Formal Verification of User-Level Real-Time Property Patterns

Ning Ge, Marc Pantel, Silvano Dal Zilio

Formal Verification of User-Level Real-Time Property Patterns

Ning Ge
School of Software, Beihang University
Beijing, China
gening@buaa.edu.cn

Marc Pantel
IRIT/INPT
Toulouse, France
marc.pantel@enseeiht.fr

Silvano Dal Zilio
LAAS-CNRS
Toulouse, France
dalzilio@laas.fr

***Abstract*—To ease the expression of real-time requirements, Dwyer, and then Konrad, studied a large collection of existing systems in order to identify a set of real-time property patterns covering most of the useful use cases. The goal was to provide a set of reusable patterns that system designers can instantiate to express requirements instead of using complex temporal logic formulas. A limitation of this approach is that the choice of patterns is more oriented towards expressiveness than efficiency; meaning that it does not take into account the computational complexity of checking patterns. For this purpose, we define a set of verification-dedicated, atomic property patterns for qualitative and quantitative real-time requirements. End-user requirements can then be expressed as a composition of these patterns using a predefined meta-model and a mapping library. These properties can be checked efficiently using a set of elementary observers and a model checking approach.**

***Index Terms*—real-time requirements, property pattern, observer, model checking, Time Petri net**

I. Introduction and Related Work

Real-time requirements commonly used during the development of concurrent systems can be, for the most part, covered by a finite set of properties, such as: worst case execution time, worst case traversal time, state duration, schedulability, etc. [1]. As illustrated by Dwyer et. al. [2], property patterns provide a valuable way to handle these requirements. First, they ease the use of formal methods by providing reusable solutions to recurrent specification and verification problems. This is especially true for novice users. Then, they help decomposing complex properties into a set of simpler ones, with a lower complexity, and thus may help decrease the verification cost.

In this paper, we define a new set of property patterns, supporting the definition of timing constraints, that can be easily composed. Each combination of pattern can be translated into a verification observer; therefore providing a way to automatically prove a requirement using a more standard model-checker.

Property patterns are an established concept. In their seminal work, Dwyer et. al. [2] performed a large-scale study of specifications containing over 500 temporal requirements. They noticed that over 90% of them could be classified under one of the short list of patterns that they had identified. They initially proposed only "qualitative" temporal properties, meaning that they focused on logical time and could not use quantitative concepts, such as time interval or durations.

More recent works [3], [4], [5] have proposed an extension of Dwyer's patterns with real-time constraints. For instance, Konrad et al. [3] mapped quantitative time property patterns into three real-time temporal logics: MTL [6], TCTL [7], and RTGIL [8], thus delegating the verification problem to model-checkers able to handle these timed logic. They also defined pattern templates to ease reuse. These works were mostly oriented towards providing more expressiveness to users, but did not really address the verification cost associated to each new pattern. Indeed, model checking for MTL, TCTL or RTGIL is expensive in practice and is only available on a very limited number of model-checkers, most of them not maintained anymore. An observer-based approach can help in solving this problem, since it can reduce the initial model checking problem to a much simpler one, like for example a reachability property.

Abid et al. [5] proposed an observer-based verification framework for real-time properties based on Dwyer's work. From a requirement engineering point of view, their patterns are not atomic, meaning that they could be further decomposed into more essential components. Also, they cannot be nested or composed together, but can only be composed through classical disjunction or conjunction of properties. We can illustrate this limitation with the case of the "end-to-end" real-time property *(Exist A After B Within I)*, where A and B are events and I is a given time interval $[T_{min}, T_{max}]$. This pattern means that the first occurrence of B (if it exists) is necessarily followed by an occurrence of A in a delay that is within I. In our setting, we can decompose this property into four more basic elements that can be freely composed: *(Exist A)*, *(Exist B)*, $(T_{AB} \geq T_{min})$ and $(T_{AB} \leq T_{max})$, where T_{AB} stands for the possible time interval between the first occurrences of A and B. The definition of a more basic library of patterns simplifies the definition of new patterns and their semantics. It also simplifies the verification problem. In [9], Castillos et al. defined a set of compositional automata-based semantics for property patterns, and proposed a composition operation in such a way that the property semantics is defined by composing the automata. This work is limited to "qualitative" temporal properties based on Dewyer's patterns, but does not yet address the "quantitative" ones.

978-1-5386-1925-4/17/$31.00

Our contributions are twofold: (1) the definition of a set of atomic pattern combinators for the formal expression of real-time requirements; and (2) the definition of a set of elementary observers in order to check our real-time patterns. In this work, we use Time Petri Nets (TPN), and an extension of TPN with data called TTS (Time Transition System), as the verification format. We also rely on the model-checking toolbox TINA [10] for the verification part. We define a set of event-based observers at TPN level (12 observers), a set of state-based observers at TTS level (4 observers). Our observers take advantage of the highest possible level of abstraction provided by TINA (marking abstraction in our case) to reduce the size of the abstract state space that needs to be explored during verification. The results of this work has been applied to our UML-MARTE real-time verification framework [11], [12], which provides a toolchain including property-specific model translation [13], property-specific state space reduction [14], probabilistic failure analysis [15], etc.

