Automating the Verification of Realtime Observers using Probes and the Modal mu-calculus*

Silvano Dal Zilio[1,2] and Bernard Berthomieu[1,2]

[1] CNRS, LAAS, 7 avenue colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. A classical method for model-checking timed properties—such as those expressed using timed extensions of temporal logic—is to rely on the use of observers. In this context, a major problem is to prove the correctness of observers. Essentially, this boils down to proving that: (1) every trace that contradicts a property can be detected by the observer; but also that (2) the observer is innocuous, meaning that it cannot interfere with the system under observation. In this paper, we describe a method for automatically testing the correctness of realtime observers. This method is obtained by automating an approach often referred to as *visual verification*, in which the correctness of a system is performed by inspecting a graphical representation of its state space. Our approach has been implemented on the tool Tina, a model-checking toolbox for Time Petri Net.

1 Introduction

A classical method for model-checking timed behavioral properties—such as those expressed using timed extensions of temporal logic—is to rely on the use of observers. In this approach, we check that a given property, P, is valid for a system S by checking the behavior of the system composed with an observer for the property. That is, for every property P of interest, we need a pair (Obs_P, ϕ_P) of a system (the observer) and a formula. Then property P is valid if and only if the composition of S with Obs_P, denoted $(\mathsf{S} \parallel \mathsf{Obs}_P)$, satisfies ϕ_P. This approach is useful when the properties are complex, for instance when they include realtime constraints or involve arithmetic expressions on variables. Another advantage is that we can often reduce the initial verification problem to a much simpler model-checking problem, for example when ϕ_P is a simple reachability property.

In this context, a major problem is to prove the correctness of observers. Essentially, this boils down to proving that every trace that contradicts a property can be detected. But this also involves proving that an observer will never block the execution of a valid trace; we say that it is *innocuous* or non-intrusive. In other words, we need to assure that the "measurements" performed by the observer can be made without affecting the system.

* This work was partly supported by the ITEA2 Project OpenETCS

In the present work, we propose to use a model-checking tool chain in order to check the correctness of observers. We consider observers related to linear time properties obtained by extending the pattern specification language of Dwyer et al. [7] with hard, realtime constraints. In this paper, we take the example of the pattern "**Present** a **after** b **within** $[d_1, d_2[$", meaning that event a must occur after d_1 units of time (u.t.) of the first occurrence of b, if any, but not later than d_2. Our approach can be used to prove both the soundness and correctness of an observer when we fix the values of the timing constraints (the values of d_1 and d_2 in this particular case).

Our method is not enough, by itself, to prove the correctness of a verification tool. Indeed, to be totally trustworthy, this will require the use of more heavy-duty software verification methods, such as interactive theorem proving. Nonetheless our method is complementary to these approaches. In particular it can be used to debug new or optimized definitions of an observer for a given property before engaging in a more complex formal proof of its correctness.

Our method is obtained by automating an approach often referred to as *visual verification*, in which the correctness of a system is performed by inspecting a graphical representation of its state space. Instead of visual inspection, we check a set of branching time (modal μ-calculus) properties on the discrete time state space of a system. These formulas are derived automatically from a definition of the pattern expressed as a first-order formula over timed traces. The gist of this method is that, in a discrete time setting, first-order formulas over timed traces can be expressed, interchangeably, as regular expressions, LTL formulas or modal μ-calculus formulas.

This approach has been implemented on the tool Tina [4], a model-checking toolbox for Time Petri Net [11] (TPN). This implementation takes advantage of several components of Tina: state space exploration algorithms with a discrete time semantics (using the option `-F1` of Tina); model-checkers for LTL and for modal μ-calculus, called *selt* and *muse* respectively; a new notion of *verification probes* recently added to Fiacre [3,5], one of the input specification language of Tina. While model checkers are used to replace visual verification, probes are used to ensure innocuousness of the observers.

Outline and contributions. The rest of the paper is organized as follows. In Sect. 2, we give a brief definition of Fiacre and the use of probes and observers in this language. In Sect. 3, we introduce the technical notations necessary to define the semantics of patterns and timed traces and focus on an example of timed patterns. Before concluding, we describe the graphical verification method and show how to use a model-checker to automate the verification process[1].

The theory and technologies underlying our verification method are not new: model-checking algorithms, semantics of realtime patterns, connection between path properties and modal logics, ... Nonetheless, we propose a novel way to combine these techniques in order to check the implementation of observers and

[1] Code is available at `http://www.laas.fr/fiacre/examples/visualverif.html`

in order to replace traditional “visual” verification methods that are prone to human errors.

