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 pattern specification language of Dwyer et al. Our contributions are twofold.
First, we provide a formal pattern 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 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 propose a real-time extension to the pattern specification language of Dwyer et al. [10]. This pattern language has been developed in the context of a verification toolchain for Fiacre [6], a formal modeling language for real-time systems. 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 from various languages, such as SDL, SPEM or AADL [7]. Fiacre is also an input language for two verification toolboxes—CADP and TINA, the TIme Petri Net Analyzer toolset [5]—that provides equivalence checking tools ; model-checkers for various temporal logics, such as LTL or the μ-calculus ; etc. While we take into account the timing aspects of the Fiacre language when exploring the state space of a model, none of these tools directly support the verification of timed requirements. For instance, we do not provide model-checkers for a timed extension of a temporal logic. The framework described in this paper is a way to solve this shortcoming.

We define a formal pattern 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 the analysis of reactive systems: compliance to deadlines, bounds on the worst-case execution time, etc.

As in the seminal work of Dwyer et al., our catalogue of patterns is partitioned into several categories, like *existence patterns*, used to express that some configuration of events must happen, or *order patterns*, that talks about the order in which multiple events must occur. For each class of patterns, 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) [15]) ; (2) a definition based on a graphical, non-ambiguous notation called TGIL ; and (3) a denotational interpretation based on first-order formulas over timed traces that is our reference definition. The primary goal of this property description language is to simplify the expression of time related requirements. While patterns enable novice users to read and write formal specifications for realistic systems, they also facilitate the conversion of specifications between 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 pattern language, we provide a verification method based on model-checking. This approach makes use of observers in order to reduce the verification of timed patterns to the verification of Linear Temporal Logic (LTL) formulas[1]. 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. In our approach, we model both the system and the observer using Time Transition Systems (TTS), a generalization of Time Petri Nets with priorities and data variables that is an internal format supported in our model-checking tools. The technical background needed for the definition of patterns and the semantics of TTS is briefly described in Sect. 2. More information on the modeling framework, the Fiacre language and the semantics of timed traces can be found in a companion work [2], where we detail our verification technique. Another contribution made in [2] 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. 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 that lead to a better time complexity in our verification benchmarks.

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 pattern 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 pattern

[1] For the patterns presented in this paper, we only use simple reachability properties, but are able to prove (unbounded) liveness properties.

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. Finally, our work has been integrated in a complete verification toolchain for the Fiacre modeling language and has already been used in different instances.

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 [2].

2.1 Metric Temporal Logic

Metric Temporal Logic (MTL) [15] 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 [18] 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. [9]. 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 $\longmapsto$, 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, - - - - -$_{A}$▶ 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 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 else the specification fails.

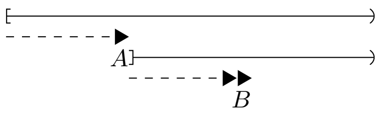


Fig. 1. An example of TGIL diagram.

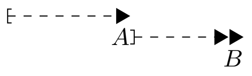


Fig. 2. A shorter notation for the diagram in Fig. 1.

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 $A \longmapsto$)) 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.

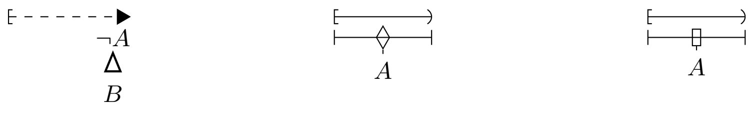


Fig. 3. Formulas operators in TGIL.

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, $\square$, is used for expressing a constraint on all the instants of a context. Finally, we can also combine formulae 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).

The final element in TGIL—that actually sets it apart from the Graphical Interval Logic of Dillon et al. [9]—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 B should follow the first A after a delay t that is in the time interval I.

