Who Checks the Model-Checkers ?

Nouha Abid[1,2], 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. We describe a method for automatically testing a model-checker for timed behavioral properties. We consider the case of an observer-based model-checker, meaning that the relationship between a model and its specification is interpreted as the composition of the model with an observer of its behavior. In this context, a major problem is to prove the correctness of observers.
In this work, we deal with systems expressed using Fiacre, a formal modelling language for realtime, reactive systems. For requirements, we consider specifications expressed using a set of realtime verification patterns, which are translated into observers. We describe a graphical verification method that has been used to gain confidence on our interpretation of patterns into observers. Our method provides a formal, automatic way to check that an observer for a specification pattern is correct, that is, a proof that an observer faithfully captures the semantics of its associated pattern. This general approach is complementary to the use of more heavy-duty verification methods, such as interactive theorem prover, and can be used to debug the implementation of new observers.

1 Introduction

A number of model-checking tools rely on the use of observers, meaning that the relationship between a model and its specification is interpreted as the composition of the model with an observer of its behavior. However, few of these tools provide a method to check the correctness of their observers. We are faced with a common problem, so judiciously summarized in Plato's famous conundrum: "who guards the guardians?" Our (partial) solution to this question is quite simple, since we propose to manipulate our verification toolchain in order to check itself. While this method is not enough to prove the correctness of a verification tool, it is complementary to the use of more heavy-duty software verification methods, such as interactive theorem prover, and can be used to debug new or optimized implementations.

In this paper, we describe our method on an integrated verification toolchain for Fiacre [5,7], a formal specification language used to model reactive systems with hard, realtime constraints. For verification purposes, Fiacre models can be compiled into to a generalization of Time Petri Nets [13] (TPN) with priorities and data variables and then checked using Tina, the TIme Petri Net Analyzer. The Tina toolbox [6] includes several tools for the edition and verification of extended TPN, such as *muse* (a modal μ-calculus model-checker) and *selt* (a model-checker for State/Event LTL). We will make use of these two tools in our work (see Sect. 4).

More recently, the Fiacre toolchain has been extended to support the verification of timed behavioral properties. Instead of adapting model-checking algorithms for timed extensions of temporal logic, such as TCTL, we propose a new requirement specification language based on a set of *timed patterns* that extends the specification language of Dwyer et al. [9] with hard, realtime constraints. For example, we define a pattern "**Present** A **after** B **within** $[d_1, d_2[$" to express that event A must occur within d_1 units of time (u.t.) of the first occurrence of B, if any, but not later than d_2 u.t. In our toolchain, specifications expressed using verification patterns are translated into observers (see [1,2] for a complete description of timed patterns and observers). From this, we transform the problem of checking timed patterns into the simpler problem of checking LTL properties on the composition of the system with an observer.

The idea is not to automatically generate the observers from a formal definition of patterns but, rather, to provide several candidates for each pattern and then to choose the best observer, performance-wise. To prove the correctness of observers—and also the correctness of their implementation—we propose a "graphical verification method" that can been used in order to gain confidence on our verification toolchain. Our method provides a formal, automatic way to test whether an observer for a specification pattern is correct, meaning that: (1) the verification results obtained with the observer are *sound* with respect to the semantics of patterns (if the pattern is not satisfied then the observer will detect a problem); and that (2) the observer is *innocuous*, or non-intrusive (the observer will inspect the whole state space of the system and cannot interfere with it).

To check the correctness of an observer, we have defined in [2] a complete 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 [10], 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 expansive to test new observers. The graphical method described in this paper 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 transciption of our algorithm into actual code.

Outline. The rest of the paper is organized as follows. In Sect. 2, we give a brief definition of Fiacre and introduce the technical notations necessary to define the semantics of patterns and time traces. In Sect. 3, we present a selection of timed patterns and define their corresponding observers using the Fiacre syntax. Before concluding, we describe our graphical verification method and show how to use *muse*—our modal μ-calculus model-checker—to automatize the verification process.