The paper is organised as follows: Sect. II gives some technical background on model-checking TPN; Sect. III introduces the catalog of real-time property patterns while Sect. IV explains the design of observers; Sect. V illustrates the definition of pattern modifiers and the set of observers; Sect. VI presents the specification and verification on an example of real-time property; Sect. VII gives some concluding remarks.

II. Technical Background

Time Petri nets (TPN) [16] are an extension of Petri nets with timing constraints on the transitions. We use an example to explain its syntax and semantics.

The TPN in Fig. 1 models the concurrent execution of a process with two tasks scheduled in parallel. Each transition in a TPN is decorated with a (static) time interval that constraints its firing time. In this net, place P_{init} is initially marked with one token. Hence transition *fork* is enabled and should fire immediately (with a delay included in the interval $[0,0]$). Upon firing, transitions $task_1$ and $task_2$ start at the same date. Intuitively, each transition is associated with a local clock that starts once it is enabled; then the transition can fire when the clock value is in the time interval associated with the transition. For instance, due to their time constraints, *Task*1*_ends* always fire before *Task*2*_ends*. Once the two tasks finish (there are two tokens in place P_{join}), the system can either exit or restart the whole execution.

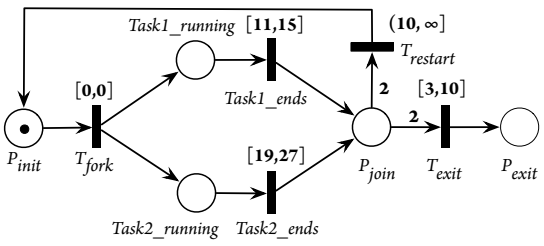

Figure 1. Time Petri Net Example

Time Petri nets can be composed like ordinary Petri net, by combining transitions with a common label. This will be used to add an observer to an existing model. Finally, TTS extend the semantics of TPN by allowing guards on shared data variables in transitions and expressions that are evaluated when a transition fires.

Time Petri Nets provide a formal framework to capture the real-time behavior of concurrent real-time system. There exists several analysis tools, such as TINA or Romeo. Our work relies on TINA (the time Petri net analyzer), a toolbox for the edition and analysis of Petri nets. TINA also supports nets with inhibitor and read arcs, priorities, stopwatches, and its extension TTS. The TINA toolset includes tools for the exploration of reachability graphs (**tina** and **sift**) that support a large choice of state abstractions; model-checkers for LTL (**selt**); for CTL and an existential fragment of μ-calculus (**muse**); etc.

Next, we define some core concepts used in the specification of patterns: occurrence, predicate, scope, events and states. A pattern describes constraints on three main kind of elements. What is the object of the constraint (predicate)? When should it happen (scope)? And how does it compare with the other events (occurrence)? The occurrence of a predicate could be specified as existence, absence, always (exist), or exist with a bounded number of occurrences. For instance, in our initial example of pattern given in Sect. I, *(Exist A After B Within I)*, the occurrence is *Exist*, the predicate is the event *A*, and the scope modifier is *(After B Within I)*.

In this context, an *event* is an instantaneous and atomic occurrence of an action at a point in time. Our patterns can also deal with *states*, that is identifiers that designate when some given invariant conditions hold. States can change as a result of an event, and we will often talk about the "enter event" of a state. In the remainder of this text, and to avoid ambiguity, a scope modifier will always be applied to an event, while we allow both states and events in a predicate. In particular, if *A* is a state then the scope *After A* will apply to the corresponding enter event of *A*.

III. Catalog of Real-Time Property Patterns

A real-time property is composed of two main components: a pattern and a scope. Roughly speaking, when considering the state graph of the system, the scope operator is used to select a subset of the reachable states. These (state) candidates are then qualified by the given pattern. Patterns comprises the system devised by Dwyer, that is based on eight patterns (*Absence, Existence, Bounded Existence, Precedence, Response, Chain Precedence* and *Chain Response*) and five scope modifiers (*Global, Before, After, Between* and *After-Until*). Konrad et al. extended this system to also include quantitative requirements by introducing five quantitative modifiers (*Minimum Duration, Maximum Duration, Bounded Recurrence, Bounded Response* and *Bounded Invariance*). We add to this set a new scope modifier, *Periodically*, for the specification of periodic events related requirements. We also add three real-time suffixes, *At least* and *At most* for the specification of bounded time on state

related predicates, and the suffix *Within* for the specification of time intervals. The addition of these new modifiers allows us to express more elaborate patterns, such as *(At least d Before E)* or *(Absent A Precedes B Periodically)*; etc.

