A Real-Time Specification Patterns Language

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan

A Real-Time Specification Patterns Language

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 propose a real-time extension to the patterns specification language of Dwyer et al. Our contributions are twofold.
First, we provide a formal patterns specification language that is simple enough to ease the specification of requirements by non-experts and rich enough to express general temporal constraints commonly found in reactive systems, such as compliance to deadlines, bounds on the worst-case execution time, etc. For each pattern, we give a precise definition based on three different formalisms: a denotational interpretation based on first-order formulas over timed traces; a definition based on a non-ambiguous, graphical notation; and a logic-based definition based on a translation into a real-time temporal logic.
Our second contribution is a framework for the model-based verification of timed patterns. Our approach makes use of new kind of observers in order to reduce the verification of timed patterns to the verification of Linear Temporal Logic formulas. This framework has been integrated in a verification toolchain for Fiacre, a formal modeling language for real-time systems.

1 Introduction

We define a formal requirements specification language aimed at the verification of reactive systems with real-time constraints. A first objective, in the design of this language, is to propose an alternative to timed extensions of temporal logic when model-checking real-time systems. While the language is rich enough to express general temporal constraints commonly found in the analysis of real-time systems (such as compliance to deadlines; bounds on the worst-case execution time; etc.), it is also designed to be simple in terms of both clarity and computational complexity. Indeed, we designed the language to be intuitive and easy to grasp—to ease the communication between the requirements analysis and system validation phases—as well as easy to check—because we provide a simple verification approach based on the use of observers and on-the-fly model-checking techniques.

Our language can be viewed as a real-time extension of the specification patterns language of Dwyer et al. [10]. Like in this seminal work, we define a list of patterns classified into main categories, such as *existence patterns*, used to express that some configuration of events must happen, or *response patterns*, that deals with causality between events. A main contribution of our work is to take into account timing constraints, which are not expressible with simpler pattern languages. For instance, we can express constraints on the time between two events, or on the duration a given condition is met.

For each pattern in our language, we give a precise definition based on three different formalisms: (1) a logical interpretation based on a translation into a real-time temporal logic (we use Metric Temporal Logic (MTL) [14] in this work); (2) a definition based on a graphical, non-ambiguous notation called Timed Graphical Interval Logic (TGIL); and (3) a denotational interpretation based on First-Order formulas over Timed Traces (FOTT) that is our reference definition. Another contribution is to provide a *reference implementation* for our language, that is a toolchain for checking whether a (model of a) system satisfies a given requirement in our language. Our patterns language has been integrated into Fiacre [5], a formal modeling language for real-time systems (`http://homepages.laas.fr/nabid/pfrac.html`). Fiacre is the intermediate language used for model verification in Topcased [11], an Eclipse based toolkit for critical systems, where it is used as the target of model transformation engines for various high-level modeling languages, such as SDL, BPEL or AADL [4]. In this context, the benefit of a high-level specification language becomes apparent, since it facilitates the interpretation of requirements between different formalisms. Indeed, patterns can be used as an intermediate format in the translation from high-level requirements (expressed on the high-level models) to the low-level formalisms understood by model-checkers.

In addition to the definition of a real-time patterns language, we present a verification method based on model-checking. This approach makes use of new kind of observers for Time Transition System models (Sect. 2.3) based on data in order to reduce the verification of timed patterns to the verification of Linear Temporal Logic (LTL) formulas. In this context, the second contribution of our paper is the definition of a set of *observers*—one for each pattern—that can be used for the model-based verification of timed patterns. This paper is devoted to the definition of the patterns language and its semantics. In another work [1], we describe more thoroughly the semantics of timed traces and study the experimental complexity of our model-checking approach based on several verification benchmarks. Another contribution made in [1] 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.

Before concluding, we review some works related to specification languages for reactive systems. We can list some distinguishing features of our approach. Most of these works focus on the definition of the patterns language (and generally relies on an interpretation using only one formalism) and leave out the problem of verifying properties. At the opposite, we provide different formalisms to reason about patterns and propose a pragmatic technique for their verification. When compared with verification based on timed temporal logic, the choice of a patterns language also 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.

2 Technical Background

We give a brief overview of the formal background needed for the definition of patterns. More information on TTS, our modeling and verification models, and on the semantics of timed traces can be found in [1].

2.1 Metric Temporal Logic

Metric Temporal Logic (MTL) [14] is an extension of Linear Temporal Logic (LTL) where temporal modalities can be constrained by an interval of the extended (positive) real line. For instance, the MTL formula $A\ \mathbf{U}_{[1,3[}\ B$ states that the event B must eventually occur, at a time $t_0 \in [1, 3[$, and that A should hold in the interval $[0, t_0[$. In the following, we will also use a weak version of the "until modality", denoted $A\ \mathbf{W}\ B$, that does not require B to eventually occur.

