View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science 153 (2006) 117–133
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Patterns for Timed Property Specifications

Volker Gruhn[2] , Ralf Laue[3]

Chair of Applied Telematics / e-Business
Computer Science Faculty, University of Leipzig
Leipzig, Germany [1]

Abstract

Patterns for property specification enable non-experts to write formal specifications that can be used for automatic model checking. The existing patterns identified in [6] allow to reason about occurrence and order of events, but not about their timing. We extend this pattern system by patterns related to time. This allows the specification of real-time requirements.

Keywords: patterns, formal specification, real-time, verification

1 Introduction

The formal specification of real-time requirements is an error-prone task. Often developers are not familiar with existing formalisms and regard it as too difficult to use timed temporal logics to specify a system. On the other hand, model checking tools that can be used to verify the correctness of a system often require specifications given in temporal logics formulas.

Property specification patterns were successfully used to bridge this gap between practitioners and model checking tools. The existing pattern system does not yet consider information about time. We present a catalogue of patterns for real-time requirements. For each pattern, we construct observer automata (observers) that can be applied directly in timed model checking

[1] The Chair of Applied Telematics / e-Business is endowed by Deutsche Telekom AG
[2] Email: gruhn@ebus.informatik.uni-leipzig.de
[3] Email: laue@ebus.informatik.uni-leipzig.de

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2005.10.035

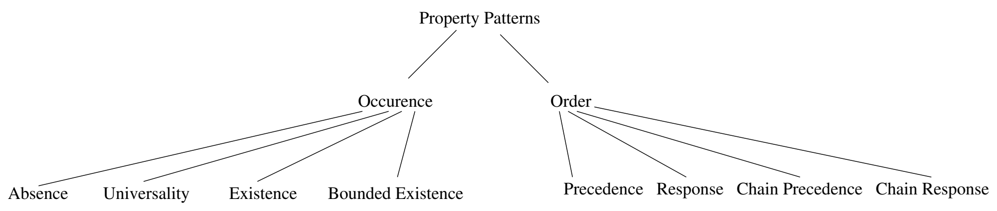

Fig. 1. Pattern Hierarchy

tools. The user who uses the pattern system does not have to care about these observers, it is possible to construct them automatically. This is an advantage to current approaches which often allow to construct a model in the input language of a model checker automatically from high level languages like UML but do not offer tool-support for specifying the requirements that have to be verified.

2 Untimed Specification Patterns

Dwyer and his colleagues at Kansas State University developed a pattern system for property specification [6]. They collected 555 specifications from several sources and found that 92% of them matched one of the patterns from their system. This pattern system enables people who are not experts in temporal logic to read and write formal specifications in a variety of formalisms. With the help of this system, properties like “An occurrence of event A must be followed by an occurrence of event B” can be expressed. The pattern system does, however, not include timed properties like “An occurrence of event A must be followed by an occurrence of event B within k time units”. Before discussing about extending the pattern system to include information about time, we will outline the pattern system. It is explained more deeply in [6] and [5].

The patterns are organised in a hierarchy, see Fig. 1. A property specification consists of a *pattern* (which describes *what* must occur) and a *scope*, which describes *when* the pattern must hold.

In the following survey, the patterns are described briefly. The meaning of variables (like P) is always: “An event P from a given disjunction of events occurs”.
Absence: P does not occur within a scope.
Universality: P occurs throughout a scope.
Existence: P must occur within a scope.
Bounded Existence: P must occur at least / exactly or at most k times within a scope.
Precedence: P must always be preceded by Q within a scope.

Response: P must always be followed by Q within a scope.
Chain Precedence / Chain Response: A sequence $P_1, \dots P_n$ must always be preceded /followed by a sequence $Q_1, \dots, Q_m$ within a scope.

Scopes define, *when* the above patterns must hold:
global: the pattern must hold during the complete system execution.
before: the pattern must hold up to a given event X.
after: the pattern must hold after the occurrence of a given X.
between: the pattern must hold from the occurrence of a given X to the occurrence of a given Y.
until: the same as "between", but the pattern must hold even if Y never occurs.

