Verification of Real-Time Specification Patterns on Time Transition Systems

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan

Verification of Real-Time Specification Patterns on Time Transition Systems

Nouha Abid[1,2], Silvano Dal Zilio[1,2], and Didier Le Botlan[1,2]

[1] CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ; Toulouse, France

Abstract. We address the problem of checking properties of Time Transition Systems (TTS), a generalization of Time Petri Nets with data variables and priorities. We are specifically interested by time-related properties expressed using real-time specification patterns, a language inspired by properties commonly found during the analysis of reactive systems. Our verification approach is based on the use of observers in order to transform the verification of timed patterns into the verification of simpler LTL formulas. While the use of observers for model-checking timed extensions of temporal logics is fairly common, our approach is original in several ways. In addition to traditional observers based on the monitoring of places and transitions, we propose a new class of observers for TTS models based on the monitoring of data modifications that appears to be more efficient in practice. Moreover, we provide a formal framework to prove that observers are correct and non-intrusive, meaning that they do not affect the system under observation. Our approach has been integrated in a verification toolchain for Fiacre, a formal modeling language that can be compiled into TTS.

1 Introduction

A distinctive feature of real-time systems is to be subject to severe time constraints that arise from critical interactions between the system and its environment. Since reasoning about real-time systems is difficult, it is important to be able to apply formal validation techniques early during the development process and to define formally the requirements that need to be checked. In this work, we follow a classical approach for explicit-state model checking: we use a high-level language able to describe systems of communicating processes; our goal is to check the validity of specifications expressed in a logical-based formalism; the verification of a system consists in compiling its description and its requirements into a low-level model for which we have the appropriate theory and the convenient tooling.

We propose a new treatment for this traditional approach. We focus on a dense real-time model and use a patterns specification language instead of a timed extension of a temporal logic. One of our contribution is to propose a new decidable verification procedure for a real-time extension to the patterns language of Dwyer et al. [12]. Instead of using real-time extensions of temporal

logic, we propose a patterns language, inspired by properties commonly found during the analysis of reactive systems, that can facilitate the specification of requirements by non-expert. This patterns language can be used to express constraints on the timing as well as the order of events, such as the compliance to deadline or minimum time bounds on the delay between events. While we may rely on timed temporal logics as a way to define the semantics of patterns, the choice of a patterns language has some advantages. For one, we do not have to limit ourselves to a decidable fragment of a particular logic—which may be too restrictive—or have to pay the price of using a comprehensive real-time model-checkeing, that have a very high theoretical complexity.

We address the problem of checking patterns on Time Transition System (TTS), a generalization of Time Petri Nets with data variables and priorities. Our verification approach is based on the use of observers in order to transform the verification of timed patterns into the verification of simpler reachability properties. For the purpose of this work, we focus on a simple *deadline pattern*, named leadsto, and define different classes of observers that can be used to check it. We have applied our approach to a much richer patterns language, with operators that can express requirements on the occurrence and order of events in a system during its execution. A complete description of the patterns language can be found in [1] (we also include a global presentation of the patterns language in Appendix A).

While the use of observers for model-checking timed extensions of temporal logics is fairly common, our approach of the problem is original in several ways. In addition to traditional observers that monitor places and transitions, we propose a new class of observers for Time Transition System (TTS) models that monitor data modifications, and which appears to be more efficient in practice. Another contribution is the definition of a formal framework to prove that observers are correct and non-intrusive, meaning that they do not affect the system under observation. This framework is useful for adding new patterns in our language or for proving the soundness of optimizations.

Beside this theoretical framework, we also provide experimental results. The complete framework defined in this paper has been integrated into a verification toolchain for Fiacre [7]—the high-level modeling language, in our context (see Fig 1). Fiacre is the intermediate language used for model verification in Topcased [13]—an Eclipse based toolkit for critical systems—where it is used as the target of model transformation engines from various languages, such as SDL, UML or AADL [5]. Fiacre is also the source language of compilers into two verification toolboxes: TINA, the TIme Petri Net Analyzer toolset [9], and CADP [14]. For the low-level model, we rely on Time Transition System (TTS), a generalization of Time Petri Nets with data variables and priorities that is one of the input formats accepted by TINA and the output of Frac compiler (the Fiacre compiler for the TINA toolbox). In our toolchain (Fig 1), the Fiacre program, which is combined with patterns, is compiled into TTS model using Frac compiler. The TTS generated is verified then with TINA toolbox. We give some experimental results on the impact of the choice of observers on the size

of the state graphs that need to be generated, that is on the space complexity of our verification method, and on time for verification which refers to the total time needed to generate and to verify the system.

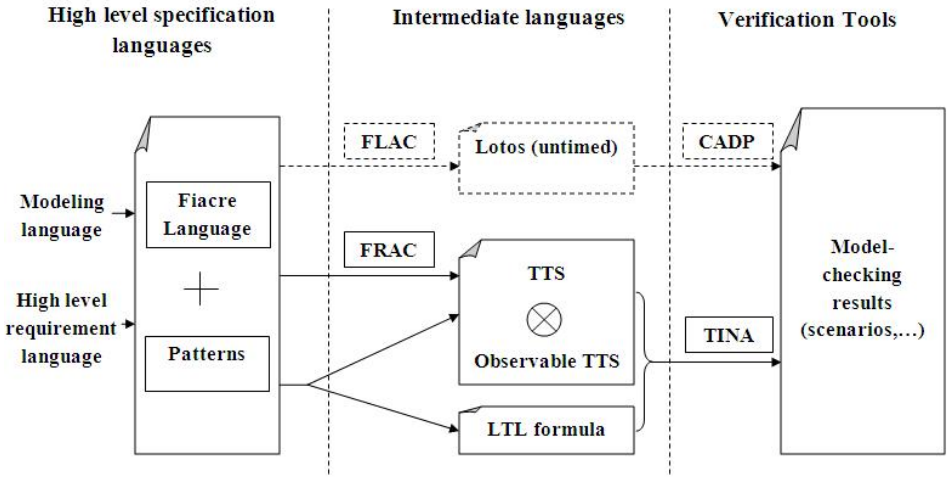


Fig. 1. The global verification toolchain

Outline we define the TTS model and the semantics of timed traces in Section 2. Sections 3 and 4 describe our property specification language and the verification framework. In Section 5, we give some experimental results on the use of the leadsto pattern. We conclude with a review of the related work, an outline of our contributions and some perspectives on future work.

2 Time Transition Systems

We briefly describe the Fiacre formal verification language and show the connection between this high-level language and Time Transition Systems (TTS), an internal format used in our model-checking tools.

2.1 The Fiacre Language

Fiacre is a formal specification language designed to represent both the behavioral and timing aspects of real-time systems. The design of the language is inspired by Time Petri Nets (TPN) for its timing primitives, while the integration of time constraints and priorities into the language can be traced to the BIP framework [10]. A formal definition of the language is given in [7,8]. 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. The listings in Fig. 1.1 and 1.2 give a simple example that models the behavior of a mouse button with double-clicking. 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.). Note that we use a dense-time model.

Listing 1.1. Process

```
process P [click, single,
           double, 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
```

Listing 1.2. Component

```
component Mouse [click, single, double : none] is
   port delay : none in [1,1]
   priority delay > click
   par
      P [click, single, double, delay]
   end
```

Fig. 2. A double-click example in Fiacre

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). 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 (choice and non-deterministic assignments); communication events on ports; and jump to next state. For example, Listing 1.1 declares a process named P, 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 that in state s1 the process may either: (1) receive a click event from the environment, set dbl to true and stay in state s1; or (2) receive an event delay and move to s2.

A *component* is defined as the parallel composition of processes and/or other components, expressed with the operator par ... end. In a composition, processes can interact both through synchronization (message-passing) and accesses 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 Listing 1.2, the declaration of the local port delay means that—for the instance of process P defined in Mouse—the transition from s1 to s2 should take exactly one unit of time.

We assume the reader is already familiar with the usual vocabulary of Time Petri nets [21] (such as the standard notions of markings and time intervals).