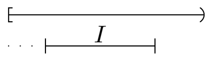


Fig. 4. Interval context

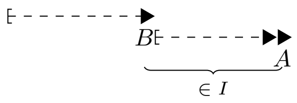


Fig. 5. Time constrained search

Another real-time extension of the Graphical Interval Logic, called RTGIL, has been proposed by Moser et al. in [17]. The most significant difference with this work lies in the way the semantics of diagrams is defined. Also, we use two real-time operators, whereas RTGIL only provide an equivalent of the "delay constraint" operator. In particular, RTGIL is not expressive enough to define all the patterns that are given in Sect. 3.

2.3 Time Transition Systems and Timed Traces

In this section, we describe the formal framework, the TTS, used for modeling the system and for "implementing" the observer associated to a pattern. The semantics of TTS is expressed as a set of timed traces.

Time Transition Systems (TTS) are a generalization of Time Petri Nets [16] with priorities and data variables. We illustrate the semantics of TTS using an example, using a graphical notation for TTS inspired by Petri Nets. In Fig. 6, we display a TTS that corresponds to a simple model for an airlock. The system 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.

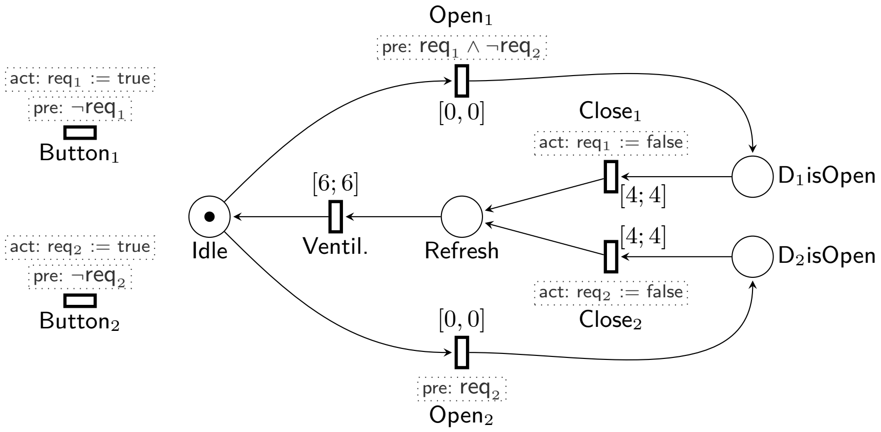


Fig. 6. A TTS example of an airlock system

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 the place connected to a transition. 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 and Button_i (for i in 1..2), and modified by post-actions on Button_i and Close_i. For instance, the pre-condition $\neg\mathsf{req}_2$ on 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.

A complete formal definition of TTS can be found in [2]. 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 token 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.

Expressing the Semantics of TTS with Timed Traces. A *trace* σ of a TTS is 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'.

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

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.

Definition 2 (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 happen at the same date (with no delay in between) while keeping a causality relation (one event is before another).

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. Our experimental results, see [2], 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.

3 Catalogue of Real-time Patterns

Following Dwyer et al. [10], we provide a comprehensive list of patterns partitioned into three main categories: *existence patterns* (used to express that some configuration of events must occur) ; *absence patterns* (the dual of existence) ;

and *response patterns* (that specify the order in which some events must occur). In each category, patterns may be refined using a scope modifier (before, after, etc.) that describes when the pattern must hold. We do not define a timed extension of *universality patterns*, since adding timing constraint does not make sense in this case. Due to the large number of possible alternatives, we restrict this catalogue to the more important patterns—those we used in real examples, during the modeling of industrial use cases—and for which we can exhibit a valid observer Furthermore, we restrict ourselves to patterns with a decidable verification procedure.