Our paper also makes some contributions at the technical level. In particular, this is the first paper that documents the notion of probe, that was only recently added to Fiacre. We believe that our (language-level) notion of probes is interesting in its own right and could be adopted in other specification languages.

2 The Fiacre Language

We consider systems modeled using the specification language Fiacre [3,5]. (Both the system and the observers are expressed in the same language.) Fiacre is a high-level, formal specification language designed to represent both the behavioral and timing aspects of reactive systems.

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. Components can be hierarchically composed. We give in Fig. 1 a simple example of Fiacre specification for a computer mouse button capable of emitting a double-click event. The behavior, 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.).

```
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
  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
```

Fig. 1. A double-click example in Fiacre

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 declare 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. 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 synchronization events. Regarding complex transitions, the expression related to state s1 of Push, for instance, declares two possible transitions from s1: (1) on a click event, set dbl to true and stay in state s1; and (2) on a delay event, change to state s2.

Components. A component is built from the parallel composition of processes and/or other components, expressed with the operator **par** $\mathsf{P}_0 \mid\mid \ldots \mid\mid \mathsf{P}_n$ **end**. In a composition, processes can interact both through synchronization (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 priorities 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. 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. (Time passes at the same rate for all the processes.) Additionally, the **priority** statement asserts that a transition on event click cannot occur if a transition on delay is also possible.

Probes and Observers. The Fiacre language has been extended, recently, to allow the definition of observers, which are a distinguished category of subprograms 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.

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 (see for example `http://www.laas.fr/fiacre/properties.html`). In our setting, observable events are instantaneous actions involved in the evolution of the system: it can be a synchronization over a 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 Mouse component of Fig. 1, a probe triggered when the (only instance of) process Push is in state s2 would have the form (Mouse/1/state s2).

The use of probes greatly simplifies the proof of innocuousness of an observer. In particular, with probes, an observer can only influence a system by "blocking

the evolution of time”, that is by performing an infinite sequence of actions in finite time. Therefore, proving that an observer is innocuous amounts to proving that it has no Zeno behaviors, which is always possible when a system is bounded.

```
process NeverTwice [a:sync] is
 states idle, once, error
 from idle a ; to once
 from once a ; to error
```

```
component Obs is
 port p:sync is Mouse/event click
 par NeverTwice [p] end
```

Fig. 2. A simple observer example

An observer is a Fiacre component 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. 2, where the component Obs monitors synchronizations on the event click. 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. 1 with the code from Fig. 2). 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).

3 Timed Traces and First-Order Formulas over Traces

The semantics of Fiacre (and the properties we want to check) are based on a notion of *timed traces*, which are sequences mixing events and time delays. In this context, a “realtime property” can be defined as a set of timed traces, which define timing and behavioral constraints on the acceptable execution of a system. In this work, we consider properties derived from realtime patterns, that can be expressed using first-order formulas over timed traces.

Timed Traces. In our context, observable events are: communication on a port; the change of state of a process; and the change of value of a variable. We use a dense time model, meaning that we consider rational time delays and work both with strict and non-strict time bounds. Hence a timed trace is a (possibly infinite) sequence of events $a, b, \dots$ and durations $\delta \in \mathbb{Q}^+$:

$$\sigma \ ::= \ \epsilon \ | \ \sigma\, a \ | \ \sigma\, \delta$$

Given a finite trace σ and a—possibly infinite—trace σ', we denote $\sigma\sigma'$ the *concatenation* of σ and σ'. We will also use the expression $\Delta(\sigma)$ to denote the duration (time length) of a trace σ, that is the sum of the individual delays in σ. The semantics of a system expressed with Fiacre, say S, can be defined as a

set $[\![\mathsf{S}]\!]$ of timed traces. We use the notation $\sigma \models \mathsf{S}$ when the trace σ is in the set $[\![\mathsf{S}]\!]$. The semantics of a property (timed pattern) will be expressed as the set of all timed traces where the pattern holds. We say that a system S satisfies a timed requirement P if $[\![\mathsf{S}]\!] \subseteq [\![\mathsf{P}]\!]$.

