# Unifying Patterns for Modelling Timed Relationships in Systems and Properties

Étienne André and Laure Petrucci

LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, France

**Abstract.** Specifying the correctness of complex concurrent and real-time systems is a crucial problem. Many property languages have been proposed to do so; however, these techniques often involve formalisms not easily handled by engineers, and furthermore require dedicated tools. We propose here a set of patterns that encode common specification or verification components when dealing with concurrent real-time systems. We provide a formal semantics for these patterns, as time Petri nets, and show that they can encode previous approaches.

## 1 Introduction

In the past few decades, many formal languages for specifying and verifying complex concurrent and real-time systems have been proposed. However, these formalisms are not always easy to handle by industry engineers. Temporal logics (*e.g.* [14,6]) offer a very powerful way of expressing correctness properties for concurrent systems but they are often considered too complicated (and maybe too rich as well) to be widely adopted by engineers. Furthermore, they generally need advanced tools dedicated to model checking.

To overcome these difficulties, several pattern-based solutions have been proposed. Patterns allow to identify frequent components of systems or properties in a standardised manner and (sometimes) to compose them so as to build more complex components. Here, we unify three sets of patterns proposed in the past.

In [11], patterns are proposed for modelling scheduling problems; they are then translated into timed automata [4]. In [7], tasks scheduling for operational planning is tackled. A coloured Petri net [10] model is then derived. Both works address the specification of systems [11,7] whereas [5] proposes real-time patterns for verification. These patterns are non-compositional, and do not aim at exhaustiveness; on the contrary, they correspond to common correctness issues met in the literature and in industrial case studies. They are translated to both timed automata [4] and Stateful timed CSP [15] , and their verification reduces to simple reachability checking.

*Contribution* In this paper, we unify previous approaches to propose a pattern-based language for the specification of real-time systems and/or properties for their verification. Each pattern has a syntax as human-readable as possible, so that engineers non-experts in formal methods can use them. Furthermore, we propose a Time Petri Net [13] semantics of these patterns for both system models and properties. For verification, the patterns are thus translated into pure reachability properties using simple observers, *i.e.* additional subsystems that observe some system actions using synchronisation and may also use time. Hence, their verification in practice avoids the use of complex verification algorithms or dedicated tools, and tool developers can implement them at little cost.

Our patterns can be used for two distinct purposes:

1. specify a system, by means of simple English-like constructs, rather than using complex formalisms. Nevertheless the translation of our patterns into time Petri nets provides a formal model of the system.
2. verify a system (not necessarily specified by our patterns), by means of the same syntax. In this situation, our patterns are again translated into time Petri nets, and can be used to verify the system model by synchronisation on transitions, and using the sole reachability of some "bad" place. This avoids the use of complex model checking algorithms.

Even though the syntax is identical for both purposes, the translation into time Petri nets for verification contains a few more places and transitions.

*Related Work* Concerning the specification of properties for verifying real-time systems, temporal logics (*e.g.* [14,6]) and their timed extensions (*e.g.* [3] among others) are by far the most commonly used, although many other formalisms have been proposed. Much more expressive than our patterns, temporal logics are more difficult to handle by non-experts. Furthermore, many tools do not actually support their full expressiveness, but only some fragments.

The idea of reducing (some) properties to reachability checking is not new: in [2], safety and bounded-liveness properties are translated into test automata, equivalent to our notion of observers. Among the differences are *i*) the fact that we do not only verify but also specify systems using our patterns, and *ii*) the fact that (as in [5]) we exhibit commonly used patterns, whereas [2] aims at completeness (the expressiveness of such reachability checking has been characterised in [1]).

In [11], typical temporal constraints dedicated to modelling scheduling problems are identified, and then translated into timed automata. In [12], 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. Differences include the choice of the target formalism (time Petri nets for us) and the fact that our patterns can encode either the system or its correctness property.

*Outline* First, Section 2 recalls the earlier definitions of patterns we base on. We then propose our new set of patterns in Section 3, and formalise them using time Petri nets in Section 4. The patterns of [11,7,5] are encoded using our unified patterns in Section 5. Finally, Section 6 concludes and gives perspectives for future work.

## 2 Earlier Works

Earlier works we base on ([11,7,5]) are concerned on the one hand with causality or timing relations between events, and on the other hand with patterns to encode behavioural properties. Hence they address altogether different aspects (model or property) that are nevertheless very intertwined.

