techniques to support the validation of observers. On the experimental side, we need to define an improved method to select the best observer. For instance, we would like to provide a tool for the “syntax-directed selection” of observers that would choose (and even adapt) the right observers based on a structural analysis of the target system.

## Chapter 6

# Checking Patterns Using Probes and Fiacre Observers

*“Success has a simple formula: do your best, and people may like it.”*

*-Sam Ewing-*

In this chapter, we present a new approach for implementing observers based on an extension of the Fiacre language with verification probes. That is to say, we provide an implementation for patterns using *Fiacre observers* instead of TTS observers as in Chapter 5. Another contribution of this chapter is the definition of a graphical method that can be used to reason about the correctness of our proposed verification method. This graphical approach is complementary to the formal framework used to check the correctness of observers.

This chapter is organised as follow. Section 6.2 introduces the notion of probes. In Section 6.3, we give the program source of the Fiacre observer for every pattern in our catalogue, together with a brief description of how the observer behaves. Section 6.4 introduces the graphical proof method used to check that an observer implementation is correct. Finally in Section 6.5, before concluding, we give the results of some experimentations that compare the performances of our two approaches for checking patterns; namely TTS and Fiacre observers.

### 6.1 Introduction

In the previous chapter, we have used observers at the TTS level for checking patterns. However this approach has two drawbacks. First, it requires to use different observers



## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

depending on whether we define a property on states or ports. The second drawback is related to implementation issues, since it is difficult to maintain the part of the pFrac that manages TTS observers. Indeed, since we do not have a textual representation for TTS, we need to use the internal representation for Fiacre processes instead.

To deal with these two drawbacks, we propose to add a transparent layer to the Fiacre language for describing observable events. This mechanism, that is called “probes”, can be used to name all the events of a Fiacre program. A probe is a language construct used to observe modifications in the system without interfering with it; probes react to the occurrence of an event without engaging in it. Finally, we will also show in Section 6.5 that the use of Fiacre observers leads to better performances in terms of memory usage.

In this chapter, we present our new approach to implement observers based on the Fiacre language. We present also the notion and syntax of probes. To the best of our knowledge, the notion of *probes* is totally new in the context of formal specification language. Paun and Chechik propose a somewhat similar mechanism in (PC99)—in an untimed setting—where they define new categories of events. However our approach is more general, as we define probes for a richer set of events, such as variables changing state.

A second contribution of this chapter is the definition of a “graphical method” to check that Fiacre observers are correct. In the previous chapters, we already defined a complete theoretical framework that comprises a formal semantics for patterns and timed traces as well as formal definitions (and proof methods) for checking the soundness and innocuousness of observers. This formal framework has been partially implemented in the Coq proof assistant, which means that we are able to prove the correctness of an observer using Coq. Nonetheless, this method can be quite tedious and problems with an observers could be detected very late during the proof, which mean that a lot of efforts could go to waste and that it is expensive to test new observers. The graphical method described in this chapter is a solution to this problem, since it allows us to “debug observers” before we prove them correct. In this respect, the two methods are complementary: we use the graphical proof to reason about the observer at an early step, before doing formal verification. Moreover, our graphical method tests our implementation of the tool (and not merely our definition of the observers), which mean

that we could detect problems that have been introduced during the transcription of our algorithm into actual code.

## 6.2 Probes and Observers in Fiacre

The last version of the Fiacre language has been extended to allow the definition of *observers*, which are a distinguished category of sub-programs that interact with other Fiacre components only through the use of *probes*. A probe is used to observe modifications in the system without interfering with it; probes react to the occurrence of an event without engaging in it.

### 6.2.1 Probes Syntax

A probe 'p' describes a state property or a set of transitions of a Fiacre specification. Probe obeys the following syntax:

```

p ::= path/obs
    | "enter" p
    | "leave" p
    | "not" p
    | p "and" p
    | p "or" p
    | p "=>" p
path ::= IDENT "/" (NAT "/" )*
obs ::= "state" IDENT
      | "value" exp
      | "event" IDENT

```

A typical probe declaration is of the form `path/obs`, where `obs` denotes the observable and `path` defines its context, that is a path to the component (or process) instance where `obs` is defined. In our setting, the observable events are instantaneous actions involved in the evolution of the system: it can be a transition on the port `p` (denoted event `p`); a process that enters the state `s` (denoted state `s`); or an expression including shared variables, say `exp`, that changes value (denoted value `exp`). For instance, in the case of the double-click program (see Chapter 3 Fig. 3.1), a probe triggered when the (first) instance of process `Push`, under the component `Mouse`, is in the state `s2` would have the form `(Mouse/1/state s2)`. Finally, probes can be composed using boolean connectives.

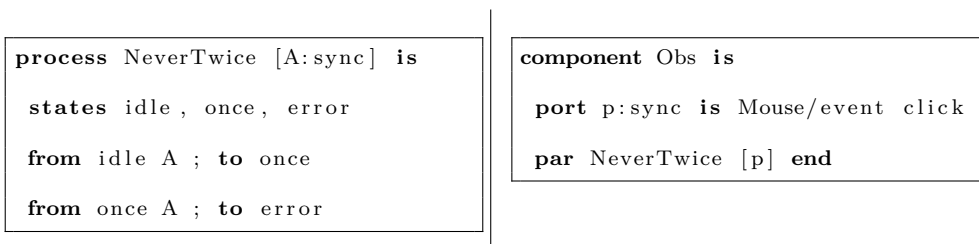
## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

For instance, the probe `not (Mouse/event click)` is triggered by any event that is not a communication over the port `click`. We can associate also a probe `p` with the keyword `enter` or `leave` to indicate that the observable concerns the act of entering or leaving `p` respectively.

### 6.2.2 Observers in Fiacre

An observer is a regular Fiacre program where ports are associated to probes (using the keyword `is`); ports associated with a probe have the reserved type `sync`. We give a naive example of observer in Fig. 6.1 using the example of the “double-click” program defined in Section 3.1 (see page 33). In this example, the component `Obs` monitors synchronisations on the event `click` presented in Fig. 3.1. In this example, the process `neverTwice` will reach the state `error` if its probe parameter, `A`, is triggered more than once. In the remainder of the text, we use the notation `(Mouse || Obs)` to denote the program obtained by concatenating the declaration of these two components (i.e. the code from Fig. 3.1 with the code from Fig. 6.1). As a consequence, we are able to detect if the system can emit two single click events just by checking if the process `neverTwice` can reach the state `error` in `(Mouse || Obs)`. This can be easily achieved using an LTL model-checker (the *slt* tool in our case) with the property  $\square \neg (\text{Obs}/1/\text{state error})$  (always, not `Obs/1` is in state `error`, where `Obs/1` is the first process instance in the definition of `Obs`).



**Figure 6.1:** A simple observer example

## 6.3 Catalogue of Fiacre Observers for Patterns Verification

In this section, we define a Fiacre observer for each pattern in our catalogue. We denote by  $E1$ ,  $E2$  and  $E3$  the events to verify and by  $[d1,d2]$  an interval where  $d1 \in \mathbb{R}^+$ ,

$d2 \in \mathbb{R}+ \cup \{+\infty\}$  and  $d2 \geq d1$ .

#### 6.3.1 Existence Patterns

- **present**  $E1$  **after**  $E2$  **within**  $[d1, d2]$

The Fiacre program of the observer and its corresponding TTS model are listed below.

```

process observer [E1, E2 : sync] is

  states idle, start, watch, error, stop

  from idle
    E2;
  to start

  from start
    wait [d1,d1];
  to watch

  from watch
    select
      E1; to stop
    unless
      wait ]d2-d1,...[; to error
    end

component obs is

  port E1 : sync in [0,0] is ...,
    E2 : sync in [0,0] is ...

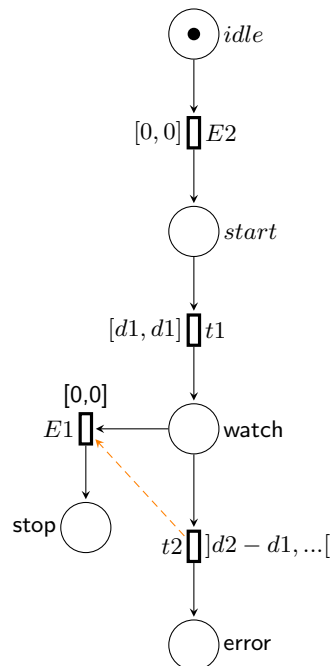
  par observer[E1, E2] end

```

To explain how the observer works, it is equivalent to look at the behavior of its TTS model (presented below). After the first occurrence of  $E2$ , the observer is in state “start”. From the state “start” and when the time interval is elapsed, it goes to state “watch”. After that, the observer waits for an occurrence of “ $E1$ ” holding in the interval  $[d1, d2]$  and moves to state “stop” otherwise it moves to state “error”. The priority relation between transition “t2” and “ $E1$ ” means that the observer takes into account the occurrences of “ $E1$ ” holding at  $d2$ . The corresponding LTL formula to verify is:  $\Box \neg error$ .

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---



- present  $E1$  before  $E2$  within  $[d1, d2]$

The Fiacre observer and its corresponding TTS model are listed below.

```

process observer1 [E1: sync] (&flag:bool, &foundE2:bool) is

  states idle, start, watch, error

  from idle
    E1;
  to start

  from start
    wait [d1,d1]; flag:=true;
  to watch

  from watch
    wait ]d2-d1,...[; flag:=false;
    on foundE2=false;
  to error

process observer2 [E2: sync] (&foundE2:bool,&flag:bool) is

  states idle, stop

  from idle
    E2;foundE2:=true;
    on flag=true;
  to stop

component obs is
  
```

### 6.3 Catalogue of Fiacre Observers for Patterns Verification

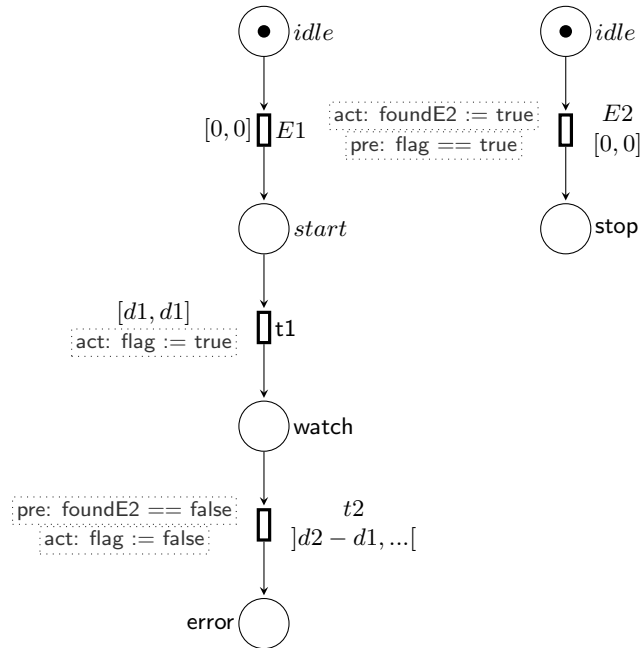
```

var  flag : bool := false ,
     foundE2 : bool := false

port E1: sync in [0,0] is ... ,
     E2: sync in [0,0] is ...

par
  observer1 [E1] (&flag , &foundE2)
||
  observer2 [E2] (&foundE2,&flag)
end

```



The observer is composed of two processes which communicate via shared variables (flag and foundE2). The process observer2 sets the value of the shared variable foundE2 to true when it sees the first occurrence of E2. In the unique transition of observer2, the operator on acts as a guard, meaning that the transition can fire only if the condition (flag = true) is true. Concurrently, the process observer1 monitors the occurrences of E1 and sets the value of flag to true for the time interval  $[d_1, d_2]$  after the first occurrence of A. The pattern is not satisfied if flag is true and foundE2 is false after  $(d_2 - d_1)$  u.t. (assuming that there would be an occurrence of E2 in the future). Therefore, the associated LTL property states that if E2 eventually occurs (eventually foundE2 is true) then observer1 must not reach its error state (always obs/1/state error is false).