In our work, we consider a dense-time model and a *point based* semantics, meaning that the system semantics is viewed as a set of (possibly countably infinite) sequence of timed events. We refer the reader to [17] for a presentation of the logic and a discussion on the decidability of various fragments.

An advantage of using MTL is that it provides a sound and non-ambiguous framework for defining the meaning of patterns. Nonetheless, this partially defeats one of the original goal of patterns; to circumvent the use of temporal logic in the first place. For this reason, we propose alternative ways for defining the semantics of patterns that may ease engineers getting started with this approach. At least, our experience shows that being able to confront different definitions for the same pattern, using different formalisms, is useful for teaching patterns.

2.2 Timed Graphical Interval Logic

We define the Timed Graphical Interval Language (TGIL), a non-ambiguous graphical notation for defining the meaning of patterns. TGIL can be viewed as a real-time extension of the graphical language of Dillon et al. [8]. A TGIL specification is a diagram that reads from top to bottom and from left to right. The notation is based on three main concepts: *contexts*, for defining time intervals; *searches*, that define instants matching a given constraint in a context; and *formulas*, for defining satisfaction criteria attached to the TGIL specification.

In TGIL, a *context* is shown as a graphical time line, like for instance with the bare context $\vdash\!\!\!\longrightarrow$, that corresponds to the time interval $[0; \infty[$.

A *search* is displayed as a point in a context. The simplest example of search is to match the first instant where a given predicate is true. This is defined using the *weak search* operator, - - - - -$_{\text{A}}$▶ which represents the first occurrence of predicate A, if any. In our case, A can be the name of an event, a condition on the value of a variable, or any boolean conditions of the two. If no occurrence

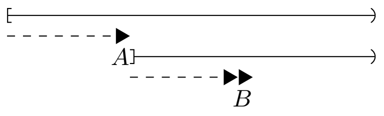


Fig. 1. An example of TGIL diagram.

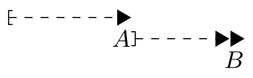


Fig. 2. A shorter notation for the diagram in Fig. 1.

of the predicate can be found on the context of a weak search, we say that the TGIL specification is valid. We also define a *strong search* operator, - - - -$_A$▶▶, that requires the occurrence of the predicate or else the specification fails.

A search can be combined with a context in order to define a sub-context. For instance, if we look at the diagram in Fig. 1, we build a context $[t_0; \infty[$, where t_0 is the time of the first occurrence of A in the context $[0; \infty[$ (also written A $\vdash\!\!\longrightarrow$) and, from this context, find the first occurrence of B. In this case, we say that the TGIL specification is valid for every *execution trace* such that either A does not occur or, if A does occur, then it must eventually be followed by B. We say that the specification is valid for a system, if it is valid on every execution trace of this system. For concision, we omit intermediate contexts when they can be inferred from the diagram. For example, the diagram in Fig. 2 is a shorter notation for the TGIL specification of Fig. 1.

We already introduced a notion of validity for TGIL with the definition of searches. Furthermore, we can define TGIL *formulas* from searches and contexts using the three operators depicted in Fig. 3. In the first diagram, the triangle below the predicate $\neg A$ is used to require that B must hold at the instant specified by the search (if any). More formally, we say that this TGIL specification is valid for systems such that, in every execution trace, either A always holds or B holds at the first point where the predicate $\neg A$ is true. We see that the validity of this diagram (for a given system) is equivalent to the validity of the MTL formula $A \mathbf{W} B$.

The other two formula operators apply to a context and a formula. The diamond operator, $\Diamond$, is used to express that a formula is valid somewhere in the context (and at the instant materialized by the diamond). In the case of the diagram in Fig. 3, we say that the specification is valid if the predicate A is eventually true. Similarly, the square operator, $\Box$, is used for expressing a constraint on all the instants of a context. Finally, we can also combine formulas using standard logical operators and group the component of a sub-diagram using dotted boxes (in the same way that parenthesis are used to avoid any ambiguity in the absence of a clear precedence between symbols).

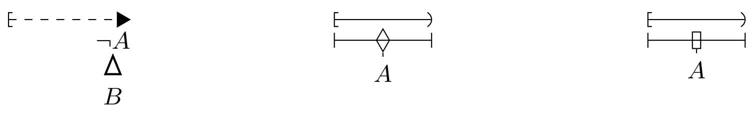


Fig. 3. Formulas operators in TGIL.

