Verification of Real-Time Specification Patterns on Time Transitions Systems

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan

Verification of Real-Time Specification Patterns on Time Transitions 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 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 with the goal to check the validity of specifications expressed in a logical-based formalism; then 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, as importantly, the convenient tooling.

We propose a new twist to this overall traditional approach. We focus on the use of a dense real-time model and base our approach on a pattern specification language instead of using a timed extension of a temporal logic. One of our contribution is to propose a decidable verification procedure for a real-time extension to the pattern language of Dwyer et al. [11] that is new. Instead

of using real-time extensions of temporal logic, we propose a pattern language, inspired by properties commonly found during the analysis of reactive systems, that can facilitate the specification of requirements by non-expert. This pattern 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 pattern 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-checker, whose complexity may be daunting.

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. (A complete description of the pattern language can be found in [1].) 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 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 [5]—the high-level modeling language, in our context. Fiacre is the intermediate language used for model verification in Topcased [12]—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 [8]. Fiacre is also the source language of compilers into two verification toolboxes: TINA, the TIme Petri Net Analyzer toolset [7], and CADP [13]. For the low-level model, we rely on Time Transition Systems (TTS), a generalization of Time Petri Nets with data variables and priorities that is one of the input formats accepted by TINA. 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.

Outline. We define our modeling language 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, Modeling and Semantics

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 [9]. A formal definition of the language is given in [5,6]. 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 a simple example in Fig. 1, 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 (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. 1. 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.

2.2 Time Transition Systems

Time Transition Systems (TTS) are a generalization of 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* [17]—hence the choice of the name—with a different syntax; we use Petri Nets instead of product of automata. We now describe the TTS model more formally. We introduce a graphical syntax for TTS using our running example and define its semantics using the notion of *timed traces*, see Definition 2. The diagram in Fig. 2 shows a TTS corresponding to the double-click example.

Ignoring at first side conditions and side effects (the pre and act expressions inside dotted rectangles), the TTS in Fig. 2 can be viewed as a TPN with one token in place s_0 as its initial marking. From this "state", a click transition may occur and move the token from s_0 to s_1. With this marking, the internal transition 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. In parallel, the transition labeled click may be fired one or more time 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 line between transitions), the click transition is disabled until the token moves from s_1 to s_2.

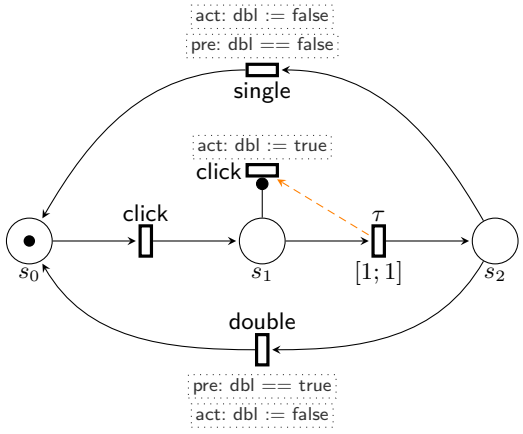


Fig. 2. The double-click example in TTS

Datas of the TTS are managed in the act and pre expressions that may be associated to each transition. Such transitions act and 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. Compared to TPN, a transition t in a TTS is enabled if there is both: (1) enough token 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 reach the place s_2 in the TTS of Fig. 2, we use the value of dbl to test whether we should signal a double click or not.

We give a more formal definition of TTS and define their semantics using a notion of timed traces.

Definition 1 (Time Transition System). *We assume two countable sets for* places *and* transitions*, denoted $\mathcal{T}$ and $\mathcal{P}$, and a set of* stores*, $\mathcal{S}$. The set of predicates over stores is denoted $\mathcal{P}re$ and the set of update functions over stores is denoted $\mathcal{A}ct$.*

A TTS is a tuple $\langle P, T, \mathcal{M}, <, \mathsf{tc}, \mathsf{enb}, \mathsf{cfl}, \mathsf{ac}\rangle$, and its state *is a triple $\langle m, s, \mathsf{dtc}\rangle$ where:*