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

- **present  $E1$  within  $[d1, d2]$**

We verify that during the execution of the system, there is an occurrence of  $E1$  holding in  $[d1, d2]$ . If we suppose that “init” is the initial state of the system then this pattern can be defined as “present  $E1$  after  $init$  within  $[d1, d2]$ ”. The observer and its TTS model are presented below. The observer has the same behaviour as the observer of the pattern “present  $E1$  after  $init$  within  $[d1, d2]$ ”. The corresponding LTL formula to verify is:  $\Box \neg error$ .

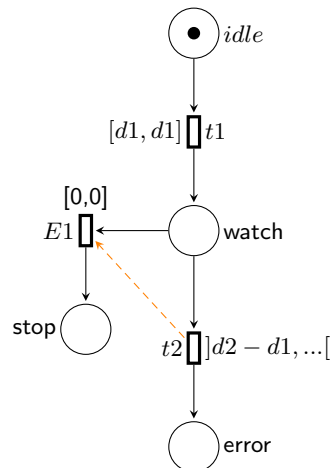
```

process observer [E1: sync] is
  states idle, watch, error, stop
  from idle
    wait [d1,d1];
  to watch
  from watch
    select
      E1; to stop
    unless
      wait ]d2-d1,...[; to error
    end
component obs is
  port E1 : sync in [0,0] is ...

  par observer[E1] end

obs

```



- **present  $E1$  lasting  $D$**

The corresponding observer and its corresponding TTS are presented below. When the system is executed, the observer waits for the first occurrence of  $E1$ . After that, it verifies that it holds true for a duration  $D$  (transition wait). If in the

### 6.3 Catalogue of Fiacre Observers for Patterns Verification

meantime, an event different of  $E1$  holds then the pattern is not verified. The corresponding LTL formula to verify is :  $\Box \neg error$ .

```

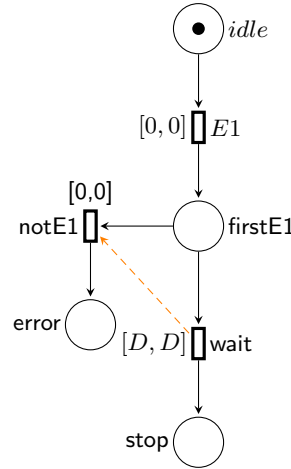
process observer [E1: sync, notE1:sync] is
  states idle, firstE1, error, stop

  from idle
    E1;
  to firstE1

  from firstE1
    select
      notE1; to error
    unless
      wait [D,D]; to stop
    end

component obs is
  port
    E1:sync in [0,0] is ...,
    notE1: sync in [0,0] is not ...
  par
    observer [E1, notE1]
  end

```



#### 6.3.2 Absence Patterns

- **absent**  $E1$  after  $E2$  for interval  $[d1, d2]$

The Fiacre observer and its corresponding TTS are presented below.

```

process observer [E1,E2 : sync] is

  states idle, start, watch, error, ok

  from idle
    E2;
  to start

```



## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

```

from start
  wait [d1,d1];
to watch

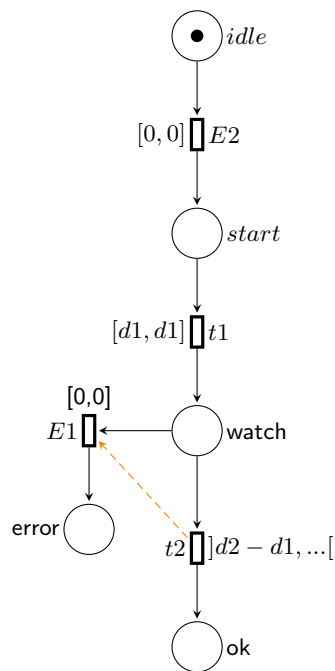
from watch
  select
    E1; to error
  unless
    wait [d2-d1,...[; to ok
  end

component obs is

  port E1 : sync in [0,0] is ...,
    E2 : sync in [0,0] is ...

  par observer[E1, E2] end

```



The observer is composed of one process that monitors the system through the ports E1 and E2 (that should be instantiated with the relevant probes). The process is initially in state *idle* and moves to *start* when E2 is triggered. When in state *start* for  $d_1$  u.t., the observer moves to state *watch* (this is the meaning of the wait operator). The select operator is a non-deterministic choice, with unless coding priorities. Hence, in state *watch*, the observer moves to *error* if an E1

### 6.3 Catalogue of Fiacre Observers for Patterns Verification

---

occurs, unless  $d_2 - d_1$  u.t. elapses, in which case it moves to *ok*. As a consequence, the pattern is false whenever process *observer* can reach state *error*. Hence the associated LTL formula is  $\Diamond E2 \Rightarrow \Diamond ok$ .

- **absent  $E1$  before  $E2$  for duration  $D$**

The observer and its corresponding TTS are presented below.

```

process observer1 [E1: sync, E3: none] (&bad: bool, &foundE2: bool) is

  states idle , ok

  from idle
  select
    E3;
    bad:=false; to ok
  []
    E1;
    bad:=true;
    foundE2:=false; to idle
  end

process observer2 [E2: sync] (&bad: bool, &foundE2: bool) is

  states idle , stop

  from idle
    E2;
    foundE2:=true;
  to stop

component obs is

  var bad: bool:=false ,
      foundE2: bool:=false

  port E1: sync in [0,0] is ... ,
      E2: sync in [0,0] is ...
      E3: none in [D,D]

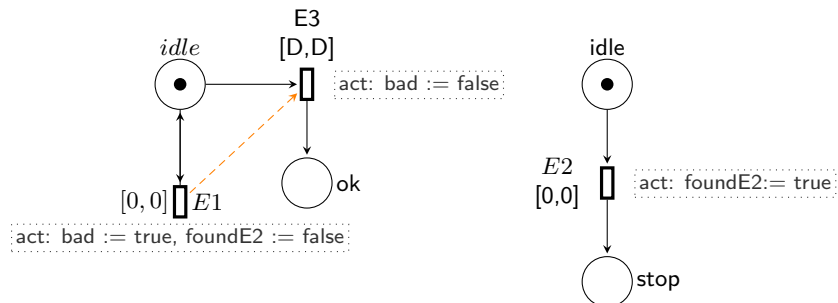
  priority E1 -> E3

  par observer1 [E1, E3](&bad, &foundE2)
    ||
    observer2 [E2](&bad, &foundE2)
  end

```

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---



The observer is composed of two process “observer1” and “observer2”. The process “observer2” captures the first occurrence of  $E2$  and sets variable “foundE2” to true. The process “observer1” captures occurrences of  $E1$  and set variable “bad” to true and “foundE2” to false. It is responsible also for controlling the duration  $D$ . The variable “bad” is set to false at the end of  $D$ . The LTL formula to verify is  $\neg \Diamond (foundE2 \wedge bad)$ .

- **absent  $E1$  for interval  $[d1, d2]$**

This pattern is equivalent to the pattern “absent  $E1$  after init for interval  $[d1, d2]$ ”. The observer and its corresponding TTS are presented below. The LTL formula to verify is  $\Box \neg error$ .

```

process observer [E1: sync] is

  states idle, watch, error, ok

  from idle
    wait [d1,d1];
  to watch

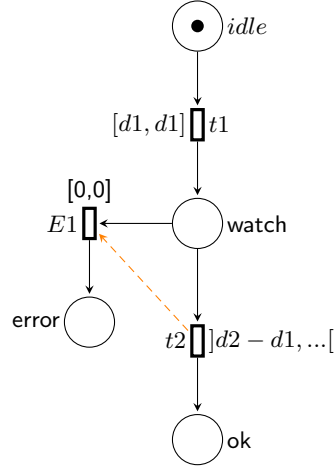
  from watch
    select
      E1; to error
    unless
      wait ]d2-d1,...[; to ok
    end

component obs is

  port E1 : sync in [0,0] is ...

  par observer[E1] end

```



### 6.3.3 Response Patterns

- ***E1 leadsto E2 within***  $[d1, d2]$

The observer and its corresponding TTS model are presented below.

The observer moves to state *start* when *E1* is triggered. Then, after  $d_1$  u.t., it moves to state *watch* where it waits for an occurrence of *E2* before  $d_2 - d_1$  u.t. elapses, in which case it is reinitialised. If no *E2* occur, the process moves to state *error*. Like in the first case, the pattern is false if observer can reach state *error*. The LTL formula to verify is  $\Box \neg error$ .

```

process observer [E1, E2 : sync] is

  states idle, start, watch, error

  from idle
    E1;
  to start

  from start
    wait [d1,d1];
  to watch

  from watch
    select
      E2; to idle
    unless
      wait ]d2-d1,...[; to error
    end

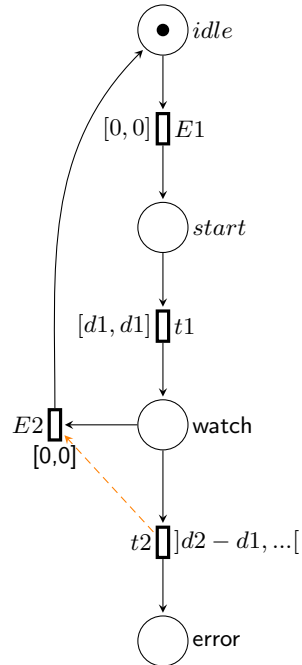
component obs is

  port E1 : sync in [0,0] is ...,
    E2 : sync in [0,0] is ...

  par observer[E1, E2] end
    
```

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---



- $E1$  leadsto  $E2$  within  $[d1, d2]$  before  $E3$

The observer and its corresponding TTS are listed below.

```

process observer1 [E1, E2 : sync] (&foundE3: bool) is

  states idle , start , watch , error

  from idle
    E1;
    on foundE3=false;
  to start

  from start
    wait [d1,d1];
  to watch

  from watch
    select
      E2; on foundE3=false; to idle
    unless
      wait ]d2-d1,...[; to error
    end

process observer2 [E3: sync] (&foundE3: bool) is

  states idle , stop

  from idle
    E3;
    foundE3:=true;
  to stop
  
