

**Runtime Verification of Critical Systems
Based on Temporal Patterns**

Research Report

István Majzik

**Budapest University of Technology and Economics
2018.**

Contents

- 1 Introduction 3
- 2 The General Context..... 3
 - 2.1 On-line Verification 3
 - 2.2 Monitor Synthesis 4
 - 2.3 A Temporal Logic Variant for Capturing Requirements 5
 - 2.3.1 The Operators of the Logic..... 5
 - 2.3.2 The Semantics of the Logic..... 8
- 3 Describing Event Patterns..... 9
 - 3.1 Previous Work 9
 - 3.2 The Pattern Library.....10
 - 3.3 Abstract Syntax for a Graphical Pattern Language15
 - 3.4 Tool Support to Combine Patterns.....17
- 4 Conclusions19
- 5 References20

1 Introduction

This research report describes a technology for supporting the runtime verification of the behaviour of critical embedded and cyber-physical systems. Here a *pattern-based approach* is described for capturing the requirements that form the basis of runtime monitoring. On the basis of the captured requirements, the source code of monitor components can be generated automatically. The pattern based approach captures *safety* and *liveness* properties (for monitoring safe and correct behaviour during execution).

In Section 2, the general context of the work is presented, including the temporal logic variant that is supported. Section 3 describes the event patterns and the specification of a graphical tool to support the composition of patterns.

2 The General Context

This section describes the general concept of runtime verification in order to put into context the definition of the pattern based requirements capturing approach presented in the subsequent sections of the document.

2.1 On-line Verification

On-line verification by runtime monitoring addresses the detection of errors and malfunctions that manifest themselves in runtime (e.g., due to random hardware faults, configuration faults, operator faults, faults in adaptation etc.). Such kind of error detection is especially important in safety-critical systems, where one of the basic principles for assuring safe behaviour is *reactive fail-safety*: proper detection and handling of hazardous errors that occur in system components implementing a safety-related function. This principle appears, among others, in IEC 61508, the generic standard for safety-related electronic systems.

Accordingly, on-line verification uses runtime monitor components that observe the behaviour of the components (the trace of states, events, actions, and the perceived context), detect the hazardous situations, and trigger a reaction to maintain safety (e.g., to stop the system). In the typical case, these monitor components are implemented as additional software components.

Automated construction of monitors (by the synthesis of their source code) is possible on the basis of a proper language that captures the requirements (rules for safe and correct behaviour). For this purpose, high-level languages like sequence diagrams, reference automata, temporal logics can be used. On the basis of this specification, the synthesis tool automatically generates the source code of the monitor components that detect the critical situations.

Note that monitors are useful not only in runtime (to detect operational faults), but also during testing: The monitors form part of the test oracle that decides whether the behaviour is acceptable considering the execution of a given test suite (i.e., a set of test traces is evaluated).

The following use cases of monitoring and runtime verification are supported:

- *Behaviour monitoring of software components*: In this use case the goal is to check the internal behaviour of the component, detecting in this way all errors that influence the states (state variables) and the control flow of the component. The monitored component is instrumented in order to send to the monitor information (signatures) that allow the identification of the internal states. The monitor receives this run-time information and compares it with the reference behaviour that defines the states and state transitions that are allowed (i.e., accepted by the monitor without detecting an error).
- *Trace-based monitoring of single components*: In this use case the goal is to check the externally observable behaviour of a component (when the instrumentation nec-

essary for checking its internal behaviour is not possible). The monitor observes the timed sequence of inputs and outputs on the interface of the component (together with context and configuration related information) and decides whether the run-time sequence of these events is conformant with the reference information that is given as a set of allowed traces. Basically, reference traces capture the externally observable operation of a single component.

- *Trace-based monitoring of interacting components*: In this use case the goal is to check the interactions between components. The monitor observes the sequences of inputs and outputs on the interfaces of multiple interacting components (together with context related information) and decides whether the run-time sequence of these events is conformant with the reference information that is given as a set of allowed traces. Basically, traces represent here the correct execution of interactions (protocols) among multiple components.

A typical application of these monitoring approaches is the checking of *safety properties* (“something bad never happens”) and *liveness properties* (“something good will eventually happen”). In this case the monitors observe the execution and evaluate the reachability of specified situations. Reachability is described using operators like “eventually”, “always”, “until”, “potentially always”, and “leads to”, potentially with time information. An example reachability property that can be checked by a monitor is the following: “Whenever a state Stop is reached, it implies that no Speedup actions are executed until the event Restart is received”. The source code of the monitors is generated in such a way that they observe and evaluate the execution trace on the basis of these expressions.

2.2 Monitor Synthesis

As mentioned above, on-line monitoring and verification is supported by the automated generation of the source code of the monitor components on the basis of the properties to be monitored. To do this, it is necessary to define the language that is used to describe the properties relevant for on-line verification. Moreover, it is necessary to develop the algorithms to be used by the monitors to evaluate the properties. These algorithms will be realized by the monitor source code generator tool (Figure 1) that (together with the component instrumentation technologies) is made available to the developers.