2 Technical Background

We consider systems modeled using Fiacre (see Sect. 2.1) and requirements expressed using specification patterns, which define timing and behavioral constraints on the execution of a system. The requirements, and the semantics of Fiacre are based on a notion of *timed traces* (defined in Sect. 2.2), which are sequences mixing events and time delays. In the remainder of the text, we will use two alternative methods to define sets of timed traces: Metric Temporal Logic—a timed extension of LTL—and first-order formulas over timed traces.

2.1 The Fiacre Language

Fiacre (`http://projects.laas.fr/fiacre/`) is a formal specification language designed to represent both the behavioral 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 [3]. A formal definition of the language is given in [5,7]. 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. 1 a simple example of Fiacre specification for a mouse button with double-click. 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.).

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. 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 for s1, for instance, declares two possible behavior 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.

```
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

Components: a component is built from the parallel composition of processes and/or other components, expressed with the operator **par** P_0 || . . . || 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 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. 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.

Probes and observers: the Fiacre language has been extended, recently, 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.

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 Oobs 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, 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. For instance, the probe **not** (Mouse/event click) is triggered by any event that is not a communication over the port click.

```
process NeverTwice [A:sync] is          component Obs is
 states idle, once, error                 port p:sync is Mouse/event click
 from idle A ; to once                    par NeverTwice [p] end
 from once A ; to error
```

Fig. 2. A simple observer example

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. 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). This can be easily achieved using an LTL model-checker (the *selt* tool in our case) with the property `[]-(Obs/1/state error)` (always, not Obs/1 is in state error, where Obs/1 is the first process instance in the definition of Obs).

2.2 Timed Traces, Metric Temporal Logic and First-Order formulas over Traces

A timed trace is a (possibly infinite) sequence of events and time delays. In our context, observable events are: communication on a port; the state of processes; and the value of variables. We use a dense time model, meaning that we consider rational time delays and work both with strict and non-strict time bounds. In this setting, a timed trace σ is a possibly infinite sequence of events $A, B, \dots$ and duration $d(\delta)$ with $\delta \in \mathbb{Q}^+$. 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 function $\Delta(\sigma)$, that returns the duration (time length) of a trace σ. The semantics of a system expressed with Fiacre, say S, can be defined as a set of timed traces: $\sigma \models$ S if the trace σ is in S. We say that a system S satisfies a timed requirement P if the trace in S are also in P.

The semantics of a timed pattern will be expressed as the set of all timed traces where the pattern holds. A first approach to define set of traces is to use timed extensions of temporal logic, such as Metric Temporal Logic (MTL) [12]. MTL is is an extension of LTL where temporal modalities can be constrained by a time interval. For instance, a system satisfies the MTL formula $A \ \mathbf{U}_{[1,3[} \ B$ if, for all its traces, the event B must occur at a time $t_0 \in [1, 3[$ and A holds everywhere in the interval $[0, t_0[$; we have $\sigma \models A \ \mathbf{U}_{[1,3[} \ B$ if and only if $\sigma = \sigma_1 B \sigma_2$ with $\Delta(\sigma_1) \in [1, 3[$ and, for all event ω in σ_1, $\omega = A$. (In the following, we will also use a weak version of the "until modality", denoted $A \ \mathbf{W} \ B$, that does not require B to eventually occur.) We refer the reader to [14] for a presentation of the logic and a discussion on the decidability of various fragments of MTL.

An advantage of using MTL is that it provides a sound and unambiguous framework for defining the meaning of 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). For instance, when referring to a timed trace σ and an event A, we can define the "scope" σ **after** A—that determines the part of σ located after the first occurrence of A—as the trace σ_2 denoted by the first-order formula $F(\sigma, \sigma_2) \overset{\text{def}}{=} \exists \sigma_1 . \sigma = \sigma_1 A \sigma_2 \wedge A \notin \sigma_1$.

Since the model-checking problem for MTL is undecidable [14], it is not enough to simply translate each pattern into a MTL formula to check whether a pattern is true for a Fiacre model. This situation can be somehow alleviated. For instance, the problem is decidable if we disallow punctual timing constraints, of the form $[d, d]$. Still, while we may rely on timed temporal logics as a way to define the semantics of patterns, it is problematic to have to limit ourselves to a decidable fragment of a particular logic—which may be too restrictive—or to rely on multiple realtime model-checking algorithms—that all have a very high complexity in practice. To solve this problem, we propose to rely on *observers* in order to reduce the verification of timed patterns to the verification of LTL formulas. This is, approximately, what we exemplified with the observer NeverTwice defined in Fig. 2. In the next section, we provide for each pattern, P, a pair (ObsP, ϕ_{P}) of a Fiacre observer and a LTL formula such that, for any Fiacre model S, we have that S satisfies P if and only if (S || ObsP) satisfies ϕ_{P}.

The idea is not to provide a generic way of obtaining the observer from a formal definition of the pattern. Rather, we seek, for each pattern, 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.

3 A Catalog of Realtime Patterns

We have defined a catalog of patterns, using a hierarchical classification borrowed from Dwyer [9]. 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 kind of *timing modifiers* that limit the possible dates of events referred in the pattern: **within** I—used to constraint the delay between two given events to be in the time interval I—and **lasting** D—used to constraint the length of time during which a given condition holds (without interruption) to be greater than D. Due to the limited space, we present only a selection of timed patterns; a complete catalog is available in [1].

Next, we give examples of existence, absence and response patterns. Existence patterns are used to express that, in every trace of the system, some events must occur. On the opposite, absence patterns are used to express that some condition should never occur. Finally, 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.

In the following, we 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 $[d_1, d_2]$, $]d_1, \cdots[$, or in the case $d_1 = d_2$—are essentially the same, except for some slight modifications on the priorities and on the time constraints of some ports. We describe a selected number of patterns. In each case, we define the semantics of the pattern, P, using a MTL formula and a FOTT formula. In the case of FOTT, we use formulas, $F(\sigma)$, with only one free variable, σ, with the intended meaning that $\sigma \models \mathsf{P}$ if and only if $F(\sigma)$ is true.

Present A after B within I

The pattern holds for traces such that A occurs at a date t_0 after the first occurrence of B, with $t_0 \in I$. The pattern is also satisfied if B never holds.

MTL def.:	$(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_I\ A)$
FOTT def.:	$\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 A \sigma_4 \wedge \Delta(\sigma_3) \in I$
LTL formula:	`[]-(Present/state error)`

