Real-Time Specification Patterns and Tools

Nouha Abid, Silvano Dal Zilio, Didier Le Botlan

Real-Time Specification Patterns and Tools*

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

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. An issue limiting the adoption of model checking technologies by the industry is the ability, for non-experts, to express their requirements using the property languages supported by verification tools. This has motivated the definition of dedicated assertion languages for expressing temporal properties at a higher level. However, only a limited number of these formalisms support the definition of timing constraints. In this paper, we propose a set of specification patterns that can be used to express real-time requirements commonly found in the design of reactive systems. We also provide an integrated model checking tool chain for the verification of timed requirements on TTS, an extension of Timed Petri Nets with data variables and priorities.

1 Introduction

An issue limiting the adoption of model checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the verification tools. Indeed, there is often a significant gap between the boilerplates used in requirements statements and the low-level formalisms used by model checking tools; the latter usually relying on temporal logic. This limitation has motivated the definition of dedicated assertion languages for expressing properties at a higher level (see Section 5). However, only a limited number of assertion languages support the definition of timing constraints and even fewer are associated to an automatic verification tool, such as a model checker.

In this paper, we propose a set of real-time specification patterns aimed at the verification of reactive systems with hard real-time constraints. Our main objective is to propose an alternative to timed extensions of temporal logic during model checking. Our patterns are designed to express general timing constraints commonly found in the analysis of real-time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both clarity and computational complexity. In particular, each pattern should correspond to a decidable model checking problem.

* This work was partially supported by the JU Artemisia project CESAR and the FNRAE project Quarteft

Our patterns can be viewed as a real-time extension of Dwyer's specification patterns [11]. In his seminal work, Dwyer shows through a study of 500 specification examples that 80% of the temporal requirements can be covered by a small number of "pattern formulas". We follow a similar philosophy and define a list of patterns that takes into account timing constraints. At the syntactic level, this is mostly obtained by extending Dwyer's patterns with two kind of *timing modifiers*: (1) P within I, which states that the delay between two events declared in the pattern P must fit in the time interval I; and (2) P lasting D, which states that a given condition in P must hold for at least duration D. For example, we define a pattern Present A after B within $]0, 4]$ to express that the event A must occur within 4 units of time (u.t.) of the first occurrence of event B, if any, and not simultaneously with it. Although seemingly innocuous, the addition of these two modifiers has a great impact on the semantics of patterns and on the verification techniques that are involved.

Our second contribution is an integrated model checking tool chain that can be used to check timed requirements. We provide a compiler for Fiacre [6], a formal modelling language for real-time systems, that we extended to support the declaration of real-time patterns. In our tool chain, Fiacre is used to express the model of the system while verification activities ultimately relies on Tina [3], the TIme Petri Net Analyzer. This tool chain provides a reference implementation for our patterns when the systems can be modeled using an extension of Time Petri Nets with data variables and priorities that we call a TTS (see Sect. 2.1). This is not a toy example; Fiacre is the intermediate language used for model verification in Topcased [13], an Eclipse-based toolkit for system engineering, where it is used as the target of model transformation engines for various high-level modelling languages, such as SDL or AADL [4]. In each of these transformations, we have been able to use our specification patterns as an intermediate format between high-level requirements (expressed on the high-level models) and the low-level input languages supported by the model checkers in Tina.

The rest of the paper is organized as follows. In the next section, we define technical notations necessary to define the semantics of patterns. Section 3 gives our catalog of real-time patterns. For each pattern, we give a simple definition in natural language as well as an unambiguous, formal definition based on two different approaches. Before concluding, we review the results of experiments that have been performed using our verification tool chain in Sect. 4.

2 Technical Background

Since patterns are used to express timing and behavioral constraints on the execution of a system, we base the semantics of patterns on the notion of *timed traces*, which are sequences mixing events and time delays, (see Def. 1 below). We use a dense time model, meaning that we consider rational time delays and work both with strict and non-strict time bounds.

The semantics of a pattern will be expressed as the set of all timed traces where the pattern holds. We use two different approaches to define set of traces:

(1) Time Transition Systems (TTS), whose semantics relies on timed traces; and (2) a timed extensions of Linear Temporal Logic, called MTL. In our verification tool chain, both Fiacre and specification patterns are compiled into TTS.