```

### 6.3 Catalogue of Fiacre Observers for Patterns Verification

```

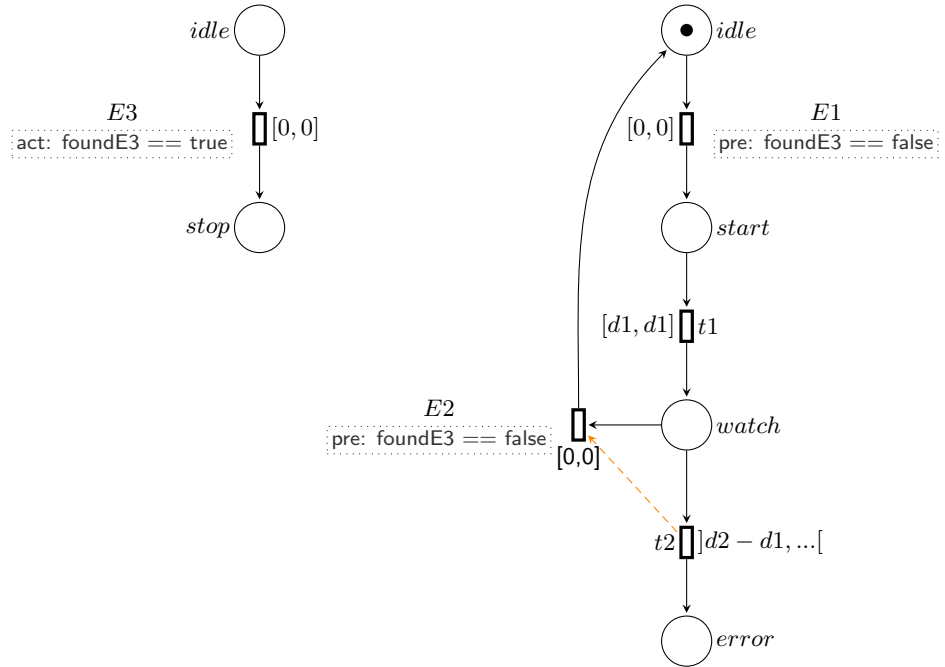
component obs is

  var foundE3:bool:=false

  port E1 : sync in [0,0] is ...,
        E2 : sync in [0,0] is ...,
        E3 : sync in [0,0] is ...

  par
    observer1[E1, E2] (&foundE3)
  ||
    observer2[E3] (&foundE3)
  end
end

```



The observer has the same behaviour as the observer of the pattern ‘ $E1$  leadsto  $E2$  within  $[d1, d2]$ ’. The only difference is that it verifies if there is an occurrence of  $E3$  after  $E2$ . The LTL formula to verify is  $\Diamond E3 \Rightarrow \Box \neg error$ .

- **$E1$  leadsto  $E2$  within  $[d1, d2]$  after  $E3$**

The observer and its corresponding TTS are presented below.

```

process observer [E1, E2, E3 : sync] is

  states idle, start, watch, beginObs, error

  from idle

```

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

```

    E3;
  to start

  from start
    E1;
  to beginObs

  from beginObs
    wait [d1,d1];
  to watch

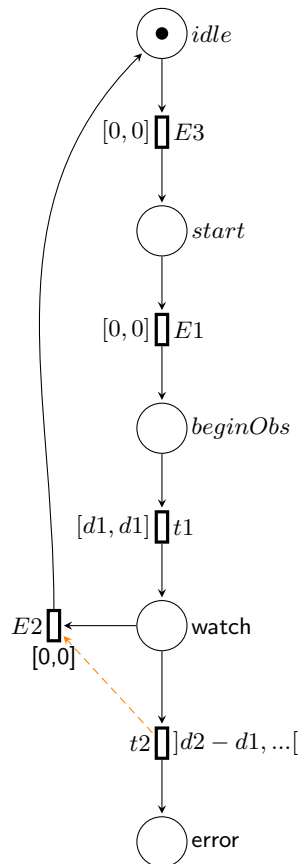
  from watch
    select
      E2; to idle
    unless
      wait ]d2-d1,...[; to error
    end

component obs is

  port E1 : sync in [0,0] is ... ,
        E2 : sync in [0,0] is ... ,
        E3 : sync in [0,0] is ...

  par observer[E1, E2, E3] end

```



The observer has the same behaviour as the observer of the pattern ‘ $E1$  leadsto  $E2$  within  $[d1, d2]$ ’ expect the fact that it verifies if there is an occurrence of  $E3$  before  $E1$ . The LTL formula to verify is  $\Diamond E3 \Rightarrow \Box \neg error$ .

## 6.4 Proving the correctness of observers

To prove that an observer *Obs* for the pattern *P* is correct, we need to prove that, for every system *S*, the program (*S* || *Obs*) satisfies the LTL formula  $\phi_P$  if and only if for all trace  $\sigma$  in *S*,  $\sigma \models P$ . In Chapter 3, we define a theoretical framework to prove exactly these kind of properties. Nonetheless, formal proofs of correctness can be quite tedious. Therefore, to detect possible problems with an observer early on (that is, before spending a lot of efforts doing a formal proof of correctness) we also propose a “graphical verification method”. This is akin to debugging our observers.

In the remainder of this section, we describe our method using the particular case of the pattern *Present E1 after E2 within*  $[4, 5[$ . Therefore, we assume that *Obs* is the observer defined in Fig. 6.3.1 (with  $d_1 = 4$  and  $d_2 - d_1 = 1$ ).

### 6.4.1 Universal Program.

The first step, in our method, is to get rid of the universal quantification on all possible systems, *S*, that is introduced by our definition of correctness. The idea is to check the observer on a particular Fiacre program—called *Universal*—that can generate all possible combinations of delays and events between the pair of events *E1* and *E2*. We give an example of universal process in Listing 6.1 (where we already compose this process with the observer).

The process *Universal* has only one state and three possible transitions. Each transition changes the value of a shared integer variable, *x*. The first and second transitions of *Universal* can be fired without time constraints. In our context, the probe *E1* will be triggered to the event “setting *x* to 1” and *E2* to “setting *x* to 2”. The third transition reset the value of *x* to 0 immediately.

### 6.4.2 Graphical Verification.

The next step is to use our verification tool chain to generate the state graph for the program (*Universal* || *Present*). The state graph should be generated with a “discrete



## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

```

process Universal (&x : nat) is

  states s0

  from s0 select
    x := 1; to s0
  [] x := 2; to s0
  unless
    on (x <> 0); wait [0,0]; x := 0; to s0
  end

component Main is

  var x : nat := 0

  port E1 : sync is value (x = 1), E2 : sync is value (x = 2)

  par Universal (&x) || observer [E1, E2] end

```

**Listing 6.1:** Universal program in Fiacre

time” abstraction, where special transitions (labeled with  $t$ ) are used to model the flow of time. Label  $t$  stands for the “tick” of the logical clock: a transition  $t$ , between two states, asserts that 1 u.t. has passed. This construction can be obtained using the tool *tina* (BRV04) with its flag `-F1` (with *tina*, it is also possible to generate the state graph with many different abstractions, including dense time models).

The resulting graph is displayed in Fig. 6.2. This state graph has been generated and printed using the tool *nd*, which is also part of the Tina tool set; *nd* is an editor and animator for extended Time Petri Nets that can export nets and state graphs in several, machine readable formats. This graph has only 26 states and can therefore be easily managed manually. The main factor commanding the number of states is the value of the timing constraints used in the pattern; in our observations, all the generated state graphs were of manageable size.

The transitions in the state graph are also quite straightforward: transitions labeled with E1 or E2 are the observable events (we call E1, E2 and  $t$  the *external transitions*); label  $z$  denotes internal transitions in the system (in the case of *Universal*, it is the transition that reset  $x$  to 0); the remaining labels correspond to transitions in the observer *Present*. The transition from state 2 to 3 corresponds to the observer entering the state *start*; likewise for the transitions labeled with *watch*, *stop* and *error*.

We can already debug the pattern *Present E1 after E2 within [4,5[* by visually inspecting the state graph. For *soundness*, we need to check that, when the pattern is

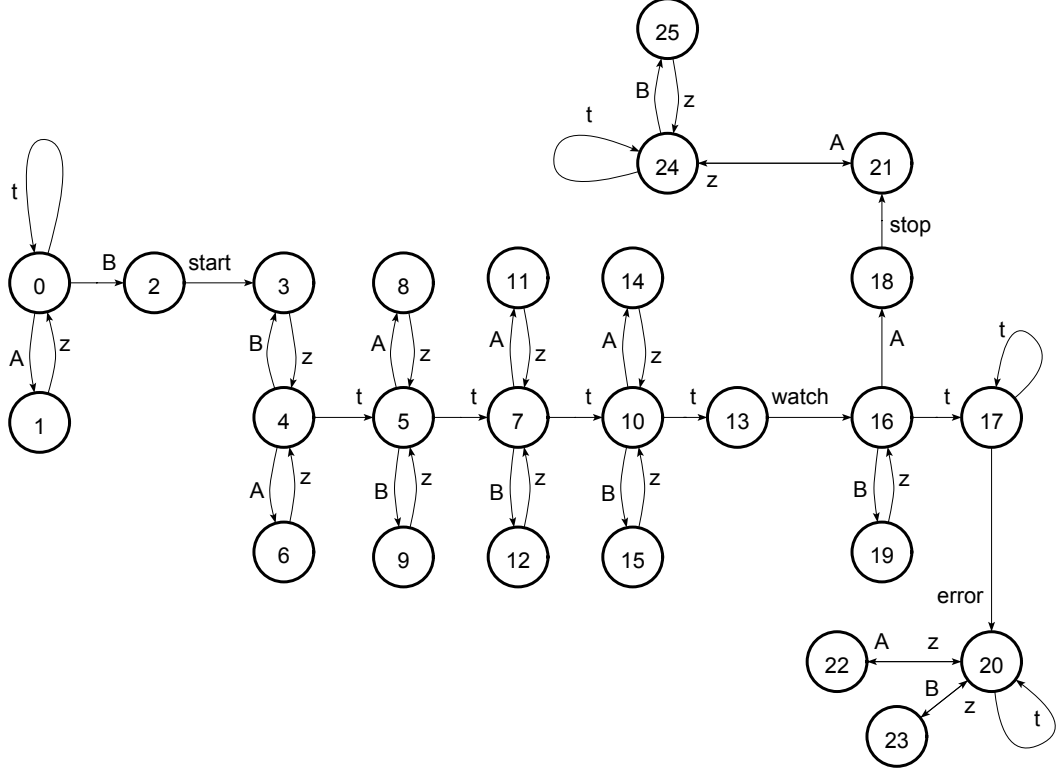


Figure 6.2: State graph for (Universal || Present)

not satisfied (for traces  $\sigma$  such that  $\sigma \not\models P$ ), then the observer will detect a problem (observer Present eventually reaches the state error.) Actually we can observe that, starting from the initial state of the system (labeled 0), after the first occurrence of a E2 (in state 2), and after 4 units of time (after 4 transitions labeled  $i$ ), the system reach a state (numbered 13 in the state graph) such that: (1) if we do not see an E1 before 1 u.t. then we have an error; and (2) if we see an E1 then we will never see an error. In this context, to “see an E1 before 1 u.t.” correspond to following a path—starting from state 13—where a transition labeled with E1 is before any transition labeled with  $i$ . Likewise, “errors” correspond to any state where error can be observed (state 17 in our case) or that cannot be reached without first observing error (i.e. states 20, 22 and 23). We will make these definitions more formal in the next paragraphs.