In order to cross the boundary between property specification and verification and reduce the verification cost, we define a set of elementary constructs for property patterns. In the proposed pattern system (see Fig. 2), real-time requirements are specified using either atomic or composite patterns. Composite patterns are built using binary operators (or, and, imply). Atomic patterns are composed of three elementary constructs: *occurrence modifier*, *basic predicate* and *scope modifier*. (To avoid any ambiguities we will reserve the use of the term "pattern" to mean "property pattern".) Basic predicates are defined based on state and event modifiers, and scope modifiers are defined based on event modifiers.

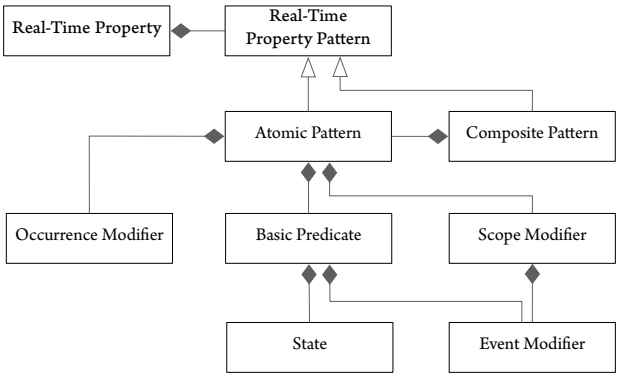

Figure 2. Real-Time Property Specification System

IV. Design Principles of TPN/TTS Observers

Before illustrating the set of elementary constructs of property patterns and defining the set of observers, we first explain the design principles of the observers in our model checking approach.

A. Structure of Observer

A TPN/TTS observer is a sub-net that will be composed with the net capturing the behavior of the system. To assess a property specification based on events, we use a composition described by the diagram of Fig. 3. A TPN/TTS observer is associated to the system through its arcs, joined at the transitions labelled T_A and T_B in components A & B. On the opposite, a TTS observer for state-based properties is not composed with the system but simply put in parallel (an operation usually referred to as free product). This operation is depicted using arcs with dotted-line in Fig. 3.

The abstract structure of our observers contains a place, namely P_{tester}, which allows properties to be assessed by using accessibility assertions declared within a modal μ-calculus (*mmc*) formula. This *mmc* formula checks the existence of a specific marking and whether a given set of transitions can be fired. In this context, the most basic *mmc* formulae that we use are of the form $[T](P_{tester} = 1)$ or $\langle T\rangle(P_{tester} = 0)$, meaning that for all (respectively for at least one) successor state the observer is (respectively is not) in state P_{tester}. All the necessary formulas are checked on-the-fly using the **muse** model checker.

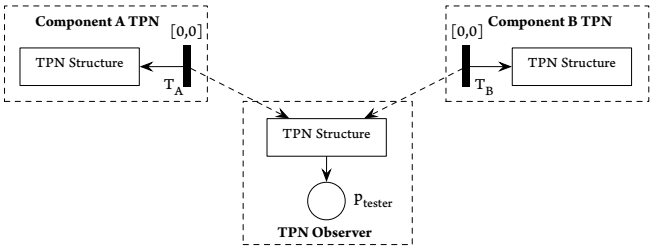

Figure 3. Observer Structure

Compared with the work of Abid et al. [5], we avoid completely the use of LTL model-checking with our observers. Indeed, the use of LTL model-checking requires to use state space abstraction that preserve the set of traces of a language, which can be less effective than abstractions that only preserve state reachability. This has a huge impact on the efficiency of model checking.

B. Soundness of Observer

Soundness here means that (1) an observer should not impact the system's behavior by introducing extra semantics or removing original semantics; and (2) observers preserve time divergence, meaning that an observer should not be able to stop the evolution of time (introducing some kind of time deadlocks). Indeed, an observer that is synchronized with a place of the system may impact its behavior by adding or removing tokens in an unlawful way. Our approach avoids this problem by interacting with the system only by its transitions. Moreover, the observers work in a "read-only" mode, guaranteed by the design "linked *from* TPN transitions".

Hence the soundness of observers is achieved using an high-level property of innocuousness that could be proved formally on each observers, independently from the system. This proof follows the same structure that the one found in the work of Abid et al. [17]. Actually, our group has also conducted experiments on the use of interactive theorem prover to prove this kind of properties [18].

C. Efficiency of Observer

We follow three principles to ensure the efficiency of our observers. *First*, a system with integrated observers should be able to generate state class graphs with a high-level abstraction (i.e. marking abstraction for TINA). This graph should preserve the required semantics of the targeting property. This principle is achieved by forbidding some elements in TPN during the design, such as the priority arcs. We rely on the marking graphs and *mmc* formulae to transform the quantitative verification problems to reachability problems

using the **muse** model checker. *Second*, the generating state space of a single observer shall be as small as possible. This principle requires us to experiment on different encoding of an observer, in order to select the relatively optimal one. Some experiences are summarized, such that the stopwatch and stopwatch inhibitor arcs are more expensive than the other three types (regular, read, inhibitor arcs). *Third*, the checking of each property pattern shall be independent to promote parallel computation.