For our purposes, we will use an event-based formalism. This allows us to write something like "the point of time, when event P occurs". We will abbreviate this point of time by t(P) and use terms like $t(P) \pm k$ for "k time units after/before the occurrence of P".

If we add information about time to the pattern system, we have to consider the following elements of specification patterns:

(i) **The events**: Instead of just specifying that "X occurs", it should be possible to reason about "combined" events like "X occurs twice in n time units". We will discuss this in section 5.

(ii) **The patterns**: We want to be able to specify properties like time-bounded Response ("P must always be followed by Q within k time units"). We will discuss such patterns in section 4.

(iii) **The scopes**: We want to delimit the period of validity for a pattern by scopes like "after t(P)+k". Scopes will be discussed in section 6.

Before we start to examine the events, patterns and scopes in detail, we will discuss the use of timed observers and give some definitions in the following chapter.

3 Timed Observer Automata

The main result of Dwyer's work was the pattern catalogue [6,5]. This catalogue gives the mappings of property specifications into various formalisms. For example, the property specification "Globally, S responds to P" is expressed as $AG(P \Rightarrow AF(S))$ in CTL or as $\Box(P \Rightarrow \Diamond S)$ in LTL. Because many model checking tools allow to check properties specified in these formalisms, the pattern catalogue can be applied directly with these tools.

Of course, it would be possible to use timed temporal logics to built a similar catalogue for timed property specifications. For example, we can use TCTL [10] to express the time-bounded response property “Globally, S responds to P within k time units” as $AG(P \Rightarrow AF_{<k}(S))$.

While such a catalogue could be interesting for theoretical purposes, its practical use would be limited, because timed model checking tools do not offer the same support for timed temporal logics as untimed model checking tools do with untimed temporal logics.

For this reason, we use the concept of (timed) observer automata (observers) to describe the desired system behaviour. Intuitively, observers run in parallel with the model under verification. They reach a certain state if and only if some property can be violated in the model.

Timed automata (TA) were introduced in the thesis of Alur [2] as an extension of Büchi automata. We assume familiarity with the concept of TA and only give a brief (incomplete) review of the definition of TA. We also use urgency and communication between different TA using synchronisation labels as extensions to the original definition of TA.

As a finite Büchi automaton, a TA has a finite set L of locations and a set $T \subseteq L \times L$ of transitions. One location $l_0 \in L$ is a distinguished initial location. The transitions are labeled by a function $Label : L \rightarrow 2^{AP}$ which assigns a set of atomic propositions to each location. Additionally, there is a finite set C of clocks running at the same speed. Transitions can be associated with a subset of C, indicating the clocks that have to be reset to zero when the transition is taken.

Clock conditions are of the form $x \sim y$ where $\sim$ stands for an operator from the set $\{<, >, =, \leq, \geq\}$, $x \in C$, and $y \in C$ or y is a natural number. Clock conditions can be associated to a transition. A transition may only be taken if its clock conditions are true.

TA can be executed as follows: In the beginning, the automaton is in the initial location, and all clocks are set to zero. Afterwards, TA can proceed in two ways: The location can be changed by taking a transition or time can progress while staying in the same location. Mostly, there is more than one possible way to proceed: It is possible to stay in the location and let time pass or to take one of the possible transitions from this location. This choice is made non-deterministically. This non-determinism is, however, restricted if transitions are marked as urgent. Urgent transitions *must* be taken as soon as possible. If more than one urgent transition can be taken, one of them must be selected at random. A non-urgent transition cannot be taken if it would be possible to take an urgent one.