2.1 Time Transition Systems and Timed Traces

Time Transition Systems (TTS) are a generalization of Time Petri Nets [17] with priorities and data variables. We describe the semantics of TTS using a simple example. Figure 1 gives a model for a simple airlock system consisting of two doors (D_1 and D_2) and two buttons. At any time, at most one door can be open. This constraint is modeled by the fact that at most one of the places $\mathsf{D_1isOpen}$ and $\mathsf{D_2isOpen}$ may have a token. 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. This behavior is modeled by adding timing constraints on the transitions $\mathsf{Open_1}$, $\mathsf{Open_2}$ and $\mathsf{Ventil.}$. Moreover, requests to open the door D_2 have higher priority than requests to open D_1. This is modeled using a priority (dashed arrow) from the transition $\mathsf{Open_2}$ to $\mathsf{Open_1}$. A shutdown command can be triggered if no request is pending.

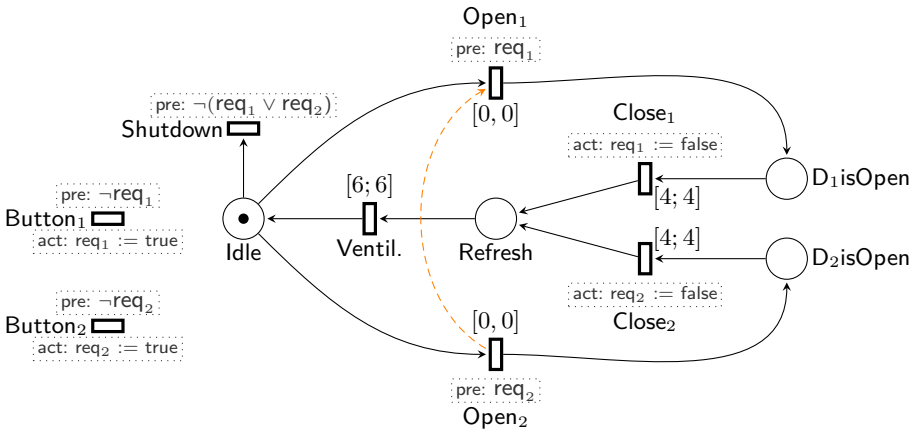


Fig. 1. A TTS model of an airlock system

To understand the model, the reader may, at first, ignore side conditions and side effects (the pre and act expressions inside dotted rectangles). In this case, a TTS is a standard Time Petri Net, where circles are places and rectangles are transitions. A transition is enabled if there are enough tokens in its input places. A time interval, such as $I = [d_1; d_2[$, indicates that the corresponding transition must be fired if it has been enabled for d units of time with $d \in I$. As a consequence, a transition associated to the time interval $[0; 0]$ must fire as soon as it is enabled. Our model includes two boolean variables, $\mathsf{req_1}$ and

req_2, indicating whether a request to open door D_1 (resp. D_2) is pending. Those variables are read by pre-conditions on transitions Open_i, Button_i, and $\mathsf{Shutdown}$ and are modified by post-actions on transitions Button_i and Close_i. For instance, the pre-condition $\neg\mathsf{req}_2$ on Button_2 is used to disable the transition when the door is already open. This implies that pressing the button while the door is open has no further effect.

We introduce some basic notations used in the remainder of the paper. (A complete, formal description of the TTS semantics can be found in [2].) Like with Petri Nets, the state of a TTS depends on its marking, m, that is the number of tokens in each place. We write $\mathcal{M}$ the set of markings. Since we manipulate values, the state of a TTS also depends on its *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. The behavior of a TTS is defined by the set of all its (timed) traces. In this particular case, a trace will contain information about fired transitions (e.g. Open_1), markings, the value of variables, and the passing of time. 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 events. The set of non-negative rational numbers is written $\mathbb{Q}^+$.