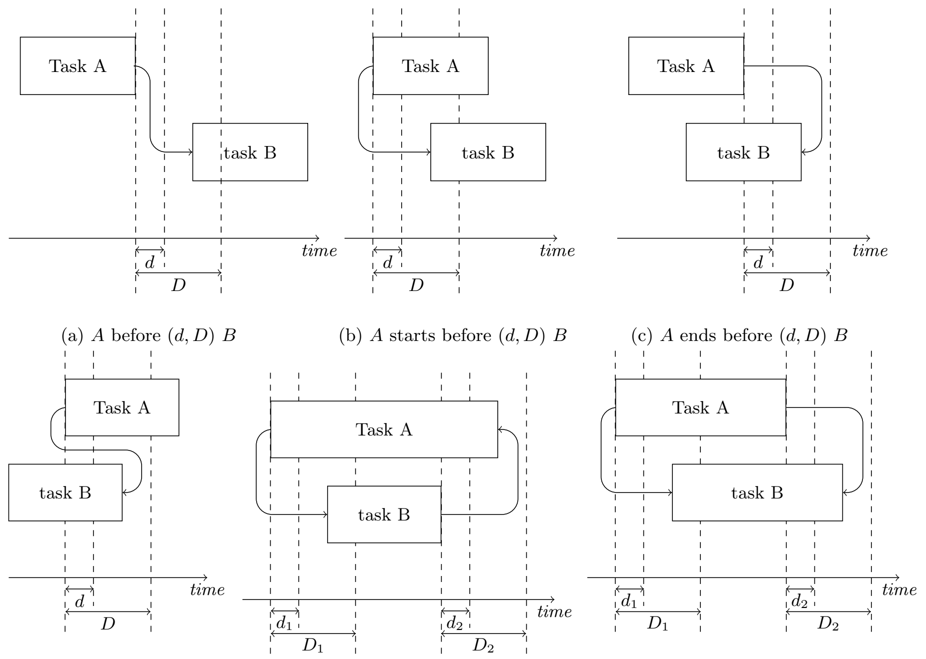
### 2.1 Scheduling Patterns [11]

In [11], planning constraints are mapped into timed automata, using elementary rules. A set of 17 interval-based temporal relations is defined. As in [11], we group these 17 patterns in 6 categories.

The temporal relations [11] allow for stating timing constraints between tasks. These tasks are composed of two events, that are the start and the end of the task. Time is often specified as an interval $(d, D)$, to express that an event happens between $d$ and $D$ units of time w.r.t. its reference.

The temporal relations from [11] are the following ones (the words in brackets are added in order to improve readability). The major ones (one per category) are represented in Fig. 1.

1. "$A$ [ends] before $(d, D)$ $B$ [starts]" (Fig. 1a)
2. "$A$ meets $B$" [*i.e.* $A$ ends exactly when $B$ starts] (degenerated case of rule 1 with $d = D = 0$)
3. "$B$ [starts] after $(d, D)$ $A$ [ends]" (inverse temporal relation of rule 1)
4. "$B$ met by $A$" [*i.e.* $A$ ends exactly when $B$ starts] (degenerated case of rule 3 with $d = D = 0$, and inverse of 2)
5. "$A$ starts before $(d, D)$ $B$ [starts]" (Fig. 1b)
6. "$A$ starts $B$" [*i.e.* $A$ starts exactly when $B$ starts] (degenerated case of rule 5 with $d = D = 0$)
7. "$B$ starts after $(d, D)$ $A$ [starts]" (inverse temporal relation of rule 5)
8. "$A$ ends before $(d, D)$ $B$ [ends]" (Fig. 1c)
9. "$A$ ends $B$" [*i.e.* $A$ ends exactly when $B$ ends] (degenerated case of rule 8 with $d = D = 0$)
10. "$B$ ends after $(d, D)$ $A$ [ends]" (inverse temporal relation of rule 8)
11. "$A$ starts_before_end $(d, D)$ $B$", *i.e.* $A$ starts $(d, D)$ time units before $B$ ends (Fig. 1d)
12. "$T_2$ ends_after_start $(d, D)$ $T_1$", *i.e.* $B$ ends $(d, D)$ time units after $A$ starts (inverse temporal relation of rule 11)
13. "$A$ contains $((d_1, D_1)(d_2, D_2))$ $B$", *i.e.* $A$ starts $(d_1, D_1)$ time units before $B$ starts, and $A$ ends $(d_2, D_2)$ time units after $B$ ends (Fig. 1e)
14. "$B$ contained_by $((d_1, D_1)(d_2, D_2))$ $A$" (inverse temporal relation of rule 13)
15. "$A$ equals $B$" (degenerated case of rule 13 with $d_1 = D_1 = d_2 = D_2 = 0$)
16. "$A$ parallels $((d_1, D_1)(d_2, D_2))$ $B$", *i.e.* $A$ starts $(d_1, D_1)$ time units before $B$ starts, and $A$ ends $(d_2, D_2)$ time units before $B$ ends (Fig. 1f)
17. "$B$ paralleled_by $((d_1, D_1)(d_2, D_2))$ $A$" (inverse temporal relation of rule 16)



(a) $A$ before $(d, D)$ $B$ (b) $A$ starts before $(d, D)$ $B$ (c) $A$ ends before $(d, D)$ $B$

(d) $A$ starts_before_end $(d, D)$ $B$ (e) $A$ contains $((d_1, D_1)(d_2, D_2))$ $B$ (f) $A$ parallels $((d_1, D_1)(d_2, D_2))$ $B$

Fig. 1: Some temporal planning constraints

## 2.2 Patterns for Operational Planning [7]

As in [11], [7] was concerned with tasks scheduling for operational planning, each task having a beginning and an end. Both of these can be timely related to those of another task, as stated in the grammar and figures below.

```
// synchronisation between tasks
synch = BilateralSynch | UnilateralSynch
// lists of tasks
Tasks = task | Tasks, task
```

The synchronisation between tasks can be either bilateral (the execution of any of them is related to the execution of the others), or unilateral (the execution of a task is related to that of the other one, but the converse is not true).