For *innocuousness* we need to check that, from any state, it is always possible to reach a state where event  $E1$  (respectively  $E2$  and  $i$ ) can fire. Indeed, it means that the observer cannot censor the observation of a particular sequence of external transitions

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

or the passing of time.

This graphical verification method has some drawbacks. It relies on a discrete time model and it only works for fixed time intervals (we have to fix the value of  $d_1$  and  $d_2$ ). Nonetheless, it is usually enough to catch some errors in the observer before we try to prove the observer correct more formally.

In practice, we do not simply rely on a visual inspection of the state graph. We can use properties expressed in the  $\mu$ -calculus to check the state graph (the Tina tool set includes a model-checker for the  $\mu$ -calculus called muse). Informally, we can define the set of traces where *Present* holds using the union of two regular languages: first the traces where E2 never occurs (the expression  $(\neg E2)^*$ ); then the traces where there is an E1 4 u.t. after the first E2 (the expression  $(\neg E2)^* \cdot E2 \cdot ((\neg i)^* \cdot i)^4 \cdot E1 \cdot \top^*$ ). Each expression can be interpreted as a  $\mu$ -calculus formula. For instance, the absence of E2 corresponds to the formula  $\mu X.(\langle \neg E2 \rangle \vee X)$ , which is true for the states 0 and 1 in the state graph. Likewise, using a maximal fix-point formula, we can define the “errored states”, that is the states where an error has been signaled or is inevitable. Finally, it is enough to check that all valid traces cannot lead to an errored state and that all invalid traces (the complement language) inevitably lead to an errored state.

### 6.5 Experimentation

In this section, we present some experimentations using pFrac, the Fiacre language compiler which takes into account properties. We will test some examples using Data observers and Fiacre observers in the case of valid property:

- The TRAIN example has been presented in Section 5.2 Chapter 5. We check in this part the case of the valid property presented in Section 5.2.
- The CAR (P1), CAR (P2) and CAR (P3) represent the defined properties presented in Section 4.4 Chapter 4.
- The AIRLOCK example represents a model for a simple airlock system consisting of two doors and two buttons. At any time, at most one door can be opened. Additionally, an open door is automatically closed after exactly 4 units of time (u.t.), followed by a ventilation procedure that lasts 6 u.t. The valid property, for

the AIRLOCK example, checks that the duration between opening a door and closing it does not exceed 11 u.t.

- The USINE example is a model describing a manufacturing plant composed of two command lines sharing some of their machines. In this case, timing constraints arise from a combination of safety issues—workers should not work more than 35 minutes in a row—and performance issues—machines perform a task in a time between 5 and 10 minutes and should be maintained after 15 cycles. The valid property, for the FACT example, states that the delay between two successive breaks should not exceed 35 minutes.
- The CITY example is a model obtained from a business workflow describing the delivery of identity documents in a French city hall. In this example, timing constraints arise from delays in the communication between services and time spent to perform administrative procedures. The valid property, in this case, states that the minimal possible delay for obtaining an id is 30 hours.

We compare the two approaches in Fig. 6.3 and we present the system size growth in bite and the execution time in second. Experimentations have been done using a machine of 6Go RAM and an Intel Core TM i5-2430M 2,4GHz.

Examples	Data Observer		Fiacre Observer	
	State Space Size (KB)	Time (ms)	State Space Size (KB)	Time (ms)
Train	508.547	40	510.056	44
CAR (P1)	668.274	66	663.922	62
CAR (P2)	695.166	57	688.364	52
CAR (P3)	5663.708	362	689.686	62
AIRLOCK	3.604	11	6.279	22
USINE	312.697	32	300.435	29
CITY HALL	43.302	18	46.080	19

**Figure 6.3:** Experimentation using Fiacre observer and data observer

Experimentations have shown that Fiacre observers are more performed in practice than data observers in all most the cases. When we compare the two approaches, we noticed that Fiacre observers are easy to maintain. We should just modify the Fiacre program. However, for data observer, we should modify the implementation of the

## 6. CHECKING PATTERNS USING PROBES AND FIACRE OBSERVERS

---

observer (modify the code). In the other side, Ficare observer are difficult to define in some cases due to some lack in Fiacre language which is not the case for data observer.

### 6.6 Conclusion

In this Chapter, we have introduced first the notion of probes and then presented our catalogue of observers based on Fiacre language. The Fiare observers have some drawbacks in practice related to Fiacre language limitation. However, they can be easily maintained. TTS observers in the other side are difficult to maintain however we can express easily observers.

We have described also a simple, graphical verification method that can be used to gain confidence on the implementation of our model-checking tools. We show how to automatise this method so as to avoid human errors. This result also prove the usefulness of having access to a complete toolbox, with different kind of tools (editors, model-checkers, ...), and working with common file formats. The use of “graphical” verification methods is part of the model-checking folklore; the technique is known by some people but has never really been documented. The overall method is quite close in spirit to refinement-based techniques.

In many respects, we apply a generic bootstrapping technique, by which we use our existing LTL model-checker to implement and check a model-checker for a more complex temporal logic. While we describe our method on a particular specification language, and for a particular set of tooling, our method is quite general and could be applied on a different setting.

## Chapter 7

# Conclusions

*“The real danger is not that computers will begin to think like men, but that men will begin to think like computers.”*

*-Sydney J. Harris-*

This thesis describes my contributions to the definition and the verification of timed requirements on the formal models of critical embedded systems. We proposed a complete framework to specify and verify properties in the context of real time systems. This framework includes a pattern language to specify properties; a method to check patterns on a model (that has been implemented in a tool chain for the language Fiacre); and an approach to prove the correctness of our tools.

In Chapter 4, we have defined a set of high-level specification patterns for expressing requirements on systems with hard real time constraints. Our patterns are simple to use since they are based on natural language. Moreover, we are based on Dwyer’s patterns. A recent study (BGPS12) has shown that Dwyer’s patterns are the most used on practice. So we are confident about the usefulness of our approach in practice. We also believe that our notation is interesting in its own right and can be reused in different contexts (for example with other modeling languages than Fiacre).

There are several directions for future works. We plan to define a compositional pattern language inspired by the “denotational interpretation” used in the definition of patterns (see Chapter 4). The idea is to define a lower-level pattern language, with more composition operators, that is amenable to an automatic translation into a verification method.

## 7. CONCLUSIONS

---

In Chapter 5 and Chapter 6 of this thesis, we propose our approach to verify properties. Our approach is based on observers. We have defined, first, a set of TTS observers for our catalogue of real time specification patterns. To choose the best way to verify a pattern, we defined, for each pattern, a set of non-intrusive observers. Second, we have presented our observers based on Fiacre language. After comparing the two approaches, we noticed that while TTS observers are quite expressive, it is more difficult to maintain a tool that is based on TTS observers rather than on Fiacre observers and that the difference of performance is negligible. Fiacre observers, indeed, can be maintained easily since they are defined using the same host language.

Our verification approach has been integrated into a complete verification tool chain for the Fiacre modelling language and can therefore be used in conjunction with Topcased (FGC<sup>+</sup>06). We give several experimental results based on the use of this tool chain in Sect. 5.2. The fact that we implemented our approach has influenced our definition of the observers. Indeed, another contribution of our work is the use of a pragmatic approach for comparing the effectiveness of different observers for the same property. Our experimental results seem to show that data observers look promising.

We are following several directions for future work. A first goal is to define a new low-level language for observers—adapted from the TTS model—equipped with more powerful optimization techniques and with easier soundness proofs. On the experimental side, we need to define an improved method to select the best observer. For instance, we would like to provide a tool for the “syntax-directed selection” of observers that would choose (and even adapt) the right observers based on a structural analysis of the target system.

Finally, we have presented in Chapter 3 and Chapter 6 two complementary methods for checking the correctness of our verification method. While the “semantics-based” method rely on an heavyweight mathematical proof, the graphical method can be used to quickly gain confidence on the implementation of our model-checking tools. We use this method to reason about the observers before doing mathematical proof. For future works, we are currently looking into the use of mechanized theorem proving techniques to support the validation of observers.

## Appendix A

# Catalogue of Real Time Patterns

In this section, we remind our catalogue of real time specification patterns. We present each pattern with its corresponding TTS observer and Fiacre observer as well as the LTL formula that needs to be checked on the composition of the system with the observer.

In the following, we use  $E1$  and  $E2$  to present the transitions corresponding to events  $A$  and  $B$  respectively.

### A.1 Existence Patterns

An existence pattern is used to express that, in every trace of the system, some events must occur.

#### **Present $A$ after $B$ within $[d1, d2]$**

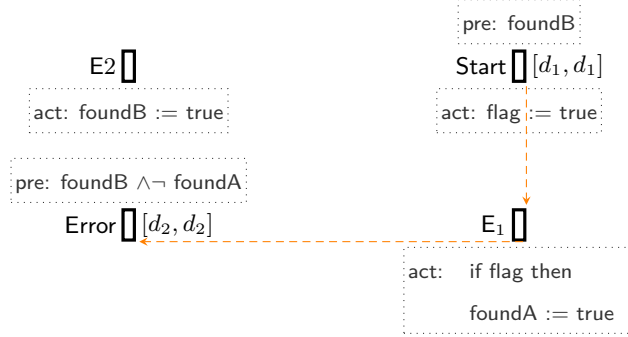
An occurrence of predicate  $A$  must hold between  $d1$  and  $d2$  u.t after the first occurrence of  $B$ . The pattern is also satisfied if  $B$  never holds.

- TTS Observer: The LTL formula to verify is  $\Box \neg \text{Error}$



## A. CATALOGUE OF REAL TIME PATTERNS

---



- Fiacre Observer: The LTL formula to verify is:  $\Diamond E2 \Rightarrow \Diamond ok$

```

process observer [E1, E2 : sync] is

  states idle, start, watch, error, stop

  from idle
    E2;
  to start

  from start
    wait [d1,d1];
  to watch

  from watch
    select
      E1; to stop
    unless
      wait ]d2-d1,...[; to error
    end

component obs is

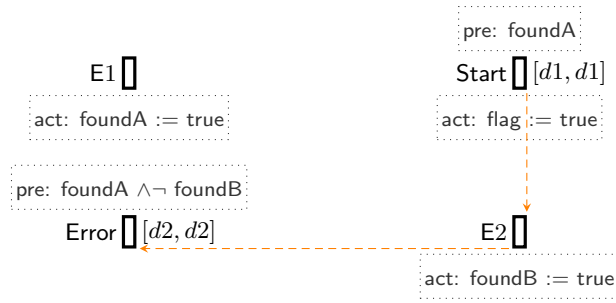
  port E1 : sync in [0,0] is ...,
    E2 : sync in [0,0] is ...

  par observer[E1, E2] end
  
```

## Present $A$ before $B$ within $[d1, d2]$

The first occurrence of predicate  $A$  holds between  $d1$  and  $d2$  u.t. before the first occurrence of  $B$ . The pattern is also satisfied if  $B$  never holds. (The difference with Present  $B$  after  $A$  within  $[d1, d2]$  is that  $B$  should not occur before the first  $A$ ).