2.2 Informal Introduction

Time Transition Systems (TTS) are a generalization of Time Petri Nets (TPN) with priorities and data variables. This computational model is very close to the abstract model for real-time systems defined by Henzinger *et al* [18]—hence the choice of the name—with a different syntax; we use Petri Nets instead of product of automata. We introduce a graphical syntax for TTS using a simple example that models the behavior of a mouse button with double-clicking, as pictured in Fig. 3. 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.). Its precise semantics, expressed in terms of *timed traces*, is defined further (2.4).

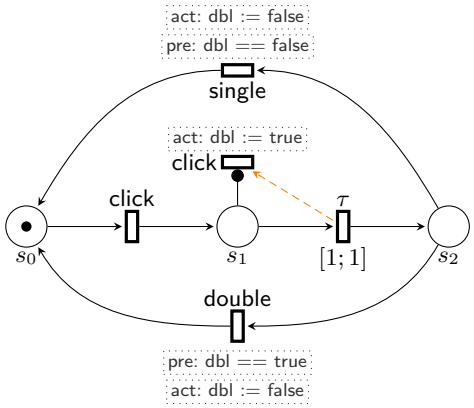


Fig. 3. The double-click example in TTS

Ignoring at first side conditions and side effects (the pre and act expressions inside dotted rectangles), the TTS in Fig. 3 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 τ 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 be fired 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* 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, we use the value of dbl to test whether we should signal a double click or not.

2.3 Formal Definition

Notation : we write $\mathbb{R}^+$ the set of nonnegative real numbers, and $\mathcal{I}(\mathbb{R}^+)$ the set of non-empty convex subsets of $\mathbb{R}^+$ (that is, intervals of $\mathbb{R}^+$, including all open or closed variants, as well as infinite intervals). Given an element I of $\mathcal{I}(\mathbb{R}^+)$ and an element δ of $\mathbb{R}^+$, we define $I - \delta$ as the set $\{x \in \mathbb{R}^+ \mid x + \delta \in I\}$. If $I - \delta$ is not empty, it is itself an element of $\mathcal{I}(\mathbb{R}^+)$.

Definition 1 (Time Transition System). *A TTS is a 9-tuple* $\langle P, T, S, \mathcal{M}, <, \mathsf{tc}, \mathsf{enb}, \mathsf{cfl}, \mathsf{ac}\rangle$*, and its* state *is a triple* (m, s, dtc) *where:*

(a) P*,* T*, and* S *are three disjoint sets, of* places*,* transitions*, and* stores*, respectively.*
(b) $\mathcal{M}$ *is the set of* markings *of the TTS, that is, mappings from* P *to* $\{0, 1\}$*.*
(c) $<$ *is a binary, transitive relation over* T *which encodes the (static) priority relation between transitions.*
(d) tc *(*static time constraint*) is a mapping from* T *to* $\mathcal{I}(\mathbb{R}^+)$*.*
(e) enb *(enable predicate) is a predicate over* $T \times \mathcal{M} \times S$*;* cfl *(conflict predicate) is a predicate over* $T \times \mathcal{M} \times T$ *and* ac *(action) is a mapping from* $T \times \mathcal{M} \times S$ *to* $\mathcal{M} \times S$*.*
(f) The TTS state (m, s, dtc) *is an element of* $\mathcal{M} \times S \times (\mathcal{I}(\mathbb{R}^+))^T$*.* dtc *is a mapping called the* dynamic time constraint*.*

The initial state of a TTS is written $(m^{init}, s^{init}, \mathsf{tc})$*. It requires the dynamic time constraint of every transition to be equal to its static time constraint.*

We make some comments on this definition. Condition (b) implies that the underlying Petri Net is one-safe. The interval $\mathsf{dtc}(t)$ is used to record the time elapsed waiting for the firing of t; therefore we require $\mathsf{tc}(t) = \mathsf{dtc}(t)$ to hold in the initial state for all transitions t in T. In Fig.3, transitions without time constraints are associated to the time interval $[0, +\infty[$. In condition (e), the predicate $\mathsf{enb}(t, m, s)$ indicates whether t is enabled under marking m and store s; $\mathsf{cfl}(t_1, m, t_2)$ indicates whether firing t_1 under marking m should reset the dynamic time interval of t_2 to its default value $\mathsf{tc}(t_2)$; and $\mathsf{ac}(t, m, s)$ returns a

new marking m' and a new store s' corresponding to the effect of firing t under marking m and store s.

It is easy to show that the TTS model includes Time Petri Nets: take an empty store and define $\mathsf{enb}(t, m, \emptyset)$ as the predicate $m \geqslant \mathsf{pre}(t)$, $\mathsf{cfl}(t, m, t')$ as the predicate $m - \mathsf{pre}(t) < \mathsf{pre}(t')$ and $\mathsf{ac}(t, m, \emptyset)$ as the function $m - \mathsf{pre}(t) + \mathsf{post}(t)$.

2.4 Semantics of TTS expressed as Timed Traces

The behavior of a TTS is abstracted as a set of traces, called timed traces. In contrast, the behavior 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 s_0 is marked (using an accessibility check), or that transition click is followed by either single or double within one time unit (using an observer, as described in Sec. 3). 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 ω as a triple (t, m, s) recording the marking and store immediately after the transition t has been fired. We denote Ω the set $T \times \mathcal{M} \times S$ of possible events.

Definition 2 (Timed trace). *A timed trace σ is a possibly infinite sequence of events $\omega \in \Omega$ and durations $d(\delta)$ with $\delta \in \mathbb{R}^+$. Formally, σ is a partial mapping from $\mathbb{N}$ to $\Omega^* = \Omega \cup \{d(\delta) \mid \delta \in \mathbb{R}^+\}$ such that $\sigma(i)$ is defined whenever $\sigma(j)$ is defined and $i \leq j$. The domain of σ is written* $\mathsf{dom}\,\sigma$.

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

Definition 3 (Duration). *Given a finite trace σ, 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 Δ to infinite traces, by defining $\Delta(\sigma)$ as the limit of $\Delta(\sigma_i)$ where σ_i are growing prefixes of σ.

Infinite traces are expected to have an infinite duration. Indeed, to rule out Zeno behaviors, we only consider traces that let time elapse. Hence, the following definition:

Definition 4 (Well-formed traces). *A trace σ is* well-formed *if and only if* $\mathsf{dom}(\sigma)$ *is finite or $\Delta(\sigma) = \infty$.*

The following definition provides an equivalence relation over timed traces. This relation guarantees that a well-formed trace (not exhibiting a Zeno behavior) 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 5 (Equivalence over timed traces). *For each $\delta > 0$, we define $\equiv_\delta$ as the smallest equivalence relation over timed traces satisfying $\sigma.d(0).\sigma' \equiv_\delta \sigma.\sigma'$, $\sigma.d(\delta_1).d(\delta_2).\sigma' \equiv \sigma.d(\delta_1 + \delta_2).\sigma'$, and $\sigma.\sigma' \equiv_\delta \sigma.\sigma''$ 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 [21]. It is expressed as a binary relation between states labeled by elements of Ω^*, and written $(m, s, \mathsf{dtc}) \longrightarrow^{l} (m', s', \mathsf{dtc}')$, where l is either a delay $d(\delta)$ with $\delta \in \mathbb{R}^+$ or an event $\omega \in \Omega$. We say that transition t is *enabled* if $\mathsf{enb}(t, m, s)$ is true. A transition t is *fireable* if it is enabled, *time-enabled* (that is $0 \in \mathsf{dtc}(t)$) 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, dtc) may progress in two ways:

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

$$(m, s, \mathsf{dtc}) \longrightarrow^{d(\delta)} (m, s, \mathsf{dtc}')$$

- *A fireable transition t fires.* Let (m', σ') be $\mathsf{ac}(t, m, s)$, and dtc' be a new mapping such that $\mathsf{dtc}'(t') = \mathsf{tc}(t')$ for all newly enabled transitions t' and for all transitions t' in conflict with t (such that $\mathsf{cfl}(t, m, t')$ holds). For other transitions, we define $\mathsf{dtc}'(t') = \mathsf{dtc}(t')$. Under these hypotheses, we have

$$(m, s, \mathsf{dtc}) \longrightarrow^{(t, m', s')} (m', s', \mathsf{dtc}')$$

We inductively define $\longrightarrow^\sigma$, where σ is a finite trace: $\longrightarrow^\epsilon$ is defined as the identity relation over states, and $\longrightarrow^{\sigma\omega}$ is defined as the composition of $\longrightarrow^\sigma$ and $\longrightarrow^\omega$ (we omit details). We write $(m, s, \mathsf{dtc}) \longrightarrow^\sigma$ whenever there exist a state (m', s', dtc') such that $(m, s, \mathsf{dtc}) \longrightarrow^\sigma (m', s', \mathsf{dtc}')$. Given an infinite trace σ, we write $(m, s, \mathsf{dtc}) \longrightarrow^\sigma$ if and only if $(m, s, \mathsf{dtc}) \longrightarrow^{\sigma'}$ holds for all σ' finite prefixes of σ. Finally, the set of traces of a TTS N is the set of well-formed traces σ such that $(m^{\text{init}}, s^{\text{init}}, \mathsf{tc}) \longrightarrow^\sigma$ holds. This set is written $\Sigma(N)$.

