A Real-Time Specification Patterns Language

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan

▶ To cite this version:

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan. A Real-Time Specification Patterns Language. 2011. hal-00593965v2

HAL Id: hal-00593965
https://hal.science/hal-00593965v2

Submitted on 5 Nov 2011 (v2), last revised 21 Dec 2011 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 propose a real-time specification patterns language based on the work of Dwyer et al. [10]. Our language 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. Dwyer's work presents a list of patterns specification for finite-state verification. These patterns are classified into 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. Dwyer's patterns are used in many works but they do not take in account real-time concepts.

In this paper, we extend the work of Dwyer et al. [10] to take in account real-time concepts. As in the seminal work of Dwyer et al., our catalog of patterns is partitioned into several categories. 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.

Our first contribution is a formal definition of the real-time patterns language. This patterns 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 provide equivalence checking tools ; model-checkers for various temporal logics, such as LTL or the μ-calculus ; etc. While we take in 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. The framework described in this paper is a way to solve this shortcoming as we provide a prototype implementation [21] for model checking properties expressed using real-time patterns on Fiacre models.

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 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. We use three categories of observers. 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 have proved more efficient in practice. This paper is devoted to the definition of the patterns language and its semantics. In another work [2], 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 [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.

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 verifi-

cation. 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. 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 $\vdash\!\!\longrightarrow$, 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, - - - - -▶ 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.

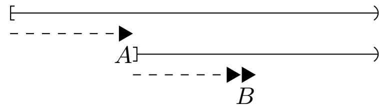

Fig. 1. An example of TGIL diagram.

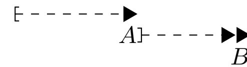

Fig. 2. A shorter notation for the diagram in Fig. 1.

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 ⊢——→) 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.

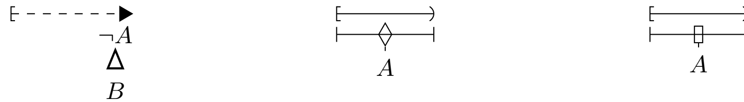

Fig. 3. Formulas operators in TGIL.

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, $\Box$, 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).

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.

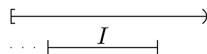

Fig. 4. Interval context

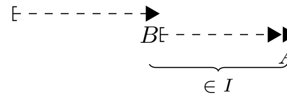

Fig. 5. Time constrained search

Another real-time extension of the Graphical Interval Logic, called RTGIL, has been proposed by Moser et al. [17]. We provide a comparison between the two logics in Sect.4. The most significant difference with this work lies in the way the semantics of diagrams is defined. 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—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 [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.

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

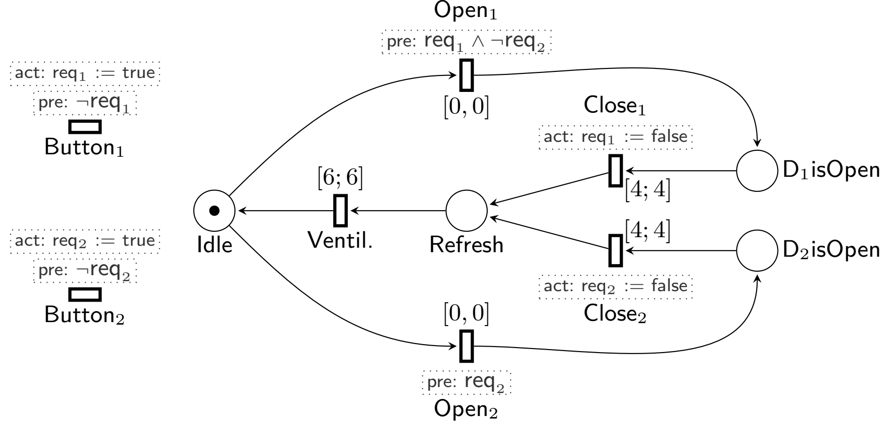

Fig. 6. A TTS example of an airlock system

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. 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:*

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

We extend Δ to infinite traces, by defining $\Delta(\sigma)$ as the limit of $\Delta(\sigma_i)$ where σ_i are growing prefixes of σ.