V. Elementary Observers for the Verification of Property Patterns

A. Basic Event Modifiers

Predicates are specified based on events and states. An event can be an atomic element E, or a composite one, called event modifier, e.g. E^i for the definition of the i^{th} occurrence of event E. Here is a more complex composite observer: *t u.t. (unit of time) after event* E^{i-k}. It contains three basic event modifiers: (1) E^i (the i^{th} occurrence of event E); (2) E^{i-k} (the event delayed k times from the current event E^i); and (3) $E^{i-k}+t$ (the event delayed t *u.t.* from current event E^{i-k}). We have defined a set of extensible basic event modifiers. As shown in Fig. 4, a generic observer structure for event modifiers are defined, where the transition E is the observable part of the system, and the transition E' is the "extensible point" for integrating other event modifiers.

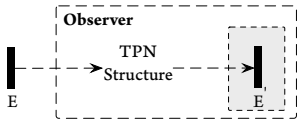

Figure 4. Observer Structure of Event Modifiers

With respect to the above principle, we have defined a set of basic event modifiers.

1) E^i: the i^{th} occurrence of event E: This event modifier, illustrated by Fig. 5, requires that occurrence of E is finite. Note that E^1 stands for the first occurrence of E, while E stands for the event type. If no occurrence is specified, E is regarded as E^1.

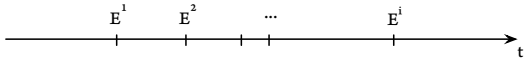

Figure 5. The i^{th} Occurrence of E

Its observer is defined as the TPN in Fig. 6. When E has occurred i times, the place P_{occ} has i tokens, and the transition E^i is enabled. This design ensures that event E^i occurs at the same time as the i^{th} occurrence of event E. The place P_{once} with one token controls the occurrence times of E^i. It allows E^i to occur only once. The number of tokens in P_{once} can be replaced by another finite value to enable E^i several times.

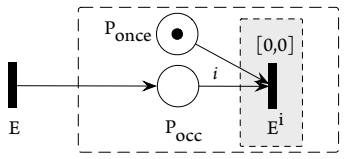

Figure 6. Event Observer: i^{th} Occurrence of E

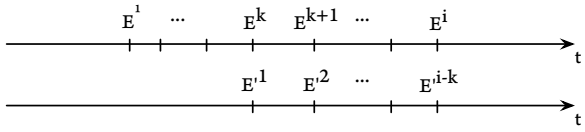

Figure 7. k Times Delay of E

2) E^{-k}: k^{th} delay of E: The event modifier E^{-k} stands for a delay of k times compared to event E, illustrated by Fig. 7.

The observer of this event modifier is defined in Fig. 8, where the place P_{occ} stores tokens representing the occurring times of event E. Each time P_{occ} has k tokens, the read arc enables the transition E^{-k}, which consumes one token in P_{occ}.

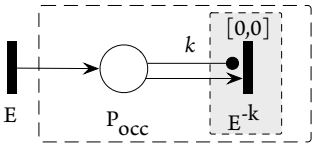

Figure 8. Event Observer: k Times Delay of E

3) $E^{/k}$: k times slower sub-occurrence of E: The event modifier $E^{/k}$ stands for the sub-occurrence of event E, with a frequency k times slower than E, illustrated by Fig. 9.

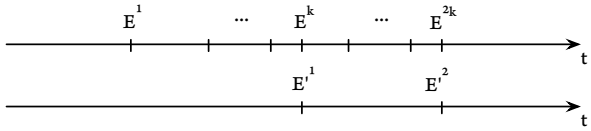

Figure 9. Sub-Occurrence of E

Its observer is designed as Fig. 10. When E occurs k times, the place P_{occ} accumulates k tokens, and the transition $E^{/k}$ is fired. Simultaneously, all the k tokens in P_{occ} are consumed.

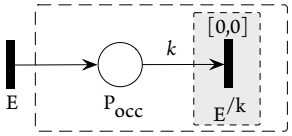

Figure 10. Event Observer: k times slower sub-occurrence of E

4) I+t: time t elapsed since system initialization: This event modifier *I+t* stands for the absolute time instant measured from the initial time of the system, illustrated by Fig. 11. It is used to assess properties such as worst/best case execution time.

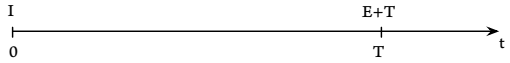

Figure 11. T Time Units after System Initialization

Its observer is designed as Fig. 12, which is composed of two parts: (1) the place P_{Init} representing the initialization of the system; (2) the transition E' representing that t *u.t.* has elapsed.