Realtime Properties and their Semantics. We propose to define properties using First-Order Formulas over Timed Traces (FOTT). A FOTT formula $\Phi(\boldsymbol{x})$, with free variables $\boldsymbol{x} = (x_1, \ldots, x_n)$, is a first-order logic formula over traces with equality between traces ($\sigma = \sigma'$), comparison between a duration and an interval ($\Delta(\sigma) \in I$) and concatenation ($\sigma = \sigma_1\, \sigma_2$).

$$\Phi(\boldsymbol{x}) \ ::= \ \Phi \wedge \Phi' \ \mid \ \neg\Phi \ \mid \ \exists x \,.\, \Phi \ \mid \ (x = \sigma) \ \mid \ (x = y\, z) \ \mid \ (\Delta(x) \in I)$$

For instance, when referring to a timed trace σ and an event a, the following formula is a tautology if the event a does not occur in σ:

$$(a \notin \sigma) \ \stackrel{\text{def}}{=} \ \neg\,(\exists x_1, x_2, x_3 \,.\, (\sigma = x_1\, x_2) \wedge (x_2 = a\, x_3))$$

Likewise, we can define the "scope" σ **after** b—that determines the part of a trace σ located after the first occurrence of b—as the trace σ' denoted by the first-order formula: $\exists x, y \,.\, (\sigma = x\, y) \wedge (y = b\, \sigma') \wedge (b \notin x)$.

The semantics of a formula $\Phi(x_1, \ldots, x_n)$ is a set of valuation functions ς associating a trace $\sigma_i = \varsigma(x_i)$ to each of the variables x_i with $i \in 1..n$, also denoted $[x_i \mapsto \sigma_i]_{i \in 1..n}$. The semantics of Φ can be defined inductively as follows:

$$\begin{array}{llll}
[\![\Phi(\boldsymbol{x}) \wedge \Psi(\boldsymbol{x})]\!] = [\![\Phi(\boldsymbol{x})]\!] \cap [\![\Psi(\boldsymbol{x})]\!] & & [\![x = \sigma]\!] & = \{\varsigma \mid \varsigma(x) = \sigma\} \\
[\![\exists y \,.\, \Phi(\boldsymbol{x})]\!] & = \{\varsigma \mid \varsigma + [y \mapsto \sigma] \in [\![\Phi(\boldsymbol{x})]\!]\} & [\![x = y\, z]\!] & = \{\varsigma \mid \varsigma(x) = \varsigma(y)\, \varsigma(z)\} \\
[\![\Delta(x) \in I]\!] & = \{\varsigma \mid \Delta(\varsigma(x)) \in I\} & &
\end{array}$$

With these definitions, a *regular set of timed traces* is the set of traces "solutions" of an existential FOTT formula with a single free variable, $\Phi(x)$; that is the set of traces σ such that the valuation $[x \mapsto \sigma]$ is in $[\![\Phi(x)]\!]$.

In this paper, we will mainly restrict ourselves to the special case of timed traces where events occur at integer dates; i.e. we restrict delays δ to be in $\mathbb{N}$ rather than in $\mathbb{Q}^+$. These traces can be generated using a "discrete time" abstraction of the models, where special transitions (labeled with $\mathtt{t}$) are used to model the flow of time. Label $\mathtt{t}$ stands for the "tick" of the logical clock.

The discrete time semantics will be enough to prove all the properties needed in our study. Indeed, when a model contains only "closed timing constraints" (of the kind $[d_1, d_2]$ or $[d_1, \infty[$), the discrete time semantics is enough to check reachability properties.

With discrete time, a delay δ can be replaced by sequences of δ $\mathtt{t}$'s, and therefore a finite timed trace can be simply interpreted as a word. In the remainder, we also consider a special symbol, z, that stands for internal actions of the system. Hence it is possible to interpret the semantics of (discrete) FOTT specification as a language over the alphabet $A = \{\mathsf{z}, \mathtt{t}, \mathsf{a}, \mathsf{b}, \ldots\}$. Actually, in the discrete case, we can show that a regular set of timed traces is also a regular language. For

example, the semantics of the formula $\exists y, z, w \ . \ ((x = y\,z) \wedge (z = a\,w))$ is the regular language corresponding to the expression $A^* \cdot \mathsf{a} \cdot A^*$.

This connection between different type of logics is at the core of our approach. Our method could be applied to more high-level property languages, such as timed extension of temporal logic [10], but would require a more complex encoding into LTL when modalities can be nested.

Our Running Example: the Present Pattern. Users of Fiacre have access to a catalog of specification patterns based on a hierarchical classification borrowed from Dwyer [7]. Patterns are built from five basic categories—existence, absence, universality, response and precedence—and can be composed using logical connectives. In each category, generic patterns may be specialized using *scope modifiers*—such as before, after, between—that limit the range of the execution trace over which the pattern must hold. Finally, timed patterns are obtained using one of two possible kinds of *timing modifiers* that limit the possible dates of events referred in the pattern: **within** I—used to constrain the delay between two given events to be in the time interval I—and **lasting** d—used to constrain the length of time during which a given condition holds (without interruption) to be greater than d.