Notation In the following, we usually write A and B for events. As a first approximation, one may consider that events equates transitions so that, for instance, when referring to a timed trace σ, "after A" designates the part of σ located after the first occurrence of transition A. This may be written, $\sigma = \sigma_1 A \sigma_2 \wedge A \notin \sigma_1$ using our notation. More generally, we can consider predicates on events, that is function from Ω to booleans. Such predicates can be used to refer to the occurrence of a given transition, but also to state conditions on the values of the variables and the marking. For example, A could be the predicate $\mathsf{req}_2 \wedge \neg \mathsf{D}_1\mathsf{isOpen}$. In this more general setting, were A is a predicate, we use $\sigma = \sigma_1 A \sigma_2$ as an abuse of notation for the relation $\sigma = \sigma_1 \omega \sigma_2 \wedge A(\omega)$. Similarly, $A \notin \sigma_1$ means that there is no ω in σ_1 such that $A(\omega)$ holds.

3.1 Existence patterns

Existence patterns are used to describe that we can find a given desired property in every possible run of a system. We use the first pattern to describe our approach in more details.

Present A after B within $[d_1, d_2]$ — This pattern asserts that an event, say A, must occur between d_1 and d_2 units of time after an occurrence of the event B. The pattern is also satisfied if B never occurs.

Example of use: as an example, we can check that if a door is opened then the airlock is ventilated after at most 4 units of time: **present** Ventil. **after** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **within** $[0; 4]$.

Denotational definition: more formally, we say that the pattern is true for a system N if and only if, for every timed-trace σ of N, we have:

$$\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 A \sigma_4 \wedge d_1 \leqslant \Delta(\sigma_3) \leqslant d_2$$

Logical definition: although the denotational approach is very convenient for a tool developer since it is self-contained and only relies on the definition of timed traces, we can alternatively provide an equivalent definition based on a MTL formula: $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_{[d_1,d_2]}\ A)$.

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 time interval is not punctual (of the form $[d, d]$); we add a pattern in our catalogue as long as we can provide a suitable observer for it.

TGIL definition: equivalently, this pattern can be represented by the diagram on Fig. 5, that we used as an example of TGIL in Sect. 2, where I is $[d_1, d_2]$.

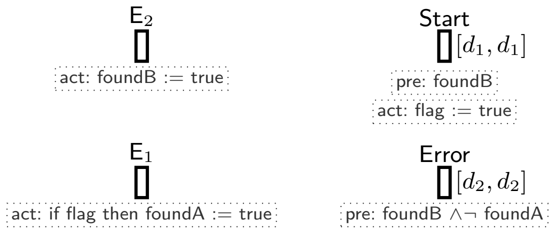
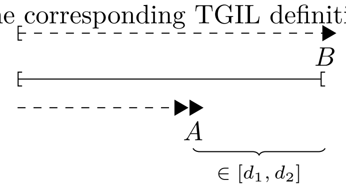


Fig. 7. Observer for the pattern **present** A **after** B **within** $[d_1; d_2]$.

Observer: the observer that is used to verify the compliance of a system to this pattern is given in Fig. 7, where Error and Start are transitions of the observer, while E$_1$ (resp. E$_2$) represents all transitions of the system that match condition A (resp. B). The observer uses three boolean variables foundA, foundB, and flag, whose initial values are set to false. Variables foundA and foundB are used to record if the events A and B have been met. Variable flag is true only in the time interval $[d_1; \infty[$ after the first occurrence of B. By adding adequate priorities between the transitions E$_1$, Start and Error (see [2] for a description of priorities), it is possible to derive an observer for the property **present** A **after** B **within** I where I is one of the time intervals $]d_1, d_2]$, or $]d_1, d_2[$, etc.

To check that a TTS satisfies this pattern, we check that the composition of the TTS with the observer satisfies the LTL formula $\Box\neg$Error, meaning that the observer never fires the transition Error.