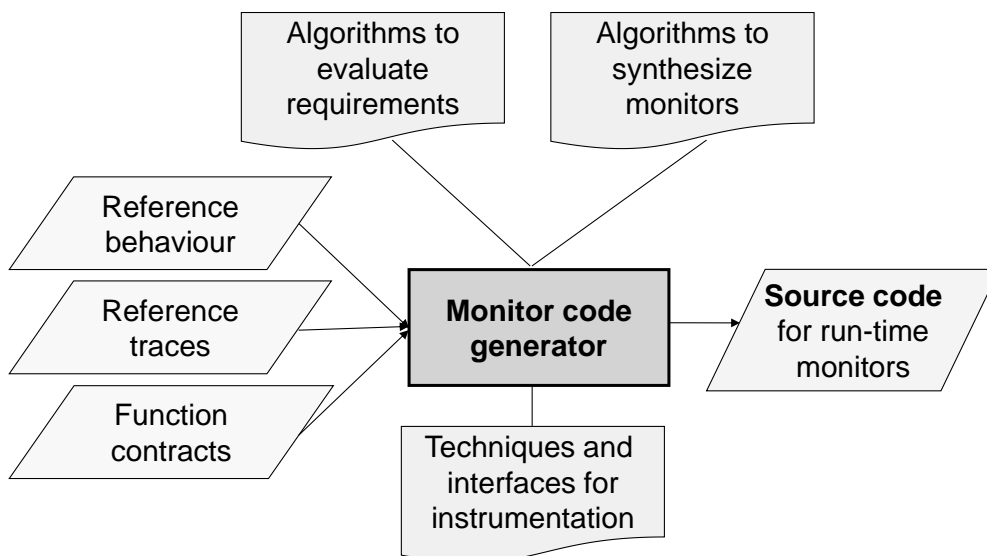


Figure 1. Supporting on-line verification by monitor code generation

In order to apply this kind of on-line verification, the following inputs are needed:

- Description of properties (hazardous situations) to be monitored. These will be formalized and form the basis of monitor code generation.
- Interfaces to observe the trace of states/events/actions to be monitored, or source code of the components for instrumentation.

In the following we focus on the description of properties based on the states/events/actions that are defined as atomic propositions.

2.3 A Temporal Logic Variant for Capturing Requirements

We use a temporal logic language to represent requirements for runtime monitoring. A new variant of linear temporal logics called Context-aware Timed Propositional Linear Temporal Logic (CaTL) is used that (besides having the usual temporal operators as “next”, “until”, “eventually” and “globally”) may include observable events, actions, perceived context, configuration, and also time. This way the context-aware real-time behaviour of systems can be monitored.

Propositional Linear Temporal Logic (PLTL) [1] is extensively used for defining requirements, and particularly popular in runtime verification frameworks. PLTL expressions can be introduced as logic expressions that can be evaluated on a trace of steps, in which each step can be characterized by atomic propositions. Here atomic propositions are local characteristics of the step that may include all elements of a monitored execution trace that are relevant from the point of view of property monitoring: function call, function return, input or output signal, message received or sent, timer started or expired, state entered or left, context change, configuration change, predicate on the value of a variable etc. Later, we will call these atomic propositions in general as “events”, and the trace of steps is the “trace of events”.

Besides the usual Boolean language operators, PLTL has the following temporal operators:

- X: “Next” operator ($X P$ means that the next step in the trace shall be characterized by the atomic proposition P).
- U: “Until” operator ($P U Q$ means that a step characterized by the atomic proposition Q shall eventually occur, and until that occurrence all steps of the trace shall be characterized by P).
- G: “Globally” operator ($G P$ means that each step in the trace shall be characterized by P).
- F: “Future” or “Eventually” operator ($F P$ means that eventually a step shall occur in the trace that is characterized by P).
- W: “Weak until” operator ($P W Q$ means: that either there is no step in the trace characterized by Q , or a step characterized by the atomic proposition Q shall eventually occur and until that occurrence all steps of the trace shall be characterized by P).

In spite of there are several extensions of PLTL, for various purposes, there is no context-aware temporal logic that can be found in the literature. For this reason previously we defined a new extension of PLTL, the *Context-aware Timed Propositional Linear Temporal Logic* (CaTL).

2.3.1 The Operators of the Logic

Context is referenced in the CaTL formulas using context fragments. A context fragment is an instance model of the context metamodel. It can be represented as a UML package with the name of the context fragment. As the context view consists of instances from the context metamodel, the creation of the context metamodel becomes a significant part of creating requirements. At first the context metamodel contains the classes, its properties and the links between them. Moreover the context metamodel shall define the well-formedness and se-

mantic constraints. *Well-formedness constraints* define constraints that must be satisfied by any context model, otherwise conceptual rules or the laws of physics are violated. *Semantic constraints* are derived from the requirements of an application, this way these are only pre-conditions or expectations about the context that can be violated in particular cases (e.g., when the robustness of an autonomous system is exercised).

The basic vocabulary of CaTL consist of a finite set P of propositions, a finite set T of static timing variables and a finite set C_M of static context variables.

Each $c_i \in C_M$ is an instance of M context metamodel ($c_i \propto M$). A context metamodel is defined as a 2-tuple $M = (N, R)$, where N represents the set of classes in the metamodel and R represents the relations (i.e., association or generalization) in the model. An $n_i \in N$ is a class, which has a set of properties. Each property has a name and a type (e.g., Boolean or string). One can create an E_M set of predefined contexts, where $e_i \propto M$ for all $e_i \in E_M$. The context variables and the predefined contexts contain instances of the classes (objects) from M . Each object has a unique identifier and an $n_i \in N$ class. The values of the properties can be defined by property constraints (defined later), which refer to the objects by the unique identifiers given in the variables. It is important, that if two context variables contain two objects with the same identifier, then that two objects must be equivalent.