(a) P is a finite set of places, $P \subset \mathcal{P}$, and T is a finite set of transitions, $T \subset \mathcal{T}$.
(b) $\mathcal{M}$ is the set of markings *of the TTS, that is, mappings from P to $\{0, 1\}$.*
(c) the binary relation $<$ is a partial order over T that encodes the priority relation between transitions.
(d) tc is a mapping that associates, to each transition t in T, a static *time interval $\mathsf{tc}(t)$ with rational (or infinite) bounds.*
(e) enb is an enabling predicate over $T \times \mathcal{M} \times \mathcal{S}$; cfl is the conflict predicate over $T \times \mathcal{M} \times T$ and ac is an action *function mapping $T \times \mathcal{M} \times \mathcal{S}$ to $\mathcal{M} \times \mathcal{S}$.*
(f) $m \in \mathcal{M}$, $s \in \mathcal{S}$, and dtc is a mapping that associates, to each transition t in T, a dynamic *time interval $\mathsf{dtc}(t)$.*

The initial state of a TTS is written $(m_{init}, s_{init}, \mathsf{tc})$. It requires the static and dynamic time intervals of every transition to be equal.

We make some comments on the conditions accompanying the definition of TTS. 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 assume that $\mathsf{tc}(t) = \mathsf{dtc}(t)$ in the initial state for all transitions t in T. A transition without time constraints is associated to the time interval $[0, +\infty[$. For 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.

We know consider briefly the dynamic semantics of TTS, which is similar to the semantics of Time Petri-Nets [21]. We say that transition t is *enabled* if $\mathsf{enb}(t, m, s)$ is true. A transition t can be *fired* 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 $\delta \in \mathsf{dtc}(t)$, meaning that all enabled transitions t are not urgent. In that case, all dynamic time intervals of enabled transitions are shifted by $-\delta$.
- *A transition t fires.* The current marking and the current store are updated with $\mathsf{ac}(t, m, s)$. Besides, for all transitions t', $\mathsf{dtc}(t')$ is reset to $\mathsf{tc}(t')$ whenever t' is newly enabled, or t conflicts with t', that is $\mathsf{cfl}(t, m, t')$ holds.

It is easy to show that the TTS model includes classical Time Petri Net: 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 $m - \mathsf{pre}(t) + \mathsf{post}(t)$. The TTS model is also a good target for compiling the Fiacre language: a process P is compiled to a TTS with one place for every state in P, while parallel composition of processes is modeled by composition of TTS (see Sect. 2.4). The reference semantics of Fiacre [6] is defined using a structural approach to operational semantics. This semantics has been implemented in a tool called *frac*, that compiles a program into a Timed Transition System. No formal presentation of the semantics of Fiacre using TTS was ever published before the present work.

2.3 Expressing the Semantics of TTS with Timed Traces

A *trace* σ of a TTS is basically a sequence of transitions and time elapses. We extend this simple definition to also keep track of the state of the system after a transition has been fired. Formally, we define an event ω as a triple (t, m, s) recording the marking and store after the transition t has been fired. We denote Ω the set $T \times \mathcal{M} \times \mathcal{S}$ of possible events. We use classic notations for sequences: the empty sequence is denoted ϵ and $\sigma(i)$ is the i^{th} element of σ; given a finite sequence σ and a—possibly infinite—sequence σ', we denote $\sigma.\sigma'$ the *concatenation* of σ and σ'. The concatenation operator is associative.

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 $\mathsf{dom}\,\sigma$.*

Given a finite trace σ, we can define the *duration* of σ, written $\Delta(\sigma)$, that is the function inductively defined by the 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. We say that an infinite trace σ is *well-formed* if and only if $\Delta(\sigma) = \infty$ or, equivalently, if for all $\delta > 0$, there exists σ_1, σ_2 such that $\sigma = \sigma_1.\sigma_2$ and $\Delta(\sigma_1) > \delta$. Finite traces are always well-formed.

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 have 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 3 (Equivalence over timed traces). *For each $\delta > 0$, we define $\equiv_\delta$ as the smallest equivalence relation over timed traces satisfying $\sigma.d(0).\sigma' \equiv_\delta \sigma.\sigma'$, $\sigma.d(t).d(t').\sigma' \equiv_\delta \sigma.d(t+t').\sigma'$, and $\sigma.\sigma' \equiv \sigma.\sigma''$ 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 that happens at the same date (with no delay in between) while keeping a causality relation (one event is before another).