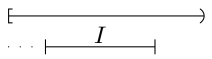


Fig. 4. Interval context

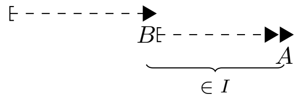


Fig. 5. Time constrained search

The final element in TGIL—that actually sets it apart from the Graphical Interval Logic of Dillon et al. [8]—is the presence of two real-time operators. The first operator (see Fig. 4) builds a context $[t_0{+}a; t_0{+}b]$ from the interval $I = [a; b]$ and an initial context of the form $[t_0; t_1]$. We also assume that $t_0 + b \leq t_1$. The second operator, represented by a curly bracket, is used to declare a delay constraint between two instants in a context, or two searches. We use it in the diagram of Fig. 5 to state that the first A after B should follow the first B after a delay t that is in the time interval I.

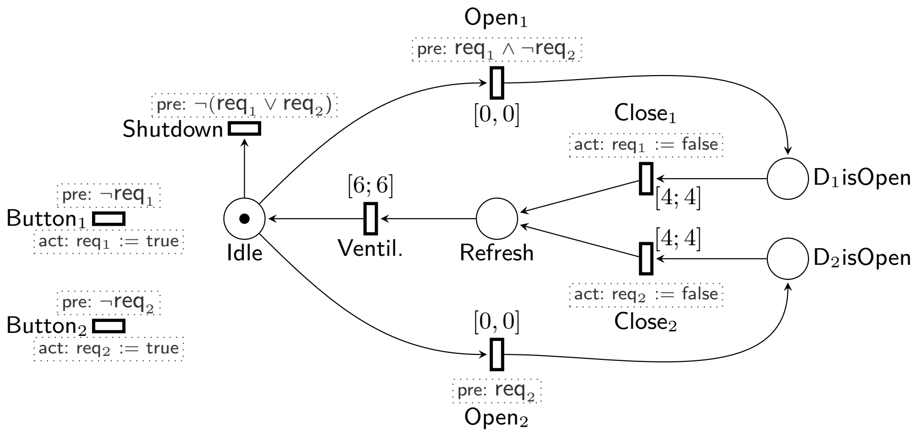
Another real-time extension of the Graphical Interval Logic, called RTGIL, has been proposed by Moser et al. in [16]. In comparison, TGIL is more expressive since it provides the notion of grouping and time constrained search (see Fig.5). As a consequence, patterns like 'Present first A before B within $[d1; d2]$' (see Sect. 3) can be expressed in TGIL, but not in RTGIL.

2.3 Time Transition Systems and Timed Traces

In this section, we describe the formal framework—called Time Transition System (or TTS)—used for modeling systems and for "implementing" observers. The semantics of TTS is expressed as a set of timed traces.

Time Transition Systems (TTS) are a generalization of Time Petri Nets [15] with priorities and data variables. Fig. 6 depicts a TTS example, using a graphical notation similar to Petri Nets. It proposes a simple model for an airlock, which consists in two doors (D_1 and D_2) and two buttons. At any time, at most one door can be open. Additionally, an open door is automatically closed after exactly 4 units of time (u.t.), followed by a ventilation procedure that lasts 6 u.t. Moreover, requests to open the door D_2 have higher priority than requests to open door D_1. A shutdown command can be triggered if no request is pending.

At first, the reader may ignore side conditions and side effects (the pre and act expressions inside dotted rectangles), considering the above diagram as a standard Time Petri Net, where circles are places and rectangles are transitions. Time intervals, such as $[4; 4]$, indicate that the corresponding transition must be fired if it has been enabled for exactly 4 units of time. A transition is enabled if there are enough tokens in its input places. Similarly, a transition associated to the time interval $[0; 0]$ must fire as soon as its pre-condition is met. The model includes two boolean variables, req_1 and req_2, indicating whether a request to open door D_1 (resp. D_2) is pending. Those variables are read by pre-conditions on transitions Open_i, Button_i (for i in 1..2), and $\mathsf{Shutdown}$. They are modified by post-actions on Button_i and Close_i. For instance, the pre-condition $\neg\mathsf{req}_2$ on



Open$_1$
pre: $\mathsf{req}_1 \wedge \neg\mathsf{req}_2$
$[0,0]$
pre: $\neg(\mathsf{req}_1 \vee \mathsf{req}_2)$
Shutdown
Close$_1$
act: $\mathsf{req}_1 := \mathsf{false}$
D$_1$isOpen
pre: $\neg\mathsf{req}_1$
Button$_1$
act: $\mathsf{req}_1 := \mathsf{true}$
$[6;6]$
$[4;4]$
Idle
Ventil.
Refresh
$[4;4]$
D$_2$isOpen
act: $\mathsf{req}_2 := \mathsf{false}$
Close$_2$
pre: $\neg\mathsf{req}_2$
Button$_2$
act: $\mathsf{req}_2 := \mathsf{true}$
$[0,0]$
pre: req_2
Open$_2$