In addition, one can use two dynamic variables: t , which represents a *clock* and e , which represents the *context* of the system. M must be defined in such a way, which ensures that e is always a valid instance of M ($e \propto M$).

A_f is the set of atomic formulas, which consists of propositions from P , atomic timing constraints, context constraints and property constraints.

- Propositions are labels, referring to properties of a system. Each proposition can be evaluated to true or false in each state of the system.

Examples: *initialized, connected*

- The timing constraints are defined in the following form: $t \sim u$, where t is the dynamic clock variable, $\partial \sim \{<, >, =\}$, $u \in \{t_i + c, c\}$, $t_i \in T$ and $c \in N$.

Examples: $t = t_0$, $t < t_0 + 5$

- The context constraints are defined in the following form: $x \approx y$, where $x \in E_M \cup V_M$ and $y \in C_M \cup \{e\}$. In this notation \approx is a compatibility relation (meaning x is compatible with y) and V_M is a set of context definitions. Context definitions are instances of the M metamodel. A context definition can be one of the followings:

- a static context variable ($c_i \in C_M$), or
- a new context, created from a static context variable, with one of the following operators:
 - *Node exclusion*: $z - v$, where z is a context definition and v is a present class instance of z ,
 - *Node addition*: $z + w$, where z is a context definition and w is an instance of the classes of M ,
 - *Connection exclusion*: $z - - a$, where z is a context definition and a is a present connection in z ,
 - *Connection addition*: $z + + b(c, d)$, where z is a context definition and b is connection between c and d , compatible with M .

Examples: $e_0 \approx e$, $e_0 - x \approx e$

- The property constraints are expressions over properties of an object. The following syntax is defined to unambiguously select a p property: context.object.p. The syntax

of the property constraints is: $p \sim v$, where p is a property, v is a value, which has to be from the same type as the property, and \sim is a comparison operator, which can be evaluated to a Boolean value.

Examples: $e_0.a.connected = true$, $e_1.b.speed < 10$

For each atomic formula, Υ assigns the modality of that atomic formula (a so-called “temperature”): $\Upsilon : A_f \rightarrow \{\text{hot, cold}\}$. An atomic formula with *hot temperature* is a mandatory, while *cold formulas* are optional. The notation of the modality is the following. If no additional notation is given, then the modality of the atomic formula is hot (mandatory). The cold (optional) modality of the a_f atomic formula is written like $< a_f >$.

A ϕ CaTL formula can be one of the followings:

- Atomic formula: $a_f \in A_f$
- Disjunction: $\phi \vee \phi$
- Negation: $\neg\phi$
- “Next” operator: $X \phi$
- “Until” operator: $\phi_1 U \phi_2$

All static variables used in a formula are implicitly quantified with a universal quantifier. Additional operators can be defined with the previously defined ones as syntactical abbreviations. The most commonly used abbreviations are defined as follows:

- Conjunction: $a \wedge b = \neg(\neg a \vee \neg b)$
- Implication: $a \rightarrow b = \neg a \vee b$
- “Eventually” operator: $F \phi = true U \phi$, where “true” denotes the Boolean true value
- “Globally” operator: $G\phi = \neg(F \neg\phi)$
- “Weak until” operator: $\phi_1 W \phi_2 = (G \phi_1) \vee (\phi_1 U \phi_2)$

To be able to represent CaTL formula in machine-readable textual format, we use the following concrete syntax of the operators (Table 1).

Operator	Concrete textual syntax
\neg	not
\wedge	and
\vee	or
\rightarrow	implies
X	Next
U	Until
G	Globally
F	Eventually
W	Until*
\approx	Compatible

• **Table 1: Concrete textual syntax for PLTL operators**

The other operators ($=$, $<$, $>$, $+$, $++$, $-$, $--$) are used in their usual mathematical form.

2.3.2 The Semantics of the Logic

Formally, the CaTL formulas are interpreted over finite traces of *Context-aware Kripke-structures* (CaKS). A CaKS is the extended version of the classical Kripke-structure, which is the mathematical abstraction of finite state-transition systems with labelled states. A CaKS for a P set of labels and an M context metamodel is defined as a 6-tuple: $\text{CaKS} = (S, T, I, L, C, E)$, where

- S is a finite set of states.
- $T \subseteq S \times S$ is the state transition relation.
- $I \in S$ is the initial state of the system.
- L is a labelling function, which assigns labels to states $L: S \rightarrow 2^P$.
- C function assigns clock value to each state: $C: S \rightarrow \mathbb{N}$, and the clock value assigned to the initial state is 0 ($C(I) = 0$) and if $(a, b) \in T$, then $C(b) \geq C(a)$, so the time is not decreasing.
- E function assigns a context to each state: $E: S \rightarrow C$, where C is the set of context instances ($\forall c \in C: c \propto M$).

Thus a CaKS model is a finite state-transition system, where a context and a clock value are assigned to each state of the system.