2.4 Composition of TTS and Composition of Traces

We study the composition of two TTS and consider the relation between the traces of a composed system and the semantics of its parts. This operation is important since, in the context of this work, both the system and the observer are TTS and we use composition to graft an observer to a system. In particular, we look for conditions ensuring that the behavior of the observer cannot interfere with the behavior of the observed system.

The composition of two TTS is basically the same than for TPN: we consider a function lab associating a label, taken in a countable set $\mathcal{L}$, to every transition and we synchronize transitions bearing the same label. Additionally, we require that every synchronized transition t has no time constraint and there is no transition t' with $t' < t$. In this case we say the two systems are *composable*.

We extend lab to events and duration by defining $\mathsf{lab}(t, m, \sigma) = \mathsf{lab}(t)$ and $\mathsf{lab}(d(\delta)) = d(\delta)$. In the same way that systems can be composed, it is possible to synchronize 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 unrelated events—events that are not shared between N_1 and N_2—can only be synchronized with $d(0)$ (meaning they are not synchronized with an observable event).

Definition 4 (Composable traces). *Let Ω_1^* and Ω_2^* be the set of possible events of two TTS N_1 and N_2. We define the relation $\bowtie$ between events of Ω_1^* and Ω_2^* by the following inference system:*

$$\frac{\mathsf{lab}(\omega_1) = \mathsf{lab}(\omega_2)}{\omega_1 \bowtie \omega_2} \qquad \frac{\mathsf{lab}(\omega_2) \in \mathcal{L} \backslash \mathsf{lab}(T_1)}{d(0) \bowtie \omega_2} \qquad \frac{\mathsf{lab}(\omega_1) \in \mathcal{L} \backslash \mathsf{lab}(T_2)}{\omega_1 \bowtie d(0)}$$

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* $\Delta(\sigma_1) = \Delta(\sigma_2)$*.*

If two TTS N_1 and N_2 are composable, we write $N_1 \otimes N_2$ their composition. Likewise, we can define a composition operation over two timed traces. However, by lack of space, and since our results do not actually rely on the definition of composition, we do not provide the formal definition of $\otimes$. Yet, we assume it satisfies the compositionality property (Property 1), meaning that a composition $N_1 \otimes N_2$ cannot exhibit behaviors that are not part of either N_1 or N_2. As a consequence, any composition operator that satisfies this property is suitable.

Property 1 (Compositionality). Assume N_1 and N_2 are composable systems with events in Ω_1 and Ω_2 respectively. Then there exists a bijection f between Ω, the events of $N_1 \otimes N_2$, and $\Omega_1 \times \Omega_2$ such that: σ is a trace of $N_1 \otimes N_2$ if and only if there exists a pair of traces (σ_1, σ_2) of $N_1 \times N_2$ such that $\sigma_1 \bowtie \sigma_2$, $\mathsf{dom}\,\sigma = \mathsf{dom}\,\sigma_1$, and for all $i \in \mathsf{dom}\,\sigma$, if $\sigma(i) = d(\delta)$, then $\sigma_1(i) = \sigma_2(i) = d(\delta)$, otherwise $f \circ \sigma(i) = (\sigma_1(i), \sigma_2(i))$.

In the following, by abuse of notation, we use (σ_1, σ_2) to refer to a timed trace of $N_1 \times N_2$ in the case where σ_1 and σ_2 are composable.

The TINA verification toolbox [7] offers several tools to work with TTS files, including both a model-checker for a State-Event version of LTL and a model-checker for the μ-calculus. 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 the next section, we introduce a specification language for Fiacre that makes it easier to express real-time requirements on systems. Each pattern in this language can be compiled into an observer expressed as a TTS. Then we use the TTS composition operation and reduce the verification of timed patterns into the verification of simple LTL properties, for which we have the adequate tooling.

3 A Real-Time Specification Patterns Language

We describe the specification patterns language available in our framework. A comprehensive description of this language is given in [1]. In this paper, we focus on an observer-based approach for the verification of patterns on a Fiacre program. 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. [11] with the ability to express time delays between the occurrences of events. In our context, observable events at the Fiacre level are: a process entering or leaving a state; a variable update; a communication through a port. The result 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 pattern language follows the classification introduced in [11], with patterns arranged in categories such as universality, bounded existence, etc. In the following, we give some examples of *absence* and *response* patterns. We will 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 an event, say ω_2, cannot occur between d_1–d_2 units of time after the occurrence of an event ω_1. This requirement corresponds to a basic absence pattern in our language:

$$\textbf{absent}\ \omega_2\ \textbf{after}\ \omega_1\ \textbf{within}\ [d_1; d_2]\ . \qquad \text{(absent)}$$

