Observer Patterns for Real-Time Systems

Étienne André
Université Paris 13, Sorbonne Paris Cité, LIPN
F-93430, Villetaneuse, France
Web page: http://lipn.univ-paris13.fr/~andre/

***Abstract*—In the past few decades, many formal techniques for verifying complex concurrent and real-time systems, as well as many property languages, have been proposed. Unfortunately, many of these techniques involve formalisms that are not always easy to handle by engineers; furthermore, they generally need dedicated tools. We propose here a set of correctness patterns encoding common properties met when verifying concurrent real-time systems. We show how to translate these patterns into pure reachability problems, thus avoiding the use of complex verification algorithms. Furthermore, we provide an instantiation of these patterns in both timed automata and stateful timed CSP, to show the applicability of our approach.**

Keywords*—Specification; Real-Time Systems; Verification; Timed Automata; CSP; Tool Support

I. Introduction

Verifying complex concurrent and real-time systems is an important challenge. In the past few decades, many formal techniques for verifying complex concurrent and real-time systems, as well as many property languages, have been proposed. Unfortunately, many of these techniques involve formalisms that are not always easy to handle by industry engineers. In particular, temporal logics (e.g., [Pnu77], [BK08]) offer a very powerful way of expressing correctness properties for concurrent systems. However, they are often considered to be too complicated (and maybe too rich as well) to be widely adopted by engineers. Furthermore, they generally need advanced tools dedicated to model checking their properties.

In this paper, we identify commonly used properties of correctness for real-time systems, gathered from several years of experience in verification, in particular with engineers from the industry. We propose for each pattern a syntax as human-readable as possible, so that engineers non-experts in formal methods can use them. Furthermore, we show how to translate them into pure reachability properties using simple observers, that is additional subsystems that observe some system actions and may also make use of time. Whereas one class of patterns must be translated into "must-reach" observers (i.e., for which a good state must be reached in each run), all other patterns can be translated into non-reachability observers (i.e., the property is correct if a given bad state is never reachable). Hence, their verification in practice avoids the use of complex verification algorithms or dedicated tools, and tool developers can implement them at little cost. Furthermore, we provide an instantiation of these patterns in both timed automata [AD94] and stateful timed CSP [SLD+13], to show the applicability of our approach; these observers can then be used in tools that only handle reachability analysis. Finally, we implemented these patterns into IMITATOR [AFKS12], a tool relying on (a parametric extension of) timed automata. The contributions of this paper are summarized below:

1) We identify commonly used properties of correctness for real-time systems;
2) We propose an abstract syntax for each pattern;
3) We translate each pattern to an observer instantiated in both timed automata and stateful timed CSP;
4) We provide a concrete syntax for the patterns, implemented in IMITATOR.

Related Work: Many formalisms have been used to model complex real-time systems, in particular timed automata [AD94], timed extensions of CSP [Hoa85], [HO02], [SLD+13] or Z [MD99], and various timed extensions of Petri nets [Mer74]. Recent works also include formalisms able to model compositional or hierarchical systems [BP99], [DHQ+08], [JK09], [DLL+10]. In [DHQ+08], timed automata patterns are proposed to model common real-time system behaviors such as deadline, timeout, and timed interrupt. These patterns are compositional and help building a system in a hierarchical manner. Our patterns do not aim at modeling the system, but at expressing its correctness; furthermore, our patterns are not, and shall not be, compositional (see discussion in Sections III and V). We choose as examples of instantiations for our patterns timed automata [AD94] and a timed extension of CSP [SLD+13], since these two classes of formalisms are both commonly used. Numerous tools are also available for them (e.g., UPPAAL [LPY97], for timed automata, and PAT [SLDP09] for stateful timed CSP). Finally, they have a quite different syntax (timed automata are graph-based, whereas CSP is a process algebra), and hence show two different instantiations of our observers.

Concerning the specification of properties for verifying real-time systems, temporal logics (e.g., [Pnu77], [BK08]) and their timed extensions (e.g., [ACD93] among others) are by far the most commonly used, although many other formalisms have been proposed too. Much more expressive than our patterns, temporal logics are also more difficult to handle by non-experts. Furthermore, many tools do not actually support their full expressiveness, but only some fragments. Stateful timed CSP [SLD+13] has been designed to specify not properties, but models. Nevertheless, our patterns share some similarities with stateful timed CSP, in particular the user-friendly English-like syntax, and the simple way to express common timing behaviors such as "deadline" or "within". Closer to our approach is the logical-algebraic specification language for dynamic systems CASL-LTL defined in [RAC03]

This paper is the author (and slightly improved) version of the paper with the same name published in the proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS'13). The final version is available at http://ieeexplore.ieee.org/.

and used, e.g., in [CR06]: although its expressiveness is again very large, its natural language-like syntax makes it adoptable by non-experts. However, although CASL-LTL can express temporal behaviors, it cannot express timed behaviors (with quantitative values) in contrast to our patterns.

The idea of reducing (some) properties to reachability checking is not new: in [ABL98], safety and bounded-liveness properties are translated to test automata, equivalent to our notion of observers. Among the differences are the fact that we exhibit commonly used patterns, where as [ABL98] aims at (near to) completeness (the expressiveness of such reachability checking has been characterized in [ABBL98]). Furthermore, we do not consider properties only based on non-reachability, but also on reachability.

The word "pattern" has been used with different semantics. Our patterns share similarities with the design patterns for software engineering [GHJV95]: they aim at characterizing common correctness properties, they certainly do not aim at exhaustiveness nor at novelty, and they are not compositional. However, whereas the design patterns for software engineering can be inserted into freely written code, our patterns shall be standalone. Although different from the patterns from [DHQ+08], our patterns share with this approach the ability to define quantitative timed behaviors.

In [KMH01], typical temporal constraints dedicated to modeling scheduling problems are identified, and then translated into timed automata. In [MGT09], patterns for specifying the system correctness are defined using UML statecharts, and then translated into timed automata. As in our approach, their correctness reduces to reachability checking. The differences rely in the choice of a graphical specification in [MGT09], as well as the target formalism (timed automata only); furthermore, we exhibit common patterns based on experience on industrial models.

Outline: Section II recalls the formalisms of timed automata and stateful timed CSP, and gives a definition of observers. Section III introduces our observer patterns and, for each of them, gives an instantiation in both timed automata and stateful timed CSP. Section IV briefly discusses the implementation of the patterns in IMITATOR. Section V concludes the paper and gives perspectives. The concrete syntax implemented in IMITATOR is given in Appendix.

II. PRELIMINARIES

In this section, we briefly recall the notion of clocks and constraints (Section II-A), as well as the formalisms of timed automata (Section II-B) and stateful timed CSP (Section II-C). Finally, we propose a general definition of observers in these two formalisms (Section II-D).