```
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 **Present** A **after** B **within** $[d_1, d_2[$

The observer (see Listing 1.1) 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 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 ok if an A occurs, unless $d_2 - d_1$ u.t. elapses, in which case it moves to error. As a consequence, the pattern is false whenever process Present can reach state error. Hence the associated LTL formula, ϕ_{P}, is []-(Present/state error).

Present first A before B within I

*If B occurs then the first occurrence of A holds t_0 u.t. before the first occurrence of B with $t_0 \in I$. (The difference with **Present** B **after** A **within** I is that B should not occur before the first A.)*

MTL def.:	$(\Diamond B) \Rightarrow (\,(\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_I\ B))\,)$
FOTT def.:	$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in I$
LTL formula:	`(<> Obs/value (foundB=true)) => ([]-(Obs/1/state error))`

```
process Obs1 [A: sync] (&flag:bool, &foundB:bool) is
   states idle, start, watch, error
   from idle A; to start
   from start wait [d1, d1]; flag:=true; to watch
   from watch on (foundB=false); wait [d2 − d1, ...[; to error

process Obs2 [B: sync] (&flag:bool, &foundB:bool) is
   states idle, stop
   from idle  on (flag=true); B; foundB:=true; to stop

component Obs[A:sync, B:sync]  is
   var flag:bool := false, foundB:bool := false
   par
      Obs1 [A] (&flag, &foundB)
   || Obs2 [B] (&flag, &foundB)
   end
```