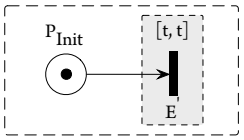

Figure 12. Event Observer: Time Elapsed since System Initialization

5) E+t: time t elapsed since the occurrence of event E: The event modifier $E + t$ stands for the moment when t *u.t.* has elapsed since the occurrence of E, illustrated by Fig. 13. When using this modifier, the occurring times of E should be finite. This event modifier is used to specify the scope *within* and the predicates such as *at least/at most*. For example, the property *After E Within I* ($[t_{min}, t_{max}]$) is specified as *After* $E + t_{min}$ **and** *Before* $E + t_{max}$.

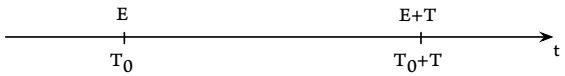

Figure 13. T Time Units after E

Its observer is designed as Fig. 14.

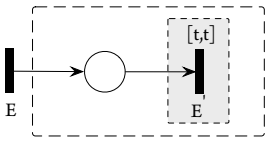

Figure 14. Event Observer: Time Elapsed since E

6) S^S & S^E: entering and exiting events of a State S: The TTS observer for the entering and exiting events of a state is designed as Fig. 15, where the transitions S^S/ S^E represent the entering/exiting events of the state S. When a system enters the state S, the assertion S in PRE(S) is true, which is the guard to enable the transition S_S and thus transit the token in the place P_S to the place P_E. Similarly, when the system exits state S, the assertion $\neg S$ in PRE($\neg S$) is true, which transits the token in the place P_E to the place P_S.

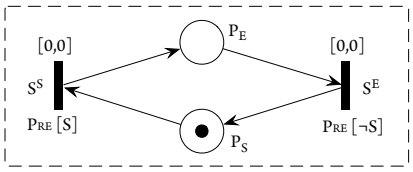

Figure 15. Event Observer: Entering and Exiting Events of S

B. Basic Predicates

The specification of basic predicates relies on events and states, where an event can be a single event modifiers or a composition of several event modifiers. We have defined a set of basic predicates used by our property patterns. The generic TPN structure of predicate observers is defined as Fig. 16, where the transition E_M is an event, and the predicate is assessed using the observer and a set of *mmc* assertions.

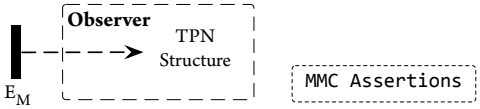

Figure 16. Predicate Observer Pattern

1) $O(E^i)$ = true for the occurrence of event E^i: In Fig. 17, the place P_{occ} linked from transition E_M is used to observe the occurrence times of event E_M. Once the transition E_M has fired i times, the token in P_{occ} is observed, which is assessed using the *mmc* assertion $P_{occ} \geq i$. Note that TINA takes P_{occ} as the number of tokens in the place P_{occ}.

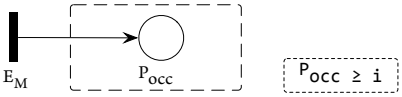

Figure 17. Predicate Observer: Occurrence of E^i

2) isFinite(E) = True for the bounded occurrence of E: This predicate is used to assess whether the occurrence of an event is finite. In Fig. 18, the place P_{occ} accumulates the occurrence times of event E_M. If the transition $T_{Overflow}$ is not fired, no overflow is detected, as E_M does not exceed the occurring bound Occ_{max}, i.e. a predefined threshold value which is usually determined by the estimation on system's behavior.

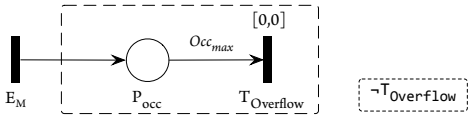

Figure 18. Predicate Observer: Occurrence of E is bounded

3) Freq(E_A) · N_A = Freq(E_B) · N_B for equivalent occurrence of E_A and E_B: This predicate is used to identify equivalent occurrences between two periodic events with different (or equal) frequencies. Suppose two periodic events E_A and E_B exhibit respectively occurrence frequency F_A and F_B. There exists minimal coefficients N_A and N_B ($N_A, N_B \in \mathbb{Z}^+$) that makes $F_A \cdot N_A = F_B \cdot N_B$. N_A and N_B can be computed using the Least Common Multiple (lcm) and the Greatest Common Divisor (gcd).

$$N_A = \frac{lcm(F_A, F_B)}{gcd(lcm(F_A, F_B), F_A)} \quad (1)$$

$$N_b = \frac{lcm(F_A, F_B)}{gcd(lcm(F_A, F_B), F_b)} \quad (2)$$

A real-time property may require to limit the time difference between two periodic events. If these two events exhibit the same frequency, N_A equals to N_B. Otherwise, N_A and N_B should be introduced to identify the corresponding occurrence between E_A and E_B.