Due to limited space, we study only one example of timed pattern, namely **Present** a **after** b **within** $[d_1, d_2[$. A complete catalog is available in [1]. This is a simple example of existence patterns. Existence patterns are used to express that, in every trace of the system, some events must occur. This pattern holds for traces such that the event a occurs at a date t_0 after the first occurrence of b with $t_0 \in [d_1, d_2[$. The property is also satisfied if b never holds. Hence traces σ that satisfy this pattern are models of the existential FOTT formula:

$$\mathrm{Pres}(x) \ \stackrel{\text{def}}{=} \ (b \notin x) \ \vee \ \exists y, z, w \ . \ ((x = y\,b\,z\,a\,w) \wedge (b \notin y) \wedge (\Delta(z) \in [d_1, d_2[))$$

```
process Present [a:sync, b:sync] is
   states idle, start, watch, error, stop
   from idle  b;  to start
   from start wait [d1, d1]; to watch
   from watch select
                 a; to stop
               unless
                 wait [d2 − d1, ···[; to error
               end
```

Listing 1.1. Observer for the pattern: **Present** a **after** b **within** $[d_1, d_2[$

With the discrete semantics, formula $\mathrm{Pres}(x)$ matches exactly the words of the form $w_1\,\mathsf{b}\,w_2\,\mathsf{a}\,w_3$ where w_1 contains no occurrences of b and w_2 contains exactly k occurrences of t with $k \in [d_1, d_2[$. (This is a regular language.) We show

in the next section how to (semi-)automatically generate the regular expression corresponding to such FOTT formulas.

We give an example of observer associated to this pattern in Listing 1.1. This observer is composed of one process that monitors the system through the ports a and b (that should be instantiated with the relevant probes). The process is initially in state idle and moves to start when b is triggered. When in state start for d_1 unit of time, 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 stop if an a occurs, unless a duration equals to $(d_2 - d_1)$ elapses, in which case it moves to the state error. As a consequence, the pattern is false whenever the probe (Present/state error) is reachable. Hence the formula associated to the pattern is $\phi_P \stackrel{\text{def}}{=}$ [] - (Present/state error).

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 formula ϕ_P if and only if $[\![\mathsf{S}]\!] \subseteq [\![\mathsf{P}]\!]$. In [1], we have defined a mathematical framework to formally prove these kind of properties, but this framework relies on manual proofs and is not supported by any tooling. Efforts are also under way to completely mechanize these proofs using the Coq proof assistant [8]. 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 rely on a "visual" verification method, that is akin to debugging our observers.

In the next section, we show how to apply the visual verification approach on our running example. One of the objectives of our work is to replace this visual verification step with a more formal approach. This is done in Sect. 5.

4 Visual Verification of Observers

In the remainder of this section, we describe the visual verification method using the particular case of the pattern **Present** a **after** b **within** $[4, 5[$; we assume that Obs is the observer Present defined in Listing 1.1, that $d_1 = 4$ and that $d_2 = 5$.

To prove that the observer Present is correct, we need to prove, for every system S, the equivalence between two facts: (1) the state (Present/state error) is not reachable in the program (S || Present[a, b]); and (2) the traces of S are valid for the property Pres, i.e. $[\![\mathsf{S}]\!] \subseteq [\![\mathrm{Pres}]\!]$.

The first step 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 a, b and z. We give an example of universal process in Listing 1.2. 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 a will be triggered to

the event "setting x to 1" and b to "setting x to 2". The third transition resets the value of x to 0 immediately and corresponds to the internal event z.

```
process Universal (&x : nat) is
   states s0
   from s0 select
           x := 1; to s0       /* setting x to 1 */
        [] x := 2; to s0       /* setting x to 2 */
        unless
           on (x<> 0); wait [0,0]; x := 0; to s0
        end

component Main is
   var x : nat := 0
   port a : sync is value (x = 1),  b : sync is value (x = 2)
   par Universal (&x) || Present [a, b] end
```

Listing 1.2. Universal program in Fiacre

We can now use our verification toolchain to generate the state graph for the program (Universal || Present) using a discrete time exploration construction. This can be obtained using the flag -F1 in Tina (it is possible to generate a state graph with many different abstractions with Tina, including dense time models).

The resulting graph is displayed in Fig. 3. This state graph has been generated and printed using the tool *nd*, which is also part of the Tina toolset; 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: we find the visible and internal transitions as before, labeled with a, b, z and t. For ease of reading, we have also changed the labels of internal transitions in the observer Present. For instance, 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. The states where the observer is in state error (the states that contradict the property $\phi_P \stackrel{\text{def}}{=}$ [] - (Present/state error)) are $Errors = \{20, 22, 23\}$.