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

Definition 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 [16]. 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 $(m^{\mathrm{init}}, s^{\mathrm{init}}, \mathsf{tc}) \longrightarrow^{\sigma}$ 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. 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 Catalog of Real-time Patterns

In this section, we present our real-time patterns language (see Tables below). We add time to Dwyer's definition. Patterns may be refined using a scope modifier

(before, after, etc.) that describes when the pattern must hold. Due to the large number of possible alternatives, we restrict this catalog 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. For each pattern, we give a denotational interpretation based on first-order formulas over timed traces (FOTT), a logical definition (MITL), a graphical definition (TGIL) and the corresponding observer. We define also the LTL formula to verify and we give an example based on Figure 6 to illustrate the use of pattern in practice.

Table 1: Existence patterns

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

- **Textual Definition:** An event, say A, must occur between d_1 and d_2 units of time (u.t) after an occurrence of the event B. The pattern is also satisfied if B never occurs.
- **Example:** **present** Ventil. **after** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **within** $[0; 4]$

- **MITL Definition:** $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_{[d_1,d_2]}\ A)$
- **TGIL Definition:** see Fig. 16
- **FOTT Definition:** $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 A \sigma_4 \wedge d_1 \leqslant \Delta(\sigma_3) \leqslant d_2$

- **Observer:** see Fig. 7
- **LTL formula to verify:** $[]\neg\mathrm{Error}$

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

- **Textual Definition:** The first occurrence of A holds within $[d_1, d_2]$ u.t. before the first occurrence of B. It also holds if B does not occur.
- **Example:** **present first** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **before** Ventil. **within** $[0; 4]$

- **MITL Definition:** $(\Diamond B) \Rightarrow (\,(\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_{[d_1,d_2]}\ B))\,)$
- **TGIL Definition:** see Fig. 17
- **FOTT Definition:** $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4\ .\ \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in [d_1, d_2]$

- **Observer:** see Fig. 8
- **LTL formula to verify:** $(\Diamond B) \Rightarrow \neg \Diamond(\mathsf{Error} \vee (\mathsf{foundB} \wedge \neg \mathsf{flag}))$

3. Present A lasting D

- **Textual Definition:** The goal of this pattern is to assert that from the first occurrence of A, the predicate A remains true for at least duration D. It makes sense only if A is a state predicate (that is, on the marking and store), and which does not refer to any transition (since transitions are instantaneous, we cannot require a transition to last for a given duration).
- **Example:** **present** Refresh **lasting** 6

- **MITL Definition:** $(\neg A)\ \mathbf{U}\ (\Box_{[0,D]} A)$
- **TGIL Definition:** see Fig. 18
- **FOTT Definition:** $\exists \sigma_1, \sigma_2, \sigma_3\ .\ \sigma = \sigma_1 \sigma_2 \sigma_3 \wedge A \notin \sigma_1 \wedge \Delta(\sigma_2) \geqslant D \wedge A(\sigma_2)$

- **Observer:** see Fig. 9
- **LTL formula to verify:** $\Box \neg \mathsf{Error}$

4. Present A within I

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

5. Present A between B and C within I

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

6. Present A after B until C within I

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

Table 2: Absence patterns

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

- **Textual Definition:** This pattern asserts that an event, say A, must not occur between d_1–d_2 u.t. after the first occurrence of an event *this*. 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)
- **Example: absent** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **after** $\mathsf{Open}_1 \vee \mathsf{Open}_2$ **for interval** $]4, 6[$

- **MITL Definition:** $\neg B \ \mathbf{W} \ (B \wedge \Box_{d_1,d_2} \neg A)$
- **TGIL Definition:** see Fig. 20
- **FOTT Definition:** $\forall \sigma_1, \sigma_2, \sigma_3, \omega \ . \ (\sigma = \sigma_1 B \sigma_2 \omega \sigma_3 \wedge B \notin \sigma_1 \wedge \Delta(\sigma_2) \in [d_1, d_2]) \Rightarrow \neg A(\omega)$