To model communication and synchronisation between various TA, transitions can be labeled with synchronisation labels `X!` (signalising an event X) and `X` (the corresponding reaction to the event X). If a transition labeled with `X!` in a TA is taken, this means that *all* transitions labeled with `X` in other automata (that are not forbidden by clock conditions or urgency of other transitions) can be taken without delay. An automaton with a transition with a label `X` is blocked until a corresponding transition labeled with `X!` is taken. On the other hand, a transition labeled with `X!` will not be blocked due to the fact that no transition labeled with `X` can be taken. Transitions can have at most one synchronisation label of the form `X!`, and one transition label of the form `X`. (In our event-based formalism we do not allow two events to coincide.)

Synchronisation labels are important for synchronising the model under verification with observers "querying" the system behaviour: If "something interesting" occurs in the model, the observers can react immediately.

4 Pattern Catalogue: Patterns

We will see that the majority of our specification patterns deal with so called safety properties. In order to prove that such a property is true, it is sufficient to check that the observer cannot reach some location(s). A violation of the specification (a counterexample) is detected when the observer *can* reach such a location.

For liveness properties (like "every occurrence of P is followed by an occurrence of Q"), reasoning about infinite runs is necessary. As usual, we use *acceptance* conditions for this purpose: Some locations in the observer are marked as accepting locations. A counterexample is detected, if there is a non-Zeno run entering an accepting location infinitely often. (We omit the formal definition of non-Zenoness here. Intuitively, a non-Zeno run is one where it is forbidden to take infinitely many transitions in a finite amount of time.)

We follow the usual convention to depict the locations of the automata by circles and the transitions by arrows. A violation of a specification is detected when a "bad" location can be reached in the observer. We call such locations *error locations* and mark them with the word ERROR at the circle. Accepting locations are indicated with a double circle, and we define that the leftmost location in a figure showing a TA is always the initial location. We call locations that are neither error not accepting locations *normal locations.*

4.1 Absence and Universality Patterns

A violation of the Absence property (P does not occur within a scope) is detected when P occurs: The observer automaton is very simple:

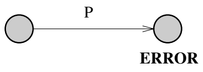

Fig. 2. Absence

Universality is a dual of the Absence property. In the event-based logic used in this paper, "some property p holds throughout a scope" is the same as Absence of the event "p becomes false".

4.2 Existence Patterns

A violation of the Existence property (P must occur within a scope) means, that a run exists which never reaches the location depicted by the right circle of the observer in Fig. 3. This means that a counterexample for the Existence property is a non-Zeno run of the automaton that passes through the accepting location infinitely often.

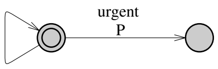

Fig. 3. Existence

The most important timed Existence property is "Starting from the current point of time, P must occur within k time-units", expressed as formula $\Diamond_{\leq k}P$ in TLTL. The observer for this property is shown in Fig. 4.

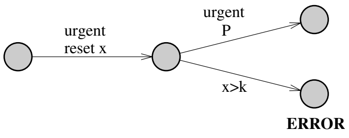

Fig. 4. Time-bounded existence

4.3 Response Patterns

The Response property (P must always be followed by Q within a scope) was the most common property in the 555 example specifications collected for [6]. An observer for this property is shown in Fig. 5.

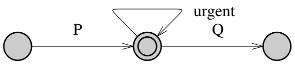

Fig. 5. Response

4.3.1 Time-Bounded Response: response must occur before t(P)+k

Often, we do not just want to specify that P must be followed by Q, but Q must occur within a given time span. Fig. 6 shows an observer for "P must be followed by Q within k time-units after the occurrence of P". Observers for similar properties like "response must occur after t(P)+k" can be constructed in a similar way.

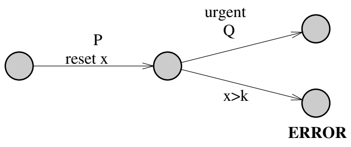

Fig. 6. Time-Bounded Response

4.4 Precedence Patterns

The Precedence property (P must always be preceded by Q within a scope) requires that no P occurs before it is "enabled" by a preceding Q. Fig. 7 shows an observer for this property.