The TINA verification toolbox [9] offers several tools to work with TTS files, including a model-checker for a State-Event version of LTL. However, a

strong limitation of LTL model-checking is that it does not allow the user to express timing constraints, for example, that some deadline between significant events is met. In Sect. 4, we introduce a specification language that makes it easier to express real-time requirements. Each pattern in this language can be compiled into an observer expressed as a TTS, which is then composed with the original system, using a composition operator defined next. Thus, we reduce the verification of timed patterns into the verification of simple LTL properties on the composed system, for which we have the adequate tooling.

2.5 Composition of TTS and Composition of Traces

We study the composition of two TTS and consider the relation between traces of the composed system and traces of its 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 in conditions ensuring that the behavior of the observer does not interfere with the behavior of the observed system.

The composition of two TTS is basically the same as for TPN: we assume given a function lab associating a label, taken in a countable set $\mathcal{L}$, to every transition. Then, transitions bearing the same label are synchronized. However, some conditions must be ensured so that composition is sound, as defined next:

Definition 6 (Composable TTS, synchronized transitions). *We consider two TTS, namely N_1 and N_2, defined as $\langle P_i, T_i, S_i, \mathcal{M}_i, <_i, \mathsf{tc}_i, \mathsf{enb}_i, \mathsf{cfl}_i, \mathsf{ac}_i\rangle$ for i=1, 2, respectively. The set of* synchronized *transitions of N_1 is $\{t_1 \in T_1 \mid \exists t_2 \in T_2 \ . \ \mathsf{lab}(t_1) = \mathsf{lab}(t_2)\}$. We define the set of synchronized 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 synchronized transition t_i of N_i is such that $tc_i(t_i) = [0, +\infty[$, 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, synchronization occurs through transitions only. As stated by the second condition, in every pair of synchronized transitions, both transitions must have a trivial time constraint $[0, +\infty[$. This condition as well as the condition on priorities is necessary to ensure compositionality, as stated by Property 1 below.

Definition 7 (Composition). *Assuming N_1 and N_2 are defined as above, let N be the TTS corresponding to their composition, which we write $N = N_1 \otimes N_2$. It is defined by the 9-tuple $\langle P, T, S, \mathcal{M}, <, \mathsf{tc}, \mathsf{enb}, \mathsf{cfl}, \mathsf{ac}\rangle$ where:*

1. *$P = P_1 \cup P_2$*

2. *Let $\bot$ be an element not in $T_1 \cup T_2$. Let $T_1^\bot$ be $T_1 \cup \{\bot\}$ and $T_2^\bot$ be $T_2 \cup \{\bot\}$. We define T as the following subset of $T_1^\bot \times T_2^\bot$:*
   $T = \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, \mathsf{lab}(t_1) = \mathsf{lab}(t_2)\}$
   $\cup \{(t_1, \bot) \mid t_1 \in T_1 \text{ and } t_1 \text{ is not synchronized}\}$
   $\cup \{(\bot, t_2) \mid t_2 \in T_2 \text{ and } t_2 \text{ is not synchronized}\}$
   We remark that $(\bot, \bot) \notin T$.
3. $S = S_1 \times S_2$
4. $\mathcal{M} = \{0,1\}^P$. *By noticing that $\mathcal{M}$ is in bijection with $\mathcal{M}_1 \times \mathcal{M}_2$, we may freely consider that elements of $\mathcal{M}$ are of the form (m_1, m_2) with $m_1 \in \mathcal{M}_1$ and $m_2 \in \mathcal{M}_2$.*
5. *$<_1$ is a binary relation on T_1. We may freely consider it as a binary relation on $T_1^\bot$ (and so there is no $t \in T_1^\bot$ with $t < \bot$ or $\bot < t$). Similarly, $<_2$ is considered as a binary relation on $T_2^\bot$. Then, $<$ is defined by: for all $(t_1, t_1', t_2, t_2') \in (T_1^\bot)^2 \times (T_2^\bot)^2$, 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).*
6. *For all (t_1, t_2) in T, we define $\mathsf{tc}(t_1, t_2)$ as $\mathsf{tc}_1(t_1)$ if $t_1 \neq \bot$, and as $\mathsf{tc}_2(t_2)$ otherwise.*
7. *$\mathsf{enb}((t_1, t_2), (m_1, m_2), (s_1, s_2))$ is defined as $(t_1 = \bot \vee \mathsf{enb}_1(t_1, m_1, s_1)) \wedge (t_2 = \bot \vee \mathsf{enb}_2(t_2, m_2, s_2))$.*
8. *$\mathsf{cfl}((t_1, t_2), (m_1, m_2), (t_1', t_2'))$ is defined as $(t_1 \neq \bot \wedge t_1' \neq \bot \wedge \mathsf{cfl}_1(t_1, m_1, t_1')) \vee (t_2 \neq \bot \wedge t_2' \neq \bot \wedge \mathsf{cfl}_2(t_2, m_2, t_2'))$*
9. *By convention, for $i = 1, 2$, and for any m and s, we define $\mathsf{ac}_i(\bot, m, s)$ as (m, s). Then, $\mathsf{ac}((t_1, t_2), (m_1, m_2), (s_1, s_2))$ is defined as the pair $((m_1', m_2'), (s_1', s_2'))$, where (m_1', s_1') is $\mathsf{ac}_1(t_1, m_1, s_1)$ and (m_2', s_2') is $\mathsf{ac}_2(t_2, m_2, s_2)$.*

Additionally, the initial state is defined as $((m_1^{init}, m_2^{init}), (s_1^{init}, s_2^{init}), \mathsf{tc})$.

We now define composition of 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 lab to events and duration by defining $\mathsf{lab}(t, m, \sigma) = \mathsf{lab}(t)$ and $\mathsf{lab}(d(\delta)) = d(\delta)$. Additionally, we say that an event ω is synchronized if and only if ω is a delay $d(\delta)$ or ω is (t, m, σ) and t is synchronized (as defined in Def. 6).

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 unsynchronized events—events that are not shared between N_1 and N_2—can only be composed with $d(0)$ (meaning they are not synchronized with an observable event).

Definition 8 (Composable traces). *Let Ω_1 and Ω_2 be the set of events of two TTS N_1 and N_2, respectively. We define the relation $\bowtie$ between Ω_1^* and Ω_2^* as the smallest relation satisfying the following inference system:*

$$\frac{\mathsf{lab}(\omega_1) = \mathsf{lab}(\omega_2)}{\omega_1 \bowtie \omega_2} \qquad \frac{\omega_2 \text{ not synchronized}}{d(0) \bowtie \omega_2} \qquad \frac{\omega_1 \text{ not synchronized}}{\omega_1 \bowtie d(0)}$$

This relation can be extended to pairs of traces (σ_1, σ_2) *of* $N_1 \times N_2$ *as follows. We say that* σ_1 *and* σ_2 *are* composable*, which we write* $\sigma_1 \bowtie \sigma_2$*, if and only if* $\mathsf{dom}\,\sigma_1 = \mathsf{dom}\,\sigma_2$ *and* $\sigma_1(i) \bowtie \sigma_2(i)$ *holds for all* $i \in \mathsf{dom}\,\sigma_1$*. Notice that* $\sigma_1 \bowtie \sigma_2$ *implies* $\Delta(\sigma_1) = \Delta(\sigma_2)$*.*

We are now able to state the compositionality 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 Ω_1 and Ω_2 respectively. Let N be $N_1 \otimes N_2$; we write Ω its set of events. Then there exists a bijection f between Ω and a subset of $\Omega_1^* \times \Omega_2^*$ such that: $\sigma \in \Sigma(N_1 \otimes N_2)$ if and only if there exists a pair of traces (σ_1, σ_2) of $\Sigma(N_1) \times \Sigma(N_2)$ with $\sigma_1 \bowtie \sigma_2$, $\mathsf{dom}\,\sigma = \mathsf{dom}\,\sigma_1$, and $\forall i \in \mathsf{dom}\,\sigma \begin{cases} \sigma(i) = d(\delta) \text{ if } \sigma_1(i) = \sigma_2(i) = d(\delta) \\ \sigma(i) = f^{-1}(\sigma_1(i), \sigma_2(i)) \text{ otherwise} \end{cases}$.