Fig. 6. A TTS example of an airlock system

Button$_2$ is used to forbid this transition to be fired when the door is already open. It implies in particular that pressing the button while the door is open has no further effect. For the purpose of this work, we only need to define some notations. Like with Petri Net, the state of a TTS depends on its marking, m, that is the number of tokens in each place (we write $\mathcal{M}$ the set of markings). Since we also manipulate values, the state of a TTS also depends on a *store*, that is a mapping from variable names to their respective values. We use the symbol s for a store and write $\mathcal{S}$ for the set of stores. Finally, we use the symbol t for a transition and T for the set of transitions of a TTS.

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 req_2 is never true when $D_2isOpen$ is marked (using an accessibility check) (using an observer, as described in Sec. 3). As a consequence, traces must contain information about fired transitions (e.g. $Open_1$), markings (e.g. $m(Refresh)$), store (e.g. current value of req_1), 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 1 (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 2 (Duration). *Given a finite trace σ, we define its* duration*, $\Delta(\sigma)$, using the following inductive rules:*

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

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 3 (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 4 (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 [15]. 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 $\Sigma(N)$.

Observers as a special kind of TTS. In the next Section, we define observers at the TTS level that are used for the verification of patterns. We make use of the whole expressiveness of the TTS model: synchronous rendez-vous (through 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. 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 (see Sec 3) 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, ...). We have compared also the total verification time. It refers to the time spended generating the complete state space of the system and verifying the property but we will present, in this paper, only the complexity result. More details about the different kinds of observers and the time results are available in [1].

Our experimental results have shown that observers based on data modifications appear to be more efficient in practice and this is the class of observers that we present in this paper (for each pattern, we have tested several possible implementations before selecting the best candidate). The idea is to use shared boolean variables that change values when observed events are fired.

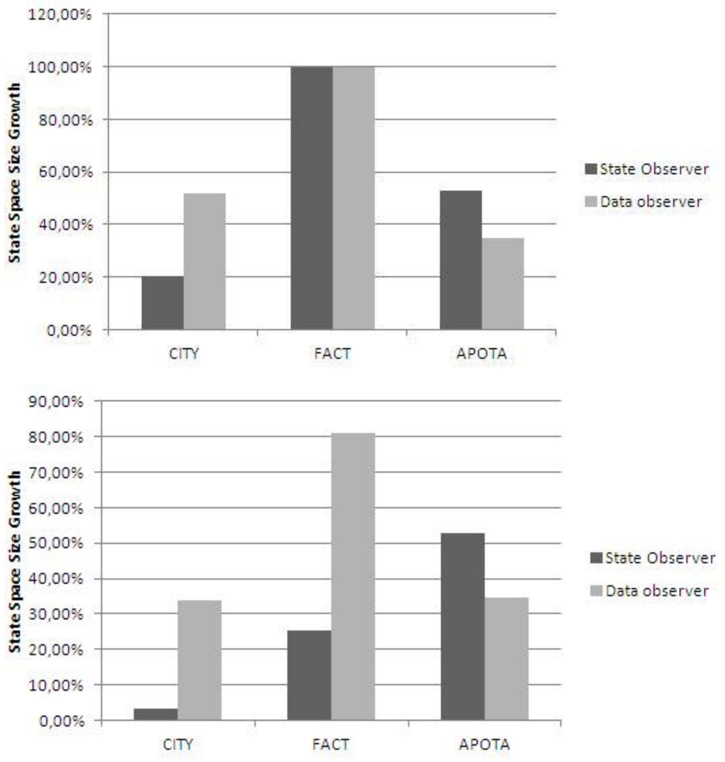


Fig. 7. Complexity for the data and state observer classes in percentage of system size growth—average time for invalid properties (above) and valid properties (below).

3 Catalog of Real-time Patterns

We introduce our real-time patterns language. Patterns may be refined using a scope modifier (before, after, etc.) that describes the context in which the pattern must hold. In the definition of patterns, we use the letters A and B to range over predicates on events $\omega \in \Omega$, that is first-order expressions on the markings of the considered system, on values of variables and on transitions. For instance, the expression $\mathsf{Open}_2 \vee \mathsf{req}_2$ is a predicate for the system shown in Fig. 6. In the definition of observers, a predicate A is interpreted as the set of transitions of the system that match A.

Due to the large number of possible alternatives, we restrict this catalog to the most important patterns. For each pattern, we give its denotational interpretation based on first-order formulas over timed traces (denoted FOTT in the following), a logical definition based on MTL, and a graphical definition (TGIL). We also provide the observer that should be combined with the system in order to check the validity of the pattern, as well as the corresponding LTL formula that has to be checked against the composed system. Some examples based on Fig. 6 are given (note that we give examples of both valid and invalid requirements).