In other words, in a bilateral synchronisation, all tasks occur or none of them does. Note that in [7] tasks can synchronise either at their beginning (*i.e.* they start together) or at their end (*i.e.* they finish at the same time). In this paper, we will be interested in events (beginning or end) and not in a full task. Therefore, bilateral synchronisation only specifies a set of interrelated events.

```
BilateralSynch = BILATERALSYNCH(Tasks)
```

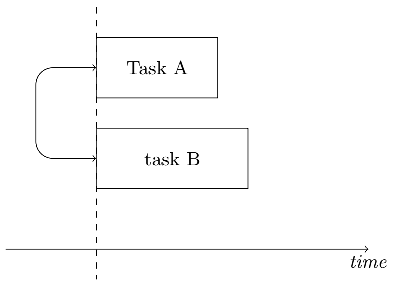


Fig. 2: Bilateral synchronisation between tasks A and B, on their start date.

Fig. 2 depicts the bilateral synchronisation `BILATERALSYNCH(A,B)`.

When the synchronisation is unilateral, a task occurs w.r.t. either a timing or another task, possibly with a delay.

```
UnilateralSynch = Relation(task1, task2, delay)
                | Relation(task, delay)
Relation = AFTER | BEFORE | AT
```

Examples in Fig. 3 depict the following relations:

(a) `AT(A,10)`: task A starts at time 10;
(b) `AFTER(A,10)`: task A starts after time 10;
(c) `AT(A,B,15)`: task A begins 15 units of time after the end of task B;
(d) `AFTER(A,B,10)`: task A begins at least 10 units of time after the end of task B.

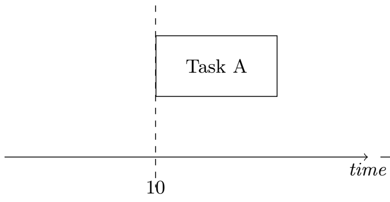
## 2.3 Observer Patterns for Real-Time Systems

In [5], we proposed a set of observer patterns encoding common properties encountered when verifying concurrent real-time systems. These patterns are based on observers, hence can be translated into pure reachability problems, thus avoiding the use of complex verification algorithms. These patterns are non-compositional, do not aim at completeness, but rather at exhibiting common properties met in the case studies of the literature.

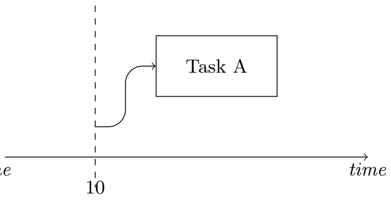
The main patterns from this section are depicted in Fig. 4 where the arrows show that the occurrence of an event implies the occurrence of the related event.

**BeforeDeadline** The first pattern relates an event with an absolute timing.

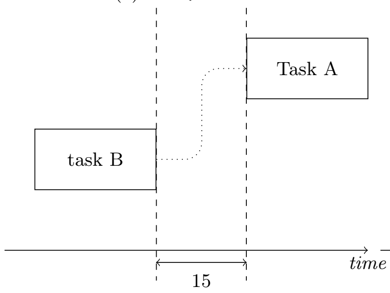
```
BeforeDeadline = a no later than d
```



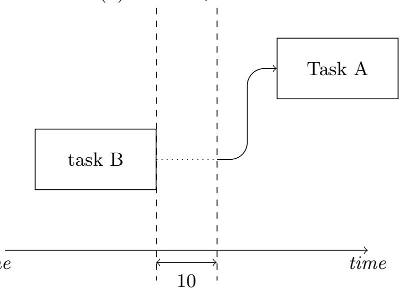
(a) `AT(A,10)`



(b) `AFTER(A,10)`

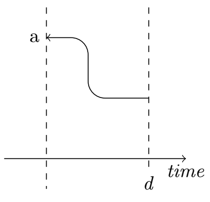


(c) `AT(A,B,15)`

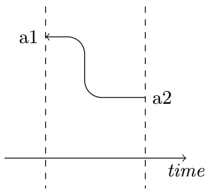


(d) `AFTER(A,B,10)`

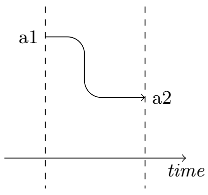
Fig. 3: Some unilateral synchronisations.



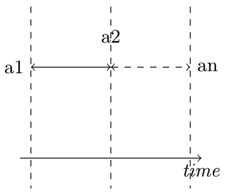
(a) BeforeDeadline



(b) Precedence



(c) EventualResponse



(d) Sequence

Fig. 4: Some observer patterns for real-time systems

**Precedence** The following patterns allow for expressing the precedence of the current event by another, with or without an explicit time frame.

The cyclic version of these patterns denotes that a pattern is repeatedly valid, whereas in the strict cyclic version the pattern is not only repeatedly valid, but no event mentioned in the pattern can happen in between (*i.e.* the events mentioned in the pattern are alternating).

The `precedence` pattern requires that, whenever event `a2` happens, then the event `a1` must have happened before (at least once). Note that `a2` is not required to happen. In the `CyclicPrecedence` pattern, every time `a2` happens, then the event `a1` must have happened before (at least once) since the last occurrence of `a2`. In the strict cyclic version, every time `a2` happens, then event `a1` must have happened exactly once since the last occurrence of `a2`, *i.e.* `a1` and `a2` alternate. (They do not need to alternate forever though.) For example, in the `CyclicPrecedence`, the sequence `a1 a1 a2` can happen but not `a1 a2 a2`, while in the `StrictCyclicPrecedence` none of them can happen.