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 $\sigma_1 \otimes \sigma_2$. Thus, this property characterizes 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 Ω and E defined as

- if $t_1 \neq \bot \wedge t_2 \neq \bot$, 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 $t_1 \neq \bot$, then $f((t_1, \bot), (m_1, m_2), (s_1, s_2))$ is $((t_1, m_1, s_1), d(0))$.
- if $t_2 \neq \bot$, then $f((\bot, t_2), (m_1, m_2), (s_1, s_2))$ 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 σ is a trace of N. Let us define σ_1 and σ_2 by $\mathsf{dom}\,\sigma_1 = \mathsf{dom}\,\sigma_2 = \mathsf{dom}\,\sigma$ and for all $i \in \mathsf{dom}\,\sigma$, if $\sigma(i)$ is $d(\delta)$ (for some δ), then $\sigma_1(i) = \sigma_2(i) = d(\delta)$, otherwise $\sigma(i)$ is an event ω, 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 σ_1 (resp. σ_2) is a trace of N_1 (resp. N_2). We show only the result for σ_1, the proof for σ_2 being similar. This is a consequence of the two following implications, where dtc_1 is the function dtc restricted to the domain T_1 (and similarly for dtc_1'), and where $f_1(\omega)$ is the first projection of $f(\omega)$. To ease the reading, we display these implications as inference rules:

$$\frac{((m_1, m_2), (s_1, s_2), \mathsf{dtc}) \longrightarrow^{d(\delta)} ((m_1, m_2), (s_1, s_2), \mathsf{dtc}')}{(m_1, s_1, \mathsf{dtc}_1) \longrightarrow^{d(\delta)} (m_1, s_1, \mathsf{dtc}_1')}$$

$$\frac{((m_1, m_2), (s_1, s_2), \mathsf{dtc}) \longrightarrow^{\omega} ((m_1', m_2'), (s_1', s_2'), \mathsf{dtc}')}{(m_1, s_1, \mathsf{dtc}_1) \longrightarrow^{f_1(\omega)} (m_1', s_1', \mathsf{dtc}_1')}$$

We only sketch the proof of these rules. The delicate part concerns dtc (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 enb in Def. 7. 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 $\mathsf{dtc}'_1(t_1) = \mathsf{dtc}(t_1)$ whereas the expected value (according to the semantics of N_1) would be $\mathsf{dtc}'_1(t_1) - \delta$ (in the first rule) or $\mathsf{tc}_1(t_1)$ (in the second rule). Fortunately, these transitions t_1 must be synchronized transition (as a consequence of the definition of enb), therefore their time interval is always unconstrained, that is $\mathsf{tc}_1(t_1) = \mathsf{dtc}_1(t_1) = [0; +\infty[$, as required by Def. 6, and by remarking that $[0; +\infty[-\delta = [0; +\infty[$ for any δ 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 Def. 7 and of the condition on priorities in Def.6.

Conversely, we assume given two composable traces σ_1 and σ_2 of N_1 and N_2, respectively. The trace σ is defined as stated in the property. It remains to be shown that σ 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 Sec. 4.2.

3 A Real-Time Specification Patterns Language

We introduce the specification patterns language available in our framework. A global presentation of the patterns is given in Appendix A. More details about this language are given in [1]. In this paper, we focus on an observer-based approach for the verification of patterns over TTS models. We also follow a pragmatic approach for studying several possible implementations for observers—and selecting the most sensible one—which we believe is new. Our language extends the property specification patterns of Dwyer et al. [12] with the ability to express time delays between the occurrences of transitions. The framework is expressive enough to define properties like the compliance to deadlines, bounds on the worst-case execution time, etc. The advantage of proposing predefined patterns is to provide a simple formalism to non-experts for expressing properties that can be directly checked with our model-checking tools.

The patterns language follows the classification introduced in [12], with patterns arranged in categories such as universality, bounded existence, etc. In the following, we give some examples of *absence* and *response* patterns. More precisely, we focus on the "response pattern with delay" to show how patterns can be formally defined and to explain our different classes of observers.

Absence pattern with delay this category of patterns can be used to specify delays within which activities must not occur. A typical pattern in this category

can be used to assert that a transition labeled with say E_2, cannot occur between d_1 and d_2 units of time after the first occurrence of a transition labeled with E_1. This is expressed as follows in our language:

$$\textbf{absent}\ E_2\ \textbf{after}\ E_1\ \textbf{for interval}\ [d_1; d_2]\ . \qquad \text{(absent)}$$

An example of use for this pattern would be an (unsatisfied) requirement that we cannot have two double clicks in less than 2 units of time (u.t.), that is **absent** double **after** double **for interval** $[0; 2]$. A more contrived example consists in requiring that if there are no single clicks in the first 10 u.t. of an execution then there should be no double clicks at all. This requirement can be expressed using the composition of two absence patterns using the implication operator and the reserved transition init (that identifies the start of the system):

$$\begin{array}{l}\big(\ \textbf{absent}\ \mathsf{single}\ \textbf{after}\ \mathsf{init}\ \textbf{for interval}\ [0; 10]\big)\\ \quad \Rightarrow \big(\ \textbf{absent}\ \mathsf{double}\ \textbf{after}\ \mathsf{init}\ \textbf{for interval}\ [0; \infty[\big)\ .\end{array}$$

Response pattern with delay this category of patterns can be used to express delays between transitions. The typical example of response pattern states that every occurrence of a transition labeled with E_1 must be followed by an occurrence of a transition labeled with E_2 within a time interval I. (We consider the first occurrence of E_2 after E_1.)

$$E_1\ \textbf{leadsto}\ E_2\ \textbf{within}\ I\ . \qquad \text{(leadsto)}$$

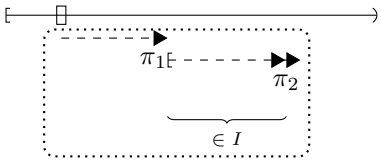
For example, using a disjunction of patterns, we can bound the time between a click and a mouse event: click **leadsto** (single $\vee$ double) **within** $[0, 1]$.

Interpretation of patterns we can use different formalisms to define the semantics of patterns. In this work, we focus on a denotational interpretation, based on first-order formulas over timed traces. We illustrate our approach using the pattern E_1 **leadsto** E_2 **within** I. For the "denotational" definition, we say that the pattern is true for a TTS N if and only if, for every timed-trace σ of N, we have:

$$\begin{array}{l}\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1\pi_1\sigma_2) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_2 = \sigma_3\pi_2\sigma_4\\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \Delta(\sigma_3) \in I \wedge \pi_2 \notin \sigma_3\end{array}$$

where π_1 (resp., π_2) is an atomic proposition that matches all events containing a transition t whose label is E_1 (resp., E_2). The denotational approach is very convenient for a "tool developer" (for instance to prove the soundness of an observer implementation in TTS) since it is self-contained and only relies on the definition of timed traces. As an alternative description of the pattern, we may use an MTL formula (see e.g. [20] for a definition of the logic): $\Box(\pi_1 \Rightarrow (\neg\pi_2)\ \mathbf{U}_I\ \pi_2)$, which reads like a LTL formula enriched by a time constraint on the $\mathbf{U}$ operator. An advantage of our pattern-based approach is that we do not

have to restrict to a particular decidable fragment of the logic. For example, we do not require that the interval I is not punctual (of the form $[d, d]$); we add a pattern in our language only if we can provide a suitable observer for it, so that no decidability issue may arise. Last, we propose a graphical interpretation of patterns, mostly inspired by the Graphical Interval Logic (GIL) [11], called Time Graphical Interval Logic (TGIL), as pictured in the following diagram:



The diagram reads as a recipe (from top to bottom): from any point in time (marked with a $\square$), if there exists an occurrence of the atomic proposition π_1 in the future (take the first one), then there must exist a farther point for which π_2 holds and the delay between these two points is in the interval I. Given this TGIL diagram and a TTS N, the property holds if and only if every timed-trace σ of N satisfies the recipe. This graphical representation is expected to be more convenient than MITL when explaining the semantics of patterns to non-expert.