A. Clocks and Constraints

Let $\mathbb{R}_+$ be the set of non-negative real numbers. We assume that X is a set of *clocks*. A clock is a variable with value in $\mathbb{R}_+$. All clocks evolve linearly at the same rate. An inequality (over X) is $e \prec e'$, where $\prec \in \{<, \leq\}$, and e, e' are two linear terms of the form $\sum_{1 \leq i \leq N} \alpha_i x_i + d$ with $x_i \in X$, $\alpha_i \in \mathbb{N}$ for $1 \leq i \leq N$, and $d \in \mathbb{N}$. A constraint (over X) is a conjunction of inequalities.

B. Timed Automata

Timed automata are finite-state automata augmented with clocks, i.e., real-valued variables increasing uniformly, that are compared within guards and invariants with timing delays [AD94].

Definition 1: A timed automaton $\mathcal{A}$ is $(\Sigma, Q, q_0, X, I, \rightarrow)$ with Σ a finite set of actions, Q a finite set of locations, $q_0 \in Q$ the initial location, X a set of clocks, I the invariant assigning to every $q \in Q$ a constraint over X, and $\rightarrow$ a step relation consisting of elements (q, g, a, ρ, q'), where $q, q' \in Q$ are the source and destination location respectively, $a \in \Sigma$ is the transition action, $\rho \subseteq X$ is the set of clocks to be reset, and the guard g is a constraint over X.

The symbolic semantics of a timed automata $\mathcal{A}$ is defined in terms of runs, i.e., alternating sequences of symbolic states and actions. Symbolic states are pairs (q, C) where $q \in Q$ and C is a constraint over X (see, e.g., [AS13]).

In practice, timed automata are often composed with each other using the parallel composition $\|$. Such a network of timed automata results in a timed automaton. We assume a (common) semantics where all automata using a given action a must fire a (local) transition labeled with a together.

C. Stateful Timed CSP

We briefly recall stateful timed CSP [SLD+13], a timed extension of timed CSP capable of specifying hierarchical real-time systems. We assume a finite set *Var* of finite-domain *variables* and a finite set Σ of *actions*. A *variable valuation* is a function assigning to each variable a value in its domain. A process P is defined by the grammar in Fig. 1, where $d \in \mathbb{N}$. $\mathcal{P}$ denotes the set of all possible processes.

Syntax	Description
$P \doteq$ `Stop`	inaction
$\mid$ `Skip`	termination
$\mid e \rightarrow P$	event prefixing
$\mid a\{program\} \rightarrow P$	data operation
$\mid P \,\Box\, Q$	external choice
$\mid P \setminus E$	hiding
$\mid P; Q$	sequential composition
$\mid P \;\vert[E]\vert\; Q$	parallel composition
$\mid$ `Wait`$[d]$	delay
$\mid P$ `timeout`$[d]$ Q	timeout
$\mid P$ `interrupt` Q	interrupt
$\mid P$ `interrupt`$[d]$ Q	timed interrupt
$\mid P$ `within`$[d]$	timed responsiveness
$\mid P$ `deadline`$[d]$	deadline
$\mid Q$	process referencing

Fig. 1. Syntax of STCSP processes

Definition 2: A stateful timed CSP model is a tuple $\mathsf{M} = (Var, V_0, P_0)$ where V_0 is the initial variable valuation, and $P_0 \in \mathcal{P}$ is a process.

Process `Stop` does nothing but idling. Process `Skip` terminates, possibly after idling for some time. Process $e \rightarrow P$ engages in action e first and then behaves as P. In order to seamlessly integrate data operations, sequential programs may be attached with actions. Process $a\{program\} \rightarrow P$ executes the sequential *program* whilst generating action a, and then behaves as P. The program may be a simple procedure

updating data variables (e.g., $a\{v_1 := 5; v_2 := 3\}$, where $v_1, v_2 \in \mathit{Var}$) or a more complicated sequential program. Process $P \,\Box\, Q$ offers an external choice[1] between P and Q. Process $P; Q$ behaves as P until P terminates and then behaves as Q immediately. $P \setminus E$ hides occurrences of events in E. Parallel composition of two processes is written as $P \;|[E]|\; Q$, where P and Q may communicate via multi-party action synchronization (following CSP rules [Hoa85]) or shared variables. We use $P \parallel Q$ for $P \;|[\Sigma(P) \cap \Sigma(Q)]|\; Q$, where $\Sigma(P)$ denotes the alphabet of actions used by P.

Process $\mathtt{Wait}[d]$ idles for d time units. In process $P\ \mathtt{timeout}[d]\ Q$, the *first* observable action of P shall occur no later than d time units. Otherwise, Q takes over the control after exactly d time units. Process $P\ \mathtt{interrupt}\ Q$ behaves exactly as P until the first observable action in Q, and then Q takes over. Process $P\ \mathtt{interrupt}[d]\ Q$ behaves exactly as P until d time units, and then Q takes over. Process $P\ \mathtt{within}[d]$ must react within d time units, i.e., an observable action must be engaged by process P within d time units. Process $P\ \mathtt{deadline}[d]$ constrains P to terminate, possibly after engaging in multiple observable events, before d time units.

D. Observers

We propose here a definition for observers, in both timed automata and stateful timed CSP. Observers are standard subsystems, with some assumptions. An observer must not have any effect on the system, and must not prevent any behavior to occur. In particular, it must not block time, nor prevent actions to occur, nor create deadlocks that would not occur otherwise. As a consequence, observers must be complete: in the example of timed automata, all actions declared by the observer must be allowed in any of the locations.

In the following, we differentiate the *original* model (i.e., the model to verify) from the *global* model (i.e., the original model plus the observer).

In the formalism of timed automata, an observer is a standard timed automaton (see Definition 1) with some restrictions:

- the observer uses (at most) one local clock x_{obs} (the case with more than one local clock could be possible, but is not used in this work), and no shared clock;
- the observer may contain a special bad location (denoted by l_b), that has no outgoing transition (except possibly self-loops);
- the observer may contain a set of good locations (denoted by l_g, l'_g, etc.).

The global model is defined as $\mathcal{A}_1 \parallel \ldots \parallel A_n \parallel \mathcal{A}_{obs}$, where $\mathcal{A}_1 \ldots \mathcal{A}_n$ are the timed automata modeling the original model, and $\mathcal{A}_{obs}$ is the observer automaton. The parallel composition ensures that the actions shared by the observer and the original model will synchronize.

In the case of stateful timed CSP, an observer is a standard process (see Definition 2) with some restrictions:

- the observer cannot refer to any existing process defined by the original model;
- the observer cannot write any existing variable used in the original model;
- the observer features two special booleans, neither read nor written by any other process defined by the original model; these booleans, denoted by v_{bad} and v_{good}, act as a flag, and will be true ("T") when a bad and a good behavior are detected, respectively – and false ("F") otherwise. We assume here that initially, we have $v_{bad} = F$ and $v_{good} = T$ (the value for v_{good} comes from the fact that, in all observers making use of v_{good}, the initial state is good).