This pattern is extended to a timed version in a straightforward manner.

```
Precedence = if a2 then a1 has happened before
CyclicPrecedence = everytime a2 then a1 has happened before
StrictCyclicPrecedence = everytime a2 then
                             a1 has happened exactly once before

TimedPrecedence = if a2 then
                      a1 has happened
                      at most d units of time before
CyclicTimedPrecedence = everytime a2 then
                            a1 has happened
                            at most d units of time before
StrictCyclicTimedPrecedence = everytime a2 then
                                  a1 has happened exactly once
                                  at most d units of time before
```

**Response** Expressing that the current event will be followed by a response is formulated by the following pattern. This pattern is equivalent to the "eventually" in linear temporal logics. None of the two events is required to happen; however, if the first one does, then second must eventually happen too. The cyclic and strictly cyclic versions are defined as for precedence. A timed extension (as in timed temporal logics) is also defined.

```
EventualResponse = if a1 then eventually a2
CyclicEventualResponse = everytime a1 then eventually a2
StrictCyclicEventualResponse = everytime a1 then
                                   eventually a2 once before next a1

TimedResponse = if a1 then eventually a2 within d
CyclicTimedResponse = everytime a1 then eventually a2 within d
StrictCyclicTimedResponse = everytime a1 then
                                eventually a2 within d
                                once before next a1
```

**Sequence** Events can also be ordered as a sequence. None of the $n$ events is required to happen; however, if some (or all) do, then they must follow exactly the order defined by the sequence. The cyclic version is straightforward. However, no strict cyclic version is defined, as it would be identical to the cyclic version.

```
Sequence = SEQUENCE a1, ..., an
CyclicSequence = always SEQUENCE a1, ..., an
```

```
eventlist =
      eventlist EVENT
    | EVENT

interval = (d, D) | [d, D] | [d, D) | (d, D]
timing = WITHIN interval

simplePattern =
      EVENT AT timing
    | EVENT EVENTUALLY timing EVENT
    | EVENT timing AFTER EVENT
    | SEQUENCE (eventlist)

pattern =
      pattern OR simplePattern
    | pattern AND simplePattern
    | ALWAYS simplePattern
    | simplePattern

SYNTACTIC SUGAR:
    AT LEAST d = WITHIN [d, infinity)
    AT MOST d  = WITHIN [0, d]
    EXACTLY d  = WITHIN [d, d]
```

Fig. 5: A grammar for unified patterns

**Unreachability** The last pattern of [5] is rather different from others, as it only expresses the model safety (*i.e.* non-reachability of a undesired state). It was considered in [5] because this property is by far the most commonly met in case studies from the literature, and because all other patterns can be reduced to (non-)reachability.

```
Unreachable = UNREACHABLE(Bad)
```

## 3 Towards a More Complete Patterns Language

The primitives in the grammars of Section 2.1 and Section 2.2 are dedicated to temporal or causal relations between tasks which are characterised by both their starting and ending times. But, in practice, most systems are concerned with individual events, as is the case in [5]. We present in this section a unified version of patterns previously introduced. We will show in Section 5 that our patterns subsume the primitives from Section 2.

We introduce a grammar for unified patterns Fig. 5. Our grammar considers individual *events* that can form a *list of events* in order to construct a *sequence*. The *timing* of events can be specified as being *within* a time frame (from *d* to

$D$ time units, where $d \in \mathbb{R}_+$ and $D \in \mathbb{R}_+ \cup \{\infty\}$). The only restriction is that the interval $[d, D] \neq [0, \infty)$ (see Remark 1 infra).

*Simple patterns* express basic relations between individual events. An event can happen w.r.t. an absolute *timing* constraint. An event can *eventually* entail the occurrence of another event w.r.t. some *timing constraint.* Conversely, an event can occur only w.r.t. a timing *after* another event already occurred. Events can be ordered in a sequence, thus all occurring one after another.

*Simple Patterns* The `simple patterns` contain four kinds of relationships, that we describe in more details in the following.

The pattern fragment `EVENT AT` encodes an absolute timing; it is used to describe events that must happen exactly at an (absolute) time.

The pattern fragment `EVENT EVENTUALLY timing EVENT` encodes that, whenever the first event happens, then the second will eventually happen, with the timing constraint specified by `timing`. That is, if the first event happens, the second must happen. The converse is not true: if the second event happens, the first one did not necessarily happen before.

The pattern fragment `EVENT timing AFTER EVENT` encodes that, whenever the first event happens it is necessarily after the second one, together with some timing constraint. For example, `e2 WITHIN(d,D) AFTER e1` denotes that `e2` may or may not happen, but if `e2` happens, then it must be at least `d` and at most `D` time units after the first occurrence of `e1`. Also note that, if `e2` does not happen, then `e1` may or may not happen.

Finally, the `SEQUENCE` ensures that a list of events happen in the particular order specified.

*Complex Patterns* Patterns can be combined in order to form more complex ones. First, we can use Boolean `AND` and `OR` operators to express the conjunction of patterns (*i.e.* both must be executed, for patterns expressing systems, or must be valid, for patterns expressing the properties) or the disjunction (*i.e.* either one of them can be executed/valid).