We can already debug the pattern **Present** a **after** b **within** $[4, 5[$ by visually inspecting the state graph.

For *soundness*, we need to check that, when the pattern is not satisfied—for traces σ that do not satisfy formula Pres—then the observer will detect a problem (observer Present eventually reaches a state in the set *Errors*).

For *innocuousness* we need to check that, from any state, it is always possible to reach a state where event a (respectively b and t) can fire. Indeed, this means that the observer cannot selectively remove the observation of a particular sequence of external transitions or the passing of time.

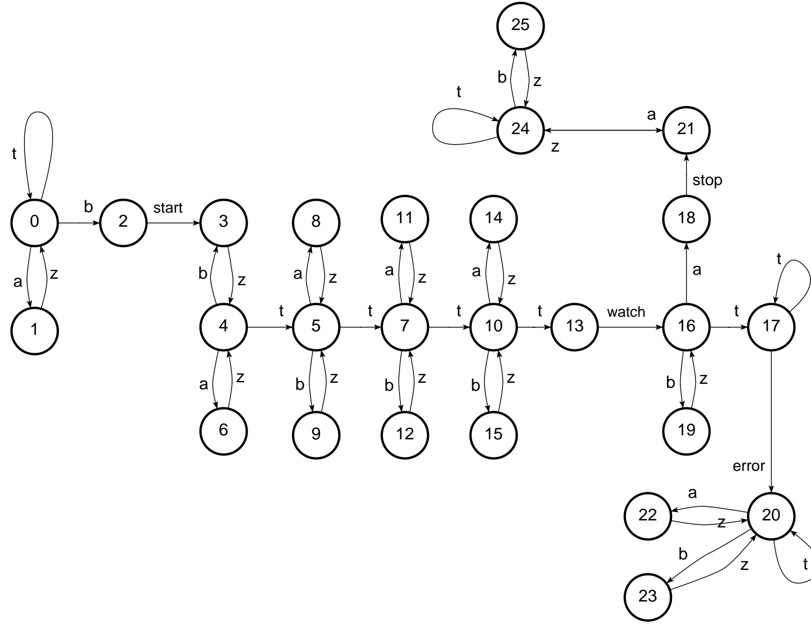

Fig. 3. State graph for (Universal || Present)

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

5 Automating the Visual Verification Method

A problem with the previous approach is that it essentially relies on an informal inspection (and on human interaction). We show how to solve this problem by replacing the visual inspection of the state graph by the verification of modal μ-calculus formulas. (the Tina toolset includes a model-checker for the μ-calculus called *muse*.) The general idea rests on the fact that we can interpret the state graph as a finite state automaton and (some) sets of traces as regular languages. This analogy is generally quite useful when dealing with model-checking problems. We start by defining some useful notations.

Label Expressions, are boolean expressions denoting a set of (transition) labels. For instance, $A_{\text{ext}} = (\mathsf{a} \vee \mathsf{b})$ denotes the external transitions, while the expression $\texttt{-}(\mathsf{a} \vee \mathsf{b} \vee \mathsf{t})$ is only matched by the silent transition label. We will also use the expression $\top$ to denote the disjunction of all possible labels, e.g.

$\top = (\text{-b}) \vee \text{b}$. The model checker *muse* allows the definition of label expressions using the same syntax.

Regular (Path) Expressions. In the following, we consider regular expressions built from label expressions. For example, the regular expression $\text{t} \cdot (\text{- t})*$ denotes traces of duration 1 with no events occurring at time 0.

$$Tick \overset{\text{def}}{=} \text{t} \cdot (\text{-t})* \tag{1}$$

We remark that it is possible to define the set of (discrete) traces where the FOTT formula Pres holds using the union of two regular languages: (1) the traces where b never occurs, $(\text{- b})*$; and (2) the traces where there is an a four units of time after the first b. The latter corresponds to the regular expression $(x = y \,\text{b}\, z \,\text{a}\, w) \wedge (\text{b} \notin y) \wedge (\Delta(z) \in [4, 5[\,)$

$$Pres \overset{\text{def}}{=} R_1 \;\vee\; R_2 \tag{2}$$

$$R_1 \overset{\text{def}}{=} (\text{- b})* \tag{3}$$

$$R_2 \overset{\text{def}}{=} (\text{-b})* \cdot \text{b} \cdot (\text{- t})* \cdot Tick \cdot Tick \cdot Tick \cdot Tick \cdot \text{a} \cdot \top* \tag{4}$$

By construction, the regular language associated to $R_1 \vee R_2$ is exactly the set of finite traces matching (the discrete semantics) of Pres. In the most general case, a regular expression can always be automatically generated from an existential FOTT formula when the time constraints of delay expressions are fixed (the intervals I in the occurrences of $(\Delta(x) \in I)$).