An example of use for this pattern would be a requirement that we cannot have more than two double clicks in less than 2 units of time (u.t.), that is **absent** double **after** double **within** $[0; 2]$. A more contrived example of requirement is to impose 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 event init (that identifies the start of the system):

$$\big(\,\textbf{absent}\ \text{single}\ \textbf{after}\ \text{init}\ \textbf{within}\ [0; 10]\big) \Rightarrow \big(\,\textbf{absent}\ \text{double}\ \textbf{after}\ \text{init}\ \textbf{within}\ [0; \infty[\big)$$

Response pattern with delay. This category of patterns can be used to express delays between events. The typical example of response pattern states that every occurrence of an event, say ω_1, must be followed by an occurrence of the event ω_2 within a time interval I. (We consider the first occurrence of ω_2 after ω_1.)

$$\omega_1\ \textbf{leadsto}\ \omega_2\ \textbf{within}\ I\ . \qquad \text{(leadsto)}$$

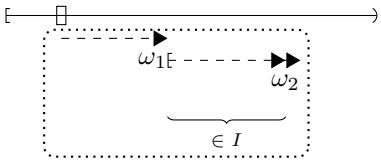
For example, using a disjunction of patterns, we can bound the time between a click and a mouse event: click **leadsto** (single $\vee$ double) **within** $[0, 1]$.

Interpretation of patterns. We use different formalisms to define the semantics of patterns: (1) a denotational interpretation, based on first-order formulas over timed traces; (2) a logical interpretation using Metric Temporal Logic (MTL); and (3) a graphical framework, based on the use of a nonambiguous diagrammatic notation. For this last method, we have defined the Timed Graphical Interval Language (TGIL) [1] that takes its inspiration from GIL, a graphical language for temporal logic (without time) defined by Dillon et al. [10]. An advantage of this approach is that, in some cases, a graphical notation is easier to understand by non-experts than temporal logic formulas.

We illustrate this approach using the pattern ω_1 **leadsto** ω_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:

$$\forall \sigma_1, \sigma_2 \ . \ (\sigma = \sigma_1 \omega_1 \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 \ . \ \sigma_2 = \sigma_3 \omega_2 \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge \omega_2 \notin \sigma_3 \ . \quad (1)$$

The denotational approach is very convenient for a "tool developper" (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. For the second method, the pattern corresponds to a simple MTL formula (see e.g. [20] for a definition of the logic): $\Box(\omega_1 \Rightarrow (\neg\omega_2)\ \mathbf{U}_I\ \omega_2)$. An advantage of our 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 and therefore we are not concerned by these decidability issues. Finally, for the graphical method, we can explain the pattern with the diagram below:



The diagram reads as a recipe (from top to bottom): from any point in time (marked with a $\Box$), if I find a point where ω_1 holds (the first one), then I have necessarily to find a point afterward where ω_2 holds and the delay between these two points is in the interval I. For the TGIL definition, we say that the pattern is true for a TTS N if and only if we can match the recipe defined by TGIL on every timed-trace σ of N.

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 two 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; the second observer monitors shared, boolean variables injected into the system. While the use of 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. The results of our experiments seem to show that, in practice, this is a promising way to implement an observer.

Transition Observer (Fig. 3). The idea is to use a special place in the observer, obs, in order to record the time since the last transition labeled E_1 occured. The place is emptied if a transition E_2 is fired otherwise the transition error is fired after *max* units of time. Proving that a TTS N satisfies this pattern amounts to checking that the system $N \otimes O$ never reaches the event error, where O is the observer. In the system displayed in Fig. 3, we use a *deterministic observer*, that examines all occurrences of E_1 for the failure of a deadline. We only give a simplified version of the correct observer, that needs to be extended if one of the transition E_1 or E_2 bears a non-trivial time constraint (that is, different from $[0, \infty[$). Also, following similar ideas, we could provide observers based on places, but we omit the details here.

By virtue of the definition of Petri Net composition, the observer in Fig. 3 duplicates each transition 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. 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 we perform an exhaustive exploration of the TTS states. 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 [14]. 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.