Present first A before B within $[d_1, d_2]$ This pattern asserts that the first occurrence of A holds within $[d_1, d_2]$ u.t. before the first occurrence of B. It also holds if B does not occur. For instance, we might require that a door opened at most 4 u.t. before the first ventilation procedure: **present first** Open$_1$ ∨ Open$_2$ **before** Ventil. **within** $[0; 4]$. This pattern is defined in MTL by: $(\Diamond B) \Rightarrow (\,(\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_{[d_1,d_2]}\ B))\,)$. Its denotational definition over timed traces is $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in [d_1, d_2]$. The corresponding TGIL definition is the following:



We check this property by adapting the previous pattern. Indeed, if B occurs, then this pattern is equivalent to **present first** B **after** A **within** $[d_1, d_2]$, taking into account the first occurrence of B in the trace (not only the first occurrence of B after A, as in the previous pattern). As a consequence, we use the same observer as above (switching A and B), replacing the action in E_1 by foundB := true, and using the following LTL formula: $(\Diamond B) \Rightarrow \neg\Diamond($Error ∨ (foundB ∧ ¬flag)$)$.

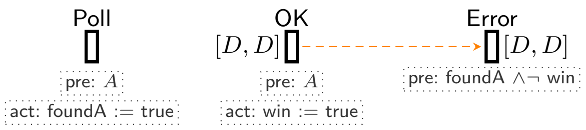
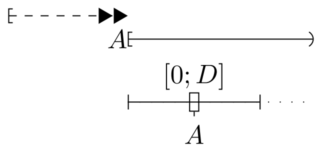


Fig. 8. Observer "**present** A **lasting** D"

Present A lasting D The goal of this pattern is to assert that from the first occurrence of A, the predicate A remains true for at least duration D. It makes sense only if A is a state predicate (that is, on the marking and store), and which does not refer to any transition (since transitions are instantaneous, we cannot require a transition to last for a given duration). We can verify for instance that the ventilation procedure lasts at least 6 u.t.: **present** Refresh **lasting** 6. The pattern is defined in MTL by: $(\neg A)\ \mathbf{U}\ (\Box_{[0,D]} A)$. It is defined over timed traces by: $\exists \sigma_1, \sigma_2, \sigma_3\ .\ \sigma = \sigma_1\sigma_2\sigma_3 \wedge A \notin \sigma_1 \wedge \Delta(\sigma_2) \geqslant D \wedge A(\sigma_2)$, where $A(\sigma_2)$ means that A holds on every event in σ_2. The corresponding TGIL definition is:



The corresponding observer is presented in Fig. 8. Initially, foundA and win are false. Transition Poll sets the former to true as soon as the predicate A holds. If A holds for at least D u.t., then OK will be fired (before Error, thanks to the priority relation, shown as a dotted arc) and win will be set to true. Otherwise, Error fires D u.t. after the first occurrence of A. Like in the previous cases, we check this pattern by verifying the LTL reachability property: $\Box\neg$Error.

Present A within I This pattern is equivalent to **present** A **after** init **within** I, where init is a special event that is always the first event in any timed trace.

3.2 Absence patterns

Absence patterns are used to specify delays within which activities must not occur.

Absent A after B for interval $[d_1, d_2]$ This pattern asserts that an event, say A, must not occur between d_1–d_2 u.t. after the first occurrence of an event B. For instance, we can check that when a door is open, no door will be open between 4 and 6 u.t. afterward: **absent** Open$_1$ $\vee$ Open$_2$ **after** Open$_1$ $\vee$ Open$_2$ **for interval** $]4, 6[$. The "denotational" definition is the following: for every timed-trace σ of N, the statement

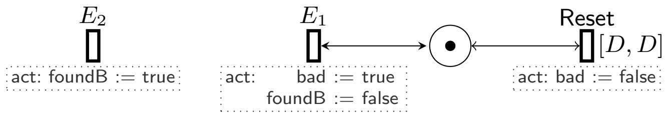
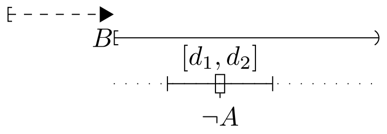