The `ALWAYS` is a sort of fixpoint, with a semantics similar as the notion of "cyclic" pattern in [5]. That is, once the pattern has been executed / verified, then it must again be executed / verified. This typically describes a cyclic behaviour. We restrict here the `ALWAYS` pattern to simple patterns (`simple pattern` in Fig. 5 ant not, *e.g.* `pattern`). The reason is on the one hand to keep our language simple[1], and on the other hand to make a translation to time Petri nets relatively easy.

Finally, it is often convenient to use some syntactic sugar for expressing timing constraints: `AT LEAST`, `AT MOST` and `EXACTLY`.

*Remark 1 (untimed patterns).* We could encode untimed patterns using our timed patterns (by allowing a syntactic sugar construct `UNTIMED = WITHIN [0,`

[1] Furthermore, while designing the patterns in [5], such `ALWAYS`-like properties were only encountered in the literature on (very) simple patterns.

`infinity)`), but we leave it out so as to keep the exposé simple. Indeed, although this does not bring theoretical problems, the translation of the untimed patterns into time Petri nets for verification purposes then exceeds the set of properties that can be checked using sole unreachability. In particular, the negation of the untimed "eventually" construct cannot be checked using the unreachability of a "bad" state, but it becomes necessary to additionally check the reachability of some "good state"; this was performed in [5]. Here, to keep the translation simple, we temporarily leave out the untimed patterns. Formalising the untimed patterns for the verification purpose will be performed in an extended version of this work.

## 4 Semantics: Translation to Time Petri Nets

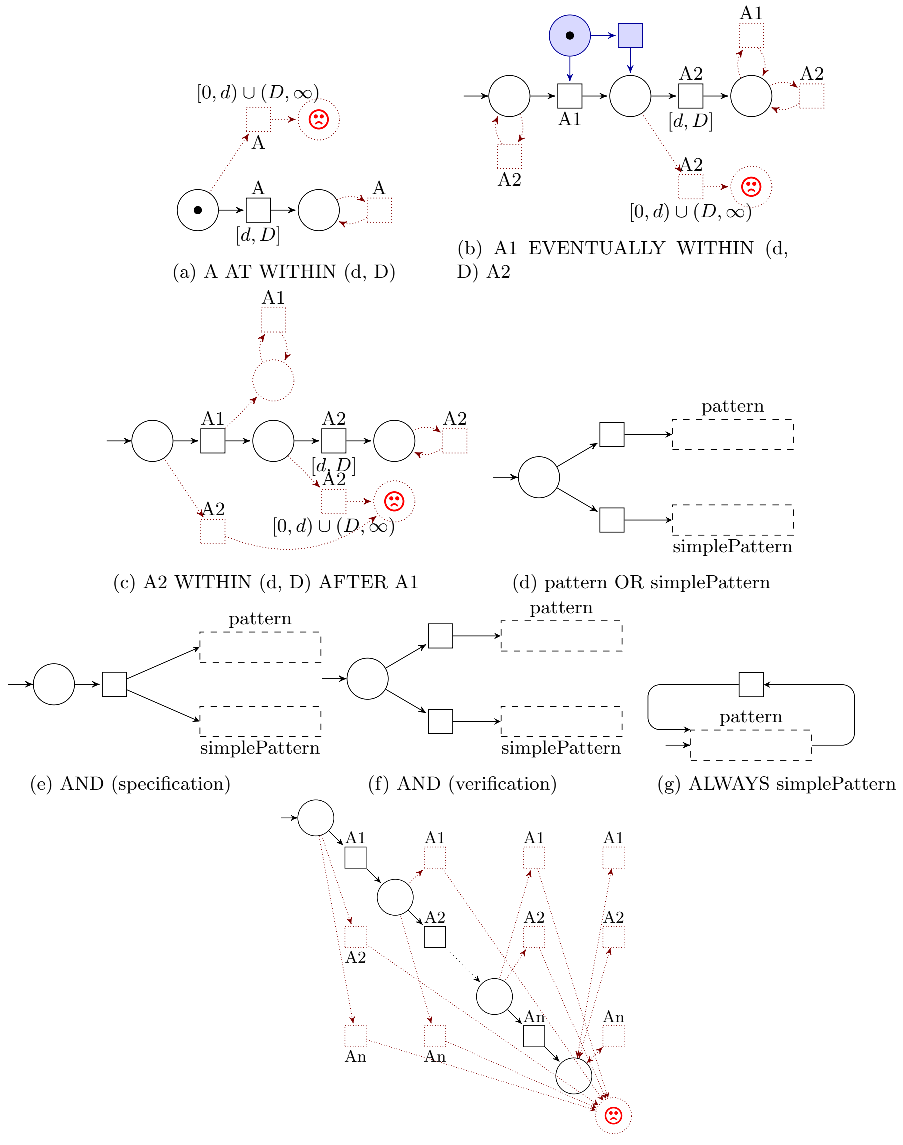
Time Petri nets [13] are Petri nets where transition are equipped with a time interval, that specifies the minimum and maximum time for the transition to be enabled before it actually fires. The different patterns in the grammar of Fig. 5 are modelled as time Petri nets in Fig. 6a–6h. Let us now describe our translation. We start with simple (*i.e.* non-compositional) patterns, and then go for complex patterns (*i.e.* that rely on others).