4 Patterns Verification

We define different types of observers at the TTS level that can be used for the verification of patterns. (This classification is mainly informative, since nothing prohibits the mix of different types of observers.) We make use of the whole expressiveness of the TTS model: synchronous or asynchronous rendez-vous (through places and transitions); shared memory (through data variables); and priorities. 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 (see the discussion in Sect. 5).

4.1 Observers for the Leadsto Pattern

We focus on the example of the leadsto pattern. We assume that some transitions of the system are labeled with E_1 and some others with E_2. We give three examples of observers for the pattern: E_1 leadsto E_2 within $[0, max[$, meaning that whenever E_1 occurs, then (the first occurrence of) E_2 must occur before max units of time. 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). While the use of places and transitions is traditionally favored when observing Petri Nets—certainly because of the definition of Petri Net composition—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 a promising way to implement an observer.

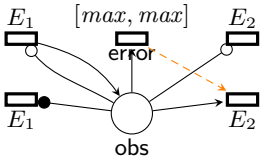


Fig. 4. Transition Observer

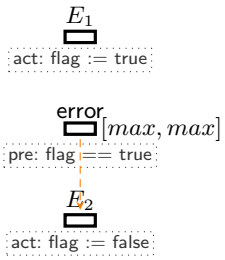


Fig. 5. Data Observer

Transition Observer (Fig. 4) the observer uses a place, obs, which records the time since the last transition labeled E_1 occurred. We use 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*.

Place obs is emptied if a transition labeled E_2 is fired otherwise the transition error is fired after max units of time. The (dashed) priority arc between error and E_2 indicates that the transition E_2 cannot fire if the transition error can fire; which mean that the property is not verified if the "first event" E_2 after E_1 occurs exactly after max units of time.

Proving that a TTS N satisfies this pattern amounts to checking that the system $N \otimes O$ (where O is the observer) never reaches an event containing error. In the system displayed in Fig. 4, we use a *deterministic observer*, that examines all occurrences of E_1 for the failure of a deadline.

By virtue of TTS composition (Def. 7), the observer in Fig. 4 duplicates each transition of N which is labeled E_1 (respectively E_2): one copy can be fired if obs is empty—as a result of the (white-circled) inhibitor arc—while the other can be fired only if the place is full. The transition error is fired if the place obs is full—there is an instance of E_1— and the interval $[max, max]$ is elapsed. It is also possible to define a non-deterministic observer, such that some occurrences of E_1 may be disregarded. This approach is safe since model-checking performs an exhaustive exploration of the TTS states, thus considering all possible scenarios. It is also quite close to the treatment obtained when compiling an (untimed) "equivalent LTL property", namely $\Box(E_1 \Rightarrow \Diamond E_2)$, into an automaton [15]. 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 optimizations.

Data observer (Fig. 5) in this observer, the transition error is conditioned by the value of a variable "flag". This shared boolean variable, which is initially set to false, is true between the firing of a transition E_1 and the following transition E_2. Additionally, transition error is enabled only if flag is true. Therefore, like in the previous case, the verification of the pattern also boils down to checking the

reachability of the event error. Notice that the whole state of the data observer is encoded in its store, since the underlying net has no place.

Place Observer (Fig. 6) in this part, we assume that events E_1 and E_2 are associated to the system entering some given states S_1 and S_2. We can easily adapt this net to observe events associated to transitions in the system. In our case, the observer will be composed with the system through its places S_1 and S_2 (we can extend the classic composition of Petri Nets over transitions to composition over places). In our observer, we use a transition labeled E_1 whenever a token is placed in S_1. Note that TTS obtained from Fiacre are one-safe, meaning that at most one token may be in S_1 at any given time. Likewise, we use a transition E_2 for observing that the system is in state S_2. The error transition is fired if the observer stays in state obs for more than max units of time, as needed.

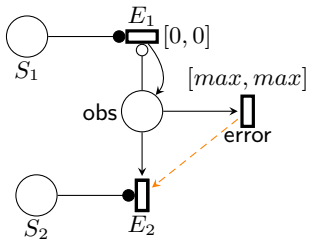


Fig. 6. Place Observer

4.2 Proving Innocuousness and Soundness of 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 σ of the observed system N is preserved in the composed system $N \otimes O$: the observer does not obstruct a behavior of the system (see Lemma 1 below). Conversely, we show that, from any trace of the composition $N \otimes O$, we can obtain a trace of N by erasing the events from O: the observer does not add new behaviors to the system. This is actually a consequence of Property 1 (see Sect. 2.5).

We write T_{sync} the set of synchronized transitions of the observer (as defined in Def. 6). Let ρ be a state of the observer, and σ a trace such that there exists ρ' with $\rho \longrightarrow^{\sigma} \rho'$. Then, we define $O(\rho, \sigma)$ as the state ρ', that is, the state reached after executing trace σ from state ρ. We use $O(\sigma)$ as a shorthand for $O(\rho_{init}, \sigma)$, where ρ_{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}, $\mathsf{tc}(t) = [0; +\infty[$, and t has no priority over other transitions.*
- *In every reachable state ρ of O, and for every l in $\mathsf{lab}(T_{sync})$, there exists a (possibly empty) finite trace σ not containing transitions in T_{sync} such that $\Delta(\sigma) = 0$ and there exists $t \in T_{sync}$ with $\mathsf{lab}(t) = l$, which is fireable in state $O(\rho, \sigma)$.*
- *There exists $\delta > 0$ such that, in every reachable state ρ of O, there exists a (possibly empty) finite trace σ not containing transitions in T_{sync} with $\rho \longrightarrow^{\sigma d(\delta)}$.*

Then, for all σ_1 in $\Sigma(N)$ there exists σ_2 in $\Sigma(O)$ such that σ_1 and σ_2 are composable.

A few comments: first condition is necessary for composition (as required by Def. 6). Second condition ensures that the observer does not prevent the firing of a (l-labeled) synchronized transition for more than 0 units of time. Note also that the observer cannot involve other synchronized transitions while reaching a state where l is firable, since this would abusively constrain the behavior of the main system–not to mention deadlock issues. Third condition ensures that the observer may always let time significantly elapse, without requiring synchronized transitions either (for the same reason).

From the compositionality property (Property 1, Sect. 2.5), it follows that the composed trace $\sigma_1 \otimes \sigma_2$ belongs to $\Sigma(N \otimes O)$, which means that every valid behavior σ_1 of the observed system N is also present in the behavior of $N \otimes O$.

Proof. We assume O satisfies the given hypotheses. We have to build a trace σ_2 in $\Sigma(O)$ such that $\sigma_1 \bowtie \sigma_2$. Since traces are considered up to equivalence, we actually build two traces σ_2 and σ_3 such that both $\sigma_2 \bowtie \sigma_3$ and $\sigma_1 \equiv \sigma_3$ hold:

> *For all σ_1 in $\Sigma(N)$, there exist σ_2 in $\Sigma(O)$ and σ_3 in $\Sigma(N)$ such that $\sigma_1 \equiv \sigma_3$, and for all $t > 0$, for all σ_3^a finite prefix of σ_3 with $\Delta(\sigma_3^a) < t$, there exists σ_2^a finite prefix of σ_2 such that $\sigma_2^a \bowtie \sigma_3^a$ holds.*

We provide an algorithm f that builds σ_2 and σ_3 incrementally.