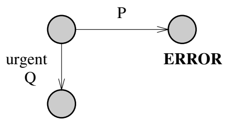

Fig. 7. Precedence

From this property, we can develop two timed properties, for which we use the term "Q enables P" to express that Q is a necessary pre-condition for P.

4.4.1 Precedence: Q enables P after a delay

For this property, P is enabled only if Q occurred and the time is greater than or equal to t(Q)+k. (It remains enabled even if another event Q occurs, i.e. the "new" event Q would not lead to a longer delay.) The observer for this property is shown in Fig. 8.

4.4.2 Precedence: Q enables P for k time units

For this property, P is enabled only if Q occurred and the time is less than or equal to t(Q)+k. This means, that Q enables P only for a time span of k time units. If P occurs when the last occurrence of Q was before more than k time units, this will violate the specification. Fig. 9 shows the observer for this property.

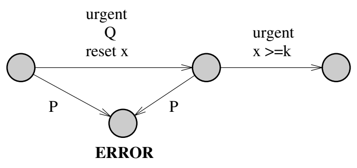

Fig. 8. Precedence with Delay

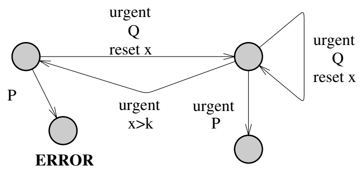

Fig. 9. Time-Restricted Precedence

4.5 Bounded Existence, Chain Precedence and Chain Response Patterns

Using the combined events "n times A" (see section 5), Bounded Existence properties can be reduced to Existence of the combined event. In the same way, Chain Precedence and Chain Response properties can be reduced to Precedence and Response properties.

5 Pattern Catalogue: Events

Using synchronisation labels, a TA can "observe" the occurrence of an event in another TA and react in some way if the event occurs and synchronisation can take place. However, often we want to react to "observations" which are related to more than one event or to the time when it occurs. We will use the term "combined events" for observations like "P occurs n times" or "both P and Q occur within a time span of no more than k time units."

To handle these combined events, we construct *reporting TA* which can take certain transitions if and only if the combined event happens. These transitions can be labeled with a new synchronisation label `M!`. In this way the reporting TA can signalise the combined event in the same way as "simple" events are signalised.

In some cases, it is useful to have two different kinds of reporting TA: Ones that *will always* report the occurrence of a combined event if it occurs, another ones that *can* report the combined event, but are not obliged to do so everytime the combined event occurs. We will call the second kind of reporting TA *lazy*.

To illustrate the use of both kinds of reporting TA, consider the Chain Response pattern "A sequence of two events A and B must always be followed by a sequence of two events C and D". We assume that one reporting TA signalises the occurrence of the A-B sequence, another one the occurrence of the C-D sequence. If we check this property with a model checker, we ask: "Is there a possible run of the system that violates the specification?". If there is such a counterexample, it *will* be detected by the model checker - even if the TA that reports the A-B sequence is lazy, because the lazy reporting TA *can* report the A-B sequence. On the other hand, the reporting TA that signalises the occurrence of the C-D sequence must not be lazy, because this could lead to a false counterexample where A-B occurs, but thereafter C-D will not be reported by the lazy reporting TA.

The use of lazy reporting TA can reduce the number of locations and transitions in the reporting TA. Unless explicitely stated as lazy, the reporting TA introduced in this chapter are non-lazy.

5.1 Chains

Dwyer [6] calls a sequence of events $E_1 \ldots E_n$ a *chain*. We consider the occurrence of such a sequence as a combined event. Our experience with real specifications is that such chains are regarded as being "non-overlapping", i.e. a new chain cannot start as long as an already started chain is not yet completed. For this reason, we will discuss only such "non-overlapping" chains here. A lazy reporting TA for a chain of three events A,B,C is shown in Fig. 10. (We get the corresponding non-lazy TA by marking all transitions as urgent).