The global model results in a new model $\mathsf{M}_{obs} = (\mathit{Var} \cup \{v_{bad}, v_{good}\}, V_0, P_0 \parallel P_{obs})$ where (Var, V_0, P_0) is the original model, and P_{obs} is the observer process. Again, the parallel composition ensures that the actions shared by the observer and the original model will synchronize. We will denote by Σ_{obs} the alphabet of the observer, i.e., the set of actions it will synchronize on. We will often make use of a special process P_S acting as a "sink", that is, accepting any action in the observer alphabet. For example, if $\Sigma_{obs} = \{a_1, a_2\}$, then $P_S = (a_1 \rightarrow P_S) \,\Box\, (a_2 \rightarrow P_S)$.

III. Observer Patterns

In this section, we introduce a library of correctness patterns, that we translate to observers, so that simple tools without complex model checking capabilities can verify them. In the following, we make the assumption that the tool is able to verify two kinds of properties. The first property is the non-reachability property, i.e., a given (bad) state is not reachable, in any of the possible runs. We assume the following abstract syntax for the verification command:

$$\mathit{assert\ unreachable}(\mathit{BAD})$$

where *BAD* is a state definition (see below).

The second property is a reachability property that requires that each run ends in a (good) state. (If the run is finite, then the notion of "end" refers to the last state; if the run is infinite, from some point, it loops with a cycle; we require all of its states to be good.) We assume the following abstract syntax:

$$\mathit{assert\ alwaysEndWith}(\mathit{GOOD})$$

where *GOOD* is a state definition. Note that both properties refer to properties that must be true (or false) for all runs (operator "A" in the CTL logic [BK08]).

We require that the notion of "state" used by some patterns must refer to the discrete part of the state only. For example, in timed automata, it is out of question to refer to the values of the system clocks when describing a state: first, it is likely that engineers may not be familiar with the formalism of timed automata; second, the timed automata may have been automatically generated (e.g., from another more user-friendly formalism), and one may not even want to have a look at their internal structure. In the following, definitions of good or bad states in timed automata will only refer to the locations of the automata; for observers, it will hence refer to its good or bad location(s). For stateful timed CSP, the notion of state will refer to the value of the variables, i.e., of the v_{bad} and v_{good} observer variables.

[1] We leave out internal, general and conditional choices (see [SLD+13]).

In the rest of this section, we propose 16 patterns organized in 7 classes. For sake of better readability, we propose a quite verbose abstract syntax. The concrete syntax can of course be refined in model checking tools (see Section IV).

A. Non-Reachability

The property of non-reachability of some bad state is by far the most common property used to characterize the correctness of real-time systems. This pattern is somehow degenerated since, for verification tools able to natively check the non-reachability of a bad state, this property does not require the use of an observer. Nevertheless, we still include it into our pattern library since it is the most common one. Given a bad state definition *BAD*, the non-reachability pattern can be described as follows:

> **Abstract syntax:**
> *assert unreachable*(BAD)
> **English description:**
> "The state *BAD* never happens."

The literature is full of such examples. Among the most common case studies, the correctness of Fischer's mutual exclusion protocol (see, e.g., a timed version in [AHV93]) is usually seen as a non-reachability property: the bad state is defined as a state where more than one process is in the critical section. Similarly, in the train crossing problem (see, e.g., a simple example in [AHV93]), the bad state corresponds to a state where the train crosses the road although the gate is still open.

B. Action Precedence

This class of patterns models the case of an action that can only happen if another one has happened before. A typical example is the case when one wants to avoid false positives for alarms: an alarm must ring only if a given action (for instance, an intrusion into a house) has happened before. (Note that this property does not mean that the intrusion will always lead to an alarm; this will be the subject of the "Eventual Response" patterns in Section III-C.)

We consider three patterns in this class. All patterns are similar, and checking their correctness always reduces to non-reachability. Since these properties are all untimed, the observers will not use any timed feature (for instance, the timed automata will be finite-state automata).

1) Acyclic Version: In this pattern, we check that if an action a_2 happens, then action a_1 has happened (at least once) before the first occurrence of a_2.

> **Abstract syntax:**
> *if* a_2 *then* a_1 *has happened before*
> **English description:**
> "If a_2 happens at least once, then a_1 has happened before the first occurrence of a_2."

Note that this pattern does not require a_2 to happen at all, even if a_1 occurred.

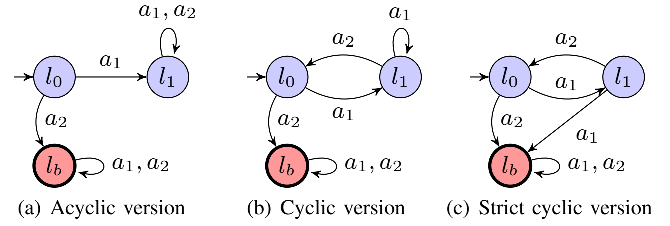


Fig. 2. Observer timed automata for "Action Precedence" patterns

We give the corresponding timed automaton observer in Fig. 2(a). The observer is simple: if a_2 occurs first, the observer enters its bad location l_b and remains there forever. If a_1 occurs, then the observer enters another sink location (l_1), not defined as bad, and remains there forever.

This observer is instantiated in stateful timed CSP below:

$$P_{obs} \doteq (a_2\{v_{bad} := T\} \to P_S) \,\square\, (a_1 \to P_S)$$

Similarly to the timed automaton observer, if a_2 occurs first, the observer sets the v_{bad} variable to T, and derives to the sink process P_S that accepts occurrences of both a_1 and a_2; otherwise, it directly derives to P_S, and v_{bad} remains false.

This pattern can be verified by a tool using the following command. For timed automata:

assert unreachable(loc[observer] = l_b*)*,

where we assume that "loc[observer]" denotes the current location of the observer automaton. In other words, the system satisfies this property if the bad location of the observer is never reached. The command for stateful timed CSP is similar:

assert unreachable($v_{bad} = T$*)*.

This verification command will be the same for all the non-reachability patterns, i.e., all patterns containing only a "bad" location (or variable), and no "good" one.

An example of use of this pattern in the hardware area is the verification over one clock cycle of a latch circuit designed by ST-Microelectronics (described in [And10]). The behavior is correct if, at the end of the clock cycle (modeled by action $CK^{\searrow}$), the output signal Q has changed before (modeled by action $Q^{\nearrow}$). Hence, the property is: *if* $CK^{\searrow}$ *then* $Q^{\nearrow}$ *has happened before*.

2) Cyclic Version: In this pattern, we check that if an action a_2 occurs, then action a_1 has happened (at least once) since the last occurrence of a_2, and so on in a cyclic manner. Again, note that this pattern does not require a_2 to happen at all, and does not guarantee that a_2 will occur an infinite number of times. Furthermore, a_1 may occur several times (at least once) between any two occurrences of a_2.

> **Abstract syntax:**
> *every time* a_2 *then* a_1 *has happened before*
> **English description:**
> "Every time a_2 happens, then a_1 has happened before, since the last occurrence of a_2 (if any)."