Listing 1.2. Observer for **Present** A **before** B **within** $[d_1, d_2[$

The observer (see Listing 1.2) is composed of two processes which communicate via shared variables (flag and foundB). The process Obs2 sets the value of the shared variable foundB to true when it sees the first occurrence of B. In the unique transition of Obs2, the operator **on** acts as a guard, meaning that the transition can fire only if the condition (flag = true) is true. Concurrently, the process Obs1 monitors the occurrences of A 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 foundB is false after $(d_2 - d_1)$ u.t. (assuming that there would be an occurrence of B in the future). Therefore, the associated LTL property states that if B eventually occurs (eventually foundB is true) then Obs1 must not reach its error state (always Obs/1/state error is false).

Absent A lasting D

There is a time interval, whose duration is at least D, where A *never occurs*

MTL def.:	$\Diamond(\Box_{[0,D]}\neg A)$
FOTT def.:	$\exists\sigma_1,\sigma_2,\sigma_3 \,.\, (\sigma = \sigma_1\sigma_2\sigma_3 \wedge A \notin \sigma_2 \wedge \Delta(\sigma_2) \geqslant D)$
LTL formula:	<> Absent/state ok

```
process Absent [A: sync] is
  states watch, ok
  from watch select
             A; to watch
           unless
             wait [$D$, $D$]; to ok
           end
```

Listing 1.3. Observer for **Absent** A **lasting** D

The observer (see Listing 1.3) is composed of one processes which is reinitialized to state watch at each occurrence of A, unless there is no occurrence of A for D u.t. (in which case Absent proceeds to state ok). This observer relies on the fact that timing constraint are reinitialized at each transition of the system (except for **loop** transitions). In this case, the pattern is satisfied if "something good happens", that is if eventually Absent reaches ok.

A leadsto first 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*).*

MTL def.:	$\Box(A \Rightarrow (\neg B)\ \mathbf{U}_I\ B)$
FOTT def.:	$\forall\sigma_1,\sigma_2 \,.\, (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists\sigma_3,\sigma_4 \,.\, \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge B \notin \sigma_3$
LTL formula:	`[]-(Leadsto/state error)`

```
process Leadsto [A: sync, B:sync] is
  states idle, start, watch, error
  from idle A; to start
  from start wait [$d_1$, ...[; to watch
  from watch select
              B; to idle
            unless
              wait [$d_2 - d_1$, ...[; to error
            end
```

Listing 1.4. Observer for A **leadsto first** B **within** $[d_1, d_2[$

The observer (see Listing 1.4) moves to state start when A is triggered. Then, after d_1 u.t., it moves to state watch where it waits for an occurrence of B before

$d_2 - d_1$ u.t. elapses, in which case it is reinitialized. If no B occur, the process moves to state error. Like in the first case, the pattern is false if Leadsto can reach state this state.

4 Testing the Correctness of Patterns

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 ϕ_{P} if and only if for all trace σ in S, $\sigma \models$ P. In [2], we define a theoretical framework to prove exactly these kind of properties. Efforts are also under way to completely mechanize these proofs using the Coq proof assistant [**?**]. 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** A **after** B **within** $[4, 5[$. Therefore, we assume that Obs is the observer Present defined in Listing 1.1 (with $d_1 = 4$ and $d_2 - d_1 = 1$).

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 A and B. We give an example of universal process in Listing 1.5 (where we already compose this process with the observer Present).

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 reset the value of x to 0 immediately.

```
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 A : sync is value (x = 1),  B : sync is value (x = 2)
   par Universal (&x) || Present [A, B] end
```

Listing 1.5. Universal program in Fiacre

Graphical Verification. The next step is to use our verification toolchain to generate the state graph for the program (Universal || Present). The state graph should be generated with a "discrete 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* [6] 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. 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: transitions labeled with A or B are the observable events (we call A, B 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** 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 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 B (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 A before 1 u.t. then we have an error; and (2) if we see an A then we will never see an error. In this context, to "see an A before 1 u.t." correspond to following a path—starting from state 13—where a transition labeled with A is before any transition labeled with i. Likewise, "errors" correspond to any state states 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 A (respectively B and i) can fire. Indeed, it means that the observer cannot censor the observation of a particular sequence of external transitions or the passing of time.