Following the classification of Dwyer, we separate the description of the patterns into five main classes: *existence patterns* (Sect. 3.1), *absence patterns* (Sect. 3.2), *response patterns* (Sect. 3.3), *universality patterns* (Sect. 3.4) and *precedence patterns* (Sect. **??**). We also define examples of *composite patterns* (Sect. 3.5).

By convention, all boolean variables occuring in observers are initially set to false. A dashed arrow between two transitions, as between transitions E_1 and Error in the next observer, indicates that the former has priority over the latter: Error cannot occur while E_1 is fireable.

3.1 Existence patterns

An existence pattern is used to express that, in every traces of the system, some configuration of events must happen. We define some conventions used when defining observers for the patterns. In the following, Error and Start are transitions that belong to the observer, whereas E_1 (resp. E_2) represents all transitions of the system that match predicate A (resp. B). We also use the symbol I as a shorthand for the time interval $[d_1, d_2]$.

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

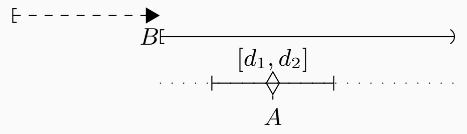
Predicate A must hold between d_1 and d_2 units of time (u.t) after the first occurrence of B. The pattern is also satisfied if B never holds.

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

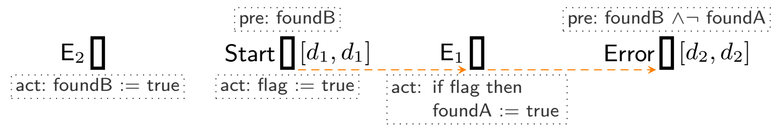
MTL def.: $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_I\ A)$

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

TGIL def.:



Observer:



The associated LTL formula is $[]\neg$Error.

Present first A **before** B **within** I

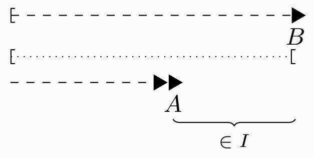
The first occurrence of predicate A holding is between d_1 and d_2 u.t. before the first occurrence of B. It also holds if B never holds. (The difference with the previous pattern is that we focus on the first occurrence of A before the first B.)

Example: **present first** Open$_1$ $\vee$ Open$_2$ **before** Ventil. **within** $[0, 10]$

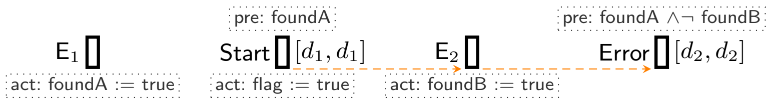
MTL def.: $(\Diamond B) \Rightarrow ((\neg A \wedge \neg B) \mathbf{U} (A \wedge \neg B \wedge (\neg B \mathbf{U}_I B)))$

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

TGIL def.:



Observer:



The associated LTL formula is $(\Diamond B) \Rightarrow \neg\Diamond(\text{Error} \vee (\text{foundB} \wedge \neg\text{flag}))$.

Present A lasting D

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

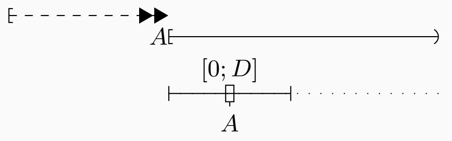
Comment: The pattern makes sense only if A is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration).

Example: **present** Refresh **lasting** 6

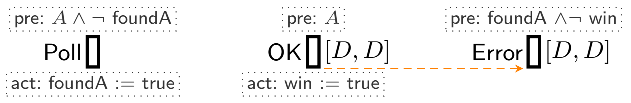
MTL def.: $(\neg A) \mathbf{U} (\Box_{[0,D]} A)$

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

TGIL def.:



Observer:



The associated LTL formula is $\Box\neg$Error.

Present A within I

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

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

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.

Present A after B until C within I

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

3.2 Absence patterns

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

Absent A after B for interval I

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

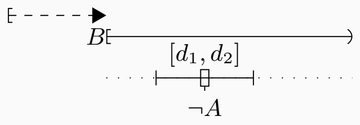
Comment: This pattern is dual to **Present** A **after** B **within** I (it is not equivalent to its negation because, in both patterns, B is not required to occur).