- The inputs of f are σ_1, σ_1^a, σ_2^a and σ_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, σ_3^a is the part of σ_1 (up to equivalence) that has already been done, whereas σ_1^a is the part remaining to be considered.
- The algorithm returns a new triple σ_1^b, σ_2^b and σ_3^b, which satisfies similar conditions, and such that σ_2^a and σ_3^a are prefixes of σ_2^b, and σ_3^b respectively.
- The algorithm is invoked iteratively with the returned triple. In the finite case (when σ_1 is finite), it eventually reaches a point where σ_1^a is empty, in which case $\sigma_1 \equiv \sigma_3^a$ holds. In the infinite case, we take σ_2 as the limit of σ_2^b, and σ_3 as the limit of σ_3^b.

We now provide the details of the algorithm, then we consider its soundness, being careful 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 σ_1' be such that σ_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:

(a) 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 $\sigma_3^b = \sigma_3^a x$.

(b) If x is an event (t, m, s) with $t \in T_{sync}$, then by hypothesis on O, there exists a finite trace σ 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 $\mathsf{lab}(t') = \mathsf{lab}(t)$. Let σ' be the sequence with the same length as σ and whose elements are $d(0)$. Let ω 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 $\sigma_3^b = \sigma_3^a\sigma' x$.

(c) If x is $d(\delta)$, then by hypothesis on O, there exists $\delta_2 > 0$ and a finite trace σ not containing transitions in T_{sync} such that $\sigma_2^a\sigma d(\delta_2)$ is in $\Sigma(O)$. Let σ' be the sequence with the same length as σ and whose elements are $d(0)$ or $d(\delta_i')$, with appropriate δ_i' such that $\sigma' \bowtie \sigma$ holds. We distinguish two subcases:

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

  (ii) either $\Delta(\sigma) + \delta_2 < \delta$, in which case we return $\sigma_1^b = d(\delta - \delta_2)\sigma_1'$, $\sigma_2^b = \sigma_2^a\sigma d(\delta_2)$, and $\sigma_3^b = \sigma_3^a\sigma' d(\delta_2)$.

There is no difficulty in checking that, for each case, the returned triple satisfies the output conditions, that is, $\sigma_1 \equiv \sigma_3^b\sigma_1^b$ and $\sigma_2^b \bowtie \sigma_3^b$, as long as σ_1^a, σ_1^b, and σ_1^c satisfy the input conditions, that is, $\sigma_1 \equiv \sigma_3^a\sigma_1^a$ 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 σ_1 infinite, which implies $\Delta(\sigma_1) = \infty$ by well-formedness of σ_1. Thus, we only have to show that $\Delta(\sigma_3) = \infty$ (that is, σ_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 σ_3 is infinite. Then, necessarily, at least one case (or subcase) of the algorithm is repeated an infinite number of steps. It cannot be subcase (ii) because δ_2 is a positive constant, and thus $d(\delta_2)$ cannot occur an infinite number of times in σ_3, whose duration is finite. As a consequence, after a finite number of iterations, all delays $d(\delta)$ occuring in σ_1 are handled by subcase (i). It cannot be subcase (i) either, because otherwise $\Delta(\sigma_1)$ would also be finite. It cannot be cases (a) nor (b) either, because otherwise σ_1 would end with an infinite sequence of events (t, m, s), with no delay, which would imply $\Delta(\sigma_1)$ is finite.

The conditions in Lemma 1 are true for the implementation of the leadsto observer, defined in Fig. 4 and in Fig. 5. Therefore these observers 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 $N \otimes O$ if and only if

the pattern is valid for N. We write $\mathsf{error} \in N \otimes O$ to mean there exists a trace $\sigma \otimes \sigma'$ in $\Sigma(N \otimes O)$ such that $\mathsf{error} \in \sigma'$.

Lemma 2. *We have $\mathsf{error} \notin N \otimes O$ if and only if for all $\sigma \in \Sigma(N)$ such that $\sigma = \sigma_1\omega_1\sigma_2$ for some traces σ_1, σ_2 and ω_1 with $\mathsf{lab}(\omega_1) = E_1$, there exist σ_3, ω_2, and σ_4 with $\sigma_2 = \sigma_3\omega_2\sigma_4$, $\mathsf{lab}(\omega_2) = E_2$, and $\Delta(\sigma_3) < max$.*

Proof. This is a consequence of the following result, where for any trace σ, we write $\mathsf{lab}(\sigma)$ for the set of labels occurring in σ, that is $\cup_{i \in \mathsf{dom}\,\sigma} \mathsf{lab}(\sigma(i))$.

Result 1 *Assume $\sigma_1 \in \Sigma(N)$ and $\sigma_2 \in \Sigma(O)$ with $\sigma_1 \bowtie \sigma_2$. We additionally assume that σ_1 and σ_2 are maximal, that is, there is no traces $\sigma_1\sigma_1' \in \Sigma(N)$ and $\sigma_2\sigma_2' \in \Sigma(O)$ such that $\sigma_1\sigma_1' \bowtie \sigma_2\sigma_2'$ holds. Under these hypotheses, we have $\mathsf{error} \in \sigma_2$ if and only if there exist σ_1^a, ω_1, σ_1^b, and σ_1^c such that $\sigma_1 = \sigma_1^a\omega_1\sigma_1^b\sigma_1^c \wedge \mathsf{lab}(\omega_1) = E_1 \wedge \Delta(\sigma_1^b) \geqslant max \wedge E_2 \notin \mathsf{lab}(\sigma_1^b)$.*

Proof. We first show the "if" way. We define three subcases: either $\Delta(\sigma_1^c) = 0$ (subcase i), or, without loss of generality, we may freely assume that either $\Delta(\sigma_1^b) > max$ (subcase ii), or that the first event of σ_1^c is labeled with E_2 (subcase iii). In all these subcases, since $\sigma_1 \bowtie \sigma_2$ holds, σ_2 is necessarily of the form $\sigma_2^a\omega_2\sigma_2^b\sigma_2^c$ with $\sigma_1^a \bowtie \sigma_2^a$, $\omega_1 \bowtie \omega_2$, $\sigma_1^b \bowtie \sigma_2^b$, and $\sigma_1^c \bowtie \sigma_2^c$ (as a consequence of Definition 8). This implies $\mathsf{lab}(\omega_2) = E_1$, $\Delta(\sigma_2^b) = \Delta(\sigma_1^b) \geqslant max$ and $E_2 \notin \mathsf{lab}(\sigma_2^b)$. Additionally, either $\Delta(\sigma_2^c) = 0$ (subcase i holds), or $\Delta(\sigma_2^b) > max$ (subcase ii), or the first event of σ_2^c is labeled with E_2 (subcase iii). Let ρ be $O(\sigma_2^a\omega_2)$. By construction of the observer, error is necessarily enabled in state ρ. Also by construction, only the firing of error or the firing of E_2 may disable error. Moreover, the time constraint on error ensures that it cannot remain enabled continuously for more than max units of time. We now consider the three subcases defined above:

- Subcase(i) : we have $\Delta(\sigma_2^c) = 0$. Necessarily, error must be fired in σ_2^b, otherwise error would be fireable at the end of σ_2, which is in contradiction with the hypothese that σ_1 and σ_2 are maximal.
- Subcase(ii) : we have $\Delta(\sigma_2^b) > max$. Since E_2 is not fired in σ_2^b, then error must be fired in σ_2^b, that is, $\mathsf{error} \in \sigma_2$.
- Subcase(iii) : we have $\Delta(\sigma_2^b) = max$, and the first event of σ_2^c is E_2. As a consequence, error cannot be fireable in $O(\sigma_2^a\omega_2\sigma_2^b)$ because it would have priority over the next event to come, namely E_2, which is a contradiction with the semantics. Thus, error must be fired in σ_2^b, that is, $\mathsf{error} \in \sigma_2$.

In all cases, we see that error is fired in σ_2^b. This concludes the "if" way.