The next step is to check that the observer agrees with every trace conforming to R_2. For this we simply need to check that, starting from the initial state of (Universal || Present), it is not possible to reach a state in the set *Errors* by following a sequence of transitions labeled by a word in R_2.

This is a simple instance of a language inclusion problem between finite state automata. More precisely, if *Present* is the set of states visited when accepting the traces in $R_1 \vee R_2$, we need to check that *Errors* is included in the complement of the set *Present* (denoted $\overline{Present}$). In our example of Fig. 3, we have that $\overline{Present} = \{17, 20, 22, 23\}$, and therefore $Errors \subseteq \overline{Present}$.

This automata-based approach has still some drawbacks. This is what will motivate our use of a branching time logic in the next section. In particular, this method is not enough to check the soundness or the innocuousness of the observer. For innocuousness, we need to check that every event may always eventually happen. Concerning soundness, we need to prove that $Errors \supseteq \overline{Present}$; which is false in our case. The problem lies in the treatment of time divergence (and of fairness), as can be seen from one of the counter-example produced when we use our LTL model-checker to check the soundness property, namely: `b.start.z.t.t.t.t.watch.t.t.`$\cdots$ (ending with a cycle of `t` transitions). This is an example where the error transition is continuously enabled but never fired.

Branching Time Specification. We show how to interpret regular expressions over traces using a modal logic. In this case, the target logic is a modal μ-calculus with operators for forward and backward traversal of a state graph . (Many temporal logics can be encoded in the μ-calculus, including CTL$*$). In this context, the semantics of a formula ψ over a Kripke structure (a state graph) is the set of states where ψ holds.

$$\psi \;::=\; \phi \wedge \psi \;\;|\;\; \neg\psi \;\;|\;\; \texttt{<A>}\psi \;\;|\;\; \psi\texttt{<A>} \;\;|\;\; X \;\;|\;\; (\min X \,|\, \psi)$$

The basic modalities in the logic are $\texttt{<}A\texttt{>}\psi$ and $\psi\texttt{<}A\texttt{>}$, where A is a label expression. A state s is in $\texttt{<}A\texttt{>}\psi$ if and only if there is a (successor) state s' in ψ and a transition from s to s' with a label in A. Symmetrically, s is in $\psi\texttt{<}A\texttt{>}$ if and only if there is a (predecessor) state s' in ψ and a transition from s' to s with a label in A. In the following, we will also use two constants, $\texttt{T}$, the true formula (matching all the states), and $\texttt{`0}$, that denotes the initial state of the model; and the least fixpoint operator $\texttt{min X | }\psi\texttt{(X)}$.

For example, the formula $\texttt{<a>T}$ matches all the states that are the source of an $\texttt{a}$-transition, likewise $\texttt{Reach_a} \stackrel{\text{def}}{=} \texttt{min X | (<a>T} \;\vee \texttt{<Z>X)}$ matches all the states that can lead to an $\texttt{a}$-transition using only internal transitions. As a consequence, we can test innocuousness by checking that the formula $\texttt{(Reach_a} \;\wedge \texttt{Reach_b} \wedge \texttt{Reach_t)}$ is true for all states.

The soundness proof relies on an encoding from regular path expressions into modal formulas. We define two encodings: $(\!(R)\!)$ that matches the states encountered while firing a trace matching a regular expression R; and $(\!(R)\!)_e$ that matches the state reached (at the end) of a finite trace in R. These encodings rely on two derived operators. (Again, we assume here that A is a label expression.)

$$\psi \;\texttt{o}\; A \;\stackrel{\text{def}}{=}\; \psi\texttt{<A>} \qquad\qquad \psi \ast A \;\stackrel{\text{def}}{=}\; \texttt{min X | } \psi \vee \texttt{X<}A\texttt{>}$$