Definition 1 (Timed trace). *A timed trace σ is a possibly infinite sequence of events $\omega \in \Omega$ and durations $d(\delta)$ with $\delta \in \mathbb{Q}^+$. Formally, σ is a partial mapping from $\mathbb{N}$ to $\Omega^* = \Omega \cup \{d(\delta) \mid \delta \in \mathbb{Q}^+\}$ such that $\sigma(i)$ is defined whenever $\sigma(j)$ is defined and $i \leq j$.*

Given a finite trace σ and a—possibly infinite—trace σ', we denote $\sigma\sigma'$ the *concatenation* of σ and σ'. This operation is associative. The semantics of patterns will be defined as a set of timed traces. Given a real-time pattern P, we say that a TTS T satisfies the requirement P if all the traces of T hold for P.

2.2 Metric Temporal Logic and Formulas over Traces

Metric Temporal Logic (MTL) [16] is an extension of LTL where temporal modalities can be constrained by a time interval. For instance, the MTL formula $A\ \mathbf{U}_{[1,3[}\ B$ states that in every execution of the system (in every trace), the event B must occur at a time $t_0 \in [1, 3[$ and that A holds everywhere 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. 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 unambiguous framework for defining the meaning of patterns. Nonetheless, this partially defeats one of the original goal of patterns, that is to circumvent the use of temporal logic in the first place. For this reason, we propose an alternative way for defining the semantics of patterns that relies on first-order formulas over traces. For instance, when referring to a timed trace σ and an event A, we can define the

"scope" σ after A–that determines the part of σ located after the first occurrence of A–as the trace σ_2 such that $\exists \sigma_1.\sigma = \sigma_1 A \sigma_2 \wedge A \notin \sigma_1$. We believe that this second approach may ease the work of engineers that are not trained with formal verification techniques. Our experience shows that being able to confront different definitions for the same pattern, using contrasting approaches, is useful for teaching patterns.

2.3 Model checking, Observers and TTS

We have designed our patterns so that checking whether a system satisfies a requirement is a decidable problem. We assume here that we work on discrete models (with a continuous time semantics), such as timed automata or time Petri Nets, and not on hybrid models. Since the model checking problem for MTL is undecidable [18], it is not enough to simply translate each pattern into a MTL formula to check whether a TTS satisfies a pattern. This situation can be somehow alleviated. For instance, the problem is decidable if we disallow simultaneous events in the system and if we disallow punctual timing constraints, of the form $[d, d]$. Still, while we may rely on timed temporal logics as a way to define the semantics of patterns, it is problematic to have to limit ourselves to a decidable fragment of a particular logic–which may be too restrictive–or to rely on multiple real-time model checking algorithms–that all have a very high complexity in practice.

To solve this problem, we propose to rely on *observers* in order to reduce the verification of timed patterns to the verification of LTL formulas. We provide for each pattern, P, a pair (T_P, ϕ_P) of a TTS model and a LTL formula such that, for any TTS model T, we have that T satisfies P if and only if $T \otimes T_P$ (the composition of the two models T and T_P) satisfies ϕ_P. 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. To this end, using our tool chain, we have compared the complexity of different implementations on a fixed set of representative examples and for a specific set of properties and kept the best candidates.

3 A Catalog of Real-Time Patterns

We describe our patterns using a hierarchical classification borrowed from Dwyer [11] but adding the notion of "timing modifiers". Patterns are built from five categories, listed below, or from the composition of several patterns (see Sect. 3.4):

- **Existence Patterns (Present):** for conditions that must eventually occur;
- **Absence Patterns (Absent):** for conditions that should not occur;
- **Universality Patterns :** for conditions that must occur throughout the whole execution;
- **Response Patterns (Response):** for (trigger) conditions that must always be followed by a given (response) condition;

– **Precedence Patterns :** for (signal) conditions that must always be preceded by a given (trigger) condition.

In each class, generic patterns may be specialized using one of five *scope modifiers* that limit the range of the execution trace over which the pattern must hold:

– **Global :** the default scope modifier, that does not limit the range of the pattern. The pattern must hold over the whole timed trace;
– **Before R :** limit the pattern to the beginning of a time trace, up to the first occurrence of R;
– **After Q :** limit the pattern to the events following the first R;
– **Between Q and R :** limit the pattern to the events occurring between an event Q and the following occurrence of an event R;
– **After Q Until R :** similar to the previous scope modifier, except that we do not require that R must necessarily occur after a Q.