- **Observer:** see Fig. 7
- **LTL formula to verify:** $\Diamond B \Rightarrow \Diamond \mathsf{Error}$

2. Absent A before B for duration D

- **Textual Definition:** This pattern asserts that no A can occur less than D u.t. before the first occurrence of B.
- **Example: absent** Open$_1$ **before** Close$_1$ **for duration** 3

- **MITL Definition:** $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0;D]} \neg B))$ $\mathbf{U}$ B
- **TGIL Definition:** see Fig. 20
- **FOTT Definition:**

- **Observer:** see Fig. 10
- **LTL formula to verify:** $\neg\Diamond$(foundB $\wedge$ bad)

3. Absent A within I

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

4. Absent A lasting D

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

5. Absent A between B and C within I

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

6. Absent A after B until C within I

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

Table 3: Response patterns

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

- **Textual Definition:** This pattern states that every occurrence of an event, say A, must be followed by an occurrence of B within a time interval $[d_1, d_2]$ (considering only the first occurrence of B after A).
- **Example:** Button_2 **leadsto first** Open_2 **within** $[0, 10]$

- **MITL Definition:** $\Box(A \Rightarrow (\neg B)\ \mathbf{U}_{[d_1,d_2]}\ B)$
- **TGIL Definition:** see Fig. 16
- **FOTT Definition:** $\forall \sigma_1, \sigma_2 \ .\ (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 \ .\ \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in [d_1, d_2] \wedge B \notin \sigma_3$

- **Observer:** see Fig. 11
- **LTL formula to verify:** $\neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$

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

- **Textual Definition:** This pattern asserts that, before the first occurrence of R, each occurrence of A is followed by B, which occurs both before R, and in the time interval $[d_1, d_2]$ after A. If R does not occur, the pattern holds.
- **Example:** Open_1 **leadsto first** Ventil **within** $[0, 5]$ **before** Open_2

- **MITL Definition:** $\Diamond R \Rightarrow (\Box(A \wedge \neg R \Rightarrow (\neg B \wedge \neg R)\ \mathbf{U}_{[d_1,d_2]}\ B \wedge \neg R)\ \mathbf{U}\ R$
- **TGIL Definition:** see Fig. 21
- **FOTT Definition:** $\forall \sigma_1, \sigma_2, \sigma_3 \ .\ (\sigma = \sigma_1 A \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 A \sigma_2 \Rightarrow \exists \sigma_4, \sigma_5 \ .\ \sigma_2 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in [d_1, d_2] \wedge B \notin \sigma_4$

- **Observer:** see Fig. 12
- **LTL formula to verify:** $\Diamond R \Rightarrow \neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$

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

- **Textual Definition:** This pattern asserts that after the first occurrence of R, "A **leadsto first** B within $[d_1, d_2]$" holds.
- **Example:** $Close_1$ **leadsto first** Ventil **within** $[0, 6]$ **after** Open_1

- **MITL Definition:** $\Box(R \Rightarrow (\Box(A \Rightarrow (\neg B)\ \mathbf{U}_{[d_1,d_2]}\ B)))$
- **TGIL Definition:** see Fig. 22
- **FOTT Definition:** $\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 R \sigma_2 A \sigma_3 \wedge R \notin \sigma_1) \Rightarrow \exists \sigma_4, \sigma_5\ .\ \sigma_3 = \sigma_4 B \sigma_5 \wedge \Delta(\sigma_4) \in [d_1, d_2] \wedge B \notin \sigma_4$

- **Observer:** see Fig. 13
- **LTL formula to verify:** $\Diamond R \Rightarrow \neg\Diamond(\mathsf{Error} \vee (B \wedge \mathsf{bad}))$