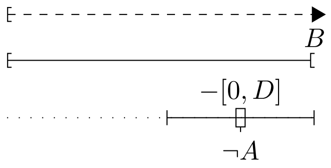
Fig. 9. Observer "**absent** A **before** B **for duration** D"

$\forall \sigma_1, \sigma_2, \sigma_3, \omega \;.\; (\sigma = \sigma_1 B \sigma_2 \omega \sigma_3 \wedge B \notin \sigma_1 \wedge \Delta(\sigma_2) \in [d_1, d_2]) \Rightarrow \neg A(\omega)$ must hold. It is equivalent to the MTL formula $\neg B \;\mathbf{W}\; (B \wedge \Box_{d_1,d_2} \neg A)$. Finally, we can explain the pattern with the next diagram:



We search the first occurrence of B and then, from the state located by the previous search, we verify that A is absent during the interval $[d_1, d_2]$. The observer we use is the one presented in Figure 7. Indeed, this pattern is dual to **Present** A **After** B **within** $[d_1, d_2]$ (but it is not strictly equivalent to its negation, because in both patterns, B is not required to occur). The model composed with the observer is checked against the LTL formula $\Diamond B \Rightarrow \Diamond \mathsf{Error}$.

Absent A before B for duration D This pattern asserts that no A can occur less than D u.t. before the first occurrence of B. For instance, if a door is closing, then it has opened more than 3 u.t. earlier: **absent** Open_1 **before** Close_1 **for duration** 3. Its MTL formula is: $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0;D]} \neg B)) \;\mathbf{U}\; B$. It is defined over timed traces by: $\forall \sigma_1, \sigma_2, \sigma_3 \;.\; (\sigma = \sigma_1 \sigma_2 B \sigma_3 \wedge B \notin \sigma_1 \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow A \notin \sigma_2$. The corresponding TGIL definition is:



The corresponding observer is depicted Figure 9. As usual, E_1 (resp. E_2) corresponds to all transitions that match predicate A (resp. B). The boolean variables foundB and bad are initially set to false. Variable bad is true after each occurrence of A for duration D. Therefore, the LTL formula we have to check is $\neg\Diamond(\mathsf{foundB} \wedge \mathsf{bad})$.

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

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

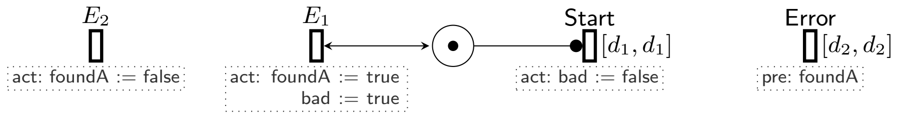


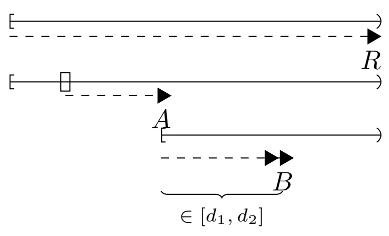
Fig. 10. Observer "A **leadsto first** B **within** $[d_1, d_2]$"

3.3 Response patterns

This category of patterns express a time constraint on a response.