Finally, timed patterns are obtained using one of two possible kind of *timing modifiers* that limit the possible dates of events referred in the pattern:

– **Within** I **:** to constraint the delay between two given events to belong to the time interval I;
– **Lasting** D **:** to constraint the length of time during which a given condition holds (without interruption) to be greater than D.

When defining patterns, the symbols $A, B, \ldots$ stand for predicates on events $\omega \in \Omega$ such as $\mathsf{Open}_2 \vee \mathsf{req}_2$. In the definition of observers, a predicate A is interpreted as the set of transitions of the system that match A. Due to the somewhat large number of possible alternatives, we restrict this catalog to the most important presence, absence and response patterns. Patterns that are not described here can be found in a long version of this paper [1].

For each pattern, we give its denotational interpretation based on First-Order formulas over Timed Traces (denoted FOTT in the following) and a logical definition based on MTL. We provide also the observer and the LTL formula that should be combined with the system in order to check the validity of the pattern. We define some conventions on observers. In the following, Error, Start, ... are transitions that belong to the observer, whereas E_1 (resp. E_2) represents all transitions of the system that match predicate A (resp. B). We also use the symbol I as a shorthand for the time interval $[d_1, d_2]$. The observers for the pattern obtained with other time intervals–such as $]d_1, d_2]$, $]d_1, +\infty[$, or in the case $d_1 = d_2$–are essentially the same, except for some priorities between transitions that may change. By convention, the boolean variables used in the definition of an observers are initially set to false.

3.1 Existence patterns

An existence pattern is used to express that, in every trace of the system, some events must occur.

Present A after B within I

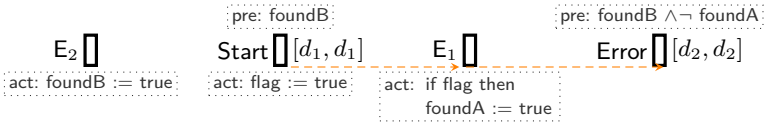
Predicate A must hold between d_1 and d_2 units of time (u.t) after the first occurrence of B. The pattern is also satisfied if B never holds.

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

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

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

Observer:



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

Explanation:

In this observer, transition Error is conditioned by the value of the shared boolean variables foundA and foundB. Variable foundB is set to true after transition E_2 and transition Error is enabled only if the predicate foundB $\wedge\neg$ foundA is true. Transition Start is fired d_1 u.t after an occurrence of E_2 (because it is enabled when foundB is true). Then, after the first occurrence of E_1 and if flag is true, foundA is set to true. This captures the first occurrence of E_1 after Start has been fired. After d_2 u.t., in the absence E_1, transition Error is fired. Therefore, the verification of the pattern boils down to checking if the event Error is reachable. The priority (dashed arrows) between Start, Error, and E_1 is here necessary to ensure that occurrences of E_1 at precisely the date d_1 or d_2 are taken in account.

Present first A before B within I

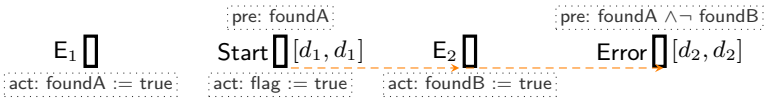
The first occurrence of predicate A holds between d_1 and d_2 u.t. before the first occurrence of B. The pattern is also satisfied if B never holds. (The difference with Present B after A within I is that B should not occur before the first A.)

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

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

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

Observer:



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

Explanation:

Like in the previous case, variables foundA and foundB are used to record the occurrence of transitions E_1 and E_2. Transition Start is fired, and variable flag is set to true, d_1 u.t. after the first E_1. Then transition Error is fired only if its precondition—the predicate foundA $\wedge\neg$ foundB—is true for d_2 u.t. Therefore transition Error is fired if and only if there is an occurrence of E_2 before E_1

(because then foundB is true) or if the first occurrence of E_2 is not within $[d_1, d_2]$ of the first occurrence of E_1.