We give the corresponding timed automaton observer in Fig. 2(b). If a_2 occurs first, the observer enters its bad location and remains there. Then, as long as at least one occurrence of

a_1 happens between any two occurrences of a_2, the observer does not enter the bad location.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1\\
P_1 &\doteq (a_2\{v_{bad} := T\} \to P_S) \,\Box\, (a_1 \to P_2)\\
P_2 &\doteq (a_1 \to P_2) \,\Box\, (a_2 \to P_1)
\end{aligned}
$$

This process is a direct translation from the timed automaton.

An example of use of this pattern in the hardware area is the verification over an arbitrary number of clock cycles of the latch circuit mentioned above. The property is: *every time* $CK^{\searrow}$ *then* $Q^{\nearrow}$ *has happened before*.

3) Strict Cyclic Version: In this pattern, we check that if an action a_2 happens, then action a_1 has happened exactly once since the last occurrence of a_2, and so on in a cyclic manner. Again, note that this pattern does not require a_2 to happen at all, and does not guarantee that a_2 will happen an infinite number of times. In other words, this pattern requires a_1 and a_2 to alternate, starting from a_1.

Abstract syntax:
every time a_2 *then* a_1 *has happened exactly once before*
English description:
"Every time a_2 happens, then a_1 has happened before, exactly once since the last occurrence of a_2 (if any)."

We give the corresponding timed automaton observer in Fig. 2(c). If a_2 happens first, the observer enters its bad location and remains there. Then, a_1 and a_2 alternate; otherwise, the observer enters the bad location.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1\\
P_1 &\doteq (a_2\{v_{bad} := T\} \to P_S) \,\Box\, (a_1 \to P_2)\\
P_2 &\doteq (a_1\{v_{bad} := T\} \to P_S) \,\Box\, (a_2 \to P_1)
\end{aligned}
$$

Again, this process is a direct translation from the timed automaton.

C. Eventual Response

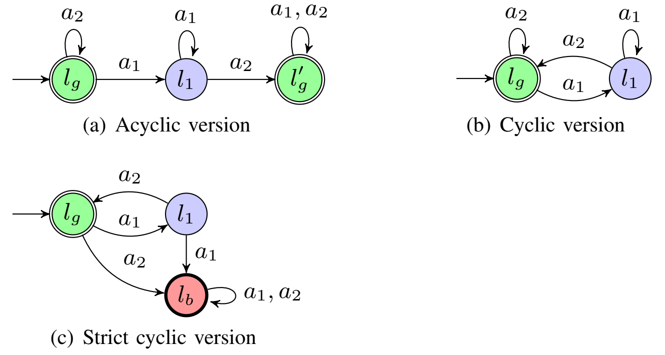
This class of properties considers the case of an action always eventually followed by another one. This class of properties is often referred to as *liveness*; however, this word has different semantics, and even model checking experts do not all agree on its meaning[2]. Here, we propose the name "eventual response" that sticks to the "eventually" operator in temporal logics.

This class of patterns is the only one in this work not to be based on non-reachability; instead, one must check that each run ends in a "good state". Again, since these properties are all untimed, the observers will not use any timed feature.

1) Acyclic Version: In this pattern, we check that if an action a_1 happens, then action a_2 eventually happens.

Abstract syntax:
if a_1 *then eventually* a_2
English description:
"If a_1 happens, then a_2 eventually happens."

[2] See, e.g., https://cs.nyu.edu/acsys/beyond-safety/liveness.htm.



(a) Acyclic version
(b) Cyclic version
(c) Strict cyclic version

Fig. 3. Observer timed automata for "Eventual Response" patterns

Note that this pattern does not require a_1 to happen. Furthermore, a_1 can happen several times before a_2 happens.

We give the corresponding timed automaton observer in Fig. 3(a). The observer starts in a good location; if a_1 never happens, it remains there. When a_1 occurs, the observer enters an intermediate location that is not good; only when a_2 occurs, the observer enters the second good location, and will remain there forever.

This observer is instantiated in stateful timed CSP below (recall that initially $v_{good} = T$):

$$
\begin{aligned}
P_{obs} &\doteq (a_2 \to P_{obs}) \,\Box\, (a_1\{v_{good} := F\} \to P_1)\\
P_1 &\doteq (a_1 \to P_1) \,\Box\, (a_2\{v_{good} := T\} \to P_S)
\end{aligned}
$$

This pattern can be verified by a tool using the following command. For timed automata:

assert alwaysEndWith(loc[observer] = l_g*).*

The command for stateful timed CSP is similar:

assert alwaysEndWith($v_{good} = T$*).*

2) Cyclic Version: In this pattern, we check that every time an action a_1 happens, then action a_2 eventually happens.

Abstract syntax:
every time a_1 *then eventually* a_2
English description:
"Every time a_1 happens, then a_2 eventually happens."

Again, this pattern does not require a_1 to happen. Furthermore, a_1 can occur several times before a_2 occurs.

We give the corresponding timed automaton observer in Fig. 3(b). This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq (a_2 \to P_{obs}) \,\Box\, (a_1\{v_{good} := F\} \to P_1)\\
P_1 &\doteq (a_1 \to P_1) \,\Box\, (a_2\{v_{good} := T\} \to P_{obs})
\end{aligned}
$$

This pattern can be verified by a tool using the same command as the acyclic version.

3) Strict Cyclic Version: In this pattern, we check that every time an action a_1 happens, then action a_2 eventually happens.

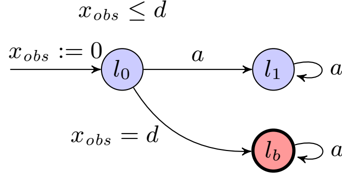


Fig. 4. Observer timed automaton for pattern "Action Before Deadline"

Abstract syntax:
every time a_1 *then eventually* a_2 *once before next* a_1
English description:
"Every time a_1 happens, then a_2 eventually happens exactly once before the next occurrence of a_1."

In this pattern, a_1 and a_2 alternate, starting with a_1 and, every time a_1 occurs, then a_2 must occur too. Again, this pattern does not require a_1 to occur at all.

We give the corresponding timed automaton observer in Fig. 3(c). This observer makes use of both the bad and the good locations. However, it is sufficient to check that the system ends in a good location. This observer is instantiated in stateful timed CSP below:

$$P_{obs} \doteq (a_2\{v_{good} := F\} \to P_S) \;\Box\; (a_1\{v_{good} := F\} \to P_1)$$
$$P_1 \;\;\doteq (a_1\{v_{good} := F\} \to P_S) \;\Box\; (a_2\{v_{good} := T\} \to P_{obs})$$

This pattern can be verified by a tool using the same command as the previous versions of this class.

This class of patterns is among the most common properties met in practice, in particular the cyclic and strict cyclic versions. To quote two examples, in Fischer's mutual exclusion protocol [AHV93], one may want to specify that "every time that access is requested, then access is eventually granted" (cyclic version). In the multi-lift system modeled in [SLD+13], one may want to specify that "every time the doors open then eventually the doors close before they open again" (strict cyclic version).