In Fig. 19, places $Tester_A$/ $Tester_B$ counts the occurring times of events E_A/E_B. The transition *Diff* may consume tokens in $Tester_A$/$Tester_B$ if tokens in $Tester_A$ are superior or equal to N_A and tokens in $Tester_B$ are superior or equal to N_B. Once $Tester_A$ contains $N_A + 1$ tokens, E_A executes at least one time faster than E_B. This exception will be detected using the *Overflow* transitions. The checking assertion is: $\neg(Overflow_A \vee Overflow_B)$.

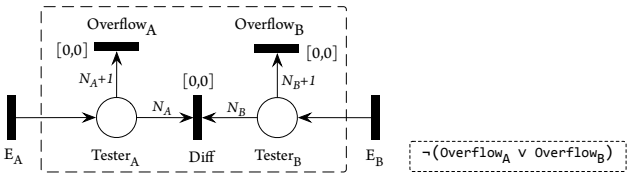

Figure 19. Predicate Observer: Same Frequency between E_A and E_B

4) T(E_A, E_B) > t for minimal Time Interval between Events: This observer is used to check that the time interval between the equivalent occurrences of E_A and E_B is at least t. E_A and E_B can be periodic or aperiodic. Its semantics is equivalent to T(E_A) - T(E_B) > t. Its observer is similar to that of equivalent occurrence between events, except that a transition T_{Delay} is added, representing the time delay for event E_A. We use the following *mmc* assertion to assess this predicate. When E_A and E_B are aperiodic, $N_A = N_B = 1$.

$$\neg(Overflow_A \vee Overflow_B) \wedge \neg((Tester_B = N_B) \wedge (Tester_A < N_A)). \quad (3)$$

5) T(E_A, E_B) < t for maximum time interval between Events: This observer is used to check the time interval between the equivalent occurrences of E_A and E_B is at most t. Its semantics equals to T(E_A) - T(E_B) < t. Its observer is designed as Fig. 21. If the assertion $\neg(Overflow_A \vee Overflow_B)$ is true, then |T(E_A) - T(E_B)| < t is valid. When E_A and E_B are aperiodic, N_A = N_B = 1.

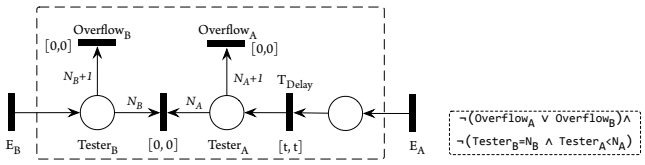

Figure 20. Predicate Observer: Minimum Time Interval between E_A and E_B

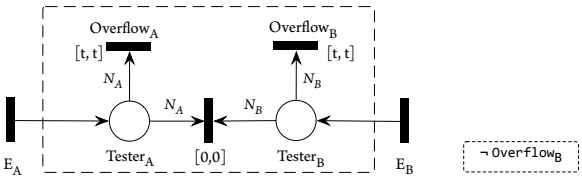

Figure 21. Predicate Observer: Maximum Time Interval between E_A and E_B

6) D(S) ≥ t & D(S) < t for minimal/maximal time duration of a state S: An efficient observer design for the predicate of state duration is to use the PRE function of TTS. In Fig. 22, the transition with constraint [t,t] is enabled when state S holds at least/at most t *u.t.*. The transition with constraint [0,0] will fire when state S does not hold any more. This transition is used to clear the marking in the place *Tester*, as state S may hold several times in the whole system's execution. The *mmc* assertions are respectively: $S \wedge (Tester = 1)$ and $S \wedge (Tester = 0)$.

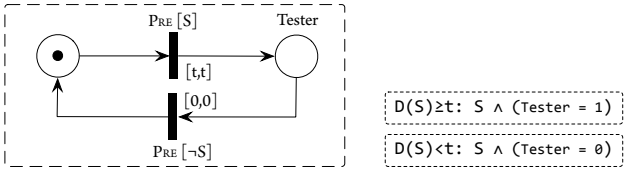

Figure 22. Predicate Observer: Time Duration of State

C. Basic Scope Modifiers

Basic scope modifiers include Global, Before E^i, After E^i, and Between E_A and E_B. Others are compositions of the basic ones.

1) Global: Global scope modifier does not need an observer in TPN. It requires all states of the system, denoted as $\mathcal{A}$.

2) Before E^i & After E^i: The observers for this pair of scope modifiers are designed as the same TPN model, as shown in Fig. 23, but depend on different logic formulae. The place *Tester* counts the occurring times of event E. We use $Tester < i$ (E^i has not yet occurred) to check Before E^i and use $Tester \geq i$ (E^i has occurred) to check After E^i. Note that this scope requires *isFinite(E)* to be true.

3) Between E_A and E_B: *Between E_A and E_B* means between the equivalent occurrences of E_A and E_B. If both E_A and E_B are periodic events, their occurrence frequencies must be equal.