A *finite trace* of a CaKS is sequence of states connected by the transition relation: $\Pi = (s_0, s_1, s_2, \dots, s_{n-1})$, where $s_i \in S$ ($i \in [0, n-1]$), $n > 0$, $s_0 = I$ and $\forall 0 < i < n : (s_{i-1}, s_i) \in T$. A Π^j *trace suffix* is defined by removing the first j steps ($j < n$) from the Π trace. By definition $\Pi^0 = \Pi$.

The inductive definition of the semantics of a ϕ CaTL formula is given below. The notation $\Pi^i \models \phi$ means, that the ϕ formula is true on Π^i trace suffix. The $x|_{a=b}$ notation is used for substituting the a variable in x with the b value.

- $\Pi \models p$ if and only if $p \in L(s_0)$, where $p \in P$ is an atomic proposition.
- $\Pi \models c$ if and only if $c|_{t=C(s_0)}$ is true, where c is a timing constraint.
- $\Pi \models d$ if and only if $d|_{e=E(s_0)}$ is true, where d is a context constraint.
- $\Pi \models f$ if and only if $f|_{e=E(s_0)}$ can be evaluated, and evaluated to true, where f is a property constraint. If f contains a property, which does not exist, then the constraint will be evaluated to false.
- $\Pi \models \neg\phi$ if and only if $\Pi \not\models \phi$ is not true
- $\Pi \models \phi_1 \vee \phi_2$ if and only if $\Pi \models \phi_1$ or $\Pi \models \phi_2$
- $\Pi \models X\phi$ if and only if length of Π is at least 2 ($n > 1$) and $\Pi^1 \models \phi$
- $\Pi \models \phi_1 \cup \phi_2$ if and only if $\exists 0 \leq i < n: \Pi^i \models \phi_2$ and $\forall 0 \leq j < i: \Pi^j \models \phi_1$

Lastly, a few terms concerning the contexts must be defined. An e_1 context is compatible with e_2 (denoted as $e_1 \approx e_2$) if, and only if, exists a bijective function between the two object sets e_1 and e_2 , which assigns a compatible object to each object. Two objects are compatible, if and only if both have the same type and have the same relations to other objects. Therefore if the compatibility function assigns $o_2 \in e_2$ to $o_1 \in e_1$ and $o_4 \in e_2$ to $o_3 \in e_1$ and there is an edge between o_1 and o_3 , then an edge must be present in e_2 between o_2 and o_4 , with the same label as in e_1 . Note that the context compatibility relation does not require the equality or compatibility of the properties of the objects, only the object types and relations are concerned.

The compatibility relation defines a bijective function, thus in $e_1 \approx e_2$ for all objects in e_1 (the O_{e_1} notation will be used in the future), there must be an object from O_{e_2} assigned, and if $o', o'' \in O_{e_1}$ are two different objects then the assigned objects from O_{e_2} must be different. It is also required to assign an object to all objects in O_{e_1} , but objects in O_{e_2} can remain without an assigned counterpart.

The objects in the context variables have unique identifiers, meaning that if two objects with the same identifier appear in two contexts then these two objects are identical. This constraint results that the assignments between the objects are immutable. Each object has only one identifier, thus two different identifiers always mean two different objects.

After getting through syntax and semantics of CaTL, let us consider some easy to understand examples. First of all, by the definitions CaTL is an extension of PLTL, thus any valid PLTL formula is a valid CaTL formula. The following formulas are all valid CaTL formulas. The meaning for each formula is also given.

- $G(\text{connected} \rightarrow F(\text{disconnected}))$

It is always true, that if the system is in the connected state, then it will eventually become disconnected.

- $G(\text{connected} \wedge t_0 = t \rightarrow F(\text{disconnected} \wedge t < t_0 + 5))$

It is always true, that if the system is connected, then it will be disconnected in 5 seconds (it is assumed that the time unit is a second).

- $G(\text{connected} \wedge e_1 \approx e \wedge X(\text{disconnected}) \rightarrow F(e_2 \approx e))$

It is always true, that if the system is connected and in the e_1 context and will be disconnected in the next state, then eventually it will be in the e_2 context. It can be also phrased as follows: If the system is connected to an object and disconnects from it, then it will eventually be connected to another object.

3 Describing Event Patterns

As formal specification languages like temporal logics or formal automata are often considered too low-level for the developers, a possible approach is the definition of easy-to-use requirement patterns. They combine the precise textual description with a graphical and/or formal representation (in a similar way like design patterns in OO architecture design).

In the following we

- identify the main patterns that are supported by our approach (as reported in [4], over 90% of the practical properties that were investigated could be expressed using these simple patterns),
- give the CaTL temporal logic based representation of these patterns (this is used to construct the complete CaTL based representation of the properties for monitor source code generation),
- propose an abstract syntax for a graphical pattern language (the concrete syntax can be elaborated in agreement with the domain specific tools).

3.1 Previous Work

The idea of property specification patterns was first suggested by Dwyer et al [3]. The motivation was to free the user from building complex temporal logic expressions that needs deep knowledge and expertise. They proposed a specification pattern system which is a hierarchical system of simple patterns. These patterns generalize commonly occurring requirements without being too abstract. The paper [4] extends this work by assessing the

method based on more than 500 real requirements, collected from literature, researchers, mailing lists and student projects. They found that 92% of these requirements were instances of their patterns. Their updated pattern collection is available online [5].