Present A **lasting** D

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

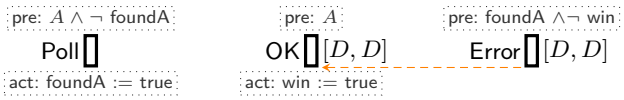
Comment: The pattern makes sense only if A is a predicate on states (that is, on the marking or store); since transitions are instantaneous, they have no duration.

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

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

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

Observer:



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

Explanation:

Variable foundA is set to true when transition *Poll* is fired, that is when A becomes true for the first time. Transition OK is used to set win to true if A is true for duration D without interruption (otherwise its timing constraint is resetted). Otherwise, if variable win is still false after D u.t., then transition *Error* is fired. We use a priority between *Error* and *OK* to disambiguate the behavior D u.t. after Poll is fired.

3.2 Absence patterns

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

Absent A **after** B **for interval** I

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

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

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

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

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

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

Explanation:

Same as the explanation for **Present** A **after** B **within** I.

Absent A **before** B **for duration** D

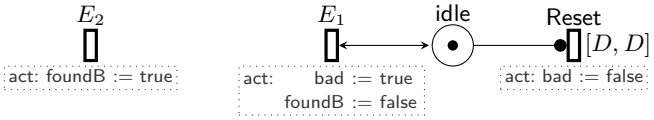
No A can occur less than D u.t. before the first occurrence of B. The pattern holds if there are no occurrence of B.

Example: **absent** Open_1 **before** Close_1 **for duration** 3

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

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

Observer:



The associated LTL formula is $\Box\neg(\mathsf{foundB} \wedge \mathsf{bad})$.

Explanation:

Variable $foundB$ is set to true after each occurrence of E_2. Conversely, we set the variables *bad* to true and $foundB$ to false at each occurrence of E_1. Therefore $foundB$ is true on every "time interval" between an E_2 and an E_1. We use transition *Reset* to set *bad* to false if this interval is longer than D. As a consequence, the pattern holds if we cannot find an occurrence of E_2 ($foundB$ is true) while *bad* is true.

3.3 Response patterns

Response patterns are used to express "cause–effect" relationship, such as the fact that an occurrence of a first kind of events must be followed by an occurrence of a second kind of events.

A **leadsto first** B **within** I

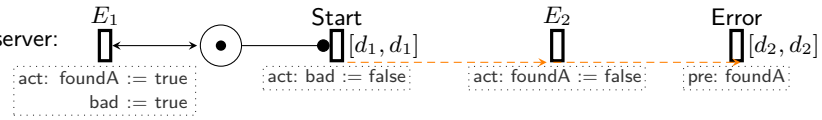
Every occurrence of A must be followed by an occurrence of B within time interval I (considering only the first occurrence of B after A).

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

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

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

Observer:



The associated LTL formula is $(\Box\neg\mathsf{Error}) \wedge (\Box\neg(B \wedge \mathsf{bad}))$.

Explanation:

After each occurrence of E_1, variables $foundA$ and *bad* are set to true and the transition *Start* is enabled. Variable *bad* is used to control the beginning of the time interval. After each occurrence of E_2 variable $foundA$ is set to false. Hence *Error* is fired if there is an occurrence of E_1 not followed by an occurrence of E_2 after d_2 u.t. We use priorities to avoid errors when E_2 occurs precisely at time d_1 or d_2.

A **leadsto first** B **within** I **before** R

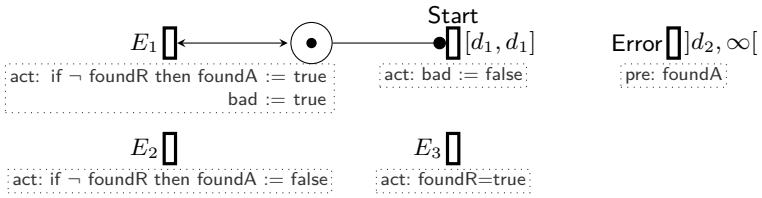
Before the first occurrence of R, each occurrence of A is followed by a B—and these two events occur before R—in the time interval I. The pattern holds if R never occur.

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

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

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

Observer:



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

Explanation:

Same explanation than for the previous case, but we only take into account transitions E_1 and E_2 occurring before E_3.