D. Action Before Deadline

This pattern models the case of an action that must occur no later than a given amount of time following the start of the system. This pattern can be seen as a subcase of pattern "Time-Bounded Action Precedence: Acyclic Version" (that will be introduced in Section III-E).

Abstract syntax:
a *no later than* d
English description:
"a will happen no later than d units of time after the system start."

We give the corresponding timed automaton observer in Fig. 4. The observer clock x_{obs} is initially set to 0. Then, if a occurs before d units of time (modeled by l_0's invariant $x_{obs} \leq d$), it enters l_1 where it will remain forever. But if a does not occur within d units of time, the observer enters the bad location.

This observer is instantiated in stateful timed CSP below:

$$P_{obs} \doteq (a \to P_S) \;\texttt{timeout}[d] \;\big((e_{obs}\{v_{bad} := T\} \to P_S) \setminus \{e_{obs}\}\big)$$

Process P_{obs} waits for an occurrence of a; if it occurs, it derives to the sink process P_S. Otherwise, timeout occurs: variable v_{bad} is set to true, and an internal action e_{obs} (local to P_{obs}) is fired, and the process then derives to the sink process. The internal action is hidden (" $\setminus \{e_{obs}\}$") to prevent any visible behavior from outside.

For example, the SPSMALL memory, designed and commercialized by chipset manufacturer ST-Microelectronics, has been modeled and verified using a network of timed automata [CEFX09]. Here, the raise of the output signal Q (action $Q^{\nearrow}$) must occur within a given amount of time after the system start.

E. Time-Bounded Action Precedence

This class of patterns models the case of an action that can only occur if another one has happened within a given interval of time before. This is a timed extension of the class of patterns "Action Precedence". To use again the example of alarms, an alarm must ring only if an intrusion has occurred within 5 seconds before. (Again, this property does not mean that the intrusion will always lead to an alarm within a given interval of time; this will be the subject of the "Time-Bounded Response" patterns in Section III-F.)

1) Acyclic Version: Here, we check that if an action a_2 happens, then action a_1 has happened (at least once) before the first occurrence of a_2 within the past d units of time.

Abstract syntax:
if a_2 *then* a_1 *has happened at most* d *units of time before*
English description:
"If a_2 happens at least once, then a_1 has happened at most d units of time before the first occurrence of a_2."

Note that this pattern does not require a_2 to happen at all, even if a_1 does.

We give the corresponding timed automaton observer in Fig. 5(a). The edge from l_0 to l_b models the fact that a_2 cannot occur first. Then, when a_1 occurs, clock x_{obs} is initialized. The rest of the automaton is similar to pattern "Action Before Deadline" (Fig. 4).

This observer is instantiated in stateful timed CSP below:

$$P_{obs} \doteq (a_2\{v_{bad} := T\} \to P_S) \;\Box\; (a_1 \to P_2)$$
$$P_2 \;\;\doteq \big((a_1 \to P_2) \;\Box\; (a_2 \to P_S)\big) \;\texttt{timeout}[d] \;\big((a_1 \to P_2) \;\Box\; (a_2\{v_{bad} := T\} \to P_S)\big)$$

2) Cyclic Version: In this pattern, we check that every time an action a_2 happens, then action a_1 has happened (at least once) before the latest occurrence of a_2.

Abstract syntax:
every time a_2 *then* a_1 *has happened at most* d *units of time before*
English description:
"Every time a_2 happens, then a_1 has happened at most d units of time before, and at least once since the latest occurrence of a_2 (if any)."

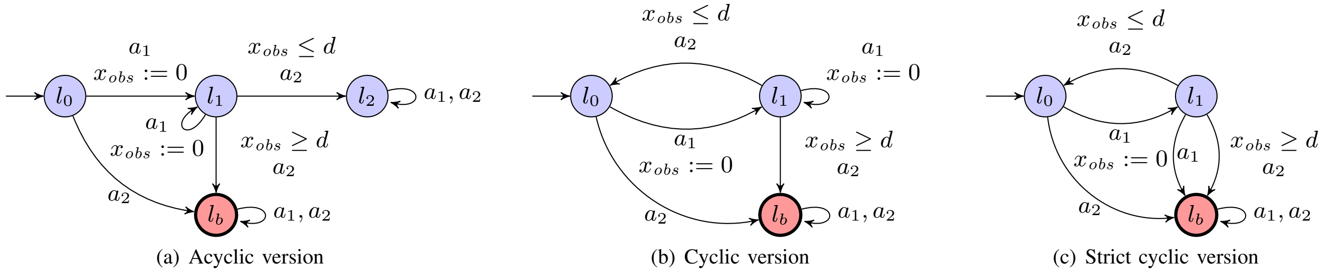


Fig. 5. Observer timed automata for "Time-Bounded Action Precedence" patterns

Note that this pattern does not require a_2 to occur at all, even if a_1 does.

We give the corresponding timed automaton observer in Fig. 5(b). The edge from l_0 to l_b models the fact that a_2 cannot occur first. Then, every time a_1 occurs, clock x_{obs} is initialized. If a_2 occurs at most d units of time later (guard $x_{obs} \leq d$) then the system continues. Otherwise (guard $x_{obs} \geq d$), the observer enters the bad location.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1 \\
P_1 &\doteq (a_2\{v_{bad} := T\} \to P_S) \,\square\, (a_1 \to P_2) \\
P_2 &\doteq \big((a_1 \to P_2) \,\square\, (a_2 \to P_1)\big) \ \texttt{timeout}[d] \\
&\quad \big((a_1 \to P_2) \,\square\, (a_2\{v_{bad} := T\} \to P_S)\big)
\end{aligned}
$$

3) Strict Cyclic Version: In this pattern, we check that every time an action a_2 happens, then action a_1 has happened exactly once before the latest occurrence of a_2, and at most d units of time since this latest occurrence.

Abstract syntax:
every time a_2 then a_1 has happened exactly once at most d units of time before
English description:
"Every time a_2 happens, then a_1 has happened at most d units of time before, and exactly once since the last occurrence of a_2 (if any)."

In other words, a_1 and a_2 alternate, starting with a_1, and with at most d units of time between a_1 and a_2. Note that this pattern does not require a_2 to occur at all, even if a_1 does.

We give the corresponding timed automaton observer in Fig. 5(c). This pattern is identical to the one in Fig. 5(b) with the difference that a_1 cannot occur twice in a row, which explains the left-hand edge (labeled with a_1) from l_1 to l_b.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1 \\
P_1 &\doteq (a_2\{v_{bad} := T\} \to P_S) \,\square\, (a_1 \to P_2) \\
P_2 &\doteq \big((a_1\{v_{bad} := T\} \to P_S) \,\square\, (a_2 \to P_1)\big) \ \texttt{timeout}[d] \\
&\quad \big((e_{obs}\{v_{bad} := T\} \to P_S) \setminus \{e_{obs}\}\big)
\end{aligned}
$$

F. Time-Bounded Response

This class of properties considers the case of an action always eventually followed by another one within some interval of time. It can be seen as a timed extension of the "Eventual Response" pattern. Again, the name of this pattern is subject to some debates. Another classical name is "Time-Bounded Liveness" (used, e.g., in [BLR05]).