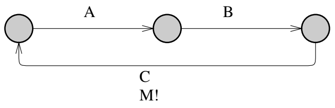

Fig. 10. Chain of three events A,B,C

5.2 Time-Bounded Chains

We call the occurrence of a sequence of events $E_1 \ldots E_n$ within a given time span less or equal to k time units a time-bounded chain. A lazy reporting TA for a time-bounded chain of three events A,B,C is shown in Fig. 11.

The non-lazy version is more complex (Fig. 12) [4]

[4] In Fig. 12, the rightmost location is not necessary. It can be deleted and its incoming transitions can be drawn to the initial location instead. While this would save one location, the TA as drawn in Fig. 12 looks less complicated.

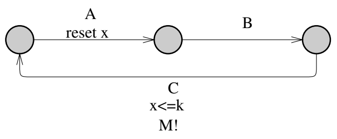

Fig. 11. Lazy reporting TA for a time bounded chain of three events A,B,C

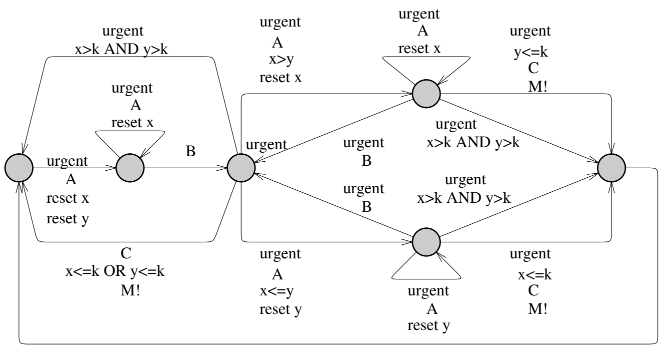

Fig. 12. Non-lazy reporting TA for a time bounded chain of three events A,B,C

5.3 *"n times A"*

The combined event "A occurs n times" is a special case of a chain. Again, we want to consider the "non-overlapping" case only, i.e. the event "A occurs 3 times" will be reported at the 3rd, 6th, 9th etc. occurrence of A, but not at the 4th, 5th etc. The reporting TA for "A occurs three times" can be derived directly from Fig. 10 (by substituting A, B and C by urgent action A).

5.4 *"n times A within k time units"*

The non-lazy reporting TA shown in Fig. 13 is much easier than the one for the A,B,C-chain in Fig. 12.

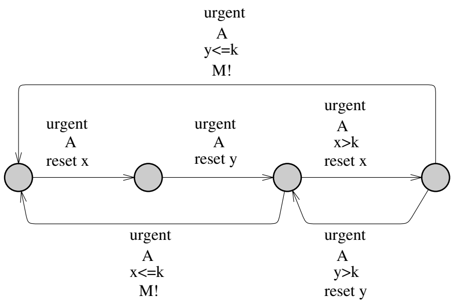

Fig. 13. three times A within k time units

Please note that the clock conditions (like `x>k`) are checked *before* the transition is taken and the "reset clock x" can take effect.

5.5 Collections

We call combined events like "A, B and C occur (regardless of the order between these events)" a collection. Fig. 14 shows how the occurrence of the collection "A, B and C occur" can be reported using a combination of four TAs: The first three TAs report the occurrence of A, B and C resp. to the fourth one which reports the collection using its synchronisation label `M!`.

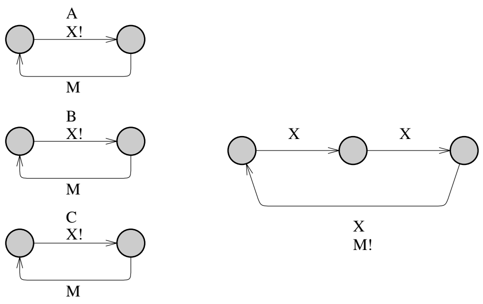

Fig. 14. Reporting TA for a collection of A, B and C

5.6 Time-Bounded Collections

Time-bounded collections like "A, B and C occur within a time-span of k time-units (regardless of the order between these events)" can be reported by a combination of TAs as shown in Fig. 15.