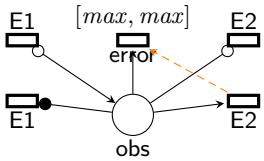


Fig. 3. Transition Observer

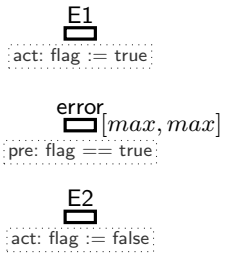


Fig. 4. Data Observer

Data observer (Fig. 4). For this other example of observer, the idea is to condition the transition error of the observer to the value of a flag variable; flag is a shared boolean variable that is true between the firing of a transition E_1 and the following transition E_2 (we also set flag to false initially). Therefore the validity of the pattern is conditioned by the predicate dtc(error) $\leq$ *max* and, 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 behavior of the data observer is encoded in its store, since the underlying net has no place.

4.2 Proving Innocuity 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. 2.4).

Let $\Sigma(N)$ be the set of well-formed traces of the TTS N. We write T_{sync} the set of synchronized transitions of the observer, that is, the set of transitions t of O such that $\mathsf{lab}(t) = \mathsf{lab}(t')$ for some transitions t' of N. Given a state ρ and a finite trace σ, we write $\rho \longrightarrow^{\sigma}$ to indicate that σ is a trace for O in state ρ. Then, $O(\rho, \sigma)$ is the state reached after executing trace σ. 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. The complete proofs are given in Appendix A.

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

1. *For every t in T_{sync}, $\mathsf{tc}(t) = [0; +\infty[$*
2. *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 with $\mathsf{lab}(t) = l$, which is fireable in $O(\rho, \sigma)$.*
3. *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 $\sigma_1 \bowtie \sigma_2$.

As a consequence, by Property 1, (σ_1, σ_2) belongs to $\Sigma(N \otimes O)$, which means that the original behavior σ_1 is preserved in the composed system.

Several remarks about the lemma:

- Condition 1 is standard when synchronizing Time Petri Nets.
- Condition 2 means that the observer must always be able to accept a transition l, performing 0-delay internal transitions if necessary (hence $\Delta(\sigma) = 0$ and the requirement that σ does not contain transitions in T_{sync}).
- Condition 3 means that the observer must always be able to let time elapse significantly (that is, at least δ units of time), performing internal transitions if necessary.

Proof sketch. Given a trace σ_1 in $\Sigma(N)$, we build a trace σ_2 in $\Sigma(O)$ that is composable with σ_1. The main difficulty concerns time elapses. Indeed, some time delays $d(\delta')$ in σ_1 are not directly synchronisable with O if δ' is too long (intuitively, the observer requests an interruption), so that $d(\delta')$ must be split in two parts $d(\delta_1)$ and $d(\delta' - \delta_1)$, where δ_1 is the delay until the observer's interruption. Additionally, one must show that the observer does not introduce an infinite number of interruptions within a finite time interval (that is, σ_2 is well-formed). As for untimed events, they are easily synchronized, introducing dumb events $\delta(0)$ where necessary. □

The conditions in Lemma 1 are true for the leadsto observer defined in Fig. 3. Therefore this observer cannot interfere with the system under observation. Next, we prove that the transition observer is sound, meaning that it reports correctly if its associated pattern is valid or not. We prove the soundness of this observer by showing that, for any TTS N, the event error does not appear in the traces of $N \otimes O$ if and only the pattern is valid for N. We write $\mathsf{error} \in N \otimes O$ to mean there exists a trace (σ, σ') in $\Sigma(N \otimes O)$ such that $\mathsf{error} \in \sigma'$.

Theorem 1. *We have* $\mathsf{error} \notin N \otimes O$ *if and only if, for all* $\sigma \in \Sigma(N)$ *such that* $\sigma = \sigma_1.E_1.\sigma_2$*, there exist* σ_3 *and* σ_4 *with* $\sigma_2 = \sigma_3.E_2.\sigma_4$ *and* $\Delta(\sigma_3) \leqslant max$*.*

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

Property 2. If there exist σ_1^a, σ_1^b, and σ_1^c such that $\sigma_1 = \sigma_1^a E_1 \sigma_1^b \sigma_1^c \wedge \Delta(\sigma_1^b) \geqslant max \wedge E_2 \notin \sigma_1^b$, then $\mathsf{error} \in \sigma_2$.