*Observers* Let us recall the concept of observers, as formalised in [5]. Observers are standard subsystems, with some assumptions. An observer must not have any effect on the system, and must not prevent any behaviour 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. Similarly, in time Petri nets, an observer must be able to synchronise at any time with any of the actions used on its transitions.

*General Idea of our Translation* Recall that our patterns aim at encoding both systems and properties. Although they are defined in a unified manner in Section 3, they must be differentiated when formalised using time Petri nets. Indeed, our patterns seen as properties reduce verification to simple reachability analysis (as in [2,1,5]).

For the verification, we define a "bad" place (labelled in Fig. 6a–6h using the "☹" symbol); this place is assumed to be unique, *i.e.* one must *fuse* all occurrences of this place when composing patterns. The verification can then be carried out as follows: given a model of the system specified using time Petri nets (but not necessarily specified using our specification patterns), and given a property of the system specified using our patterns and translated into a time Petri nets, we perform the synchronisation (on transitions) of the entire system. Then, the property (expressed by the pattern) is satisfied iff the "☹" place is unmarkable, *i.e.* cannot be marked in any marking of the synchronised net.

In order to differentiate between the specification of systems and the specification of properties, we depict in dotted red the places and transitions necessary to add to our translated patterns so as to be able to perform verification. In other words, these dotted red places and transitions shall be omitted when specifying systems and not properties. Conversely, we depict in plain light blue the places and transitions only necessary for the system specification, but that must be omitted for the verification.



(a) A AT WITHIN (d, D)

(b) A1 EVENTUALLY WITHIN (d, D) A2

(c) A2 WITHIN (d, D) AFTER A1

(d) pattern OR simplePattern

(e) AND (specification)

(f) AND (verification)

(g) ALWAYS simplePattern

(h) SEQUENCE(A1,A2,...,An)

Fig. 6: Translation of our patterns into time Petri nets

*Simple Patterns* Fig. 6a gives the translation of the `A AT WITHIN (d, D)` pattern. For the property, the translation is straightforward: a place is followed by a timed transition with firing time $[d, D]$ labelled with $A$. This correctly encodes the fact that $A$ must occur within $[d, D]$ after the system start. (We assume closed intervals in the remainder of the translation; open or semi-open intervals can be handled similarly.) Concerning the verification, in addition to the correct behaviour, we need also to specify the bad behaviour, hence in this case another transition that can occur only if the timing is not satisfied, leading to the "bad" place. That is, the bad place is reachable iff the property is violated. Additionally, for our pattern to be a good "observer" (*i.e.* that must not disturb the system), it must be able to synchronise at any time with the system on the transition the pattern declares (here only `A`). This explains the loop on transition A on the right-hand side of Fig. 6a. (In fact, self-loops should also be added to the bad place; we omit them for sake of space, but also because it is less important to block the system once the property has been proved invalid.)

Pattern `A1 EVENTUALLY WITHIN (d, D) A2` is modelled by the TPN in Fig. 6b. For the specification, two cases are admitted: one where `A1` is fired, and one where it is not. Moreover the possibility of firing `A1` depends on other actions in the system which may put a token in the initial place of the pattern (depicted by an incoming arrow). The additional places and transitions for the verification counterpart of this pattern are explained as follows: the first self-loop allows `A2` to happen anytime as long as `A1` has not happened. Then, if `A2` happens strictly before or strictly after $[d, D]$, the observer enters the bad place. Otherwise, the property is satisfied, and both `A1` and `A2` can happen anytime, which is depicted using the two self-loops.

The `A2 WITHIN (d, D) AFTER A1` pattern (given in Fig. 6c) is similar to the previous one but, in this case, it is not possible to have `A2` without `A1`. As for the verification, note that `A2` cannot occur before `A1`, hence the transition between the initial place and the bad place. Furthermore, several `A1` may occur before `A2` occurs: this is encoded using the second output from `A1` and a self-loop. Thus the time for firing `A2` is counted from the first occurrence of `A1`. The rest of the pattern is similar to the previous one.

The `SEQUENCE(A1,A2,...,An)` pattern is given in Fig. 6h. Naturally, it is made of a sequence of transitions. Additionally, the verification version is such that, as soon as a transition violates the order imposed by the sequence, the system goes to the bad place. An additional self-loop in the last good place, synchronising on any transition, makes the observer non-blocking.

*Complex Patterns* These patterns are used to combine the previous ones (eventually with complex patterns as well). In Fig. 6d to Fig. 6g, they are pictured in dashed boxes, which would also include the "bad" place. For the specification, the complex patterns are straightforward: they syntactically combine existing patterns. For the verification, this is a little less simple: first, recall that all "bad" locations must be fused into a single one. Second, the "and" verification pattern becomes identical to the... "or" specification pattern: this is because the property `P1 AND P2` is violated if `P1` is violated (*i.e.* the bad place is reachable in the corresponding pattern) *or* `P2` is violated. The "or" pattern is not translated for verification; this is because this cannot be checked with sole unreachability (see Section 6). Concerning the `ALWAYS` pattern for verification, one must fuse the last non-dotted place of the pattern (usually the right-most place in the figures) with the initial place. However, the self-loops (generally on `A1` and `A2`) on the last non-dotted place must be removed.