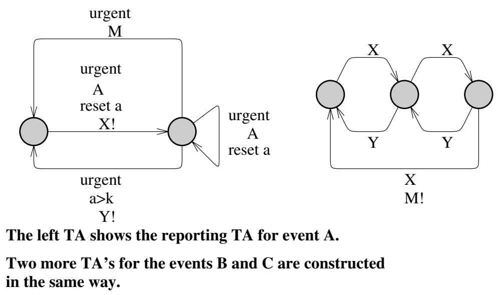
urgent
M
urgent
A
reset a
X!
urgent
A
reset a
urgent
a>k
Y!
X
X
Y
Y
X
M!
The left TA shows the reporting TA for event A.
Two more TA's for the events B and C are constructed in the same way.

Fig. 15. Reporting TA for a time-bounded collection of A, B and C

5.7 Non-occurrence in a given time span

If we switch from an event-based to a state-based view, event A can mean "a property becomes true" and event B can mean "the property becomes false". Then the specification "A is not followed by B within a time-span of k time units" means that "the property stays true for at least k time units". A lazy reporting TA for this specification is shown in Fig. 16. A non-lazy reporting TA would require to remember a potentially infinite number of times when A occurred, which prevents us from constructing one.

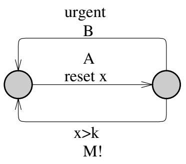

Fig. 16. A is not followed by B within k time units

6 Pattern Catalogue: Scopes

Let **A** be the observer for some property. It can observe whether the property holds *globally*, i.e. during the entire execution of the model. In this section, we will show how **A** can be changed in order to check the property *over a given scope*. Following the definition of untimed scopes given by Dwyer (see section 2), we will show how **A** can be changed to check the validity of a property before, after and until $t(D) \pm k$, where k is an integer, expressing some units of time. This allows us to write timed specifications like "Something must happen within at least 10 minutes after the system has been started".

We define the time intervals defined by the scopes as open at both ends, for example "before t(D)" does not include the point of time t(D).

Note that D can be a combined event. This adds more flexibility to the original pattern system, because it allows us to write specifications like "After the third occurrence of P, some property must hold".

The key idea to handle the scopes is to transform **A** into a scope-dependent TA **A'**. To check a property over a given scope, we either search for error runs in **A'** itself or we construct another TA **O** which acts as an observer for **A'** and reaches an error state if and only if an error run in **A'** can be found. The model must be built such that transitions in the reporting TAs can be taken *before* transitions in **A'**, and transitions in **A'** can be taken before transitions in **O**. The construction algorithms for the automata will be given in the following sections.

Before showing how **A'** and **O** can be constructed, we would like to recall two definitions from [6]:

(i) If the scope delimiters do not occur in a system execution, the specification will be true by definition. For example, the specification "P must occur before X" will always be true if there is no X at all in the system execution.

(ii) In patterns like "Before R, S responds to P", the specification is violated if P occurs before R, but the responding S occurs after R. Analogously, in patterns like "After R, P must be preceded by Q", the event Q must occur after R to "enable" P.

Due to space restrictions, we explain the procedure of constructing and using **A'** and **O** for before...-scopes only.

6.1 Scope "before t(D)"

To construct **A'** from **A**, we search for a normal location without outgoing transitions in **A** and call it $sink_{normal}$. If there is no such location, we add $sink_{normal}$ as a new normal location. Also, we search for an error location in **A** and call it $sink_{error}$. If there is no such location, we add a new error location $sink_{error}$. From each normal location in **A**, we add an urgent transition with a synchronisation label D (in order to react to the occurrence of event D) to $sink_{normal}$. From each accepting location in **A**, we add an urgent transition with a synchronisation label D to $sink_{error}$. Also we remove all self-loops without labels (clock-reset, synchronisation or guards) at the accepting locations. This procedure results in the TA **A'**. A violation of a specification within a "before t(D)"-scope is found if **A'** can reach its error state.