Property 3. If $\mathsf{error} \in \sigma_2$, then there exist σ_1^a, σ_1^b, and σ_1^c such that $\sigma_1 = \sigma_1^a E_1 \sigma_1^b \sigma_1^c \wedge \Delta(\sigma_1^b) \geqslant max \wedge E_2 \notin \sigma_1^b$.

5 Experimental Results

Our verification framework has been integrated into a prototype extension of *frac*, the Fiacre compiler for the TINA toolbox. We provide an extension of the *frac* compiler that supports the addition of real-time patterns and automatically compose a system with the necessary observers[1]. 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 of the observed system. For a system S, we define *size*(S) as the size (in bytes) of the state class graph of S generated by our verification tools. Hence *size*(S) is a good indicator of the memory footprint needed

[1] Available at `http://homepages.laas.fr/~nabid`.

for model-checking the system S. 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. In our verification tools, we use *state class graphs* [7] (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. Building on this definition, we say that the complexity of an observer O is the function C_O such that $C_O(S)$ is the quotient between the size of $(S \otimes O)$ and the size of S. This notion of complexity is useful to explain the impact of an observer during verification if we assume that model-checking a property on a system is performed in a time proportional to the size of its state class graph.

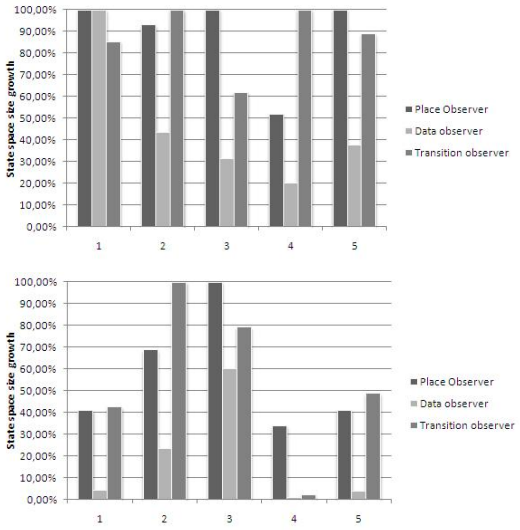


Fig. 5. Complexity for the three observer classes in percentage of system size growth—average time for invalid properties (above) and valid properties (below).

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. [14].

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. 5. We used classical examples of timed systems in these experiments: (1) a train-gate model; (2) a self-stabilizing protocol on a token ring; (3) a production cell factory; (4) a robotic control system; and (5) the alternating bit protocol. We have consistently observed that the observer based on data is the most efficient and that it seldom achieves the worst-case complexity. Actually, 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 with models of timed system.)

6 Related Work, Contributions and Perspectives

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

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 [18]: 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. [11], 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 pattern 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, the correctness of their observers remains to be proved, and the integration of their approach inside a global toolchain is lacking. 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, Toussaint et al. [23] propose a similar verification technique 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. In particular, we are not restricted to reachability properties and are able to prove liveness properties. 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 framework for proving the correctness of observers. We also introduce experimental results. Our approach is integrated in a complete verification toolchain for the Fiacre modeling language and can therefore been used in conjunction with Topcased. We have already used this tooling in a verification toolchain for a timed extension of BPMN [15]. Another contribution 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, D. Le Botlan. Definition of the Fiacre Real-Time Specification Patterns Language. Quarteft Project deliverable T2-12-B, 2011.
2. L. Aceto, A. Burgueño, K.G. Larsen. Model Checking via Reachability Testing for Timed Automata. *Int. Conf. on Tools and Alg. for the Constr. and Analysis of Systems (TACAS)*, LNCS 1384, 1998.
3. L. Aceto, P. Boyer, A. Burgueño, K.G. Larsen. The Power of Reachability Testing for Timed Automata. *Theoretical Computer Science*, 300, 2003.