- TTS Observer: The LTL formula to verify is  $(\Diamond B) \Rightarrow \neg \Diamond (\text{Error} \vee (\text{foundB} \wedge \neg \text{flag}))$



- The LTL formula to verify is:  $\Diamond E2 \Rightarrow \neg \Diamond \text{error}$

```

process observer1 [E1: sync] (&flag:bool, &foundE2:bool) is

    states idle, start, watch, error

    from idle
        E1;
    to start

    from start
        wait [d1,d1]; flag:=true;
    to watch

    from watch
        wait ]d2-d1,...[; flag:=false;
        on foundE2=false;
    to error

process observer2 [E2: sync] (&foundE2:bool,&flag:bool) is

    states idle, stop

```

## A. CATALOGUE OF REAL TIME PATTERNS

---

```

from idle
    E2; foundE2:=true;
    on flag=true;
to stop

component obs is

    var flag:bool:=false ,
        foundE2:bool:=false

    port E1: sync in [0,0] is ... ,
        E2: sync in [0,0] is ...

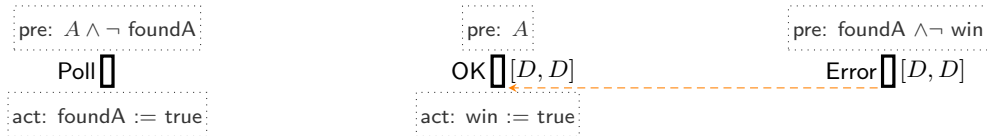
    par
        observer1 [E1] (&flag , &foundE2)
    ||
        observer2 [E2] (&foundE2,&flag)
    end

```

### Present $A$ lasting $D$

Starting from the first occurrence when the predicate  $A$  holds, it remains true for at least duration  $D$ . The pattern makes sense only if  $A$  is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration.

- TTS Observer: The LTL formula to verify is  $\Box \neg \text{Error}$



- The LTL formula to verify is:  $\Box \neg \text{error}$

```

process observer [E1: sync , notE1:sync] is
    states idle , firstE1 , error , stop

```

```

from idle
    E1;
to firstE1

from firstE1
    select
        notE1; to error
    unless
        wait [D,D]; to stop
    end

component obs is
    port
        E1:sync in [0,0] is ...,
        notE1: sync in [0,0] is not ...
    par
        observer [E1, notE1]
    end

```

### Present $A$ within $[d1, d2]$

The pattern holds if there is an occurrence of predicate  $A$  holding in the interval  $I$ . This pattern can be viewed as the pattern “present  $A$  after *init* within  $[d1, d2]$ ”

- TTS Observer: The same as “present  $A$  after *init* within  $[d1, d2]$ ”
- The LTL formula to verify is:  $\Box \neg error$

```

process observer [E1: sync] is
    states idle, watch, error, stop
    from idle
        wait [d1,d1];
        to watch
    from watch
        select
            E1; to stop
        unless

```

## A. CATALOGUE OF REAL TIME PATTERNS

---

```

        wait ]d2-d1,...[; to error
    end

component obs is
    port E1 : sync in [0,0] is ...

    par observer[E1] end

obs

```

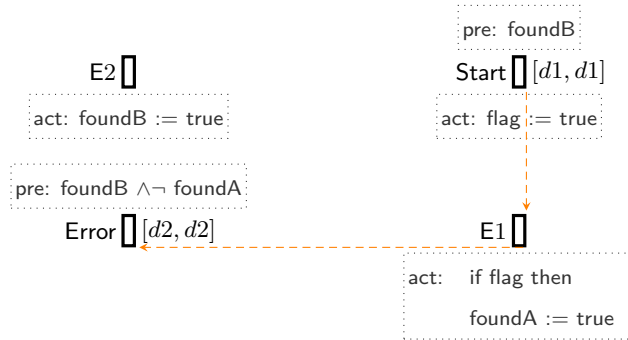
### A.2 Absence patterns

Absence patterns are used to express that some condition should never occur.

#### Absent $A$ after $B$ for interval $[d1, d2]$

Predicate  $A$  must never hold between  $d1-d2$  u.t. after the first occurrence of  $B$ . This pattern is dual to *Present  $A$  after  $B$  within  $[d1, d2]$*  (it is not equivalent to its negation because, in both patterns,  $B$  is not required to occur).

- TTS Observer: The LTL formula to verify is  $\Diamond B \Rightarrow \Diamond Error$



- Fiacre Observer: The LTL formula to verify is:  $\Diamond E2 \Rightarrow \Diamond ok$

```

process observer [E1,E2 : sync] is

    states idle , start , watch , error , ok

```

```

from idle
    E2;
to start

from start
    wait [d1,d1];
to watch

from watch
    select
        E1; to error
    unless
        wait ]d2-d1,...[; to ok
    end

component obs is

    port E1 : sync in [0,0] is ...,
        E2 : sync in [0,0] is ...

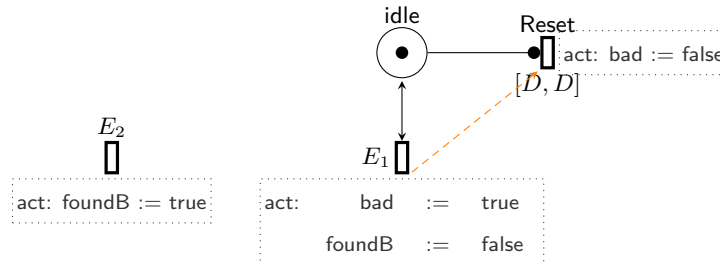
    par observer[E1, E2] end

```

## Absent $A$ before $B$ for duration $D$

No  $A$  can occur less than  $D$  u.t. before the first occurrence of  $B$ . The pattern holds if there are no occurrence of  $B$ .

- TTS Observer: The LTL formula to verify is  $\Box \neg (\text{foundB} \wedge \text{bad})$



## A. CATALOGUE OF REAL TIME PATTERNS

---

- Fiacre Observer: The LTL formula to verify is:  $\neg\Diamond(foundE2 \wedge bad)$

```
process observer1 [E1: sync , E3:none] (&bad:bool,&foundE2:bool) is

  states idle ,ok

  from idle
    select
      E3;
      bad:=false; to ok
    []
      E1;
      bad:=true;
      foundE2:=false; to idle
    end

process observer2 [E2: sync] (&bad:bool,&foundE2:bool) is

  states idle , stop

  from idle
    E2;
    foundE2:=true;
  to stop

component obs is

  var bad:bool:=false ,
      foundE2:bool:=false

  port E1: sync in [0,0] is ... ,
        E2: sync in [0,0] is ...
        E3: none in [D,D]

  priority E1 -> E3

  par observer1 [E1, E3](&bad,&foundE2)
    ||
```

```

    observer2 [E2](&bad,&foundE2)
end

```

### Absent $A$ for interval $[d1, d2]$

This pattern is equivalent to the pattern “absent  $A$  after *init* for interval  $[d1, d2]$ ”.

- TTS Observer: The same as the observer of the pattern “absent  $A$  after *init* for interval  $[d1, d2]$ ”.
- Fiacre Observer: The LTL formula to verify is:  $\Diamond ok$

```

process observer [E1: sync] is

    states idle , watch , error , ok

    from idle
        wait [d1,d1];
    to watch

    from watch
        select
            E1; to error
        unless
            wait ]d2-d1 , ... [; to ok
        end

component obs is

    port E1 : sync in [0,0] is ...

par observer[E1] end

```



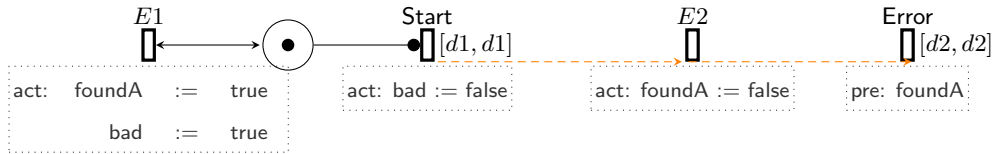
### A.3 Response patterns

Response patterns are used to express “cause–effect” relationship, such as the fact that an occurrence of a first kind of events must be followed by an occurrence of a second kind of events.

#### *A leadsto first B within $[d1, d2]$*

Every occurrence of *A* must be followed by an occurrence of *B* within time interval  $[d1, d2]$  (considering only the first occurrence of *B* after *A*).

- TTS Observer: The LTL formula to verify is  $(\Box \neg \text{Error}) \wedge (\Box \neg (B \wedge \text{bad}))$



- Fiacre Observer: The LTL formula to verify is:  $\Box \neg \text{error}$

```

process observer [E1, E2 : sync] is

  states idle , start , watch , error

  from idle
    E1;
  to start

  from start
    wait [d1,d1];
  to watch

  from watch
    select
      E2; to idle
    unless
      wait ]d2-d1,...[; to error
    end

```

```

component obs is

    port E1 : sync in [0,0] is ...,
        E2 : sync in [0,0] is ...

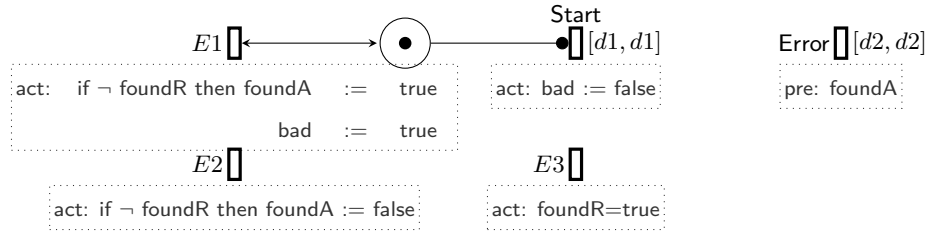
    par observer[E1, E2] end

```

#### *A* leadsto first *B* within $[d1, d2]$ before *R*

Before the first occurrence of *R*, each occurrence of *A* is followed by a *B*—and these two events occur before *R*—in the time interval  $[d1, d2]$ . The pattern holds if *R* never occur.