A similar work restricted to safety patterns can be read in [6]. It contains a hierarchical classification that can help the user to find the appropriate pattern. An example pattern definition (excerpt) can be seen in Figure 2.

Precedence	
Intent	
To describe a relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first. Also known as <i>Enables</i> .	
Example Mappings	
In these mappings <i>S</i> enables the occurrence of <i>P</i> .	
CTL <i>S</i> precedes <i>P</i>:	
Globally	$\neg E[\neg S \ U(P \wedge \neg S)]$
Before <i>R</i>	$\neg E[(\neg S \wedge \neg R) \ U(P \wedge \neg S \wedge \neg R \wedge EF(R))]$
After <i>Q</i>	$\neg E[\neg Q \ U(Q \wedge \neg E[\neg S \ U(P \wedge \neg S)])]$
Between <i>Q</i> and <i>R</i>	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R) \ U(P \wedge \neg S \wedge \neg R \wedge EF(R))])$
After <i>Q</i> until <i>R</i>	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R) \ U(P \wedge \neg S \wedge \neg R)])$

Figure 2. An example property pattern (Precedence) [6]

The work [7] applies the pattern-based requirement description to the domain of programmable logic controllers. Their main contributions are two new pattern groups (possibility and fairness), and a tool helping the users to produce the temporal logic expressions based on the patterns. Later a new pattern (liveness, that is the generalization of the possibility pattern) was proposed and applied in a real case study [8].

The work of Preusse et al. [9] follows a slightly different approach. The defined small “patterns” based on the Computational Tree Logic (CTL) that can be combined together freely. The result is a highly restricted English called *Safety-Oriented Technical Language* (SOTL). Although it is considered as a specification method, even they admit that it is not suitable for a complete specification of the behaviour, but to check critical cases.

3.2 The Pattern Library

In the following basic patterns are identified, giving the natural language representation together with the temporal logic formalization. In the description “events” mean all input or output occurrences (i.e., elements of a monitored execution trace) that are relevant from the point of view of property monitoring: function call, function return, input or output signal, message received or sent, timer started or expired, state entered or left, predicate on a variable, context change, configuration change etc.

The property patterns are divided into two groups: occurrence patterns and ordering patterns (see below). The scopes of the patterns in an execution trace are illustrated in Figure 3.

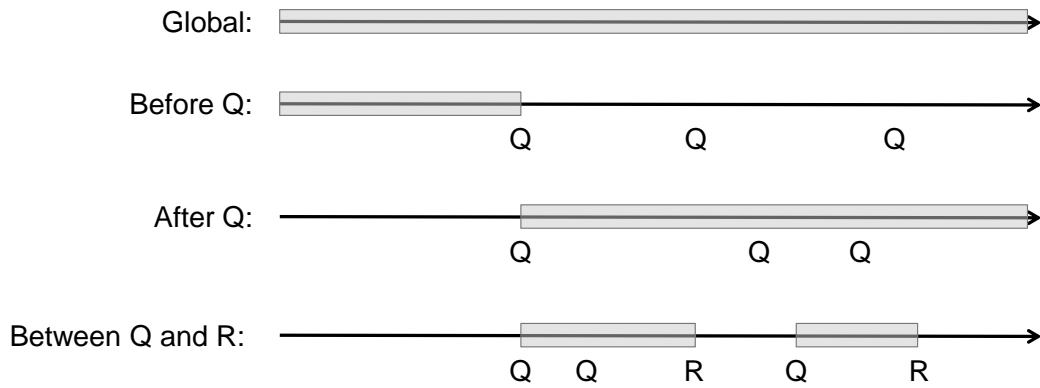


Figure 3. Scope of a pattern in a trace w.r.t. events Q and R

- *Occurrence patterns* describe the occurrence of a given event during execution. The following basic patterns are in this group:
 - *Universality* (also known as Always or Henceforth): It describes a (portion of) execution which contains only steps that are characterized with event P.

Property with scope	Formalized property in CaTL
Event P occurs in each step of the execution.	Globally P
Event P occurs in each step of the execution before event Q.	Eventually Q implies (P Until Q)
Event P occurs in each step of the execution after event Q.	Globally (Q implies Globally P)
Event P occurs in each step of the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (P Until R))

- *Absence* (also known as Never): It describes a (portion of) execution in which a certain event P does not occur.

Event P does not occur in the execution globally.	Globally (not P)
Event P does not occur in the execution before event Q.	Eventually Q implies (not P Until Q)
Event P does not occur in the execution after event Q.	Globally (Q implies Globally (not P))
Event P does not occur in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (not P Until R))

- *Existence* (also known as Eventually)¹: It describes a (portion of) execution that contains event P.

Event P occurs in the execution.	Eventually (P)
Event P occurs in the execution before event Q.	not Q Until* (P and not Q)
Event P occurs in the execution after event Q.	Globally (not Q) or Eventually (Q and Eventually P)
Event P occurs in the execution between events Q and R.	Globally (((Q and not R) and (Eventually R)) implies (not R Until* (P and not R)))

- *Bounded existence*: It describes a (portion of) execution in which an event occurs at most a specified number of times. Here the most typical case is considered when the specified number of times is 2 (where “2 times” means “twice”).