Conversely ("only if" way), we assume that $\mathsf{error} \in \sigma_2$. By construction of the observer, error must have been enabled for max units of time, that is $\sigma_2 = \sigma_2^a\sigma_2'^b\omega_3\sigma_2^c$, with $\mathsf{lab}(\omega_3) = \mathsf{error}$, $\Delta(\sigma_2'^b) = max$, and error is enabled in all events of $\sigma_2'^b$. Without loss of generality, we may take $\sigma_2'^b$ as large as possible, which implies that it starts with the first event that enabled error. $\sigma_2'^b = \omega_2\sigma_2^b$, where $\mathsf{lab}(\omega_2) = E_1$, necessarily. Additionally, since error is kept enabled for all events of σ_2^b, we must have $E_2 \notin \mathsf{lab}(\sigma_2^b)$. Finally, since $\sigma_1 \bowtie \sigma_2$ holds, there exist σ_1^a,

ω_1, σ_1^b, σ_1^c with $\sigma_1 = \sigma_1^a\omega_1\sigma_1^b\sigma_1^c$, $\sigma_1^a \bowtie \sigma_2^a$, $\omega_1 \bowtie \omega_2$, $\sigma_1^b \bowtie \sigma_2^b$ and $\sigma_1^c \bowtie \sigma_2^c$. As a consequence, $\mathsf{lab}(\omega_1) = E_1$, $\Delta(\sigma_1^b) = max$, and $E_2 \notin \mathsf{lab}(\sigma_1^b)$. This concludes the proof.

5 Experimental Results

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. (The software is available at `http://homepages.laas.fr/~nabid`.) 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 *nd*, the Time Petri Net animator provided by TINA.

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) [9] of S generated by our verification tools. In our verification tools, 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 Net associated to 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 $(S \otimes O)$ 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 (see Sect. 4.2), we can show that the SCG of S is a subgraph of the SCG of $S \otimes O$, 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 select the best observer for a real-time property. In the context of "untimed properties", this approach may be compared to the problem of optimizing the generation of Büchi Automata from LTL formulas, see e.g. [15].

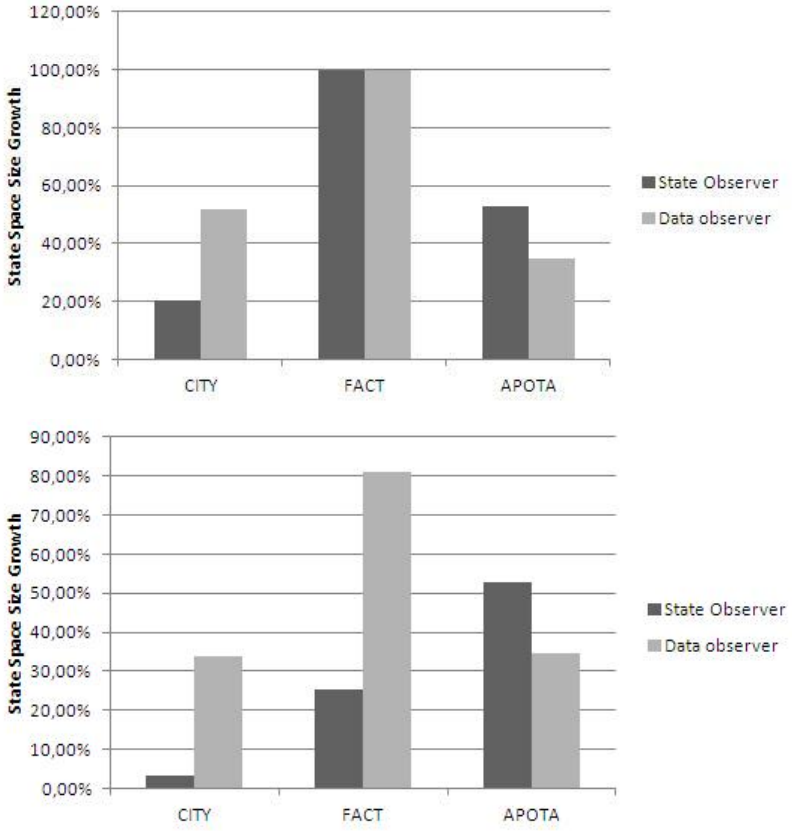


Fig. 7. Complexity for the data and state observer classes in percentage of system size growth—average time for invalid properties (above) and valid properties (below).

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. 7. For the experiments used in this paper, we use three examples of TTS selected because they exhibit very different features (size of the state space, amount of concurrency and symmetry in the system, ...). Example CITY is a TTS 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. This is a typical leadsto pattern that can

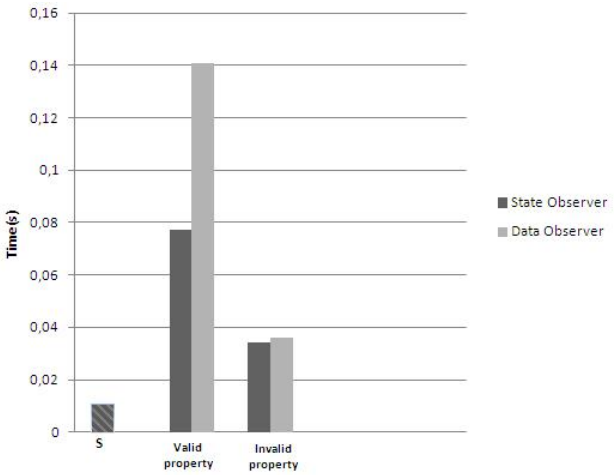


Fig. 8. Total verification time (factory example)

be written: df **leadsto** pf **inlessthan** 30, where df is a state corresponding to the start of the process and pf is a state corresponding to the delivery of the document to the applicant. The same property, with a minimum delay set to 100 instead of 30, gives our example of invalid property. Example FACT is a TTS modeling a manufacturing plant composed of two command lines sharing some of theirs 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. For the invalid property, we use the same requirement, but shortening the delay to 5 minutes. The last example, APOTA, is an industrial use case that models the dynamic architecture for a network protocol in charge of data communications between an airplane and ground stations [6]. 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.

In Fig. 7, 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 Sect. 4.1 and our three running examples. We give one figure in the case where the property is not valid and another when the property is valid. We have compared also, in Fig. 8 and 9, the total verification time for the FACT and APOTA examples. This time refers to the time spended generating the complete state space of the system and verifying the property. In Fig. 8 and 9, S refers to the initial system (the state space of the system without adding observer) while "Valid property" and

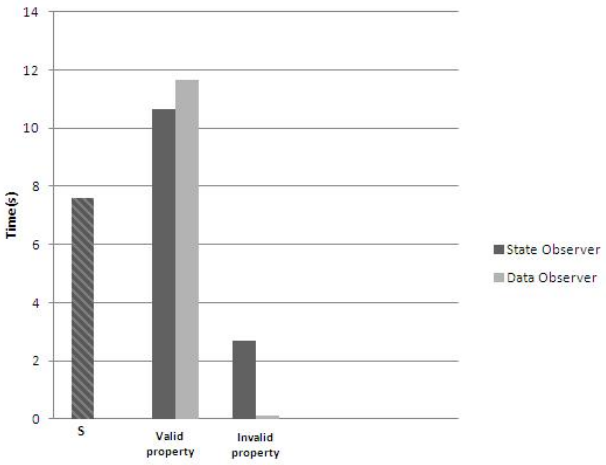


Fig. 9. Total verification time (APOTA example)

"Unvalid property" refer to the state space of the system synchronized with data observer and state observer in the case of valid and unvalid 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 average and that seldom gives the worst result. Although the diagrams shown above do not seem in favor of data observers, we have carried out other numerous tests which actually show promising results for data observers. As a matter of fact, the few poor results of the data observer can be explained by pathological cases where the time parameters used in the property are very different from those used in the system (a classical problem whith model-checking real-time systems.)

6 Related Work, Contributions and Perspectives

Two broad approaches coexist for the definition and verification of real-time properties: (1) real-time extensions of temporal logic [17]; and (2) observer-based approaches, such as the Context Description Languages (CDL) of Dhaussy et al. [22] or approaches based on timed automata [20,3,2].

Obviously, the logic-based approach provides most of the theoretically well-founded body of works, such as complexity results for different fragments of real-time temporal logics [17]: Temporal logic with clock constraints (TPTL); Metric Temporal Logic—with or without interval constrained operators—; Event Clock Logic; etc. The algebraic nature of logic-based approaches make them expressive and enable an accurate formal semantics. However, it may be impossible to express all the necessary requirements inside the same logic fragment if we ask

for an efficient model-checking algorithm (with polynomial time complexity). For example, Uppaal [4] chose a restricted fragment of TCTL with clock variables, while Kronos provide a more expressive framework, but at the cost of a much higher complexity. As a consequence, selecting this approach requires to develop model-checkers for each interesting fragment of these logics—and a way to choose the right tool for every requirement—which may be impractical.