A **leadsto first** B **within** I **after** R

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

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

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

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

Observer: It is similar to the observer of the pattern A leadsto first B within I before R . We should just replace ¬foundR in transition E_1 and E_2 by foundR. The associated LTL formula is $\Diamond R \Rightarrow (\Box\neg\text{Error} \wedge \Box\neg(B \wedge \text{bad}))$.

Explanation:

Same explanation than in the previous case, but we only take into account transitions E_1 and E_2 occurring after an E_3.

3.4 Composite Patterns

Patterns can be easily combined together using the usual boolean connectives. For example, the pattern "P_1 **and** P_2" holds for all the traces where P_1 and P_2 both hold. To check a composed pattern, we use a combination of the respective observers, as well as a combination of the respective LTL formulas. For instance, if (T_1, ϕ_1) and (T_2, ϕ_2) are the observers and LTL formulas corresponding to the patterns P_1 and P_2, then the composite pattern P_1 **and** P_2 is checked using the LTL formula $\phi_1 \wedge \phi_2$. Similarly, if we check the LTL formula $\phi_1 \Rightarrow \phi_2$ (implication) then we obtain a composite pattern $P_1 \multimap P_2$ that is satisfied by systems T such that, for all traces of T, the pattern P_2 holds whenever P_1 holds.

4 Use Cases and Experimental Results

In this section, we report on three experiments that have been performed using an extension of a Fiacre compiler that automatically compose a system with

the necessary observers. In case the system does not meet its specification, we obtain a counter-example that can be converted into a timed sequence of events exhibiting a problematic scenario. This sequence can be played back using *play* and *nd*, two Time Petri Nets animators provided with Tina.

Avionic Protocol and AADL. Our first example is a network avionic protocol (NPL) which includes several functions allowing the pilot and ground stations to receive and send information relative to the plane: weather, speed, ... AADL has been used to model the dynamic architecture for this demonstrator [5]. The AADL model includes several threads that exchange information through shared memory data and amounts to about 8 diagrams and 800 lines of code (using AADL textual syntax). The AADL code specifies both the hardware and software architecture of the system and defines the real time properties of threads, like for instance their dispatch protocol (periodic or sporadic) or their periods.

We used the AADL2Fiacre plug-in of Topcased to check properties on the NPL specification. The Fiacre model obtained after transformation takes into account the complete behavior described in the AADL model but also the whole language execution model, meaning that our interpretation takes fully into account the scheduling semantics as specified in the AADL standard. The abstract state space for the TTS generated from Fiacre has about 120 000 states and 180 000 transitions and can be generated in less than 12s on a typical development computer (Intel dual-core processor at 2GHz with 2Gb of RAM). On examples of this size, our model checker is able to prove formal properties in a few seconds. We checked a set of 22 requirements that were given together with the description of the system, all expressed using a natural language description and, in one case, a scenario based on a UML sequence diagram. Of these 22 requirements, 18 where instances of "untimed patterns", such as checking the absence of deadlock or that threads are resettable. The four remaining requirements where "response patterns" of the kind A leadsto first B within $[0, d]$. Using patterns, we were able to check the 22 patterns in less than 5min.

Service Oriented Applications. We consider models obtained from the composition of services expressed using a timed extension of BPEL, the Business Process Execution Language. Our example models a scenario from the health-care domain related to patient handling during a medical examination. The scenario involves three entities, each one managed by a service: a Clinic Service (CS); a Medical Analysis Service (MAS); and a Pharmacy Service (PS). When a patient arrives in clinic, a doctor should check with the MCS whether its social security number is valid. If so, the doctor may order some medical analyses from the MAS and, after analyzing the results, he can order drugs through the PS. Timing constraints can be added to this scenario by associating a duration to each activity of the workflow and a delay to each service invocation.

We use our patterns to express different requirements on this system. An example involving the absence pattern is that we cannot have two medical analyses for a patient in less than 10 days (240 hours): absent MAS.medicalAnalysis

after MAS.medicalAnalysis for interval $]0, 240]$. A more complicated example of requirement is to impose that if a doctor does not cancel a drug order within 6 hours, then it should not cancel drugs for another 48 hours. This requirement can be expressed using the composition of two absence patterns (see Sect. 3.4):

(absent MCS.drugsChanging after MCS.drugsAsking for interval $[0; 6]$)
$\multimap$ (absent MCS.drugsChanging after MCS.drugsAsking for interval $[0; 54]$).