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 some errors in the observer before we try to prove the observer correct more formally.

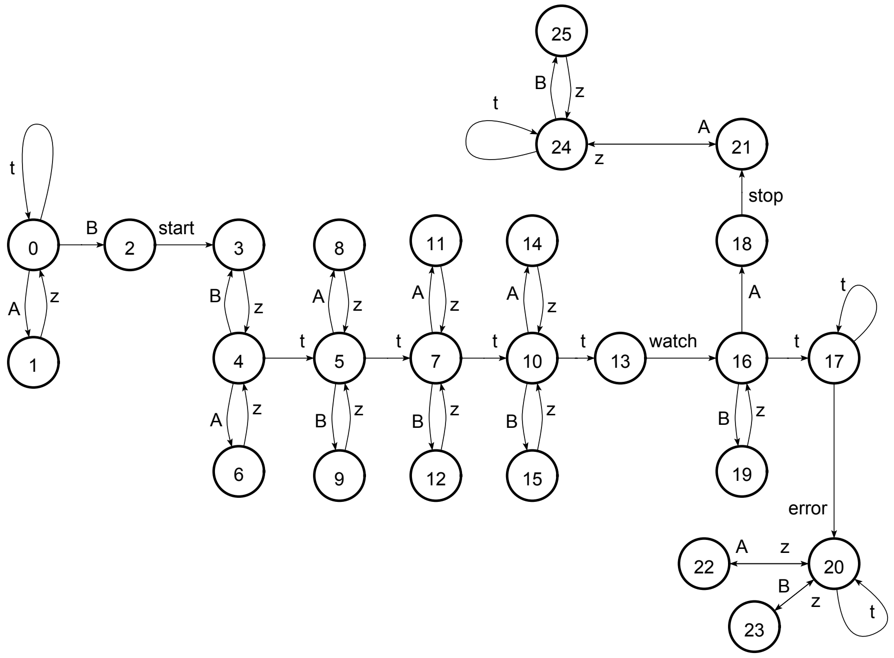

Fig. 3. State graph for (Universal || Present)

Automation of our 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, $\mathsf{L}_{\text{ext}} = (\mathsf{A} \vee \mathsf{B} \vee \mathsf{t})$ denotes the external transitions, while the expression (not B) denotes the set of all labels except B. We use the expression $\top$ to denote the conjunction of all possible labels, e.g. $\top = (\text{not } \mathsf{B}) \vee \mathsf{B}$.

Regular (path) expressions. In the following, we consider regular expressions build from label expressions. For example, the regular expression $\mathit{Tick} = \mathsf{t} \cdot (\text{not } \mathsf{t})*$ denotes a set of traces that contains only one t— which means traces of duration 1—located in the first position. We remark that it is possible to define the set of traces where Present holds using the union of two regular languages: (1) the traces where B never occurs, $R_1 = (\text{not } \mathsf{B})*$; and (2) the traces where there is an A 4 u.t. after the first B:

$$R_2 = (\mathsf{not\ B})* \cdot \mathsf{B} \cdot (\mathsf{not\ t})* \cdot \mathit{Tick} \cdot \mathit{Tick} \cdot \mathit{Tick} \cdot \mathit{Tick} \cdot \mathsf{A} \cdot \top* \tag{1}$$

It is a folklore result that regular expressions can be interpreted as temporal logic formulas. Next, we use the connection between regular expressions and LTL to check the soundness of the Present observer. We study the limitation of this approach and show that the use of a branching logic is more interesting.

Linear Time Specification. We already used LTL to define the "acceptance condition" of our observer. Therefore it is natural to use it again to check the system (Universal || Present). We assume a basic knowledge of LTL; we use X to denote the LTL next operator and T for the constant true. The idea is to define LTL formulas for the regular expressions R_1 and R_2 and to check them on the state graph of (Universal || Present). To this end, we define two useful derived operators, o and -*.