Pattern-based approaches propose a user-friendly syntax that facilitates their adoption by non-experts. However, in the real-time case, most of these approaches lack in theory or use inappropriate definitions. One of our goal is to reverse this situation. In the seminal work of Dwyer et al. [12], patterns are defined by translation to formal frameworks, such as LTL and CTL. There is no need to provide a verification approach, in this case, since efficient model-checkers are available for these logics. This work on patterns has been extended to the real-time case. For example, Konrad et al. [19] extends the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL, but they do not study the decidability of the verification method (the implementability of their approach). Another related work is [16], where the authors define observers based on Timed Automata for each pattern. However, they do not provide a formal framework for proving the correctness or the innocuousness of their observers and they have not integrated their approach inside a model-checking toolchain.

Concerning observer-based approaches, we can cite the work of Aceto et al. [2,3] 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. In the context of Time Petri Net, a similar approach has been experimented by Toussaint et al. [23], but they propose a less general model for observers and consider only two verification techniques over four kinds of time constraints.

In contrast to these related works, we make the following contributions. We reduce the problem of checking real-time properties to the problem of checking LTL properties on the composition of the system with an observer. This paper provides several theoretical results: we give the first formal account of the semantics of TTS—a low-level model used in the TINA toolset—and provide a formal framework for proving the correctness of observers. We use also this framework to check wether an observer is non-intrusive—whether it can interfere with the behaviour of the system under observation—a property that we call *innocuousness*. To the best of our knowledge, this property is totally overlooked in the related work on observer-based approaches. Moreover, we give necessary conditions for an observer to be innocuous (see Lemma 1 in Sect. 4.2).

Our approach has been integrated into a complete verification toolchain for the Fiacre modeling language and can therefore be used in conjunction with Topcased. We give several experimental results based on the use of this toolchain in Sect. 4. 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 theoretical side, we are currently looking into the use of mechanized 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.

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. A Real-Time Specification Patterns Language. Technical Report 11364, LAAS, 2011. `http://homepages.laas.fr/nabid/#Publications`.
2. L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003.
3. L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 1998.
4. G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
5. B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali, and F. Vernadat. Formal verification of aadl specifications in the topcased environment. In F. Kordon and Y. Kermarrec, editors, *Ada-Europe*, volume 5570 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2009.
6. B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, S. Heim, P. Gaufillet, and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *ERTSS 2010 – 5th International Congress and Exhibition on Embedded Real-Time Software and Systems*, May 2010.
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 *ERTS 2008*, Toulouse, France, 2008.
8. B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. The Syntax and Semantics of Fiacre – Version 2.0. 2007.
9. 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.
10. M. Bozga, V. Sfyrla, and J. Sifakis. Modeling synchronous systems in bip. In S. Chakraborty and N. Halbwachs, editors, *EMSOFT*, pages 77–86. ACM, 2009.
11. 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 Trans. Softw. Eng. Methodol.*, 3(2):131–165, 1994.

12. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
13. P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. 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. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.
15. P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.
16. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
17. T. A. Henzinger. It's about time: Real-time logics reviewed. In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 1998.
18. T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Inf. Comput.*, 112(2):273–337, 1994.
19. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 372–381. ACM, 2005.
20. O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In E. Asarin and P. Bouyer, editors, *FORMATS*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2006.
21. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974.
22. A. Raji, P. Dhaussy, and B. Aizier. Automating context description for software formal verification. In *Workshop MoDeVVa*, 2010.
23. J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraints verification methods based on time petri nets. In *FTDCS*, pages 262–269. IEEE Computer Society, 1997.

A Real-time Patterns Language

We present bellow our real-time patterns language as well as a textual definition for each patterns. More details about these patterns are provided in [1].

Table 1: Existence patterns

1. Present A after B within $[d_1, d_2]$

   – **Textual Definition:** An event, say A, must occur between d_1 and d_2 units of time (u.t) after an occurrence of the event B. The pattern is also satisfied if B never occurs.

2. Present first A before B within $[d_1, d_2]$

   – **Textual Definition:** The first occurrence of A holds within $[d_1, d_2]$ u.t. before the first occurrence of B. It also holds if B does not occur.

3. Present A lasting D

   – **Textual Definition:** The goal of this pattern is to assert that from the first occurrence of A, the predicate A remains true for at least duration D. It makes sense only if A is a state predicate (that is, on the marking and store), and which does not refer to any transition (since transitions are instantaneous, we cannot require a transition to last for a given duration).

4. Present A within I

   This pattern is equivalent to **present** A **after** init **within** I

5. Present A between B and C within I

   This pattern is equivalent to the composition of two patterns :
   present A **after** B **within** I
   and
   present A **before** C **within** I

6. Present A after B until C within I

   This pattern is equivalent to: A **leadsto** C **within** I **after** B

Table 2: Absence patterns

1. Absent A after B for interval $[d_1, d_2]$

   – **Textual Definition:** This pattern asserts that an event, say A, must not occur between d_1–d_2 u.t. after the first occurrence of an event B. This pattern is dual to **Present** A **After** B **within** $[d_1, d_2]$ (but it is not strictly equivalent to its negation, because in both patterns, B is not required to occur).

2. Absent A before B for duration D

   – **Textual Definition:** This pattern asserts that no A can occur less than D u.t. before the first occurrence of B.

3. Absent A within I

   This pattern is defined as **Absent** A **after** init **within** I

4. Absent A lasting D

   This pattern is defined as **Present** $\neg A$ **lasting** D

5. Absent A between B and C within I

   This pattern is equivalent to the composition of two patterns :
   absent A **after** B **within** I
   and
   absent A **before** C **within** I

6. Absent A after B until C within I

   This pattern is equivalent to : **absent** A $\wedge$ C **after** B **within** I

Table 3: Response patterns

Response Patterns

1. A leadsto first B within $[d_1, d_2]$

– **Textual Definition:** This pattern states that every occurrence of an event, say A, must be followed by an occurrence of B within a time interval $[d_1, d_2]$ (considering only the first occurrence of B after A).

2. A leadsto first B within $[d_1, d_2]$ before R

– **Textual Definition:** This pattern asserts that, before the first occurrence of R, each occurrence of A is followed by B, which occurs both before R, and in the time interval $[d_1, d_2]$ after A. If R does not occur, the pattern holds.

3. A leadsto first B within $[d_1, d_2]$ after R

– **Textual Definition:** This pattern asserts that after the first occurrence of R, "A **leadsto first** B within $[d_1, d_2]$" holds.

4. A leadsto B between Q and R within I

This pattern is equivalent to the composition of two patterns :
A **leadsto** B **within** I **after** Q
and
A **leadsto** B **within** I **before** R

5. A leadsto B after Q until R within I

This pattern is equivalent to : A **leadsto** B **between** Q **and** R **within** I

Table 4: Universality patterns

1. always A lasting D

This pattern is defined as **Present** A **lasting** D

2. always A within I This pattern is defined as ¬(**absent** A **after** init **for interval** I)
3. always A after B for duration D This pattern is defined as **absent** $\neg A$ **after** B **for duration** D
4. always A before B for duration D This pattern is defined as **absent** $\neg A$ **before** B **for duration** D

Table 5: Precedence patterns

1. A precedes B for duration D This pattern is defined as **absent** B **before** A **for duration** D
2. A precedes B before R for duration D – **Textual Definition:** This pattern asserts that before the first occurrence of R, no B occurs before the first occurrence of A for duration D.
3. A precedes B after R for duration D – **Textual Definition:** This pattern asserts that before the first occurrence of R, no B occurs before the first occurrence of A for duration D.
4. A precedes B between Q and R for duration D This pattern is equivalent to the composition of two patterns : A **precedes** B **after** Q **for duration** D **and** A **precedes** B **before** R **for duration** D
5. A precedes B after Q until R for duration D

This pattern is equivalent to : **absent** B **between** Q **and** R **for duration** D