1) Acyclic Version: In this pattern, we model that if an action a_1 happens, then action a_2 will happen within d units of time.

Abstract syntax:
if a_1 then eventually a_2 within d
English description:
"If a_1 happens, then a_2 will eventually happen within d units of time."

Again, this pattern does not require a_1 to happen at all, even if a_2 does.

We give the corresponding timed automaton observer in Fig. 6(a). At first, a_2 can occur anytime. Then, once a_1 occurs, the clock x_{obs} is initialized. If a_2 occurs within d units of time (guaranteed by invariant $x_{obs} \leq d$) then the system enters a sink state, and both a_1 and a_2 can occur anytime. Otherwise, after d units of time, the system enters the bad location, and remains there.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq (a_2 \to P_{obs}) \,\square\, (a_1 \to P_2) \\
P_1 &\doteq (a_1 \to P_1) \\
P_2 &\doteq \big((P_1 \ \texttt{interrupt}\ a_2 \to \texttt{Skip})\ \texttt{deadline}[d]; P_S\big) \\
&\quad \square\, \big((P_1 \ \texttt{interrupt}[d]\ e_{obs}\{v_{bad} := T\} \to P_S) \\
&\quad \setminus \{e_{obs}\}\big)
\end{aligned}
$$

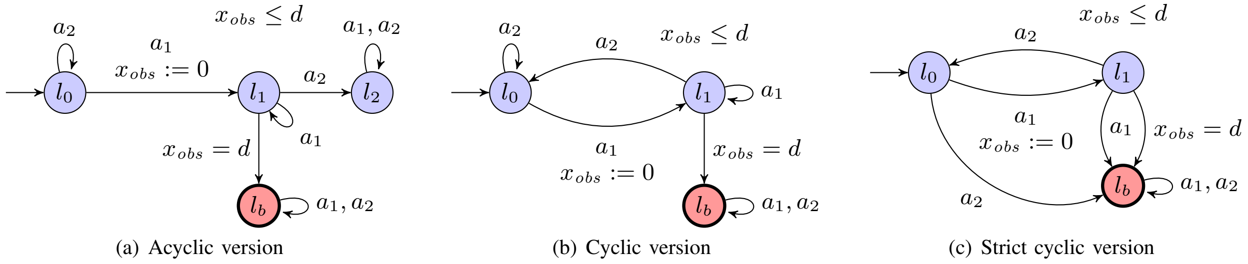
2) Cyclic Version: Here, we model that every time an action a_1 happens, then action a_2 will happen within d units of time.

Abstract syntax:
every time a_1 then eventually a_2 within d
English description:
"Every time a_1 happens, then a_2 will eventually happen within d units of time, and before the next occurrence of a_1 (if any)."

Again, this pattern does not require a_1 to occur at all, even if a_2 does.

We give the corresponding timed automaton observer in Fig. 6(b). The observer is very close to the acyclic pattern (Fig. 6(a)) with the addition of the back edge from l_1 to l_0.

This observer is instantiated in stateful timed CSP below:



(a) Acyclic version (b) Cyclic version (c) Strict cyclic version

Fig. 6. Observer timed automata for "Time-Bounded Response" patterns

$$
\begin{aligned}
P_{obs} &\doteq (a_2 \rightarrow P_{obs}) \,\square\, (a_1 \rightarrow P_2)\\
P_1 &\doteq (a_1 \rightarrow P_1)\\
P_2 &\doteq \big((P_1 \ \mathtt{interrupt}\ a_2 \rightarrow \mathtt{Skip})\ \mathtt{deadline}[d]; P_{obs}\big)\\
&\quad \square\, \big((P_1 \ \mathtt{interrupt}[d]\ e_{obs}\{v_{bad} := T\} \rightarrow P_S)\\
&\quad \setminus \{e_{obs}\}\big)
\end{aligned}
$$

3) Strict Cyclic Version: Here, we model that every time an action a_1 happens, then action a_2 will happen within d units of time.

> **Abstract syntax:**
> *every time* a_1 *then eventually* a_2 *within* d *once before next* a_1
> **English description:**
> "Every time a_1 happens, then a_2 will eventually happen within d units of time, exactly once, and before the next occurrence of a_1 (if any)."

In other words, a_1 and a_2 alternate, starting with a_1 and, every time a_1 occurs, then a_2 must occur too within d units of time. Again, this pattern does not require a_1 to occur at all.

We give the corresponding timed automaton observer in Fig. 6(c). The observer is very close to the acyclic pattern (Fig. 6(b)) with the addition of the left-hand edge from l_1 to l_b, corresponding to a second a_1 in a row.

This observer is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1\\
P_1 &\doteq (a_1 \rightarrow P_2) \,\square\, (a_2\{v_{bad} := T\} \rightarrow P_S)\\
P_2 &\doteq \big((a_1\{v_{bad} := T\} \rightarrow P_S) \,\square\, (a_2 \rightarrow P_1)\big)\ \mathtt{timeout}[d]\\
&\qquad \big((e_{obs}\{v_{bad} := T\} \rightarrow P_a) \setminus \{e_{obs}\}\big)
\end{aligned}
$$

Again, this class of correctness patterns is very common in practice. An example of use of this pattern in hardware verification is the flip-flop circuit (see [CC07, Fig. 16]). In this circuit verified over one single clock cycle, signal Q must change at most $T_{CK \rightarrow Q}$ units of time after signal CK's rising edge (this corresponds to the acyclic version of "Time-Bounded Response"). In the bounded retransmission protocol (see, e.g., [DKRT97]), one specifies that premature time-outs are not possible, i.e., a message must not come after the timer expires: hence, when the timer starts, the message must eventually arrive before d units of time, where d is the timer expiration time, and before the next timer start (cyclic version). This pattern is also common in scheduling problems (see, e.g., [FLMS12] in the setting of parametric schedulability analysis): in scheduling problems, each job i must finish within d_i time units, i.e., each action corresponding to the job start must be followed by the job end within d_i time units before it starts again (strict cyclic version). The same pattern applies again to characterize the global system execution time.

G. Sequence

This class of 2 patterns models sequences of actions. By sequence, we mean n actions followed by each other in a predefined order. The two patterns are the aylclic version, where the sequence must occur once (after which the behavior is free) and the cyclic version.

1) Acyclic Version:

> **Abstract syntax:**
> *sequence* $a_1, \dots, a_n$
> **English description:**
> "Actions $a_1, \dots, a_n$ must occur in this order."