4. G. Behrmann, R. David, K. Larsen. A tutorial on Uppaal. Springer, 2004.
5. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat. Fiacre: an intermediate language for model verification in the TOPCASED environment. *4th European Congress Embedded Real Time Software*, 2008.
6. B. Berthomieu, J.-P. Bodeveix, M. Filali, H. Garavel, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, F. Vernadat. The Syntax and Semantics of FIACRE – version 2.0. LAAS Research Report 07264, 2009.
7. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *Int. Journal of Production Research*, 42(14), 2004.
8. B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, F. Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. *Ada-Europe*, 2009.
9. M. Bozga, V. Sfyrla, J. Sifakis. Modeling synchronous systems in BIP. *ACM Int. Conf. on Embedded software*, 2009.
10. L.K. Dillon, L.E. Moser, P.M. Melliar-Smith, Y.S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology*, 3(2), 1994.
11. M.B. Dwyer, G.S. Avrunin, J.C. Corbett. Patterns in Property Specifications for Finite-State Verification. *Int. Conf. on Software Engineering*, IEEE, 1999.
12. P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, M. Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. *European Congress Embedded Real Time Software*, 2006.
13. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. *Int. Conf. on Tools and Alg. for the Constr. and Analysis of Systems (TACAS)*, LNCS 6605, 2011.
14. P. Gastin, D. Oddoux. Fast LTL to Büchi Automata translation. *Int. Conf. on Computer Aided Verification (CAV)*, 2001.
15. N. Guermouche, S. Dal Zilio. Real-Time Requirement Analysis of Services. *to appear*, hal-00578436, 2011.
16. V. Gruhn, R. Laue. Patterns for Timed Property Specifications. *international Conf. on Software engineering*, 2006.
17. T. Henzinger, Z. Manna, A. Pnueli. Timed Transition Systems. *Real Time : Theory in Practice*, LNCS 600, 1992.
18. T. Henzinger. It's about time: Real-time logics reviewed. *Int. Conf. on Concurrency Theory (CONCUR)*, LNCS 1466, 1998.
19. S. Konrad, B.H.C. Cheng. Real-time Specification Patterns. *Workshop on Quantitative Aspects of Programming Languages*, 2005.
20. O. Maler, D. Nickovic, A. Pnueli. From MITL to Timed Automata. *Int. Conf. on Formal modeling and Analysis of Timed Systems*, LNCS 4202, 2006.
21. P. M. Merlin. A study of the recoverability of computing systems. *PhD thesis, Dept. of Inf. and Comp. Sci., Univ. of California*, Irvine, CA, 1974.
22. A. Raji, P. Dhaussy, B. Aizier. Automating Context Description for Software Formal Verification. *Workshop MoDeVVa*, 2010.
23. J. Toussaint, F. Simonot-Lion, J.P. Thomesse. Time Constraints Verification Methods Based on Time Petri Nets. *IEEE Workshop on Future Trends of Distributed Computing Systems*, 1997.

A Proofs

Proof of Lemma 1

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 carefull with respect to well-formedness conditions (checking in particular that the algorithm does not pile up infinite sequences of zero-delay events).

Algorithm $f(\sigma_1, \sigma_1^a, \sigma_2^a, \sigma_3^a)$: let x and 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.

■

Proof of Property 2

First, by Definition 4 in section 3.2 and based on the fact that $\sigma_1 \bowtie \sigma_2$, $E_1 \in \sigma_1 \implies E_1 \in \sigma_2$ and $\exists \sigma_2^b$ a prefix in σ_2 such that $\sigma_1^b \bowtie \sigma_2^b$. By hypothesis, $\Delta(\sigma_1^b) \geqslant max \wedge E_2 \notin \sigma_1^b \implies \Delta(\sigma_2^b) \geqslant max \wedge E_2 \notin \sigma_2^b . \Delta(\sigma_2^b) \geqslant dtc(\mathsf{error})$ implies that place error is enabled in σ_2^b. We conclude that $\mathsf{error} \in \sigma_2$. ■

Proof of Property 3

σ_2 is an execution trace such that $\sigma_2 = \sigma_2^a \sigma_2^b \sigma_2^c$.
$\mathsf{error} \in \sigma_2$ means that error is enabled and $dtc(\mathsf{error}) \leqslant max$, we deduce that $\mathsf{error} \in \sigma_2^c$. Based on the fact that error is executed in σ_2^c, the delay of σ_2^b is greater than max, which implies that $E_2 \notin \sigma_2^b$ and $E_1 \in \sigma_2$. The σ_2 can be defined as $\sigma_2 = \sigma_2^a E_1 \sigma_2^b \sigma_2^c$. We have $\sigma_1 \bowtie \sigma_2$ implies that $\sigma_1 = \sigma_1^a E_1 \sigma_1^b \sigma_1^c$. ■