6.2 Scope "before t(D)+k"

For a scope "before t(D)+k", we construct a reporting TA that signalises the event "The point of time t(D)+k has been reached" by generating an event M, as shown in Fig. 17. This reduces the case "before t(D)+k" to "before t(M)", for which a solution has been given in 6.1.

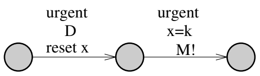

Fig. 17. Reporting TA for "time is t(D)+k"

6.3 Scope "before t(D)-k"

To construct **A'** from **A**, we add a synchronisation label `B!` (=entering a "bad" location) to all transitions in **A** that lead to an error location or to an accepting location. We add a synchronisation label `G!` (=entering a "good" location) to all transitions from an accepting to a normal location. Finally, we remove all self-loops without labels at the accepting locations.

The observer **O** shown in Fig. 18 observes **A'** and reaches its error state if and only if the property observed by **A** can be violated before t(D)-k.

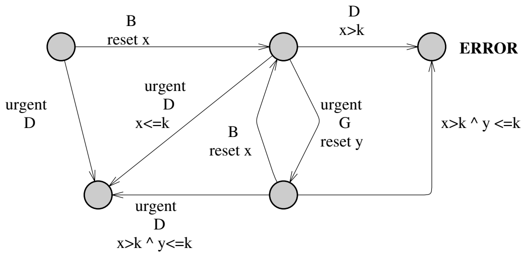

Fig. 18. Observer automaton **O** for "before t(D)-k"

7 Implementation

The general procedure for implementing observers that check a property of the model in a model-checking tool is as follows: The reporting TAs and the observers must be scheduled together with the model under verification in a round-robin manner such that each step of the model is followed by a step of each reporting TA and a step of each observer without letting time pass. Hereby, it is necessary to pay attention to the hierarchy of synchronisation labels: The TAs whose run depends on events reported by other TAs must be scheduled after the TAs that can report something to them. Usually, this means that after [5] each step of the model under verification, each reporting TA has the chance to evolve and finally the observers are scheduled. If reporting TAs are cascaded to report complex combined events, the TA that signalise an event using a synchronisation label `X!` must be scheduled before the TAs with the corresponding label `X`. For the TAs that observe the occurrence of a collection (see Fig. 14 and 15) the reporting TAs with the `X!` synchronisation labels must be scheduled before *and* after the TA with the `M!` label.

[5] Even if we use the word "after", please keep in mind that there is no time-difference between model, reporting TAs and observers.

8 Related Work

The original pattern catalogue was introduced in [6], a survey of property specifications was published in [7]. The pattern system was successfully used for creating the Bandera Specification Language [4] which allows software developers to write specifications using a structured-English language front-end.

[13] uses a similar pattern system which includes time-bounded Existence and time-bounded Response as time-related patterns. To develop a pattern system for real-time requirements was also proposed by [15], but to actually collect these patterns was not the main focus of this work.

The use of TA for specifying temporal properties is quite common and used by several authors. Examples are [14] and [3]. The latter states: "According to our experience, automata based notations turned out to be simpler than most logics for describing sequences of events". However, usually the observers have to be constructed by hand, even if the translation of the model itself into the input language of a model checker can be done automatically.

[9], [8], [11] and others[6] use timed UML models to specify the system and its properties. Thus sequence diagrams, OCL constraints or UML state machines (acting as observers) serve as property specification language. This is a great advantage for experienced UML users, but the construction of the properties must still be done by hand.

[1] introduces a visual language to specify event-based real-time requirements, and they have constructed a tool that translates these requirements into the input language of the model checker Kronos. This is an interesting approach: The user has still to learn a new formalism, but the visual language is easy to understand. However, we see a major drawback in the way how properties have to be specified: The user has to graphically describe the scenarios which *violate* the requirements. Often this is difficult even for experts, and the authors involuntarily proved this fact in one of their examples: To violate the requirement that "a stimulus e is not followed by any response within 100 time units", they described only the scenario where the response followed more than 100 time units later and forgot the case that it never occurs.