*Initial Marking* In addition to the tokens introduced by the absolute time patterns (`A AT WITHIN (d,D)`), a single initial token must be put in the top-most pattern of the composed pattern expression. (We assume that, if the top-most expression is a pattern `A AT WITHIN (d,D)`, then no further token is added.)

## 5 Encoding Previous Patterns Using our Unified Patterns

In this section we show how all primitives from Section 2 can be expressed using our new set of patterns.

In order to express the relations between tasks described in Section 2.1 and Section 2.2 with these new primitives, a task `A` is transformed into two events, specifying the task Beginning (`A.start`) and its end (`A.end`).

### 5.1 Encoding Patterns from [11]

The expression of patterns from Section 2.1 is summarised in Table 1.

Note that several rules are expressed identically. For example, rule 2 is reflecting the point of view from event A, and rule 4 the one of event B, while in our patterns we express the relation as seen from an external observer.

### 5.2 Encoding Patterns from [7]

Table 2 shows the mapping for the patterns of Section 2.2.

Note that formula 5 implies that if B does not occur, neither does A. On the contrary, if B occurs, A can occur or not. If it does, it is $d$ units of time after B ended. A similar remark applies to formula 6. Finally, in formula 7, A necessarily occurs $d$ units of time after the end of B.

### 5.3 Encoding Patterns from [5]

Patterns from Section 2.3 are presented in Table 3. Our translation is straightforward. The only "trick" is the translation of the "strict cyclic" patterns of [5], that are encoded using both the cyclic version of these patterns (using `ALWAYS`) and the `SEQUENCE` pattern, that requires `A1` and `A2` to alternate.

## 6 Conclusion

We proposed a unified pattern mechanism to both specify and verify real-time systems, together with a semantics using time Petri nets. Our new set of patterns unifies the patterns of [11,7,5] into a single homogeneous pattern language.

| | | |
|---|---|---|
| Rule 1 | $A$ [ends] before $(d, D)$ $B$ [starts] | `B.start WITHIN` $(d, D)$ `AFTER A.end` |
| Rule 2 | $A$ meets $B$ | `B.start EXACTLY 0 AFTER A.end` |
| Rule 3 | $B$ [starts] after $(d, D)$ $A$ [ends] | `B.start WITHIN` $(d, D)$ `AFTER A.end` |
| Rule 4 | $B$ met by $A$ | `B.start EXACTLY 0 AFTER A.end` |
| Rule 5 | $A$ starts before $(d, D)$ $B$ [starts] | `B.start WITHIN` $(d, D)$ `AFTER A.start` |
| Rule 6 | $A$ starts $B$ | `B.start EXACTLY 0 AFTER A.start` |
| Rule 7 | $B$ starts after $(d, D)$ $A$ [starts] | `B.start WITHIN` $(d, D)$ `AFTER A.start` |
| Rule 8 | $A$ ends before $(d, D)$ $B$ [ends] | `B.end WITHIN` $(d, D)$ `AFTER A.end` |
| Rule 9 | $A$ ends $B$ | `B.end EXACTLY 0 AFTER A.end` |
| Rule 10 | $A$ ends before $(d, D)$ $B$ [ends] | `B.end WITHIN` $(d, D)$ `AFTER A.end` |
| Rule 11 | $A$ starts_before_end $(d, D)$ $B$ | `B.end WITHIN` $(d, D)$ `AFTER A.start` |
| Rule 12 | $B$ ends_after_start $(d, D)$ $A$ | `B.end WITHIN` $(d, D)$ `AFTER A.start` |
| Rule 13 | $A$ contains $((d_1, D_1)(d_2, D_2))$ $B$ | `B.start WITHIN` $(d_1, D_1)$ `AFTER A.start AND A.end WITHIN` $(d_2, D_2)$ `AFTER B.end` |
| Rule 14 | $B$ contained_by $((d_1, D_1)(d_2, D_2))$ $A$ | `B.start WITHIN` $(d_1, D_1)$ `AFTER A.start AND A.end WITHIN` $(d_2, D_2)$ `AFTER B.end` |
| Rule 15 | $A$ equals $B$ | `B.start EXACTLY 0 AFTER A.start AND B.end EXACTLY 0 AFTER A.end` |
| Rule 16 | $A$ parallels $((d_1, D_1)(d_2, D_2))$ $B$ | `B.start WITHIN` $(d_1, D_1)$ `AFTER A.start AND B.end WITHIN` $(d_2, D_2)$ `AFTER A.end` |
| Rule 17 | $B$ parallelled $((d_1, D_1)(d_2, D_2))$ by $A$ | `B.start WITHIN` $(d_1, D_1)$ `AFTER A.start AND B.end WITHIN` $(d_2, D_2)$ `AFTER A.end` |

Table 1: Encoding patterns from [11]

*Future Works* First, translating the untimed `EVENTUALLY` and the `OR` patterns for verification purposes is in our agenda; this will be done by checking, not only the unreachability of the bad, but also the reachability of a good place.