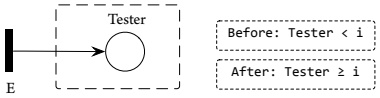

Figure 23. Scope Observer: Before E & After E

If E_A and E_B occur only once, by default their frequencies are equal. Its observer is designed as shown in Fig. 24, where the places $Tester_A$ and $Tester_B$ count the difference of the occurring times between E_A and E_B, and the formula $(Tester_A = 1) \wedge (Tester_B = 0)$ is used for the assessment.

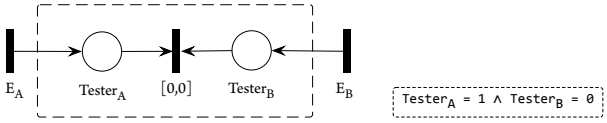

Figure 24. Scope Observer: Between two Events

4) After E_A Until E_B: This scope modifier can be represented by the above ones:

- When E_B occurs after E_A, it is equivalent to (*Exist* E_B *After* E_A) and (*Between* E_A *and* E_B);
- When E_B does not occurs after E_A, it is equivalent to (*Absent* E_B *After* E_A) and (*After* E_A).

D. Occurrence Modifiers

Occurrence modifiers are used to specify the occurrence times of given event/state modifiers within some scope. They are classified as *Exist, Absent, and Always*, and are used together with predicates and scopes. The use of observers for occurrence modifiers is not mandatory. Assume that in the state graph, $N(P)$ is the number of states that match the predicate P, $N(S)$ is the number of states that match the scope S, and $N(P \wedge S)$ is the number of states that match both the predicate and the scope. With respect to the semantics of *Exist, Absent, Always*, the assessments depend on the following assertions:

- *Exist P in S* means that P must occur within S:

$$\begin{cases} N(P \wedge S) \geq 1 & \text{if N(S) > 0;} \\ True & \text{if N(S) = 0.} \end{cases} \tag{4}$$

- *Absent P in S* means that P must not occur in S:

$$N(P \wedge S) = 0 \tag{5}$$

- *Always P in S* means that P occur through the whole S:

$$N(P \wedge S) = N(S) \tag{6}$$

Note that when $N(S) = 0$, the scope is false, then the predicate for any occurrence modifier (*Exist, Absent, Always*) is always true. The assertions for *Absent* and *Always* satisfies this fact by default. The assertion "True, if N(S)=0" for *Exist* is also defined for this purpose.

VI. Example of Real-Time Property Verification

We illustrate the proposed specification and verification approach using a simple example (see Ex. 1).

Example 1: As shown in Fig. 25, two concurrent processes are modeled in TPN. Both execute only once. The target property $\mathcal{P}$ is *Always* E_A *After* E_B *Within* $[1, 2]$.

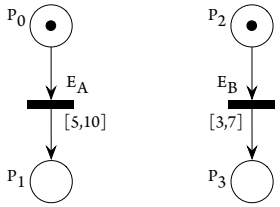

Figure 25. Observer-based Verification Example

On the basis of our specification patterns, $\mathcal{P}$ is composed by (1) the occurrence modifier *Always*, (2) the predicate E_A *occurs*, and (3) the scope *After* E_B *Within* $[1, 2]$, which is equivalent to *Between* $E_B + 1$ *and* $E_B + 2$.

Accordingly, observers for assessing $\mathcal{P}$ are generated using the set of observers defined in our approach, as shown in Fig. 26. The predicate E_A *occurs* is observed by using an atomic event-based observer $\mathbf{obs_4}$. The scope *Between* $E_B + 1$ *and* $E_B + 2$ is observed by using a composite observer with three parts: (i) $\mathbf{obs_1}$ for event modifier $E_B + 1$, (ii) $\mathbf{obs_2}$ for event modifier E_B+2, and (iii) $\mathbf{obs_3}$ for scope modifier *Between* $E_B + 1$ *and* $E_B + 2$.

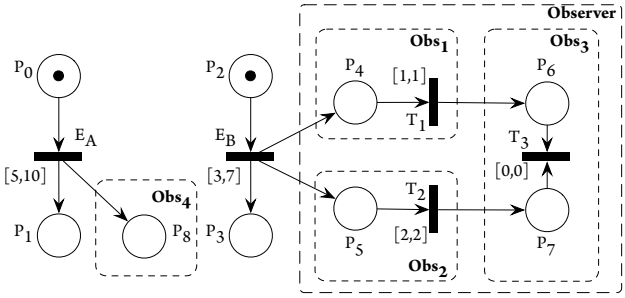

Figure 26. TPN Observers for the Example

Based on the TPN system with additional observers, a state graph of marking abstraction with 10 states and 13 transitions is generated, shown in Fig. 27. Relying on the **muse** model checker, $\mathcal{P}$ is assessed using the following *mmc* formulae:

- Assertion P for the predicate E_A *occurs*: P_8.
- Assertion S for the scope *Between* $E_B + 1$ *and* $E_B + 2$: $P_6 \wedge \neg P_7$.
- Assertion O for the occurrence modifier *Always*: if $N(P \wedge S) = N(S)$, then P is satisfied.