9 Conclusions and Directions for Future Research

We believe that the proposed pattern system helps to specify the properties for verifying the correctness of a system using model-checking tools. Because the system is most useful if the observers are generated automatically in the

[6] [9] mentions a number of other papers in the bibliography.

input language of existing model checking tools, we will develop tool-support for this task as a next step. The practitioner who uses a "translation" from specifications in structured-English into a language that can be processed by a model checking tool, does not need a deep knowledge in temporal logics.

We believe that the vast majority of real-world specifications are instances of patterns in our system. This is supported by the fact that our patterns can express all properties that can be specified using the formalism presented in [1]. We should, however, evaluate the completeness of our pattern system by surveying an appropriate number of real-world specifications. If necessary, the pattern system will be updated as a consequence of this study.

10 Postscriptum

After our paper has been presented at the QAPL workshop, it came to our attention that another research team has published a similar system for real-time specification patterns independently from us [12]. Unfortunately, we could not yet compare both approaches, but we will do so as soon as possible. We expect that both approaches have strengthes that can be exploited in further resarch.

References

[1] Alfonso, A., V. A. Braberman, N. Kicillof and A. Olivero, *Visual timed event scenarios.*, in: *26th International Conference on Software Engineering (ICSE 2004)* (2004), pp. 168–177.

[2] Alur, R. and D. L. Dill, *Automata for modeling real-time systems*, in: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming* (1990), pp. 322–335.

[3] Braberman, V. A. and M. Felder, *Verification of real-time designs: Combining scheduling theory with automatic formal verification.*, in: *ESEC / SIGSOFT FSE*, 1999, pp. 494–510.

[4] Corbett, J. C., M. B. Dwyer, J. Hatcliff and Robby, *A language framework for expressing checkable properties of dynamic software*, in: *SPIN*, 2000, pp. 205–223.

[5] Dwyer, M. B., *Specification patterns web site, available at http://patterns.projects.cis.ksu.edu.*

[6] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Property specification patterns for finite-state verification*, in: *FMSP '98: Proceedings of the second workshop on Formal methods in software practice* (1998), pp. 7–15.

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, in: *Proc. of the 21st international conference on Software engineering* (1999), pp. 411–420.

[8] Flake, S. and W. Müller, *Specification of real-time properties for UML models*, in: *Proceedings of the Hawai'i International Conference on System Sciences (HICSS-35)*, IEEE, Hawaii, USA, 2002.

[9] Graf, S., I. Ober and I. Ober, *Model checking of UML models via a mapping to communicating extended timed automata*, in: S. Graf and L. Mounier, editors, *Proceedings of SPIN'04 Workshop, Barcelona, Spain*, LNCS **2989** (2004).

[10] Henzinger, T. A., X. Nicollin, J. Sifakis and S. Yovine, *Symbolic Model Checking for Real-Time Systems*, in: *7th. Symposium of Logics in Computer Science* (1992), pp. 394–406.

[11] Knapp, A., S. Merz and C. Rauh, *Model checking - timed UML state machines and collaborations*, in: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* (2002), pp. 395–416.

[12] Konrad, S. and B. H. C. Cheng, *Real-time specification patterns*, in: *Proceedings of the 27th International Conference on Software Engineering (ICSE05)*, St Louis, MO, USA, 2005.

[13] Letier, E. and A. van Lamsweerde, *Deriving operational software specifications from system goals*, in: *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering* (2002), pp. 119–128.

[14] Ober, I. and A. Kerbrat, *Verification of quantitative temporal properties of SDL specifications*, in: *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML* (2001), pp. 182–202.

[15] Roubtsova, E. E., J. van Katwijk, W. J. Toetenel, C. Pronk and R. C. M. de Rooij, *Specification of real-time systems in UML.*, Electr. Notes Theor. Comput. Sci. **39** (2000).