If ϕ_R is the LTL formula corresponding to R and L is a label expression, then (L o ϕ_R) $\overset{\text{def}}{=}$ (L $\wedge$ X ϕ_R) is the formula corresponding to (L $\cdot$ R); it is the formula matching a trace with the head in L and the tail matching R. Likewise, we define the formula (L -* ϕ_R) $\overset{\text{def}}{=}$ (not L) U ϕ_R, that corresponds to the regular expression (not L)$*\cdot R$. With our notations, we can define the derived operator Tick(ϕ) = t o (t -* ϕ) = (t $\wedge$ X ((not t) U ϕ)), that corresponds to the expression $\mathit{Tick} \cdot R$. As a consequence, the formula corresponding to the expression R_2 defined in (1) is simply:

$$\phi(R_2) = \mathtt{B -* (B\,o\,(t -* (Tick\,(Tick\,(Tick\,(Tick\,(A\,o\,T)))))))} \tag{2}$$

The final step is to check that the observer agrees with every trace conforming to R_2. For this, we simply need to check that, for every trace σ such that $\sigma \models \phi(R_2)$, we have $\sigma \models \phi_{\mathsf{P}}$, where ϕ_{P} is the LTL formula associated to the observer. Therefore, in the case of Present, it is enough to check the formula $\phi(R_2)$ => ([]-error). This can be done with *selt*, the LTL model-checker provided with Tina, using the same syntax than in this section.

This approach has some drawbacks. First, because of the semantics of LTL, this method only works with infinite (maximal) traces, whereas regular expression also capture finite traces. An instance of this problem can be seen in the interpretation of the expression R_1 = (not B)$*$, used in our running example. Indeed, a correct choice for $\phi(R_1)$ is the formula []-B, that cannot be defined using our two derived operators, o and -*. Second, this method is not enough to check soundness or innocuousness. For innocuousness, we need to check that the transition A may always eventually happen; this is a typical example of formula that cannot be expressed in LTL (but that is expressible in CTL). Concerning soundness, we only proved half of the property. Actually, we can prove that the property ($\phi(R_1) \vee \phi(R_2)$) <=> ([] - error) is false; a trace not satisfying $\phi(R_1) \vee \phi(R_2)$ will not necessarily raise an error in the observer. The problem lies in the treatment of time divergence (and of fairness), as can be seen from the counter-example produced by our model-checker:

`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. We show how the use of a branching time logic solve these problems.

Branching Time Specification. Like in the previous paragraph, we show how to interpret regular expressions over traces using a temporal logic. In this case, the target logic is a μ-calculus with modalities for forward and backward traversal of the graph . (Many temporal logics can be encoded in the μ-calculus, including, CTL$*$) In this setting, the semantics of a formula ψ is the set of states where ψ holds. The basic modalities are `<L>`ψ and ψ`<L>`. A state s is in `<L>`ψ if and only if there is a (successor) state s' in ψ and a transition from s to s' with a label in `L`. Symmetrically, s is in ψ`<L>` if and only if there is a (predecessor) state s' in ψ and a transition from s' to s with a label in `L`. We should also use `T`, the true formula (matching all the states); `‘0`, that denotes the initial state; and the least fixpoint operator `min X |` ψ`(X)`.

For example, the formula `<A>T` matches all the states that are the source of an `A`-transition and `Reach_A` $=$ `min X | (<A>T` $\vee$ `<-(A` $\vee$ `B` $\vee$ `t)>X)` matches all the states that can lead to an `A` using only internal transitions. As a consequence, we can test innocuousness by checking that the formula `(Reach_A` $\wedge$ `Reach_B` $\wedge$ `Reach_t)` is true for all states.