Event P occurs at most 2 times in the execution.	(not P Until* (P Until* (not P Until* (P Until* Globally not P))))
Event P occurs at most 2 times in the execution before event Q.	Eventually Q implies (((not P and not Q) Until (Q or ((P and not Q) Until (Q or ((not P and not Q) Until (Q or ((P and not Q) Until (Q or (not P Until Q))))))))))
Event P occurs in the execution after event Q.	Eventually Q implies (not Q Until (Q and (not P Until* (P Until* (not P Until* (P Until* Globally not P))))))
Event P occurs in the execution between events Q and R.	Globally ((Q and Eventually R) implies (((not P and not R) Until (R or ((P and not R) Until (R or ((not P and not R) Until (R or ((P and not R) Until (R or (not P Until R))))))))))

- *Ordering patterns* describe the relative order in which multiple events occur during execution. The following basic patterns are in this group:

- *Precedence*: It describes a pair of events where the occurrence of the first event is a necessary pre-condition for an occurrence of the second event (i.e., the occurrence of the second event is enabled by an occurrence of the first event). Note that a Precedence pattern allows causes to occur without subsequent effects.

Event S precedes P in the execution.	Eventually P implies (not P Until* S)
Event S precedes P in the execution before event Q. ²	Eventually Q implies (not P Until (S or Q))

¹ In the pattern library, in order to formalize the often used natural language constructs, in some cases patterns and their negation are also considered (e.g., Absence and Existence).

² Note that here „before event Q”, „after event Q”, and „between events Q and R” are scopes of the properties as defined at the beginning of this section

Event S precedes P in the execution after event Q.	Globally not Q or Eventually (Q and (not P Until* S))
Event S precedes P in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (not P Until (S or R)))

- *Response*: It describes a pair of events where an occurrence of the first event must be followed by, or happen together with an occurrence of the second event (i.e., there is a cause-effect relationship between the first and the second event). Also known as Follows or Leads-to. Note that a Response pattern allows effects to occur without causes (this way Precedence and Response patterns are not equivalent, response is just a “converse” of Precedence).

Event S responds to P in the execution.	Globally (P implies Eventually S)
Event S responds to P in the execution before event Q.	Eventually Q implies (P implies (not Q Until (S and not Q))) Until Q
Event S responds to P in the execution after event Q.	Globally (Q implies Globally (P implies Eventually S))
Event S responds to P in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (P implies (not R Until (S and not R))) Until R)

- *Chain precedence*: Chain patterns in general describe requirements related to combinations of event relationships. In case of chain precedence, a precedence relationship is described, consisting of (sequences of) individual events. First a 2 cause – 1 effect chain precedence relationship pattern is presented.

Events S followed by T precede P in the execution. ³	Eventually P implies (not P Until (S and not P and Next (not P Until T)))
Events S followed by T precede P in the execution before event Q.	Eventually Q implies (not P Until (Q or (S and not P and Next (not P Until T))))
Events S followed by T precede P in the execution after event Q.	(Globally not Q) or (not Q Until (Q and Eventually P implies (not P Until (S and not P and Next (not P Until T))))
Events S followed by T precede P in the execution between events Q and R.	Globally ((Q and Eventually R) implies (not P Until (R or (S and not P and Next (not P Until T))))

Second, a 1 cause – 2 effects chain precedence relationship pattern is presented.

Event P precedes S followed by T in the execution.	(Eventually (S and Next Eventually T)) implies ((not S) Until P)
--	--

³ In other words, it is not allowed that P occurs before S followed by T.

Event P precedes S followed by T in the execution before event Q.	Eventually Q implies ((S and (not Q) and Next (not Q Until (T and not Q))) Until (Q or P))
Event P precedes S followed by T in the execution after event Q.	(Globally not Q) or ((not Q) Until (Q and ((Eventually (S and Next Eventually T)) implies ((not S) Until P))))
Event P precedes S followed by T in the execution between events Q and R.	Globally ((Q and Eventually R) implies ((not(S and (not R) and Next (not R Until (T and not R)))) Until (R or P)))

- *Chain response*: It describes a response relationship, consisting of (sequences of) individual events. First, a 2 stimuli – 1 response chain response relationship pattern is presented.

Event P responds to (S followed by T) in the execution.	(Eventually (S and Next Eventually T)) implies ((not S) Until P))
Event P responds to (S followed by T) in the execution before event Q.	Eventually Q implies ((not (S and (not Q) and Next (not Q Until (T and not Q)))) Until (Q or P))
Event P responds to (S followed by T) in the execution after event Q.	(Globally not Q) or ((not Q) Until (Q and ((Eventually (S and Next Eventually T)) implies ((not S) Until P))))
Event P responds to (S followed by T) in the execution between events Q and R.	Globally ((Q and Eventually R) implies ((not (S and (not R) and Next (not R Until (T and not R)))) Until (R or P)))

Second, a 1 stimulus - 2 responses chain is presented:

Events S followed by T respond to P in the execution.	Globally (P implies Eventually (S and Next Eventually T))
Events S followed by T respond to P in the execution before event Q.	Eventually Q implies (P implies (not Q Until (S and not Q and Next (not Q Until T)))) Until Q
Events S followed by T respond to P in the execution after event Q.	Globally (Q implies Globally (P implies (S and Next Eventually T)))
Events S followed by T respond to P in the execution between events Q and R.	Globally ((Q and Eventually R) implies (P implies (not R Until (S and not R and Next (not R Until T)))) Until R)

As it turns out, the CaTL expressions provide a precise description (that is needed for monitor source code generation), but their interpretation is difficult without some experience with temporal logics. The natural language description helps the designer to select the corresponding formalized property and understand its formalization. However, the natural language description is often less precise, for example, in case of the property “Event P occurs in each step of the execution before event Q”, the existence of Q is necessary for the satisfaction of this property, but this is not evident from the natural language description (one may consider that the property is satisfied when Q never occurs only a sequence of P).