- TTS Observer: The LTL formula to verify is  $\Diamond R \Rightarrow (\Box \neg \text{Error} \wedge \Box \neg (B \wedge \text{bad}))$



- Fiacre Observer: The LTL formula to verify is:  $\Diamond E3 \Rightarrow \Box \neg \text{error}$

```

process observer1 [E1, E2 : sync] (&foundE3: bool) is

    states idle, start, watch, error

    from idle
        E1;
        on foundE3=false;
    to start

    from start
        wait [d1,d1];
    to watch

    from watch
        select

```

## A. CATALOGUE OF REAL TIME PATTERNS

---

```

        E2; on foundE3=false; to idle
    unless
        wait ]d2-d1,...[; to error
    end

process observer2 [E3: sync] (&foundE3: bool) is

    states idle, stop

    from idle
        E3;
        foundE3:=true;
    to stop

component obs is

    var foundE3:bool:=false

    port E1 : sync in [0,0] is ...,
        E2 : sync in [0,0] is ...,
        E3 : sync in [0,0] is ...

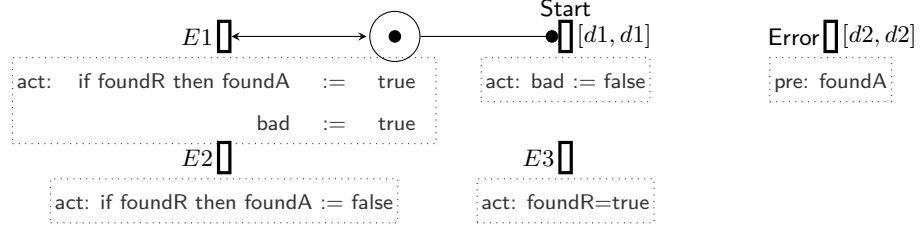
    par
        observer1 [E1, E2] (&foundE3)
    ||
        observer2 [E3] (&foundE3)
    end

```

### ***A* leadsto first *B* within $[d1, d2]$ after *R***

Same than with the pattern “*A* leadsto first *B* within  $[d1, d2]$ ” but only considering occurrences of *A* after the first *R*.

- TTS Observer: The LTL formula to verify is  $\Diamond R \Rightarrow (\Box \neg \text{Error} \wedge \Box \neg (B \wedge \text{bad}))$



- Fiacre Observer: The LTL formula to verify is:  $\Diamond E3 \Rightarrow \Box \neg error$

```

process observer [E1, E2, E3 : sync] is

    states idle, start, watch, beginObs, error

    from idle
        E3;
    to start

    from start
        E1;
    to beginObs

    from beginObs
        wait [d1,d1];
    to watch

    from watch
        select
            E2; to idle
        unless
            wait ]d2-d1,...[; to error
        end

component obs is

    port E1 : sync in [0,0] is ...,
        E2 : sync in [0,0] is ...,
        E3 : sync in [0,0] is ...

    par observer[E1, E2, E3] end
    
```

### A.4 Universality patterns

Universality patterns are used to express that some condition should always occur. Universality patterns can be viewed as the dual of absence patterns.

#### **always $A$ lasting $D$**

This pattern is equivalent to “absent  $\neg A$  after *init* for interval  $[d1, d2]$ ”.

#### **always $A$ after $B$ for duration $D$**

This pattern is equivalent to “absent  $\neg A$  after  $B$  for interval  $[D, D]$ ”.

#### **always $A$ before $B$ for duration $D$**

This pattern is equivalent to “absent  $\neg A$  before  $B$  for duration  $D$ ”.

## Appendix B

# pFrac: Frac with Real Time Properties

In this section, we describe our prototype software that has been developed in order to experiment with the verification of patterns in Fiacre. The software, called pFrac<sup>1</sup>, is a Fiacre language compiler which takes into account real time properties. It is an extension of the Frac<sup>2</sup> compiler (Fiacre language compiler). pFrac includes our method to specify properties based on patterns and our verification approach based on observers. pFrac transforms each Fiacre program (input) into a TTS model (output). The TTS model generated is composed of the TTS of the system and the TTS of the used observer. pFrac gives also the possibility to choose an observer among our proposed observers: state observer, data observer, transition observer or Fiacre observer.

Next, we start with a brief technical description of the compiler

### B.1 Technical Description

As for frac—with which it shares a large part of its code base—pFrac is a Fiacre compiler implemented in Standard ML that outputs TTS files (in the .tts format). pFrac takes as

---

<sup>1</sup><http://homepages.laas.fr/nabid/pfrac.html>

<sup>2</sup><http://projects.laas.fr/fiacre/>

## B. PFRAC: FRAC WITH REAL TIME PROPERTIES

---

input a Fiacre program (a .fcr file) and outputs a “TTS folder” containing the following files:

- .net file: contains the states, the transitions and the priorities of the system;
- .c file: contains a C program which manipulates shared variables of the systems. It associates each transition with its guard and actions.
- .ltl file: contains the LTL formulas to check (when there are properties to be verified). pFrac associates an LTL formula to each property defined in the input fiacre (.fcr) file.
- .h file: contains a set of headers for the C part of the TTS.

### B.2 Installation

pFrac is available as executable binaries for a variety of architectures (the source code is not yet freely available). Downloads are available on the author web page, see <http://homepages.laas.fr/nabid/pfrac.html>.

The following steps describes how to install pFrac:

1. download and unpack the adequate distribution from the web site. This will create a folder with PFRACDIR its path;
2. set the value of variable FRACLIB in FRACDIR/Makefile to FRACDIR/lib;
3. add directory FRACDIR to your PATH environment variable.

### B.3 Usage

pFrac binaries are command-line programs (see Figure B.1). A normal execution is performed by giving the optional flags followed by a valid input file. As input, it accepts only models specified using Fiacre language written using the .fcr format. It outputs a TTS folder of the system.

The exploration of a system will generates a .TTS output. In this case, pFrac will generate a textual File containing a description of the states, transitions and priorities

```
usage: pFrac [-h | -help]
           [-q | -v | -obsT | -obsS | -obsD | -obs -properties]
           [-o errfile] [-p | -c | -f | -t]
           [-flat | -share] [-unsafe] [-O | -g] [-strip] [-arch]
           [infile] [outfile]
```

Figure B.1: pFrac usage syntax.

of the system; a .c file containing a description of the shared variables of the system and an .ltl file containing the corresponding LTL formula to be verified when there are properties defined.

pFrac supports different versions for checking properties depending on the observer used. The choice of the observer to be used depends on the the flag used. We give a description of the set of flags used in pFrac in Fig. B.2.

To check if a system verifies a property, we add the corresponding pattern into the Fiacre specification. We use the double-click example (see Chapter 3) to demonstrate the use of our prototype. In this case, we check if we have a double click in less than 1 u.t (see listing below).

```
process Push [ click:none, single:none,
              double:none, delay:none] is
states s0, s1, s2
var dbl : bool := false
from s0 click; to s1
from s1
  select
    click; dbl := true; loop
  [] delay; to s2
end
from s2
  if dbl then double
```



## B. PFRAC: FRAC WITH REAL TIME PROPERTIES

---

```
    else single end;
    dbl := false;
to s0

component Mouse [click:none, single:none,
                double:none] is
port delay : none in [1,1]
priority delay > click
par
    Push [click, single, double, delay]
end

property double is Mouse/1/state s0 leadsto Mouse/1/state s2 within [1,1]
assert double
```

Then the output file will be the input of the TINA toolbox and selt model checker.

FLAGS	WHAT	DEFAULT
-h   -help	this mode	
error format:		
-q	quiet	-q
-v	verbose	-q
-obsT	use transition observer	-obsT
-obsS	use state observer	-obsS
-obsD	use data observer	-obsD
-obs -properties	use fiacre observer	-obs -properties
-o errfile	save errors in errfile (stderr if -)	stderr
output format:		
-p	just parses	-c
-c	just parses and checks	-c
-f   -fcr	checks then outputs fiacre source (implicit if outfile has extension .fcr)	-c
-t   -tts	checks then compiles to .tts (implicit if outfile has extension .tts)	-c
code generation options:		
-share	enables sharing of state components	-share
-flat	disables sharing of state components	-share
-unsafe	warns but dont fail on shv interferences	
-O   -g	optimizes or not the code generated	-O
other options:		
-strip	forgets properties and assert directives	
-arch	prints component hierarchy	
files:		
infile	input file	stdin
outfile	output file	stdout

Figure B.2: pFrac options

# Vérification des Propriétés

## Temps Réel dans le Langage Fiacre

### Résumé en Français

## Annexe C

# Résumé en Français

Dans cette thèse, nous nous intéressons à la problématique de la vérification formelle des systèmes critiques temps réel, c'est-à-dire des systèmes dont l'exécution dépend de certaines contraintes temporelles. La spécification formelle des exigences pour de tels systèmes, ainsi que leur vérification, reste une tâche très compliquée, surtout pour les non experts.

Plusieurs solutions ont été proposées pour faciliter la spécification et la vérification des systèmes temps-réels. Un premier type d'approche est basée sur la définition d'un ensemble de patrons de spécification qui représentent les propriétés les plus utilisées en pratique. Cependant, ce type de solutions n'est pas toujours supporté par un outillage de vérification efficace, dans le sens que les auteurs de ces langages de patrons ne fournissent pas directement une implantation pour leur langage. Un second type d'approches repose sur l'utilisation du formalisme des logiques temporelles pour spécifier les propriétés à vérifier et sur les techniques de model-checking pour leur vérification. S'agissant de systèmes temps-réels, il est dans ce cas nécessaire d'utiliser des extensions temporisées des logiques temporelles. Cependant, ces approches donnent le plus souvent lieu à des problèmes de model-checking qui sont indécidable, ou dont la complexité en pratique

## C. RÉSUMÉ EN FRANÇAIS

---

est très élevée.

Dans ce travail, nous suivons la première approche et proposons un langage de patrons de propriétés temps-réels accompagnés d'un outil de vérification par model-checking. Nous apportons plusieurs contributions à ce domaine. Nous proposons un cadre théorique complet pour la spécification et la vérification de patrons de propriétés temps réel. (Ces patrons peuvent être interprété comme une extension temps réel du langage de propriétés défini par Dwyer et al (DAC99).) Notre approche a été implantée dans le contexte du langage de modélisation Fiacre. Enfin, nous définissons deux méthodes complémentaires permettant de vérifier la correction de notre approche de vérification.

[**Mots-clés:**] vérification formelle; systèmes temps réel; langage d'exigences; patron de spécification; model-checking; observateurs

## Quelles motivations pour la vérification formelle?

Au cours des dernières années, la vérification formelle des systèmes critiques s'est progressivement établie comme une discipline à part entière de l'informatique. Cette évolution est, d'une part, motivé par le fait que ces systèmes sont de plus en plus présents dans notre vie quotidienne et qu'ils ont un impact majeur sur nos sociétés. Cette évolution s'explique également par le fait qu'il s'agit d'un domaine scientifique très vaste, source de nombreux problèmes à la fois intéressant et difficile à résoudre.

La vérification formelle peut être vu comme un domaine de l'informatique qui s'intéresse à vérifier—ou à prouver—qu'un système ou un logiciel est correct par rapport aux exigences fixées au départ. La vérification formelle est une technique largement utilisée pour vérifier la correction *d'un systèmes critiques* tels que ceux trouvés dans le domaine de l'aéronautique ou le nucléaire par exemple. Dans ce contexte, le terme critique

---

est utilisé pour souligner qu’une défaillance, un dysfonctionnement, ou un problème dans la conception du système peut avoir des conséquences catastrophiques. Sans revenir sur les exemples impliquant la mort de personnes impliqués dans des “accidents informatique”, on peut souligner l’évaluation faite par les États-Unis concernant les pertes liés aux “accidents des systèmes” et qui s’élèvent à environ 3 milliards de dollars par an. Dans un même ordre d’idées, les compagnies aériennes estiment que chaque minute de temps d’arrêt lié à un problème sur un de leurs systèmes critique coûte environ 70 k\$.