Concerning soundness, we define the equivalent of the operators `o` and `-*`. Assuming `L` is a label expression, we denote `(`ψ `o L)` the formula ψ`<L>`; if ψ_R corresponds to the regular expression R then ψ`<L>` corresponds to $R{\cdot}L$. Likewise, we use the notation `(`ψ `* L)` for the formula `min X |` $\psi \vee$ `X<L>` and `(`ψ `-* L)` instead of `(`ψ `* (not L))`. The formula `(`ψ `* L)` matches all the states reachable from states where ψ is true using (finite) sequences of transition with label in `L`. Therefore, `(`ψ_R `* L)` corresponds to $R{\cdot}$`L`$*$ and `(‘0 -* B)` corresponds to (`not B`)$*$. Finally, we can define the formula `Tick(`ψ`)` as a shorthand for `(`ψ `o t) -* t`. Using these notations, the formula corresponding to the expression R_2 defined in (1) is:

$$\psi(R_2) = \texttt{((Tick (Tick (Tick (Tick (((‘0 -* B) o B) -* t))))) o A) * T} \quad (3)$$

By construction, The formula ψ_R (corresponding to the regular expression R) is true in all the states reachable from the initial state using a sequence of transition that matches R. To check the soundness of observer Present, we need to prove that the transition error cannot be triggered from any of the state reachable by a sub-expressions of R_2. We denote Ψ_{P} the disjunction of formulas corresponding to these sub-expressions. The definition of errors is also a little bit more involved than in the previous case. We say that a state is “erorred” 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 (the formula `(T<error> * T)` $\wedge$ `(‘0 -* B)`). We denote `Errored` the formula `<error>T` $\vee$ `((T<error> * T)` $\wedge$ `(‘0 -* B))`. Finally, we can prove the soundness of Present by checking the formula Ψ_{P} `<=> Errored`.

5 Related Work, Contributions 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 algorithm, rather than on the verification of the tool itself. For example, Smaus et al. [17] 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 [16], 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.

Concerning observer-based model-checking, most of the works rely on an automatic way to build observers from a formal definition of the properties. Aceto et al. [4] 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. [18] 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 [11], where the authors define observers based on Timed Automata for each pattern.

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 [8,15]—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.

Compared to these works, we have made several contributions. We define a complete verification framework for systems with hard realtime constraints. This framework includes the definition of a set of high-level timed specification patterns; a model-checking toolset and a method to verify the correctness of the framework. Our approach eliminates the need to rely on model-checking algorithms for timed extensions of temporal logics that—when decidable—are very complex and have bad performances. This work is also our first public application of the probe technology, that was added to Fiacre only recently. We believe that

We have described a simple, graphical verification method that can be used to gain confidence on the implementation of our model-checking tools. We show how to automatize 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.

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. A Realtime Specification Patterns Language. Technical Report 11364, LAAS, 2011. `hal-00593965`
2. N. Abid, S. Dal Zilio, and D. Le Botlan. Verification of Realtime Specification Patterns on Time Transitions Systems. Technical Report 11365, LAAS, 2011. `hal-00593963`
3. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous realtime systems in BIP. In Proc. of SEFM–IEEE Software Engineering and Formal Methods, 2006.
4. 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.
5. 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://projects.laas.fr/fiacre/`, 2012.
6. 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.
7. 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.
8. M. Chechik and D.O. Paun. Events in Property Patterns. In *Theoretical and Practical Aspects of SPIN Model Checking*, vol. 3, 1999.
9. M. B. Dwyer, L. Dillon. Online Repository of Specification Patterns. At `http://patterns.projects.cis.ksu.edu/`
10. M. Garnacho, J.-P. Bodeveix and M. Filali. Mechanized Semantics of Realtime Component-Based Languages. private communication (submitted work), 2012.
11. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
12. R. Koymans. Specifying realtime properties with metric temporal logic. *Realtime Syst.*, 2:255–299, 1990.
13. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974.
14. 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.
15. D.O. Paun and M. Chechik. Events in Events in Linear-Time Properties. In *CoRR journal*, vol. cs.SE/9906031, 1999.
16. Esterel Technologies. SCADE Tool Suite. `http://www.esterel-technologies.com/products/scade-suite`
17. 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.
18. J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraints verification methods based on time Petri nets. In *Proc. of FTDCS.* IEEE, 1997.