An example of an application of a pattern is the following:

- Application specific property: For the Control component of a remotely controlled surveillance robot, receiving a StopCommand message from the Remote Operator guarantees that the StopAction signal will sent to the Motor component of the Robot.
- Pattern: Response with global scope: “Event S responds to P in the execution.”
- Instantiation of the pattern: S is the StopAction (signal), P is the StopCommand (message).
- CaTL formalization of the property: Globally (StopCommand implies Eventually StopAction)

The above listed patterns focused on the most frequently used temporal properties. These can be extended with timing (e.g., to capture the time between stimulus and response) using the timing extensions of the CaTL language.

3.3 Abstract Syntax for a Graphical Pattern Language

To describe and re-use patterns, we also propose a language (with its abstract syntax) that is inspired by [2]. It allows the developer to describe properties over the system or its components by using a combination of quantifiers, temporal patterns, and structural patterns on the domain model(s). Accordingly, the language consists of four parts.

- *Quantification of the formula* (Figure 4). Here *forall* or *exists* quantifiers can be used together with the corresponding structural patterns (see below). Accordingly, the property must be satisfied for all, or for one (depending on the quantifier) matches of the structural pattern. Quantification patterns can be nested, or can contain a temporal pattern.

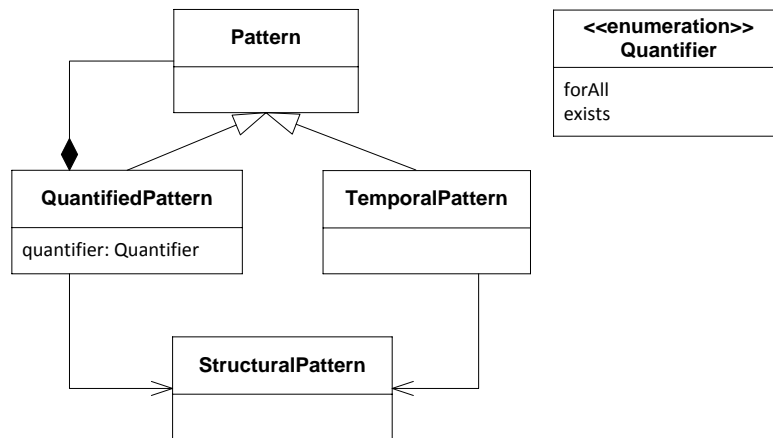


Figure 4. The quantification of the formula

- *Temporal patterns* (Figure 5). The temporal patterns consists of the typical occurrence patterns (absence, universality, existence, bounded existence) and ordering patterns (response, precedence, chain response, chain precedence) together with a scope (globally, before, after, between). As presented in Figure 4, these temporal patterns refer to structural patterns.

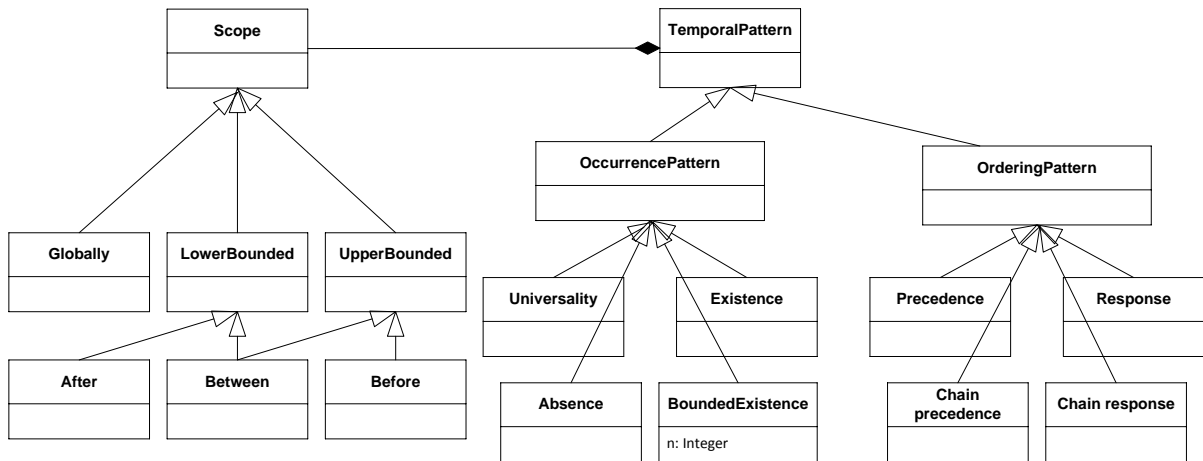


Figure 5. The temporal patterns

- *Structural patterns* (Figure 6) can be used in quantification or in a temporal pattern. A structural pattern means a query on a model (by a pattern matching algorithm). In quantification it returns all bound variables in found matches, while in case of a temporal pattern it returns true if at least one match is found or false when no match is found. The patterns presented in Figure 6 can be extended with additional language constructs if needed (the presented set of patterns is sufficient to represent the majority of practical properties).