Example: **absent** Open$_1$ $\vee$ Open$_2$ **after** Close$_1$ $\vee$ Close$_2$ **for interval** $[0, 10]$

MTL def.: $\neg B \ \mathbf{W} \ (B \wedge \Box_I \neg A)$

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

TGIL def.:



Observer: We use the same observer as for **Present** A **after** B **within** I, but here Error is the expected behavior.
The associated LTL formula is $\Diamond B \Rightarrow \Diamond$Error.

Absent A before B for duration D

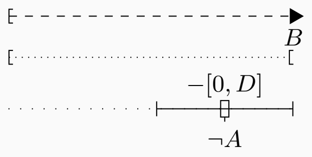
No A can occur less than D u.t. before the first occurrence of B. The pattern holds if there are no occurrence of B.

Example: **absent** Open$_1$ **before** Close$_1$ **for duration** 3

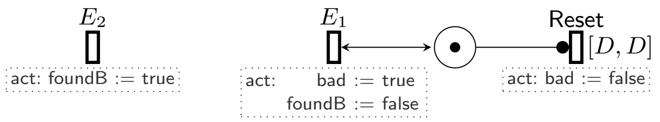
MTL def.: $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0,D]} \neg B)) \ \mathbf{U} \ B$

FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3, \omega \ . \ (\sigma = \sigma_1 \omega \sigma_2 B \sigma_3 \wedge B \notin \sigma_1 \omega \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow \neg A(\omega)$

TGIL def.:



Observer:



The associated LTL formula is $\neg\Diamond$(foundB $\wedge$ bad).

Absent A within I

This pattern is defined as absent A after init within I.

Absent A lasting D

This pattern is defined as absent A after init within I.

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

Absent A between B and C within I

This pattern is equivalent to the composition of two patterns :
absent A after B for interval $[d_1, d_2]$
and
absent A before C for duration $(d_2 - d_1)$.

Comment: We define I as the interval $[d_1, d_2]$.

3.3 Response patterns

Response patterns are used to express "cause–effect" relationship, such as the fact that an occurrence of a first kind of events must be followed by an occurrence of a second kind of events. Like in the previous patterns, we can use scopes and timing constraints to refine our requirements.

A leadsto first B within I

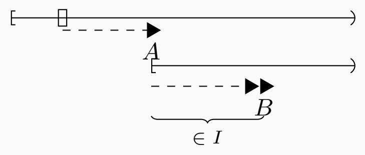
Every occurrence of A must be followed by an occurrence of B within a time interval I (considering only the first occurrence of B after A).

Example: Button_2 **leadsto first** Open_2 **within** $[0, 10]$

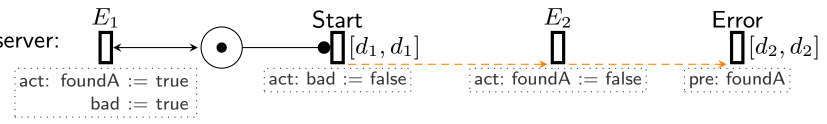
MTL def.: $\square(A \Rightarrow (\neg B)\ \mathbf{U}_I\ B)$

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

TGIL def.:



Observer:



The associated LTL formula is $\neg\Diamond(\text{Error} \vee (B \wedge \text{bad}))$.

A leadsto first B within I before R

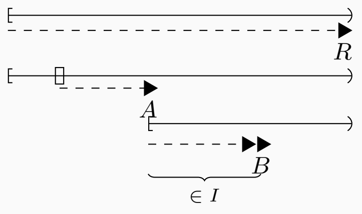
Before the first occurrence of R, each occurrence of A is followed by a B which occurs both before R, and in the time interval I after A. If R does not occur, the pattern holds.

Example: Button$_2$ **leadsto first** Open$_2$ **within** $[0, 10]$ **before** Shutdown

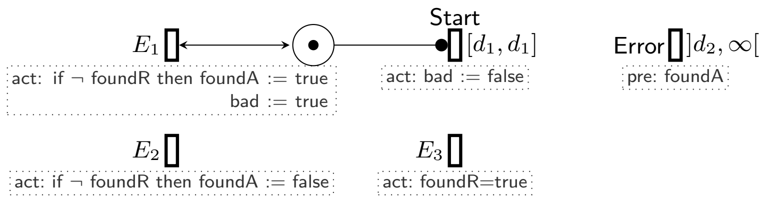
MTL def.: $\Diamond R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R)\ \mathbf{U}_I\ B \wedge \neg R)\ \mathbf{U}\ R$

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

TGIL def.:



Observer:



The associated LTL formula is $\Diamond R \Rightarrow \neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$.