4. A leadsto B between Q and R within I

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

5. A leadsto B after Q until R within I

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

Table 4: Universality patterns

1. always A lasting D

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

2. always A within I

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

3. always A after B for duration D

This pattern is defined as **absent** $\neg A$ **after** B **for duration** D

4. always A after B for duration D

This pattern is defined as **absent** $\neg A$ **after** B **for duration** D

Table 5: Precedence patterns

1. A precedes B for duration D

This pattern is defined as **absent** B **before** A **for duration** D

2. A precedes B before R for duration D

– **Textual Definition:** This pattern asserts that before the first occurrence of R, no B occurs before the first occurrence of A for duration D.
– **Example:** $Button_1 \wedge Open_1$ **precedes** $Close_1$ **before** Ventil **for duration** 4

– **MITL Definition:** $\Diamond R \Rightarrow (A \Rightarrow (\Box_{[0;D]} \neg B))\ \mathbf{U}\ R$
– **TGIL Definition:** see Fig. 23
– **FOTT Definition:** $\forall \sigma_1, \sigma_2, \sigma_3\ .\ (\sigma = \sigma_1 \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 \sigma_2 \wedge A \in \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow A \notin \sigma_2$

– **Observer:** see Fig. 14
– **LTL formula to verify:** $\neg \Diamond(\mathsf{foundB} \wedge \mathsf{bad})$

3. A precedes B after R for duration D

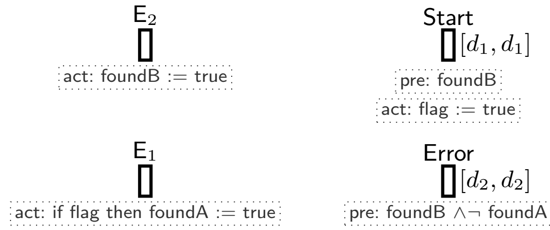

Fig. 7. Observer for the pattern **present** A **after** B **within** $[d_1; d_2]$.

- **Textual Definition:** This pattern asserts that before the first occurrence of R, no B occurs before the first occurrence of A for duration D.
- **Example:** $Button_1 \wedge Open_1$ **precedes** $Close_1$ **before** Ventil **for duration** 4

- **MITL Definition:** $\Diamond R \Rightarrow (A \Rightarrow (\Box_{[0;D]} \neg B))$ **U** R
- **TGIL Definition:** see Fig. 23
- **FOTT Definition:** $\forall \sigma_1, \sigma_2, \sigma_3$. $(\sigma = \sigma_1 \sigma_2 R \sigma_3 \wedge R \notin \sigma_1 \sigma_2 \wedge A \in \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow A \notin \sigma_2$

- **Observer:** see Fig. 14
- **LTL formula to verify:** $\neg \Diamond$(foundB ∧ bad)

4. A precedes B between Q and R for duration D

This pattern is equivalent to the composition of two patterns :
A **precedes** B **after** Q **for duration** D
and
A **precedes** B **before** R **for duration** D

5. A precedes B after Q until R for duration D

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

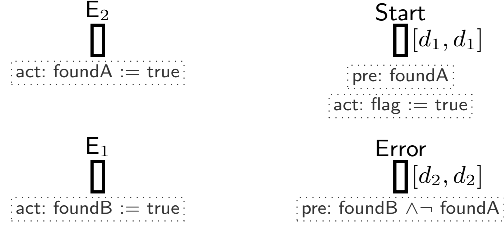

Fig. 8. Observer for the pattern **present** first A **before** B **within** $[d_1; d_2]$.

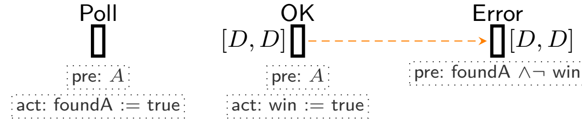

Fig. 9. Observer "**present** A **lasting** D"

3.1 Composite Patterns

We notice that patterns can be combined. To verify composed patterns, we use the combination of the necessary observers corresponding to the composed patterns. We provide below a list of composite patterns with the corresponding LTL formulas to verify. We assume that π_1 and π_2 are the LTL formulas corresponding to $pattern_1$ and $pattern_2$ respectively.