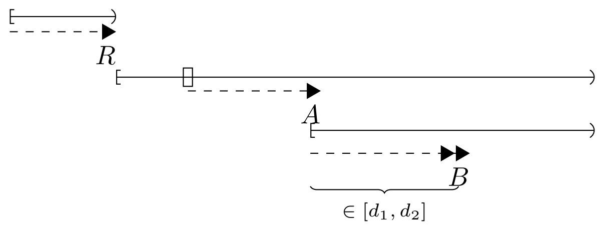
A leadsto first B within $[d_1, d_2]$ This pattern states that every occurrence of an event, say A, must be followed by an occurrence of B within a time interval $[d_1, d_2]$ (considering only the first occurrence of B after A). For instance, each time Button_2 occurs, door D_2 necessarily opens before 10 u.t.: Button_2 **leadsto first** Open_2 **within** $[0, 10]$ (notice that the same pattern with Button_1 and Open_1 does not hold because requests to open door D_2 are prior). Denotationally, we write $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in [d_1, d_2] \wedge B \notin \sigma_3$. It is equivalent to the MTL formula: $\Box(A \Rightarrow (\neg B)\ \mathbf{U}_{[d_1,d_2]}\ B)$. Graphically, this pattern is depicted Figure 5. The corresponding pattern is shown Figure 10. Both variables foundA and bad are initially set to false. The former indicates that no B has occurred since the last A occurred. The later indicates if d_1 u.t. have elapse since the last occurrence of A. The model composed with the observer is checked against the LTL formula $\neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$.

A leadsto first B within $[d_1, d_2]$ *before R* This pattern asserts that, before the first occurrence of R, each occurrence of A is followed by B, which occurs both before R, and in the time interval $[d_1, d_2]$ after A. If R does not occur, the pattern holds. As an example, we may check that before door D_2 opens, the opening of D_1 triggers ventilation in less than 5 u.t.: Open_1 **leadsto first** Ventil **within** $[0, 5]$ **before** Open_2. Note that we require A and R to be disjoint predicates, and so this pattern cannot be used to check that ventilation occurs between two occurrences of Open_1. This pattern is defined in MTL by: $\Diamond R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R)\ \mathbf{U}_{[d_1,d_2]}\ B \wedge \neg R)\ \mathbf{U}\ R$. It is defined over timed traces by: $\forall \sigma_1, \sigma_2, \sigma_3 \,.\, (\sigma = \sigma_1 A \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 A \sigma_2 \Rightarrow \exists \sigma_4, \sigma_5 \,.\, \sigma_2 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in [d_1, d_2] \wedge B \notin \sigma_4$. The corresponding TGIL definition is:



We check this pattern by using the observer already shown in Figure 10 in which the post-actions of E_1 and E_2 are encapsulated in a if statement: if $\neg$foundR then ..., and adding the post-action foundR := true to all transitions of the system that match R. The LTL formula to be verified becomes $\Diamond R \Rightarrow \neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$.

A leadsto first B within $[d_1, d_2]$ *after R* This pattern asserts that after the first occurrence of R, "A **leadsto first** B within $[d_1, d_2]$" holds. It is defined in MTL by: $\Box(R \Rightarrow (\Box(A \Rightarrow (\neg B)\ \mathbf{U}_{[d_1,d_2]}\ B)))$. It is defined over timed traces by: $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 R \sigma_2 A \sigma_3 \wedge R \notin \sigma_1) \Rightarrow \exists \sigma_4, \sigma_5\ .\ \sigma_3 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in [d_1, d_2] \wedge B \notin \sigma_4$. The corresponding TGIL definition is:



The observer used to verify this pattern is an obvious adaptation of the previous case, replacing $\neg$foundR by foundR.

To conclude with, we notice that patterns can be combined, for instance (**Absent** Button$_2$ **before** Button$_1$) $\wedge$ (**Absent** Button$_2$ **after** Button$_1$ **for interval** $[0, 6]$) $\Rightarrow$ Button$_1$ **leadsto first** Open$_1$ **within** $[0, 6]$ which verifies that Button$_1$ triggers the opening of door D_1 in less than 6 u.t. provided Button$_2$ has not been pressed before, neither in the following 6 u.t. This pattern consists in the combination of three simple patterns, and is verified by combining the necessary observer and using the combined LTL formula $\pi_1 \wedge \pi_2 \Rightarrow \pi_3$, where π_i are the LTL formula corresponding to the three simple patterns.