En même temps que ce besoin de vérification des systèmes se fait sentir, nous devons également faire face à une augmentation de la complexité des systèmes. En effet, la demande du marché pour des solutions plus efficaces et automatisées a poussé la complexité des systèmes embarqués à des niveaux jamais imaginés. Pour cette raison, nous avons besoin de nouvelles méthodes et de nouveaux outils pour vérifier les systèmes à une étape avancée de leur réalisation. Pour résoudre à ce enjeux, nous devons pouvoir répondre aux questions suivantes:

1. Comment exprimer les exigences d’un système?
2. Quelles propriétés doivent être vérifiées?
3. Comment vérifier des propriétés dans un système?
4. Comment prouver que les résultats obtenus après vérification sont correctes?

## Spécification formelle des exigences

Pour répondre à la troisième question, Edmund M. Clarke et Allen Emerson (CE82) ainsi que Joseph Sifakis et Jean-Pierre Queille (QS82) ont défini en 1980 l’approche de *model checking*. Le model-checking peut être considéré comme un ensemble de techniques automatisées pour vérifier si “le modèle d’un système” répond à ses exigences par une

## C. RÉSUMÉ EN FRANÇAIS

---

exploration exhaustive de l'espace d'état du modèle.

Les techniques de model-checking ont attiré l'attention de beaucoup de chercheurs, quel ce soit dans le monde académique ou industriel, principalement parce qu'il repose sur une approche automatique; on dit souvent qu'il s'agit d'une méthode “push button”. Un inconvénient de cette approche est le problème dit de l'explosion de l'espace d'état, ce qui signifie que l'espace d'état construit pendant la vérification du système peut augmenter exponentiellement en fonction de la taille (la complexité) du système.

Dans le contexte de l'approche de model checking, les exigences sont souvent exprimées en utilisant *des logiques temporelles* qui peuvent être considérées comme des solutions à la première et la deuxième question citée en haut. Un problème limitant l'utilisation de l'approche model checking dans l'industrie est la difficulté, pour les non experts, d'exprimer leurs besoins en utilisant les langages de spécification pris en charge par l'approche de model checking. En effet, il y a souvent un écart important entre les approches utilisées pour la déclaration des exigences et les formalismes de bas niveau utilisés par les model-checkers. Cette limitation a motivé la définition de langages de spécification dédiés, permettant d'exprimer des propriétés à un plus haut niveau d'abstraction. Toutefois, un nombre limité d'approches supportent la notion de contraintes temporelles et la plupart ne sont pas associées à un outil de vérification automatique, tel qu'un model-checker.

Une des motivations principale pour le travail accompli au cours de ma thèse est de surpasser les lacunes des solutions existantes. Le but de mon travail a été de développer un cadre complet pour spécifier et vérifier des propriétés dans le cadre des *systèmes critiques temps réel*. Dans ce travail, les systèmes sont modélisés en utilisant le langage Fiacre (BBF<sup>+</sup>08)—un langage formel pour la spécification de systèmes temps réel développés dans notre équipe—alors que nous avons utilisé la boîte à outil Tina (BRV04) pour la vérification. En particulier, Tina inclut un model-checker pour

---

une extension des réseaux de Petri temporisé. Tout au long de notre étude, nous nous sommes fixé un ensemble clair d'objectifs:

- développer une méthode pour spécifier les exigences qui soit simple à définir et à expliquer;
- proposer une approche efficace, dans la pratique, pour la vérification de ces exigences;
- définir une méthode pour vérifier que nos approches de vérification sont correctes.

Au cours de ce travail, nous avons proposé un cadre formel complet qui contient une approche pour spécifier des propriétés, une approche pour leur vérification et des méthodes pour prouver la correction des méthodes de vérification. Notre première contribution est la définition d'un ensemble de patrons de spécification qui étendent le langage de spécification de Dwyer (CDHR00, DAC99) pour prendre en compte des contraintes temps réel. Par exemple, nous définissons le patron “*Present A after B within [0, 4]*” pour exprimer que l'événement *A* doit arriver après un maximum de 4 unités de temps (ut) après la première occurrence de *B*.

Notre principal objectif est de proposer une alternative aux extensions temporisées des logiques temporelles pour des approches model checking. Nos modèles sont conçus pour exprimer des contraintes temporelles générales couramment utilisées dans l'analyse des systèmes temps réel (tels que le respect des délais, durée de l'événement, etc). Ils sont également conçus pour être simple, à la fois en terme de clarté (linguistique) et en terme de complexité de calcul. En particulier, chaque patron doit être associé à un problème de model-checking décidable.

## Contributions de la thèse

Les contributions de cette thèse peuvent être divisées en trois axes: (1) la spécification des propriétés, (2) la vérification des propriétés, et (3) la preuve de correction de la



## C. RÉSUMÉ EN FRANÇAIS

---

méthode de vérification.

Chronologiquement, nous avons commencé notre travail en étudiant l'ensemble des événements qui peuvent être pris en compte dans le langage Fiacre. Ensuite, nous avons étudié les approches existantes utilisées pour spécifier les propriétés dans le but de fournir une approche simple pour les utilisateurs non expert et de surpasser les problèmes existants avec les approches basées sur les logiques temporelles. En particulier les problèmes liés à la décidabilité ou les problèmes de complexité liés aux méthodes de vérification.

Notre première contribution est la définition d'un ensemble de patrons de spécification qui peut être considéré comme une extension temps réel des patrons de Dwyer (AZB12b). (Une étude récente (BGPS12) a montré que les patrons de Dwyer sont la classe de patrons de spécification les plus utilisés en pratique; que ce soit dans l'industrie ou dans le milieu académique.) Notre approche est originale pour divers raisons. Tout d'abord, nous présentons une manière simple de spécifier les exigences, puisque notre approche est basée sur l'utilisation du langage naturel et est restreint à l'utilisation de quelques mots clefs. Ensuite, nous proposons un environnement de vérification basé sur l'utilisation d'observateurs, en transformant chaque propriété vers la vérification d'une formule LTL sur la composition d'un système avec un observateur. Enfin, nous proposons un formalisme graphique—que nous avons nommé Timed Graphical Interval Logic (TGIL) (ADZ12)—qui permet de décrire la sémantique (le sens) des patrons. L'idée est de fournir une alternative à la définition formelle des patrons de spécification.

L'approche définie dans cette thèse a pour but de faciliter le travail des ingénieurs qui ne sont pas experts dans les techniques de vérification formelle. Nos modèles présentent un cadre simple pour spécifier des propriétés qualitatives et quantitatives. Certes notre approche est moins expressive que les logiques temporelles, mais elle est suffisante pour présenter les propriétés les plus utilisées en pratique. De plus, nous travaillons dans un

---

cadre décidable et nous fournissons des algorithmes de vérifications dont la complexité est “équivalente” à ceux utilisés pour vérifier des propriétés comportementales.

Après la définition de notre langage de patrons de spécification, nous avons étudié les méthodes de vérification existantes. Notre deuxième contribution consiste à définir une méthode de vérification pour les système temps réel basée sur l’utilisation d’observateurs (AZB12a) et de techniques de model-checking. Pratiquement, nous transformons le problème de vérifier une pattern en la vérification d’une formule LTL simple. Nous avons fait le choix d’une approche par observateurs parce qu’il s’agit d’une méthode simple à utiliser en pratique et qui ne nécessite pas de définir un autre langage que celui utilisé pour décrire le système. Bien que l’approche par observateurs soit classique, notre contribution est originale par plusieurs points. Tout d’abord, nous définissons différentes catégories d’observateurs pour chaque patron. Ensuite, nous proposons une démarche pragmatique pour sélectionner l’observateur le plus efficace en pratique en termes de temps de vérification et d’augmentation de l’espace d’états du système.

Enfin, nous avons défini un cadre formel pour prouver que nos observateurs sont corrects et non-intrusifs, ce qui signifie qu’ils n’ont pas d’impact sur le comportement du système qu’ils observent (ADZB12). Nous avons défini deux méthodes complémentaires de vérification: une preuve graphique et une preuve formelle. Les deux méthodes sont complémentaires. Nous utilisons la méthode graphique pour raisonner sur l’exactitude des observateurs à une étape avancée du processus de développement, tandis que nous utilisons la méthode formelle pour prouver formellement, en se basant sur des preuves mathématiques, que nos observateurs sont corrects.

## Brève description des chapitres de la thèse

Nous donnons un bref aperçu du contenu des chapitres de cette thèse.

**Chapitre2 — État de l’art:** ce chapitre présente le contexte général de cette thèse.

## C. RÉSUMÉ EN FRANÇAIS

---

Nous présentons brièvement les différentes étapes de vérification des systèmes temps réel (modélisation, spécification et vérification), puis nous discutons des travaux pertinents au contexte de cette thèse. Nous présentons, tout d'abord, les travaux liés à la modélisation des systèmes temps réel, puis nous présentons les travaux liés aux méthodes de vérification formelle existants. Nous essayons de mettre l'accent sur les inconvénients de chaque approche afin de pouvoir souligner les lacunes dans les approches existantes. Nous concluons ce chapitre par une présentation détaillée des contributions de cette thèse.

**Chapitre3 — Cadre formel:** ce chapitre décrit le cadre formel ainsi que les outils mathématique utilisé dans le reste de la thèse. Tout d'abord, nous commençons par une présentation du langage Fiacre et une définition formelle des *Time Transition System* (TTS). TTS est une extension des réseaux de Petri temporel prenant en compte des variables partagées et des priorités. Ensuite, nous définissons formellement certaines notions utiles dans notre approche pour prouver la correction de notre méthode de vérification. Nous présentons aussi les différentes méthodes utilisées pour définir la sémantique des propriétés.

**Chapitre4 — Patrons de spécification temps réel:** ce chapitre décrit notre catalogue de patrons de spécification temps réel. Nous commençons par définir l'ensemble des événements observables dans le langage Fiacre. Ensuite, nous définissons la liste de nos patrons de spécification temps réel. Nous présentons un cas d'utilisation pour montrer l'utilité de nos propriétés dans un cas réaliste. Avant de conclure, nous présentons le méta modèle de nos patrons de spécification.

**Chapitre5 — Vérification des patrons en utilisant des observateurs TTS:** dans ce chapitre nous décrivons notre méthode de vérification basée sur l'utilisation des observateurs au niveau TTS. Nous nous concentrons d'abord sur un patron spécifique, puis nous définissons pour ce modèle un ensemble d'observateurs. Nous

---

utilisons une méthodologie pour essayer de sélectionner l’observateur le plus efficace en pratique. Nous prouvons par la suite que nos observateurs sont corrects. Avant de conclure ce chapitre, nous définissons un observateur pour chaque patron de notre catalogue.

**Chapitre6 —Vérification des patrons en utilisant des observateurs Fiacre:** ce

Chapitre décrit une autre méthode de vérification basée sur l’utilisation d’observateurs au niveau Fiacre. Nous avons défini une nouvelle extension du langage Fiacre qui permet la définition de ce qu’on appelle “sondes”. Avec cette extension, les observateurs peuvent alors être définis comme des programmes Fiacre à part entière. Nous définissons, pour chaque patron, un observateur Fiacre. Nous présentons, par la suite, une approche graphique qui permet de vérifier la correction des observateurs à une étape avancée de leur processus de développement. Avant de conclure, nous comparons nos deux approches de vérification, les observateurs TTS d’une part et les observateurs Fiacre d’autre part.