Second, more patterns from the literature should be integrated to our encoding. Although we shall not develop too complex a pattern system, so as to avoid giving birth to a complicated property language, the patterns in [12] seem interesting to us. Furthermore, the patterns of [9] seem to fit directly in our unified pattern systems, but this should be shown formally. It would also be interesting to formally compare the expressiveness of our patterns with [2,1] or (subsets of) temporal logics such as LTL/CTL.

Third, although it is relatively easy to convince oneself that we correctly encoded the patterns of [11,7,5], formally proving their semantic equivalence would be interesting. It would also be nice to provide tool support, helping a designer to write patterns to model a system and its properties.

Finally, our translation to time Petri nets was done manually. An alternative option would be to define an *ad-hoc* domain specific language (DSL), and then to use model transformation techniques (such as in [8]) to obtain time Petri nets.

**Acknowledgement**

We are grateful to the anonymous reviewers for useful comments.

| | | |
|---|---|---|
| Formula 1 | `BILATERALSYNCH(A,B)` | `A.start EXACTLY 0 AFTER B.start` `OR A.end EXACTLY 0 AFTER B.end` |
| Formula 2 | `AT(A,d)` | `A.start AT EXACTLY d` |
| Formula 3 | `AFTER(A,d)` | `A.start AT AT LEAST d` |
| Formula 4 | `BEFORE(A,d)` | `A.start AT AT MOST d` |
| Formula 5 | `AT(A,B,d)` | `A.start EXACTLY d AFTER B.end` |
| Formula 6 | `AFTER(A,B,d)` | `A.start AT LEAST d AFTER B.end` |
| Formula 7 | `BEFORE(A,B,d)` | `B.end EVENTUALLY AT MOST d A.start` |

Table 2: Encoding patterns from [7]

## References

1. 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*, volume 1530 of *LNCS*, pages 245–256. Springer, 1998.
2. Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In *TACAS*, volume 1384 of *LNCS*, pages 263–280. Springer, 1998.
3. Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
4. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
5. Étienne André. Observer patterns for real-time systems. In *ICECCS*, pages 125–134. IEEE Computer Society, 2013.
6. Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.
7. Sébastien Bardin and Laure Petrucci. COAST : des réseaux de Petri à la planification assistée. In *AFADL*, pages 285–298, 2004.
8. Julien DeAntoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoît Combemale. Towards a meta-language for the concurrency concern in DSLs. In *DATE*, pages 313–316. ACM, 2015.
9. 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.
10. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.
11. Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping temporal planning constraints into timed automata. In *TIME*, pages 21–27. IEEE Computer Society, 2001.
12. Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*, pages 112–124. British Computer Society, 2009.
13. Philip Meir Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, 1974.
14. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
15. 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.

| | | |
|---|---|---|
| Formula 1 | `Precedence = if A2 then A1 has happened before` | `A2 AT LEAST 0 AFTER A1` |
| Formula 2 | `CyclicPrecedence = everytime A2 then A1 has happened before` | `ALWAYS (A2 AT LEAST 0 AFTER A1)` |
| Formula 3 | `StrictCyclicPrecedence =` <br> `everytime A2 then A1 has happened exactly once before` | `ALWAYS SEQUENCE(a1,a2)` |
| Formula 4 | `EventualResponse = if A1 then eventually A2` | `A1 EVENTUALLY AT LEAST 0 A2` |
| Formula 5 | `CyclicEventualResponse =` <br> `everytime A1 then eventually A2 (before next A1)` | `ALWAYS (A1 EVENTUALLY AT LEAST 0 AFTER A2)` |
| Formula 6 | `StrictCyclicEventualResponse =` <br> `everytime A1 then eventually A2 once before next A1` | `ALWAYS (A1 EVENTUALLY AT LEAST 0 A2)` <br> `AND SEQUENCE(A1,A2)` |
| Formula 7 | `BeforeDeadline = A no later than d` | `A AT AT MOST d` |
| Formula 8 | `TimedPrecedence =` <br> `if A2 then A1 has happened at most d units of time before` | `A2 AT MOST d AFTER A1` |
| Formula 9 | `CyclicTimedPrecedence =` <br> `everytime A2 then A1 has happened` <br> `at most d units of time before` | `ALWAYS (A2 AT MOST d AFTER A1)` |
| Formula 10 | `StrictCyclicTimedPrecedence =` <br> `everytime A2 then A1 has happened exactly once` <br> `at most d units of time before` | `ALWAYS (A2 AT MOST d AFTER A1)` <br> `AND ALWAYS SEQUENCE(A1, A2)` |
| Formula 11 | `TimedResponse =` <br> `if A1 then eventually A2 within d` | `A1 EVENTUALLY AT MOST d A2` |
| Formula 12 | `CyclicTimedResponse =` <br> `everytime A1 then eventually A2 within d` | `ALWAYS (A1 EVENTUALLY AT MOST d A2)` |
| Formula 13 | `StrictCyclicTimedResponse =` <br> `everytime A1 then eventually A2 within d once before next A1` | `ALWAYS (A1 EVENTUALLY AT MOST d A2)` <br> `AND ALWAYS SEQUENCE (A1, A2)` |
| Formula 14 | `Sequence = sequence A1, ..., An` | `SEQUENCE(A1,A2,...,An)` |
| Formula 15 | `CyclicSequence = always sequence A1, ..., An` | `ALWAYS SEQUENCE(A1,A2,...,An)` |

Table 3: Encoding patterns from [5]