For the assertion O, only one state (S_5) satisfies $P \wedge S$, thus $N(P \wedge S) = 1$; while two states (S_4 and S_5) satisfy S, thus $N(S) = 2$. As $N(P \wedge S) \neq N(S)$, property $\mathcal{P}$ fails.

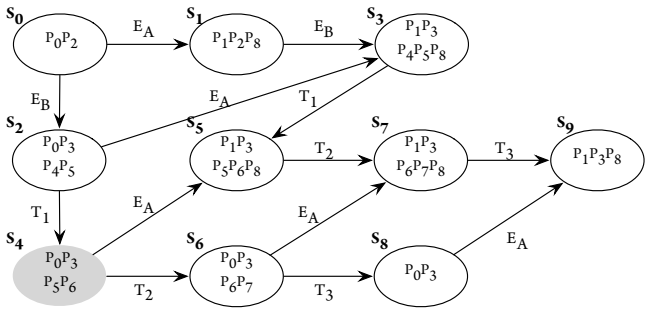

Figure 27. Reachability Graph of Verification Example

$N(P \wedge S) < N(S)$ implies that the states satisfying $\neg P \wedge S$ ($\neg P_1 \wedge P_6 \wedge \neg P_7$) are violating ones. By checking $\neg P_1 \wedge P_6 \wedge \neg P_7$, we find the violating state in *ktz*, which is the state S_4 with marking $P_0P_3P_5P_6$.

VII. Conclusion

Dwyer's and Konrad's pattern systems target expressiveness of real-time requirements for the end-users, and leave the verification related issues to the users. Accordingly, these patterns do not guarantee the efficiency of verification. We have defined a set of atomic constructs in our property pattern system and defined the associated set of elementary observers. End-user real-time requirements are expressed as compositions of these patterns based on a predefined meta-model and a set of mapping rules.

Compared to previous works from one of the author [5], our proposal allows for the automatic generation of composite observers for a composite property. It has also a positive outcome on the verification cost. Our pattern verification framework has been integrated into our UML-MARTE real-time verification framework [11], [12]. This extends our available tooling for UML-MARTE with a process for formal requirement verification and elicitation during the early design phases.

Acknowledgment

This work was funded by the FUI Project P and ITEA2 Project OPEES projects.

References

[1] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[2] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. ACM, 1999, pp. 411–420.

[3] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 372–381.

[4] V. Gruhn and R. Laue, "Patterns for timed property specifications," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 117–133, 2006.

[5] N. Abid, S. Dal Zilio, and D. Le Botlan, "Real-time specification patterns and tools," in *Formal Methods for Industrial Critical Systems*. Springer, 2012, pp. 1–15.

[6] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[7] R. Alur, "Techniques for automatic verification of real-time systems," Ph.D. dissertation, stanford university, 1991.

[8] L. E. Moser, Y. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon, "A graphical environment for the design of concurrent real-time systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 1, pp. 31–79, 1997.

[9] K. C. Castillos, F. Dadeau, J. Julliand, B. Kanso, and S. Taha, "A compositional automata-based semantics for property patterns," in *International Conference on Integrated Formal Methods*. Springer, 2013, pp. 316–330.

[10] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA–construction of abstract state spaces for Petri nets and Time Petri nets," *International Journal of Production Research*, vol. 42, no. 14, 2004.

[11] N. Ge and M. Pantel, "Time properties verification framework for uml-marte safety critical real-time systems," in *European Conference on Modelling Foundations and Applications*. Springer, 2012, pp. 352–367.

[12] N. Ge, M. Pantel, and X. Crégut, "A UML-MARTE temporal property verification tool based on model checking," in *International Conference on Embedded Real Time Software and Systems (ERTS)*, 2014.

[13] N. Ge, M. Pantel, and X. Crégut, "Time properties dedicated transformation from UML-MARTE activity to time transition system," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 4, pp. 1–8, 2012.

[14] N. Ge and M. Pantel, "Real-time property specific reduction for Time Petri net," in *International Workshop on Petri Nets and Software Engineering (PNSE@PetriNets)*, 2014, pp. 165–179.

[15] N. Ge, M. Pantel, and X. Crégut, "Automated failure analysis in model checking based on data mining," in *International Conference on Model and Data Engineering*. Springer, 2014, pp. 13–28.

[16] P. Merlin and D. Farber, "Recoverability of communication protocols–implications of a theoretical study," *Communications, IEEE Transactions on*, vol. 24, no. 9, pp. 1036 – 1043, 1976.

[17] N. Abid, S. Dal Zilio, and D. Le Botlan, "A formal framework to specify and verify real-time properties on critical systems," *IJCCBS*, vol. 5, no. 1/2, 2014.

[18] M. Garnacho, J. Bodeveix, and M. Filali-Amine, "A mechanized semantic framework for real-time systems," in *Formal Modeling and Analysis of Timed Systems FORMATS*, 2013.