$\boxed{A\ \textbf{leadsto first}\ B\ \textbf{within}\ I\ \textbf{after}\ R}$

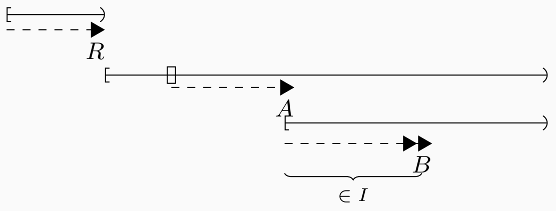
*Same than with the pattern "A **leadsto first** B **within** I" but only considering occurrences of A after the first R.*

Example: Button$_2$ **leadsto first** Open$_2$ **within** $[0, 10]$ **after** Shutdown

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

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

TGIL def.:



Observer: It is similar to the observer of the pattern A leadsto first B within I before R . We should just replace ¬foundR in transition E1 and E2 by foundR. The associated LTL formula is $\Diamond R \Rightarrow \neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$.

$\boxed{A\ \textbf{leadsto}\ B\ \textbf{between}\ Q\ \textbf{and}\ R\ \textbf{within}\ I}$

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

3.4 Universality patterns

Universality patterns are used to express that some condition should always occur.

always A **lasting** D

This pattern is defined as $\neg$*(absent A after init for interval I).*

always A **within** I

This pattern is defined as $\neg$*(absent A after init for interval I).*
Comment: *init* represents the first state of the system.

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

This pattern is defined as $\neg$*(absent A after B for interval* $[D, D]$*).*

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

This pattern is defined as $\neg$*(absent A before B for duration D).*

3.5 Composite Patterns

A main restriction of our language is that patterns cannot be nested. Nonetheless, patterns can be combined together using boolean operators. To check a composed pattern, we use a combination of the respective observers, as well as a combination of the respective LTL formulas.

For instance, if ϕ_1 and ϕ_2 are the LTL formulas corresponding to the observers for the patterns P_1 and P_2 respectively, then the composite pattern P_1 **and** P_2 is checked using the LTL formula $\phi_1 \wedge \phi_2$. Likewise, if we check the LTL formula $\phi_1 \Rightarrow \phi_2$ (implication), then we have a pattern P_1 **−o** P_2 that is valid when requirement P_2 holds for all the traces where P_1 holds (linear implication). Patterns can be combined using other connectives as well. Pnueli et al. [18] introduce the notion of tester to verify composite patterns. The difference with our work is that we are based on time event sequence however Pnueli et al. use the notion of signal. Moreover, we are able to define observer for punctual interval which is not the case for tester.

4 Case Study: Railcar System

We demonstrate the use of our specification patterns language to on a realistic use case and discuss the complexity of our approach. We take the example of an automated railcar system taken from [9]. The system is composed of four terminals connected by rail tracks in a cyclic network. Several railcars, operated from a central control center, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to the terminal to signal its arrival and, in the terminal, passengers can order a car using a button. This system has several real-time constraints: the terminal must be ready to accommodate an incoming car in 5 s; a car arriving in a terminal leaves its door open for exactly 10 s; passengers entering a car have 5 s to choose their destination; etc.

We have modeled this system using Fiacre (the model is available in the example section at `http://homepages.laas.fr/nabid/pfrac.html`). The key requirements of the railcar system are as follows:

- *P1 :* when a passenger arrives in a terminal, he must have a car ready to transport him within 15 s. This property can be expressed with a response pattern: Passenger/sendReq **leadsto** Control/ackTerm **within** [0,15], where sndReq is the state where the passenger chooses his destination and Control/ackTerm is the state where it is served.
- *P2 :* when the car starts moving, the door must be closed: **present** CarDoor/closeDoor **after** CarHandler/moving **within** [0,10]. This pattern states that when the car prepares for moving (it enters the state moving) we must see the event closeDoor within at most 10 s.
- *P3 :* when a passenger select a destination (in the car), the signal should stay illuminated until the car is arrived: **absent** Terminal/buttonOff **before** Control/ackTerm **for duration** 10, where Terminal/buttonOff is the state where the signal is turned off and Control/ackTerm is the state where the car reach its destination.

We have checked these requirements using our toolchain. The three properties *P1, P2* and *P3* are valid. This is the worst-case since it means that we need to explore the whole state space of the system combined with its observer.

We give some results on the time and space needed for model-checking these requirements. We use these results to give some indication on the complexity of checking timed patterns.