$$\begin{aligned}
(\!(R \cdot A)\!)_e &\stackrel{\text{def}}{=} (\!(R)\!)_e \;\texttt{o}\; A \\
(\!(R \cdot A*)\!)_e &\stackrel{\text{def}}{=} (\!(R)\!)_e \ast A \\
(\!(R \cdot \mathit{Tick})\!)_e &\stackrel{\text{def}}{=} \texttt{(}(\!(R)\!)_e \;\texttt{o t)} \ast \texttt{(-t)} \\
(\!(R_1 \vee R_2)\!)_e &\stackrel{\text{def}}{=} (\!(R_1)\!)_e \vee (\!(R_2)\!)_e \\
(\!(\epsilon)\!)_e &\stackrel{\text{def}}{=} \texttt{`0}
\end{aligned}
\qquad\Bigg|\qquad
\begin{aligned}
(\!(R \cdot A)\!) &\stackrel{\text{def}}{=} (\!(R)\!) \vee (\!(R \cdot A)\!)_e \\
(\!(R \cdot A*)\!) &\stackrel{\text{def}}{=} (\!(R)\!) \vee (\!(R \cdot A*)\!)_e \\
(\!(R \cdot \mathit{Tick})\!) &\stackrel{\text{def}}{=} (\!(R)\!) \vee (\!(R \cdot \mathit{Tick})\!)_e \\
(\!(R_1 \vee R_2)\!) &\stackrel{\text{def}}{=} (\!(R_1)\!) \vee (\!(R_2)\!) \\
(\!(\epsilon)\!) &\stackrel{\text{def}}{=} \texttt{`0}
\end{aligned}$$

Lemma 1. *Given a Kripke structure K, the states matching the formula $(\!(R)\!)_e$ (respectively $(\!(R)\!)$) in K are the states reachable from the initial state after firing (resp. all the states reachable while firing) a sequence of transitions matching R.*

Proof (Sketch). By induction on the definition of R. For example, if we assume that ψ correspond to the regular expression R, then $\psi \ast A$ matches all the states reachable from states where ψ is true using (finite) sequences of transition with label in A; i.e. formula $\psi \ast A$ corresponds to $R \cdot A*$. Likewise, we use the interpretation of the empty expression, ϵ, to prefix every formula with the constant $\texttt{`0}$ (that will only match the initial state). This is necessary since μ-calculus formulas are evaluated on all states whereas regular path expressions are evaluated from the initial state. □

For example, we give the formula for $(\!(R_2)\!)_e$ below, where ψ `o Tick` stands for the expression $(\psi$ `o t) * (-t)`:

$$(\!(R_2)\!)_e \stackrel{\text{def}}{=} \texttt{`0 * (-b) o b * (-t) o Tick o Tick o Tick o Tick o a * T}$$

If ψ_{Err} is a modal μ-calculus formula that matches the error condition of the observer, then we can check the correctness and soundness of the observer Present by proving that the equivalence (EQ), below, is a tautology (that it is true on every states of (Universal || Present)).

$$(\!(\text{Pres})\!) \;\Leftrightarrow\; -\psi_{Err} \tag{EQ}$$

Again, we can interpret the “error condition” using the μ-calculus. The definition of errors is a little bit more involved than in the previous case. We say that a state is in error if the transition error is enabled (the formula `<error>T` is true) or if the state can only be reached by firing the error transition (which corresponds to the formula `(T<error> * T)` $\wedge$ `(‘0 * (- error))`. Hence ψ_{Err} is the disjunction of these two properties:

$$\psi_{Err} \stackrel{\text{def}}{=} \texttt{<error>T} \;\vee\; \texttt{((T<error> * T)} \;\wedge\; \texttt{- (`0 * (-error)))}$$

The formula (EQ) can be checked almost immediately (less than 1 s on a standard computer) for models of a few thousands states using *muse*. Listing 1.3 gives a *muse* script file that can be used to test this equivalence relation.

6 Related Work and Conclusion

Few works consider the verification of model-checking tools. Indeed, most of the existing approaches concentrate on the verification of the model-checking algorithms, rather than on the verification of the tools themselves. For example, Smaus et al. [16] 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 checkers 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 [15], 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 the verification part. Finally, another possibility is to rely on a kind of “Proof-Carrying Code” approach, where the model checker can produce a deductive proof on either success or failure [12]. This proof can then be checked separately, using a tool independent from the model checker.