- $pattern_1$ **and** $pattern_2$: the corresponding LTL formula is $\pi_1 \wedge \pi_2$.
- $pattern_1$ **or** $pattern_2$: the corresponding LTL formula is $\pi_1 \vee \pi_2$.
- **not** $pattern_1$: the corresponding LTL formula is $\neg\pi_1$.
- $pattern_1$ **imply** $pattern_2$: the corresponding LTL formula is $\neg\pi_1 \vee \pi_2$.
- $pattern_1$ **xor** $pattern_2$: the corresponding LTL formula is $(\pi_1 \vee \pi_2) \wedge \neg(\pi_1 \wedge \pi_2)$.

Pnueli et al. [19] 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 Related Work and Contributions

In untimed context, the original catalog of specification patterns 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 patterns language is still supported, with several tools, an online repository of examples (`http://patterns.projects.cis.ksu.edu/`) and the

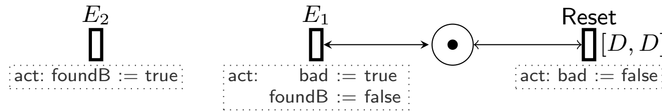

Fig. 10. Observer "**absent** A **before** B **for duration** D"

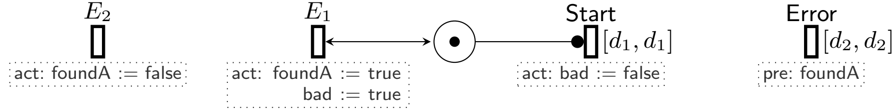

Fig. 11. Observer "A **leadsto first** B **within** $[d_1, d_2]$"

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 patterns 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. A graphical real-time logic, called RTGIL, has been introduced in [17]. RTGIL does not take in account the notion of grouping and time constrained search (see Sect.2). Moreover, in TGIL, we take in account different kind of context which we use to define patterns. The notion of closed interval is not taken in account in RTGIL. We can not define some patterns like 'Present first A before B within $[d1; d2]$' in RTGIL.

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. [20] 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. By this way, we reduce the problem of checking real-time properties to the problem of checking LTL properties on the compo-

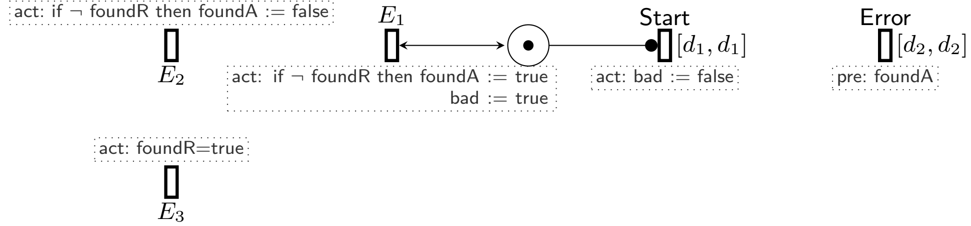

Fig. 12. Observer "A **leadsto** B **within** $[d_1, d_2]$ before R"

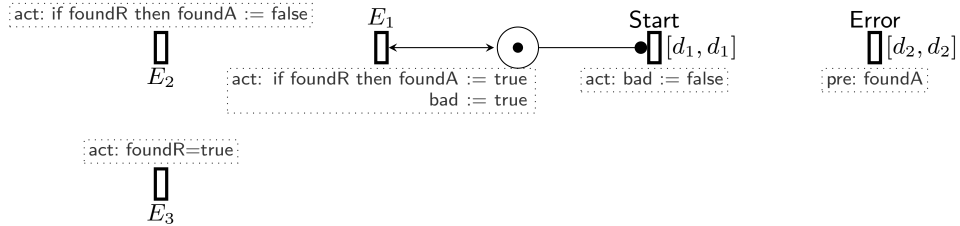

Fig. 13. Observer "A **leadsto** B **within** $[d_1, d_2]$ after R"

sition 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 patterns 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.