**Chapitre7 — Conclusion:** La thèse se conclue par une discussion sur les axes de travaux futurs.

# Bibliography

- [ABBL03] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003. 26, 29
- [ABK04] A. Alfonso, V. Braberman, and N. Kicillof. Visual timed event scenarios. In *Proceedings of the 26th International Conference on Software Engineering*, 2004. 49
- [ABL98] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In Steffen (Ste98), pages 263–280. 26, 29
- [AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming, ICALP '90*, pages 322–335, London, UK, UK, 1990. Springer-Verlag. 9, 11
- [AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 12, 21
- [ADZ12] Nouha Abid and Silvano Dal Zilio. Timed Graphical Interval Logic. Rapport de recherche, LAAS, 2012. 5, 142

- [ADZB12] Nouha Abid, Silvano Dal Zilio, and Bernard Berthomieu. Who Check the Model Checker? Rapport de recherche, LAAS, 2012. 6, 143
- [AH90] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104:390–401, 1990. 21
- [AZB12a] Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. A Verified Approach for Checking Real-Time Specification Patterns. In *VECOS 2012*, 2012. 5, 143
- [AZB12b] Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. A real-time specification patterns and tools. In *FMICS*, Lecture Notes in Computer Science. Springer, 2012. 5, 142
- [BBDZ<sup>+</sup>10] Bernard Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gauffillet, Sebastien Heim, and Francois Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *ERTSS 2010 - Embedded Real-Time Software and Systems*, pages 1–9, TOULOUSE (31000), France, May 2010. 9 pages DGE Topcased. 11, 80
- [BBF<sup>+</sup>08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gauffillet, Frederic Lang, and François Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse, France, 2008. 4, 13, 14, 33, 74, 140
- [BBFH<sup>+</sup>12] J.-P. Bodeveix B. Berthomieu, M. Fillali, G. Hubert, F. Lang, F. Peres, R. Saad, S. Jan, and F. Vernadat. The Syntax and Semantics of Fiacre – Version 3.0. Technical report, 2012. 33, 41
- [BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous

## BIBLIOGRAPHY

---

- real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society. 33
- [BDL04] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004. 24
- [BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zürich, Switzerland, pages 992–1000. IEEE Computer Society Press, June 2012. SEiP track acceptance rate: 18.5% (20/108). 5, 23, 29, 70, 115, 142
- [Bit01] Friedemann Bitsch. Safety patterns - the key to formal specification of safety requirements. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '01, pages 176–189, London, UK, UK, 2001. Springer-Verlag. 23, 29
- [BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, March 2000. 20
- [BRV<sup>+</sup>03] Bernard Berthomieu, Pierre-Olivier Ribet, François Vernadat, J. L. Bernartt, Jean-Marie Farines, Jean-Paul Bodeveix, Mamoun Filali, Gérard Padiou, Pierre Michel, Patrick Farail, Pierre Gauffilet, Pierre Dissaux, and Jean-Luc Lambert. Towards the verification of real-time systems in avionics: the cotre approach. *Electr. Notes Theor. Comput. Sci.*, 80:203–218, 2003. 9, 14
- [BRV04] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction

- of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42-No 14, 2004. 4, 24, 78, 110, 140
- [CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976. 16
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. 15
- [CCO<sup>+</sup>04] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag, 2004. 24
- [CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software, 2000. 4, 22, 141
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. 3, 9, 17, 139
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986. 20



## BIBLIOGRAPHY

---

- [CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. 17
- [Cla08] Edmund M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008. 18
- [COR<sup>+</sup>95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. April 1995. 16
- [CP99] Marsha Chechik and Dimitrie O. Paun. Events in property patterns. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 154–167, London, UK, UK, 1999. Springer-Verlag. 23, 29
- [DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999. 4, 22, 29, 56, 58, 138, 141
- [DBL97] *6th IEEE Workshop on Future Trends of Distributed Computer Systems (FTDCS '97), 29-31 October 1997, Tunis, Tunisia, Proceedings*. IEEE Computer Society, 1997. 157
- [DHQ<sup>+</sup>08] Jin Song Dong, Ping Hao, Sheng Chao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 52(1), 2008. 64, 80
- [Dij75] Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975. 13

- [DJC94] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Trans. Softw. Eng.*, 20(12):900–913, December 1994. 27
- [DKM<sup>+</sup>94] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3:131–165, 1994. 48, 49
- [DPC<sup>+</sup>09] Philippe Dhaussy, Pierre-yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In *In Springer Verlag Lecture Notes in Computer Science, 12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (MODELS'09), Volume 5795 (2009), pages 438-452.*, October 2009. 23
- [FGC<sup>+</sup>06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SysTEms Design. In *European Congress on Embedded Real-Time Software (ERTS)*, 2006. 14, 32, 74, 92, 116
- [FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute, 2006. 10, 74
- [GL02] Hubert Garavel and Frédéric Lang. Ntif: A general symbolic model for communicating sequential processes with data. In Doron Peled and Moshe Y. Vardi, editors, *FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2002. 14

## BIBLIOGRAPHY

---

- [GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006. 23, 29
- [GLMR05] Guillaume Gardey, Didier Lime, Morgan Magnin, and Olivier H. Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 418–423, Edinburgh, Scotland, UK, July 2005. Springer. 25
- [GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *in Proceedings of CASCON'97*, pages 169–183, 1997. 10
- [GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification, CAV '01*, pages 53–65, London, UK, UK, 2001. Springer-Verlag. 76, 79
- [GPV<sup>+</sup>95] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18, 1995. 26
- [Gru08] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 31–40, New York, NY, USA, 2008. ACM. 23
- [HKPM97] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, Août 1997. Version révisée distribuée avec Coq. 16

- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. 13
- [Hol81] Gerard J. Holzmann. Pan - a protocol specification analyzer. 1981. 13
- [Hol03] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003. 9, 13, 25
- [HPUB] Hesham Hallal, Alex Petrenko, Andreas Ulrich, and Sergiy Boroday. Using sdl tools to test properties of distributed systems. In *in Proc. of Formal Approches to Testing of Software (FATES'01), Workshop of the Int. Conference on Concurrency Theory (CONCUR'01)*, pages 125–140. 27
- [JM86] F Jahanian and A K Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(9):890–904, September 1986. 21
- [JMG88] C. Jard, J. F. Monin, and R. Groz. Development of veda, a prototyping tool for distributed algorithms. *IEEE Trans. Softw. Eng.*, 14(3):339–352, March 1988. 27
- [Jr.85] Richard J. Linn Jr. The features and facilities of estelle: a formal description technique based upon an extended finite state machine model. In Michel Diaz, editor, *PSTV*, pages 271–296. North-Holland, 1985. 27
- [Kat99] J.P. Katoen. *Concepts, algorithms, and tools for model checking*. Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich Alexander Universität, Erlangen- Nürnberg. Institut für Mathematische Maschinen und Datenverarbeitung, IMMD, 1999. 24
- [KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns.

## BIBLIOGRAPHY

---

- In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 372–381. ACM, 2005. 23, 29
- [Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. 15
- [Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990. 21, 46
- [KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. 13
- [ksu] <http://patterns.projects.cis.ksu.edu/>. Online Repository of Specification Patterns. 22
- [LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM. 26
- [Mer74] P.M. Merlin. *A Study of the Recoverability of Computing Systems*. University of California, Irvine, 1974. 12, 35, 36, 40
- [MMSR<sup>+</sup>96] L. E. Moser, P. M. Melliar-Smith, Y. S. Ramakrishna, G. Kutty, and L. K. Dillon. The real-time graphical interval logic toolset. In *In Proceedings of the Conference on Computer-Aided Verification, July/August*, pages 446–449. Springer-Verlag, 1996. 48, 52
- [MN08] Stephan Merz and Nicolas Navet. *Modeling and Verification of Real-Time Systems - Formalisms and Software Tools*. ISTE Publishing, 2008. 11

- [MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000. 9, 10
- [Neu94] Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *SIGSOFT Softw. Eng. Notes*, 19(1):16–29, January 1994. 2
- [NM10] N. Navet and S. Merz. *Modeling and Verification of Real-Time Systems*. John Wiley & Sons, 2010. 9, 10
- [OW06] Joël Ouaknine and James Worrell. Safety metric temporal logic is fully decidable. In *In Proceedings of TACAS 06, LNCS 3920*, pages 411–425. Springer, 2006. 46, 47
- [PBV11] Florent Peres, Bernard Berthomieu, and François Vernadat. On the composition of time petri nets. *Discrete Event Dynamic Systems*, 21(3):395–424, 2011. 42, 77
- [PC99] Dimitrie O. Paun and Marsha Chechik. Events in linear-time properties. In *In 4th Int. Conference on Requirements Engineering*, 1999. 94
- [Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962. 9, 12
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977. 19, 21
- [Pri57] A. N. Prior. *Time and Modality*. Clarendon Press, Oxford, 1957. 18
- [QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on*

## BIBLIOGRAPHY

---

- International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag. 3, 17, 139
- [Rey07] Pierre-Alain Reynier. *Vérification de systèmes temporisés et distribués : modèles, algorithmes et implémentabilité*. PhD thesis, Lab. Spécification & Vérification, ENS Cachan, France, June 2007. 12
- [Rus00a] John Rushby. From refutation to verification. In Tommaso Bolognesi and Diego Latella, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 369–374, Pisa, Italy, October 2000. Kluwer Academic Publishers. 17
- [Rus00b] John Rushby. Theorem proving for verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modelling and Verification of Parallel Processes: MOVEP 2000*, number 2067 in Lecture Notes in Computer Science, pages 39–57, Nantes, France, June 2000. springer Verlag. 16
- [RyS96] Jean-François Raskin and Pierre yves Schobbens. State clock logic: a decidable real-time logic. In *HART’97, LNCS 1201*, pages 33–47. Springer-Verlag, 1996. 49
- [SACO02] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Propel: an approach supporting property elucidation. In *24th Intl. Conf. on Software Engineering*, pages 11–21. ACM Press, 2002. 23
- [SHE01] Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *RE*, pages 14–22. IEEE Computer Society, 2001. 49
- [SMS09] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction

- of büchi automata for ltl model checking verified in isabelle/hol. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 424–439, Berlin, Heidelberg, 2009. Springer-Verlag. 30
- [SMSV83] Richard L. Schwartz, P. M. Melliar-Smith, and Friedrich H. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pages 173–186, New York, NY, USA, 1983. ACM. 20
- [Ste98] Bernhard Steffen, editor. *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*. Springer, 1998. 146
- [TSLT97] Joel Toussaint, Françoise Simonot-Lion, and Jean-Pierre Thomesse. Time constraints verification methods based on time petri nets. In *FTDCS* (DBL97), pages 262–269. 27, 29
- [VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986. 26
- [Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997. 24