Again, this pattern imposes an order, but does require the actions to happen. The sequence may also stop in the middle.

We give the corresponding timed automaton observer in Fig. 7(a). For sake of saving space, we denote by $\Sigma_{obs}^{\setminus i}$ the alphabet of the observer except action a_i, that is $\Sigma_{obs}^{\setminus i} = \Sigma_{obs} \setminus \{a_i\}$. Basically, as soon as an action non-conform to the predefined order occurs, the observer goes into the bad location.

This pattern is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1\\
P_1 &\doteq (a_1 \rightarrow P_2) \,\square\, (a_2\{v_{bad} := T\} \rightarrow P_S) \,\square\, \dots\\
&\qquad \square\, (a_n\{v_{bad} := T\} \rightarrow P_S)\\
P_2 &\doteq (a_1\{v_{bad} := T\} \rightarrow P_S) \,\square\, (a_2 \rightarrow P_3) \,\square\, \dots\\
&\qquad \square\, (a_n\{v_{bad} := T\} \rightarrow P_S)\\
&\dots\\
P_n &\doteq (a_1\{v_{bad} := T\} \rightarrow P_S) \,\square\, \dots \,\square\\
&\qquad (a_{n-1}\{v_{bad} := T\} \rightarrow P_S) \,\square\, (a_n \rightarrow P_f)\\
P_f &\doteq P_S
\end{aligned}
$$

2) Cyclic Version:

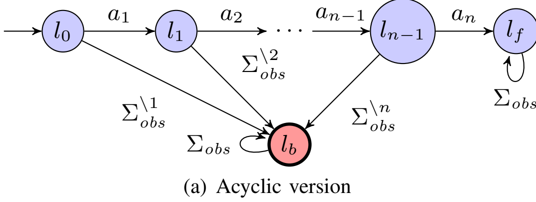
> **Abstract syntax:**
> *always sequence* $a_1, \dots, a_n$
> **English description:**
> "Actions $a_1, \dots, a_n, a_1, \dots$ must occur in this order."

Again, this pattern imposes an order, but does require the actions to happen. The sequence may also not be infinite.

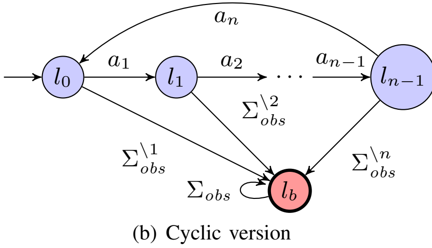
We give the corresponding timed automaton observer in Fig. 7(b). We use the same notations as for Fig. 7(a).

This pattern is instantiated in stateful timed CSP below:

$$
\begin{aligned}
P_{obs} &\doteq P_1 \\
P_1 &\doteq (a_1 \rightarrow P_2) \Box (a_2\{v_{bad} := T\} \rightarrow P_S) \Box \ldots \\
&\quad \Box (a_n\{v_{bad} := T\} \rightarrow P_S) \\
P_2 &\doteq (a_1\{v_{bad} := T\} \rightarrow P_S) \Box (a_2 \rightarrow P_3) \Box \ldots \\
&\quad \Box (a_n\{v_{bad} := T\} \rightarrow P_S) \\
&\ldots \\
P_n &\doteq (a_1\{v_{bad} := T\} \rightarrow P_S) \Box \ldots \Box \\
&\quad (a_{n-1}\{v_{bad} := T\} \rightarrow P_S) \Box (a_n \rightarrow P_1)
\end{aligned}
$$



(a) Acyclic version



(b) Cyclic version

Fig. 7. Observer timed automata for "Sequence" patterns

For example, in the "And–Or" circuit of [CC05, Figure 3], the system behavior consists in an alternating sequence of signal changes. Any other order is considered to be wrong.

IV. Implementation

A. Motivation

These patterns have been implemented in IMITATOR [AFKS12]. This tool performs parameter synthesis for a parametric extension of timed automata [AHV93], where constants in guards and invariants can be replaced with parameters (i.e., unknown constants). One of IMITATOR's features is to be able to cover a bounded parametric subspace with constraints (or tiles); this gives a *behavioral cartography* of the system [AS13]. It has been shown that the obtained tiling does not depend on any property. However, splitting the tiles into good and bad tiles does depend on a given property to be checked. Although it has been first performed in a manual manner (for small systems) or in a semi-automatic manner (using scripts) for larger systems, we finally went for an automated property-checking implemented natively in IMITATOR. An interesting advantage of an integrated bad-state oriented approach is that, as soon as a bad state is reached, a tile can be classified as bad without performing the whole analysis (under certain assumptions).

B. Patterns Syntax

When computing the behavioral cartography, IMITATOR requires the definition of the property for splitting tiles between good and bad. The syntax is: `property := [PROP_DEF]`, where "`PROP_DEF`" corresponds to the concrete syntax of one of the patterns. The concrete syntax implemented in IMITATOR is almost the same as the abstract syntax (see Appendix A). We believe that such a verbose syntax helps to be more readable by non-experts in formal methods.

Then, IMITATOR translates the correctness pattern into an observer automaton, following the rules of Section III, and launches the corresponding verification command (either *unreachable*, or *alwaysEndWith*).

The latest version of IMITATOR (2.6), implementing these patterns, can be found in IMITATOR Web page[3].

V. Final Remarks

We propose here a set of patterns commonly used to specify the correctness of complex real-time systems. The main advantages are as follows. First, the engineer non-expert in formal methods can easily specify the system correctness from a library of common properties expressed in an intuitive language. Second, the tool developer can easily verify these properties based on the sole (non-)reachability algorithm.

This work has been partially motivated by problems of parameter synthesis and extends in a straightforward manner to parametric real-time systems. In such systems, the constants appearing in the model are unknown, and called parameters. The synthesis problem consists in finding values for these parameters such that the system behaves well according to a given property. The observer patterns implemented in IMITATOR all allow the use of parameters instead of constants. In particular, one can use a parameterized deadline in the observer, and hence synthesize values for this deadline such that the system behaves well. This approach allows a double-parameterized verification, where both the model and the property are parameterized (as in, e.g., [BR07]).

So far, only patterns expressing properties that must be true (or false) for all runs (operator "A" in the CTL logic [BK08]) have been considered, because they are by far the most commonly found in the large set of case studies we considered. Extending our set of patterns (e.g., to the "E" operator) is the subject of future work.

Different from patterns in [DHQ+08], our patterns shall not be composed. Compositional patterns could be designed without theoretical problem; for example, the "sequence" pattern could be seen as the composition of n simpler sequence patterns allowing only 2 actions. Or the "Time-Bounded Response" can be seen as the composition of "Eventual Response" with a new "Deadline" pattern. Structuring our patterns can be an interesting direction of future research. Nevertheless, we see it as somehow dangerous in some cases, because our goal is precisely to discard behaviors complicated or rarely met in practice, that could be allowed by arbitrary compositions of patterns. Most importantly, we found very few examples of systems that would require compositional observer patterns, or in which case they consist in very simple compositions. An example is the industrial networked automaton system of [ACD+09], where the correctness is defined as a sequence of 3 actions (pattern "Sequence (acyclic)"),

[3] http://www.lsv.ens-cachan.fr/Software/imitator/

itself eventually followed by a fourth action within a deadline (pattern “Time-Bounded Response (acyclic)”).