4 Related Work and Contributions

The original catalog of specification pattern is defined in [10], where Dwyer et al. study the expressiveness of their approach and define patterns using different logical framework: LTL, CTL, Quantified Regular Expressions, etc. The pattern 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 [8] that provides a structured-English language front-end for the specification of properties. Some previous works have considered the addition of time inside patterns. Konrad et al. [14] extend the pattern language with time constraints and give a mapping from timed pattern to TCTL and MTL. Nonetheless, they do not study the complexity of the verification method (the implementability of their approach). Another related work is [12], where Gruhn and Laue define observers based on Timed Automata for each pattern. However, the correctness of their observers remains to be proved, and the integration of their work inside a global toolchain is lacking.

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. [3,4] 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.

In this paper, we make the following contributions. We extend the specification patterns language of Dwyer et al. with timed properties. For each pattern, we give a precise definition based on three different formal formalisms. The complete list of patterns is given in [1]. 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 we describe in Sect. 2.3 and 3 of this paper. This way, 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. 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 based on the monitoring of data modifications that appears to be more efficient in practice. Concerning tooling, we offer an EMF-built meta-model for our specification language that allow its integration within a model-driven engineering development. Moreover, 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 critical systems. We have already used this tooling in a verification toolchain for a timed extension of BPMN [13].

5 Conclusion and Perspectives

We propose a high-level pattern language that allows system architects to express common real-time properties, such as response to a request in a bounded time or absence of some events in a given time interval. Following Dwyer et al's rationale, we believe that these patterns may ease the adoption of formal verification techniques by non-experts through the definition of a less complicated language than timed temporal logics. The approach also appears to be quite efficient in practice.

For future works, we plan to consider a lower-level compositional pattern language inspired by the "denotational interpretation" used in our definition of patterns. The benefits of such a language, we hope, would include, on the one hand, automatic translation into observers and, on the other hand, would be more expressive than our finite collection of patterns. Yet, it should be kept simple enough to ease the expression of common cases. In parallel, we want to define a new modeling language for observers—adapted from the TTS framework—equipped with more powerful optimization techniques and with 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, D. Le Botlan. Definition of the Fiacre Real-Time Specification Patterns Language. Quarteft Project deliverable T2-12-B, 2011.
2. N. Abid, S. Dal Zilio, D. Le Botlan. Verification of Real-Time Specification Patterns on Time Transitions Systems. *Submitted*, 2011.
3. 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.
4. L. Aceto, P. Boyer, A. Burgueño, K.G. Larsen. The Power of Reachability Testing for Timed Automata. *Theoretical Computer Science*, 300, 2003.
5. 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.
6. 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. *European Congress Embedded Real Time Software*, 2008.
7. 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.
8. J.C. Corbett, M.B. Dwyer, J.Hatcliff, Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. *Int. SPIN Work. on Model Checking of Software*, LNCS 1885, 2000.
9. 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.
10. M.B. Dwyer, G.S. Avrunin, J.C. Corbett . Patterns in Property Specifications for Finite-State Verification. *Int. Conf. on Software Engineering*, IEEE, 1999.
11. 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.
12. V. Gruhn, R. Laue. Patterns for Timed Property Specifications. *Int. Conf. on Software Engineering*, 2006.
13. N. Guermouche, S. Dal Zilio. Real-Time Requirement Analysis of Services. *to appear*, hal-00578436, 2011.
14. S. Konrad, B.H.C. Cheng. Real-time Specification Patterns. *Workshop on Quantitative Aspects of Programming Languages*, 2005.
15. R.Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*,2(4):255–299, 1990.
16. 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.

17. L.E. Moser, Y.S. Ramakrishana, G. Kutty, P.M. Melliar-Smith, L.K. Dillon. A Graphical Environnement for the Design of Concurrent Real-Time Systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, 1997.
18. J. Ouaknine, J. Worrell. On the Decidability of Metric Temporal Logic. *Symp. on Logic in Computer Science (LICS)*, IEEE, 2005.
19. 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.