Finally, using the notation S.init and S.end to refer to a start (resp. end) event in the service S, we can express that drugs must be delivered within 48 hours of the medical examination start: MCS.init leadsto PS.sendDrugsOrder within $[0; 48]$.

The complete scenario is given in [9], where we describe a transformation tool chain from Timed BPEL processes to Fiacre. For a more complex version of the health care scenario, with seven different services and more concurrent activities, the state graph for the TTS generated from Fiacre is quite small, with only 886 states and 2476 transitions. The generation of the Fiacre specification and its corresponding state space takes less than a second. For examples of this size, the verification time for checking a requirement is negligible (half a second).

Transportation Systems. Our final example is an automated railcar system, taken from [10], that was directly modeled using Fiacre. The system is composed of four terminals connected by rail tracks in a cyclic network. Several railcars, operated from a central control center, are available to transport passengers between terminals. When a car approaches its destination, it sends a request to signal its arrival to the terminal. This system has several real-time constraints: the terminal must be ready to accommodate an incoming car in 5s; a car arriving in a terminal leaves its door open for exactly 10s; passengers entering a car have 5s to choose their destination; etc. There are three key requirements:

(P1) when a passenger arrives in a terminal, a car must be ready to transport him within 15s. This property can be expressed with a response pattern, where Passenger/sndReq is the state where the passenger requests a car and Car/ackTerm is the state where it is served:

$$\text{Passenger/sendReq } \textbf{leadsto } \text{Car/ackTerm } \textbf{within } [0, 15]$$

(P2) When the car starts moving, the door must be closed:

$$\textbf{present } \text{CarDoor/closeDoor } \textbf{after } \text{CarHandler/moving } \textbf{within } [0, 10]$$

(P3) When a passenger select a destination (in the car), a signal should stay illuminated until the car is arrived:

$$\textbf{absent } \text{Terminal/buttonOff } \textbf{before } \text{Control/ackTerm } \textbf{for duration } 10$$

We can prove that these three patterns are valid on our Fiacre model. Concerning the performances, we are able to generate the complete state space of

the railcar system in 310 ms, using 400 kB of memory. This gives an upper-bound to the complexity of checking simple (untimed) reachability properties on the system, like for instance the absence of deadlocks. The three patterns can all be checked in under 1.5 s. For instance, we observed that checking property (P1) is not more complex than exploring the complete system: the property is checked in 450 ms, using 780 kB of memory. Also, this is roughly the same complexity than checking the corresponding untimed requirement in LTL that is: $\Box(\mathsf{Passenger/sendReq} \Rightarrow \Diamond \mathsf{Control/ackTerm})$.

Conclusion. In other benchmarks, we have often found that the complexity of checking timed patterns is in the same order of magnitude than checking their untimed temporal logic equivalent. An exception to this observation is when the temporal values used in the patterns are far different from those found in the system; for example if checking a periodic system, with a period in the order of the milliseconds, against a requirement using an interval in the order of the minutes. More results on the complexity of our approach can be found in [2]. These experimentation, while still modest in size, gives a good appraisal of the use of formal verification techniques for real industrial software.

These experimental results are very encouraging. In particular, we can realistically envisage that system engineers could evaluate different design choices in a very short time cycle and test the safety of their solutions at each iteration.

5 Related Work and Contributions

We base our approach on the original catalog of specification patterns defined by Dwyer [11]. This work essentially study the expressiveness of their approach and define patterns using different logical framework (LTL, CTL, Quantified Regular Expressions, etc.). As a consequence, they do not need to consider the problem of checking requirements as they can readily rely on existing model checkers. Their patterns language is still supported, with several tools, an online repository of examples [12] and the definition of the Bandera Specification Language [8] that provides a structured-English language front-end. A recent study by Bianculli et al. [7] show the relevance of this pattern-based approach in an industrial context.