Concerning observer-based model-checking, most of the works rely on an automatic way to synthesize observers from a formal definition of the properties.

```
# Results are displayed as set of states. Use "output card" to see the cardinality
output set;

# definition of derived operators

infix X ∗ L = min Y | X ∨ Y⟨L⟩;         infix X o L = X⟨L⟩;
op TICK X = min Y | X⟨t⟩ ∨ Y⟨−t⟩;   op NEVER L = (‘0) ∗ (−L);
op EXT = a ∨ b ∨ t; # labels of the external transitions
op REACH L = min X | (⟨L⟩T) ∨ ⟨−EXT⟩X;

# INNOCUOUSNESS

op Innocuous = (REACH a) ∧ (REACH b) ∧ (REACH t);

# SOUNDNESS

op A0 = (NEVER b) o b;        op S0 = (NEVER b) ∨ A0;
op A1 = A ∗ (−t);             op S1 = S0 ∨ A1;
op A2 = TICK(A1);             op S2 = S1 ∨ A2;
op A3 = TICK(A2);             op S3 = S2 ∨ A3;
op A4 = TICK(A3);             op S4 = S3 ∨ A4;
op A5 = TICK(A4);             op S5 = S4 ∨ A5;
op A6 = A5 o a;               op S6 = S5 ∨ A6;
op A7 = A6 ∗ T;               op S7 = S6 ∨ A7;

op R1 = NEVER b;              op R2 = S7
op Pres = R1 ∨ R2;
op ERRORS = ⟨error⟩T ∨ (((T⟨error⟩) ∗ T) ∧ − ((‘0) ∗ (−error)));

Pres ⇔ (− ERRORS); # this is a tautology if all the states are listed
```

Listing 1.3. Script file for *muse* to check that ((Pres)) $\Leftrightarrow -\psi_{Err}$ is a tautology

For instance, Aceto et al. [2] propose a method to verify properties based on the use of test automata. In this framework, verification is limited to safety and bounded liveness properties since the authors focus on properties that can be reduced to reachability checking. In the context of Time Petri Net, Toussaint et al. [17] also propose a verification technique based on "timed observers", but they only consider four specific kinds of time constraints. None of these works consider the complexity or the correctness of the verification problem. Another related work is [9], where the authors define observers based on Timed Automata for each pattern. Our approach is quite orthogonal to the "synthesis approach". Indeed we seek, for each property, to come up with the best possible observer in practice. To this end, using our toolchain, we compare the complexity of different implementations on a fixed set of representative examples and for a specific set of properties and kept the best candidates. The need to check multiple implementations for the same patterns has motivated the need to develop a lightweight verification method for checking their correctness.

Compared to these works, we make several contributions. We define a complete verification framework for checking observers with hard realtime constraints. This framework has been tested on a set of observers derived from high-level timed specification patterns. This work is also our first public application of the probe technology, that was added to Fiacre only recently. 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 [6,14]—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. We believe that this (language-level) notion of probes is interesting in its own right and could be adopted by other formal specification languages. Finally, we propose a formal approach that can be used to gain confidence on the implementation of our model-checking tools and that replaces traditional "visual verification methods" that are prone to human errors.

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. A formal framework to specify and verify real-time properties on critical systems. International Journal of Critical Computer-Based Systems (IJCCBS) 5(1/2):4-30, 2014.
2. L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In *Proc. of TACAS*, vol. 1384 of *LNCS*. Springer, 1998.
3. B. Berthomieu, J.-P. Bodeveix, and M. Fillali and G. Hubert and F. Lang and F. Peres and R. Saad and S. Jan and F. Vernadat. The Syntax and Semantics of Fiacre – Version 3.0. `http://www.laas.fr/fiacre/`, 2012.
4. 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*, 42:14, 2004.
5. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Proc. of ERTS*, 2008.
6. M. Chechik and D.O. Paun. Events in Property Patterns. In *Theoretical and Practical Aspects of SPIN Model Checking*, vol. 3, 1999.
7. M. B. Dwyer, L. Dillon. Online Repository of Specification Patterns. At `http://patterns.projects.cis.ksu.edu/`
8. M. Garnacho, J.-P. Bodeveix and M. Filali. A Mechanized Semantic Framework for Real-Time Systems. In *Proc. of FORMATS*, LNCS vol. 8053, 2013.
9. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
10. R. Koymans. Specifying realtime properties with metric temporal logic. *Realtime Syst.*, 2:255–299, 1990.
11. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974.
12. K. S. Namjoshi. Certifying Model Checkers. In *Proc. of CAV*, LNCS vol. 2102, 2001.
13. J. Ouaknine and J. Worrell. On the decidability and complexity of metric temporal logic over finite words. In *Logical Methods in Computer Science*, vol. 3, 2007.
14. D.O. Paun and M. Chechik. Events in Events in Linear-Time Properties. In *CoRR journal*, vol. cs.SE/9906031, 1999.
15. Esterel Technologies. SCADE Tool Suite. `http://www.esterel-technologies.com/products/scade-suite`
16. A. Schimpf, S. Merz and J.-G. Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In *Proc. of TPHOLs*, LNCS vol. 5674, 2009.
17. J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraints verification methods based on time Petri nets. In *Proc. of FTDCS*. IEEE, 1997.