Still, a conjunction of patterns could be useful in the case when one wants to verify several properties. In the case of our patterns, it would be interesting to set up a mechanism such that different properties could be checked at the same time, using a single definition of bad state. Hence, if the bad state is not reachable, all properties are true.

Among the future directions of research is the extension of our library of patterns to hybrid systems. In hybrid systems, clocks are generalized to continuous variables, that can have a (possibly nonlinear) flow. Although our patterns extend in a straightforward manner to hybrid systems, they may not be sufficient to capture common correctness properties. Indeed, the correctness of hybrid systems is often specified in terms of variables being in between certain bounds, e.g., the level of a water tank must remain above some minimum and below some maximum. Designing correctness patterns for probabilistic extensions of real-time systems is also the subject of future work.

Acknowledgement

I am grateful to Sun Jun for his kind help when modeling some of the patterns using stateful timed CSP.

References

[ABBL98] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. The power of reachability testing for timed automata. In Vikraman Arvind and Ramaswamy Ramanujam, editors, *FSTTCS'98*, volume 1530 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 1998. 2

[ABL98] Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In *TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 1998. 2

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. 1

[ACD+09] Étienne André, Thomas Chatain, Olivier De Smet, Laurent Fribourg, and Silvain Ruel. Synthèse de contraintes temporisées pour une architecture d'automatisation en réseau. In *MSR'09*, volume 43 of *Journal Européen des Systèmes Automatisés*, pages 1049–1064. Hermès, 2009. 9

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 1, 2

[AFKS12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM'12*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, 2012. 1, 9

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *STOC'93*, pages 592–601. ACM, 1993. 4, 6, 9

[And10] Étienne André. Synthesizing parametric constraints on various case studies using IMITATOR II. Research Report LSV-10-21, Laboratoire Spécification et Vérification, ENS Cachan, France, 2010. 4

[AS13] Étienne André and Romain Soulat. *The Inverse Method*. FOCUS Series in Computer Engineering and Information Technology. ISTE Ltd and John Wiley & Sons Inc., 2013. 2, 9

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. 1, 3, 9

[BLR05] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In *FORMATS'05*, volume 3829 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2005. 7

[BP99] Patricia Bouyer and Antoine Petit. Decomposition and composition of timed automata. In *ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 210–219. Springer, 1999. 1

[BR07] Véronique Bruyère and Jean-François Raskin. Real-time model-checking: Parameters everywhere. *Logical Methods in Computer Science*, 3(1:7), 2007. 9

[CC05] Robert Clarisó and Jordi Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *ACSD'05*, pages 122–131. IEEE Computer Society, 2005. 9

[CC07] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64(1):115–139, 2007. 8

[CEFX09] Rémy Chevallier, Emmanuelle Encrenaz-Tiphène, Laurent Fribourg, and Weiwen Xu. Timed verification of the generic architecture of a memory circuit using parametric timed automata. *Formal Methods in System Design*, 34(1):59–81, 2009. 6

[CR06] Christine Choppy and Gianna Reggio. A formally grounded software specification method. *Journal of Logic and Algebraic Programming*, 67(1-2):52–86, 2006. 2

[DHQ+08] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008. 1, 2, 9

[DKRT97] Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 1997. 8

[DLL+10] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In *ATVA'10*, volume 6252 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2010. 1

[FLMS12] Laurent Fribourg, David Lesens, Pierre Moro, and Romain Soulat. Robustness analysis for scheduling problems using the inverse method. In *TIME'12*, pages 73–80. IEEE Computer Society Press, 2012. 8

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. 2

[HO02] Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In *iFM'02*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer, 2002. 1

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. 1, 3

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009. 1

[KMH01] Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping temporal planning constraints into timed automata. In *TIME'01*, pages 21–27. IEEE Computer Society, 2001. 2

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. 1

[MD99] Brendan P. Mahony and Jin Song Dong. Overview of the semantics of TCOZ. In *iFM'99*, pages 66–85. Springer, 1999. 1

[Mer74] Philip Meir Merlin. *A study of the recoverability of computing systems*. PhD thesis, University of California, Irvine, 1974. 1

[MGT09] Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS'09*, pages 112–124. British Computer Society, 2009. 2

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57. IEEE Computer Society, 1977. 1

[RAC03] Gianna Reggio, Egidio Astesiano, and Christine Choppy. CASL-LTL : A CASL extension for dynamic reactive systems version 1.0– summary. Technical Report of DISI, 2003. 1

[SLD+13] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3.1–3.29, 2013. 1, 2, 3, 6

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *CAV'09*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009. 1

Appendix

A. *Patterns Syntax in* Imitator

In the following, `a`, `a1`, `a2`, `...`, `an` must be actions declared in the model. The deadline `d` can be either a constant linear expression or a parameter, hence allowing in the latter case for synthesis of the deadline parameter.

Pattern "1. Non-reachability":
```
property := unreachable [BAD_DEF]
```

Pattern "2a. Action Precedence (Acyclic)":
```
property := if a2 then a1 has happened before
```

Pattern "2b. Action Precedence (Cyclic)":
```
property := everytime a2 then a1 has happened before
```

Pattern "2c. Action Precedence (Strict Cyclic)":
```
property := everytime a2 then a1 has happened once
before
```

Pattern "3a. Eventual Response (Acyclic)":
```
property := if a1 then eventually a2
```

Pattern "3b. Eventual Response (Cyclic)":
```
property := everytime a1 then eventually a2
```

Pattern "3c. Eventual Response (Strict Cyclic)":
```
property := everytime a1 then eventually a2 once
before next
```

Pattern "4. Action Before Deadline":
```
property := a within d
```

Pattern "5a. Time-Bounded Action Precedence (Acyclic)":
```
property := if a2 then a1 has happened within d
before
```

Pattern "5b. Time-Bounded Action Precedence (Cyclic)":
```
property := everytime a2 then a1 has happened within
d before
```

Pattern "5c. Time-Bounded Action Precedence (Strict Cyclic)":
```
property := everytime a2 then a1 has happened once
within d before
```

Pattern "6a. Time-Bounded Response (Acyclic)":
```
property := if a1 then eventually a2 within d
```

Pattern "6b. Time-Bounded Response (Cyclic)":
```
property := everytime a1 then eventually a2 within d
```

Pattern "6c. Time-Bounded Response (Strict Cyclic)":
```
property := if a1 then eventually a2 within d once
before next
```

Pattern "7a. Sequence (Acyclic)":
```
property := sequence a1, ..., an
```

Pattern "7b. Sequence (Cyclic)":
```
property := always sequence a1, ..., an
```