We are able to generate the complete state space of the railcar system in 309 ms, using 397.69 Kb of memory. This gives an upper-bound to the complexity of checking simple (untimed) reachability properties on the system, like for instance the absence of deadlocks. In our experiments, the complexity of checking property *P1* is almost the same than the complexity of exploring the complete system: the property is checked in 449 ms, using 780.21 Kb of memory. To give another comparison, this is also almost the same result than checking a similar, untimed variant of *P1* using the LTL model-checker provided in our toolbox [6] (i.e. with the formula $\Box\,\mathsf{Passenger/sendReq} \Rightarrow (\Diamond\,\mathsf{Control/ackTerm})$). Concerning the two remaining patterns, we are able to check *P2* in 440 ms and 787.17 Kb and *P3* in 538 ms and 840.39 Kb. In more general cases, we have often found that the complexity of checking timed patterns is in the same order of magnitude than checking their untimed temporal logic equivalent. An exception to this observation is when the temporal values used in the patterns are far different from those found in the system; for example if checking a periodic system, with a period in the order of the milliseconds, against a requirement using an interval in the order of the minutes. More results on the complexity of our approach can be found in [1].

5 Related Work and Contributions

Dwyer et al. have defined the original catalog of specification patterns [10], but in an untimed setting. They study the expressiveness of their approach and define patterns using different logical framework (LTL, CTL, Quantified Regular Expressions, etc.). This means that they do not need to consider the problem of checking requirements as they can rely on efficient model-checking algorithms. The patterns language is still supported, with several tools, an online repository of examples (`http://patterns.projects.cis.ksu.edu/`) and the definition of the Bandera Specification Language [7] that provides a structured-English language front-end for the specification of properties.

Some works consider the extension of patterns with timing constraints. Konrad et al. [13] extend the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL. Nonetheless, they do not consider the complexity of the verification problem (the implementability of their approach). Another related work is [12], where the authors define observers based on Timed Automata for each pattern. However, they have not integrated their language inside a toolchain or proved the correctness of their observers. (In [1], we define a framework that was used to prove the correctness of some of our observers.) We can also compare our approach with works concerned with observer-based techniques for the verification of real-time systems. Consider for example the work of Aceto et al. [3,2] based on the use of test automata to check properties on timed automata. In this framework, verification is limited to safety and bounded liveness properties since the authors focus on properties that can be reduced to reachability checking. In the context of Time Petri Net, Toussaint et al. [19] propose a verification technique similar to ours, but only consider four specific kinds of time constraints.

Compared to these related works, we make several contributions. We extend the specification patterns language of Dwyer et al. with real-time constraints. For each pattern, we give a precise definition based on different formalisms. We also address the problem of checking the validity of a pattern on a real-time system using model-based techniques: our verification approach is based on the use of observers, that are described in Sect. 2.3 and 3. Using this approach, 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 timed, liveness properties as well. While the use of observers for model-checking timed extensions of temporal logics is fairly common, our work 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. Concerning tooling, we offer an EMF-based meta-model for our specification language that allow its integration within a model-driven engineering development: our work is integrated in a complete verification toolchain for the Fiacre modeling language and can therefore be used in conjunction with Topcased [11], an Eclipse based toolkit for engineering critical system.

6 Conclusion and Perspectives

We define a high-level specification language for expressing requirements on real-time systems. Our approach appears to be quite efficient in practice. In particular, it eliminates the need to use model-checking algorithms for timed temporal logics, which may have a very high complexity. While we have concentrated our attention on model-checking, we believe our notation is interesting in its own right and can be reused in different contexts; following the same rationale than Dwyer et al. [10], we believe that our patterns may ease the adoption of formal verification techniques by non-experts.

There are several directions for future works. We plan to define a compositional patterns language inspired by the "denotational interpretation" used in the definition of patterns. The idea is to define a lower-level, easily extensible language, that is amenable to an automatic translation into observers (and therefore can dispose with the need to manually prove the correctness of our interpretation). In parallel, we plan to define a new modeling language for observers—adapted from the TTS framework—together with specific optimization techniques and easier soundness proofs. This language would be used as a new compilation target for our specification patterns language.

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. Verification of Real-Time Specification Patterns on Time Transitions Systems. Technical Report 11365, LAAS, 2011. `http://hal.archives-ouvertes.fr/hal-00593963/`.
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. 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.
5. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse, France, 2008.
6. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42-No 14, 2004.

7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *In Proceedings of the SPIN Software Model Checking Workshop, volume 1885 of LNCS*, pages 205–223. Springer, 2000.
8. L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3:131–165, 1994.
9. J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 52(1), 2008.
10. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99*, 1999.
11. 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.
12. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
13. 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.
14. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2:255–299, October 1990.
15. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974.
16. L. Moser, Y. S. Ramakrishna, G. Kutty, P. MELLIAR-SMITH, and L. K. Dillon. A graphical environment for design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6:31–79.
17. J. Ouaknine and J. Worrell. On the decidability and complexity of metric temporal logic over finite words. In *Logical Methods in Computer Science*, page 2007, 2007.
18. A. Pnueli and A. Zaks. On the merits of temporal testers. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer, 2008.
19. 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.