Some works consider the extension of patterns with hard real-time constraints. Konrad et al. [15] extend the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL. Nonetheless, they do not consider the complexity of the verification problem (the implementability of their approach). Another related work is [14], where the authors define observers based on Timed Automata for each pattern. However, they consider a less expressive set of patterns (without the lasting modifier) and they have not integrated their language inside a tool chain or proved the correctness of their observers. By contrast, we have defined a formal framework that has been used to prove the correctness of some of our observers [2]. Work is currently under way to mechanize these proofs using the Coq interactive theorem prover.

Our patterns can be viewed as a subset of the Contracts Specification Language (CSL), defined in the context of the SPEEDS project [19], which is intended as a pragmatic proposal for specifying contract assertions on HRC models. While the semantics for HRC is based on hybrid automata, the only automatic verification tools available for CSL use a discrete time model. Therefore, our verification tool chain provides a partial implementation for CSL (the part concerned by timing constraints) for a dense time model. This is an important result since more conception errors can be captured using a dense rather than a discrete time model.

Compared to these related works, we make several contributions. We extend the specification patterns language of Dwyer et al. with two modifiers for real-time constraints. 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 a set of observers, that are described in Sect. 3. Using this approach, we reduce the problem of checking real-time properties to the problem of checking simpler LTL properties on the composition of the system with an observer. Another contribution is the definition of a formal framework to prove that observers are correct and non-intrusive, meaning that they do not affect the system under observation. This framework is useful for proving the soundness of optimization. Due to space limitations, we concentrate on the definition of the patterns and their semantics in this paper, while most of the theoretical results are presented in a companion research report [2]. Finally, concerning tooling, we offer an EMF-based meta-model for our specification patterns that allow its integration within a model-driven engineering development: our work is integrated in a complete verification tool chain for the Fiacre modelling language and can therefore be used in conjunction with Topcased [13], an Eclipse based toolkit for system engineering.

6 Conclusion and Perspectives

We define a set of high-level specification patterns for expressing requirements on systems with hard real-time constraints. Our approach eliminates the need to rely on model checking algorithms for timed extensions of temporal logics that—when decidable—are very complex and time-consuming. While we have concentrated our attention on model checking—and although we only provide an implementation for TTS models—we believe our notation is interesting in its own right and can be reused in different contexts.

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

has nearly reached completion, would be used as a new target for implementing patterns verification.

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. A Real-Time Specification Patterns Language. Technical Report 11364, LAAS, 2011.
2. N. Abid, S. Dal Zilio, and D. Le Botlan. Verification of Real-Time Specification Patterns on Time Transitions Systems. Technical Report 11365, LAAS, 2011.
3. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42:14, 2004.
4. B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat. Formal verification of AADL specifications in the Topcased environment. In *Proc. of Ada-Europe*, vol. 5570 of *LNCS*. Springer, 2009.
5. B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, P. Dissaux, M. Filali, S. Heim, P. Gaufillet and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *Proc. of ERTSS 2010–5th International Congress and Exhibition on Embedded Real-Time Software and Systems*, 2010.
6. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Proc. of ERTS*, 2008.
7. D. Bianculli and C. Ghezzi and C. Pautasso and P. Senti. Specification Patterns from Research to Industry: a Case Study in Service-based Applications. In *the 34th International Conference on Software Engineering*. IEEE, 2012.
8. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN Software Model Checking Workshop*, vol. 1885 of *LNCS*. Springer, 2000.
9. N. Guermouche, and S. DalZilio . Formal Requirement Verification for Timed Choreographies. Technical Report HAL 578436, 2011.
10. J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 52(1), 2008.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ICSE*, 1999.
12. M. B. Dwyer, L. Dillon. Online Repository of Specification Patterns. At `http://patterns.projects.cis.ksu.edu/`
13. P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design. In *Proc. of ERTS*, 2006.
14. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
15. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. of ICSE*, ACM, 2005.
16. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2:255–299, October 1990.
17. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, 1974.
18. J. Ouaknine and J. Worrell. On the decidability and complexity of metric temporal logic over finite words. In *Logical Methods in Computer Science*, vol. 3, 2007.
19. V. Gafni. Contract Specification Language (CSL). In Speeds D2.5.4–Speculative and Exploratory Design in Systems Engineering, 2008.