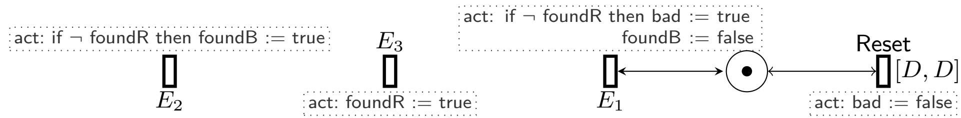

Fig. 14. Observer "A **precedes** B **for duration** D before R"

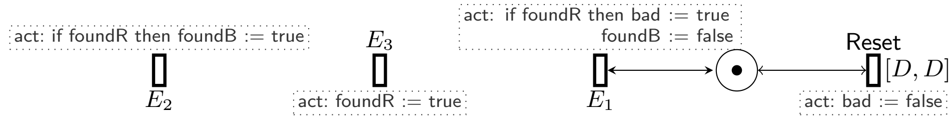

Fig. 15. Observer "A **precedes** B **for duration** D before R"

For future works, we plan to consider a lower-level compositional patterns 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. *Research Report*, hal-00593963, 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.

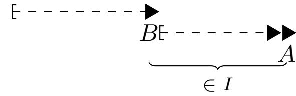

Fig. 16. Graphical presentation for "Present [first] A **after** B **within** $[d_1; d_2]$"

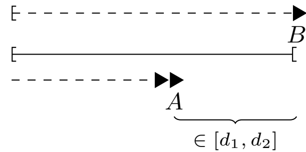

Fig. 17. Graphical presentation for "Present first A **before** B **within** $[d_1; d_2]$"

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. A. Pnueli, A. Zaks On the Merits of Temporal Testers. *Book:25 Years of Model Checking*, 172 - 195, Springer-Verlag Berlin, Heidelberg , 2008
20. 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.
21. pFrac (version 1.4.0) [software], *available from URL http://homepages.laas.fr/nabid/pfrac.html*, 2011.

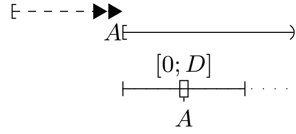

Fig. 18. Graphical presentation for "Present A **lasting** D"

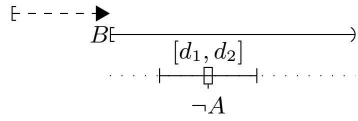

Fig. 19. Graphical presentation for "Absent A **after** B **for interval** $[d_1; d_2]$"

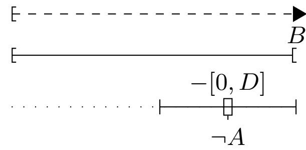

Fig. 20. Graphical presentation for "Absent A **before** B **for duration** D"

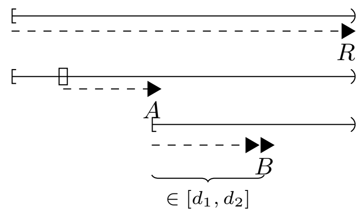

Fig. 21. Graphical presentation for "A **leadsto** B **within** I **before** R"

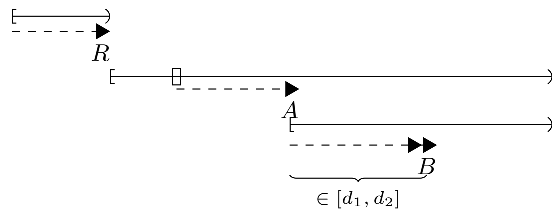

Fig. 22. Graphical presentation for "A **leadsto** B **within** I **after** R"

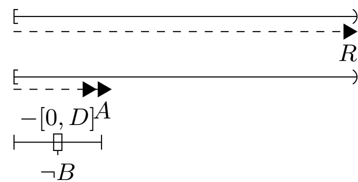

Fig. 23. Graphical presentation for "A **precedes** B **before** R **for duration** D"

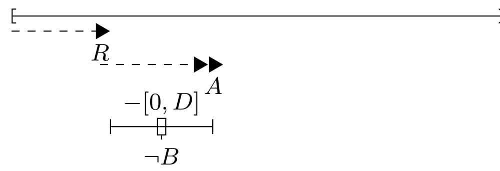

Fig. 24. Graphical presentation for "A **precedes** B **after** R **for duration** D"