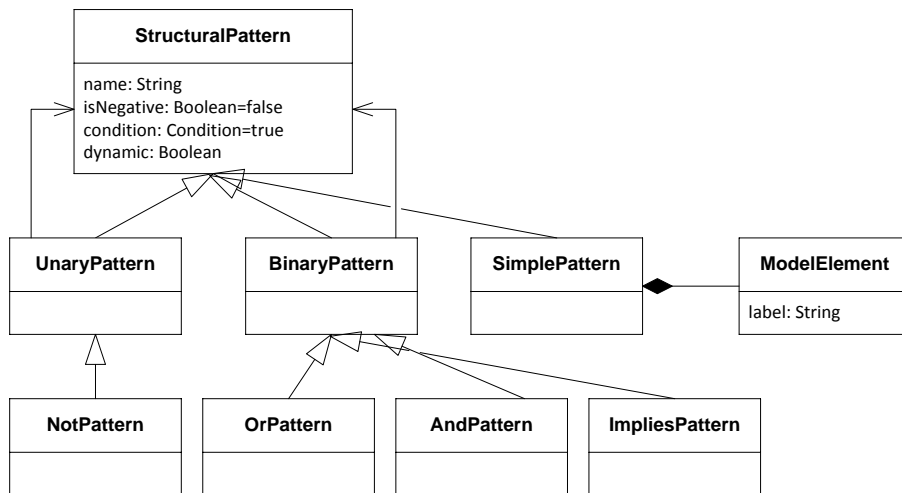


Figure 6. The structural pattern

- *Pattern elements* (Figure 7). A generic model element (ModelElement) serves as the superclass for pattern elements that are specific to the metamodel of the domain modelling language. In Figure 7 the root metamodel of statecharts is included together with specific model elements for events and actions. A straightforward extension is the inclusion of context fragments and configuration fragments as pattern elements, this way the context and configuration metamodel elements should be inserted. All classes are subclasses of ModelElement and have a label (for binding variables) and a condition.

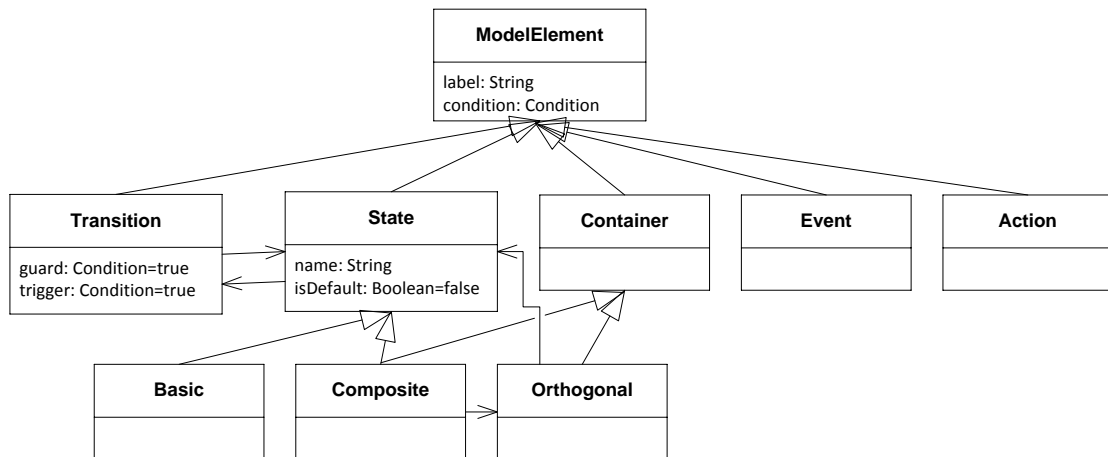


Figure 7. The pattern elements

The concrete syntax of this property language depends mainly on the concrete syntax of the domain model (context and configuration models) and the software artefacts (in this latter case the use of UML 2 model elements is a natural solution). In case of the quantifiers, temporal and structural operators, graphical as well as natural language representation can be used.

3.4 Tool Support to Combine Patterns

In the following, we specify a graphical tool that supports the composition of patterns.

The main elements of the concrete (graphic) syntax that can be used by the tool are depicted in Figure 8.







Modelled artefact	Metamodel element	Graphic representation	Example proposition
Next state	NextForm		
And operator	AndForm		
System property	Propositions		Disconnected
Timing constraint	TimingConst		$t < t_0 + 5$
Context constraint	ContextConst		$e_1 \sim e$
Object property	PropertyConst		$e_1.a.speed < 10$

Figure 8. The concrete syntax of the pattern language

As example, the representation of the property “After start, the next event is ‘Connected’ that is followed by the event ‘Disconnected’ in less than 5 time units, where speed of object a is less than 10 in the e_1 context” is given in Figure 9.

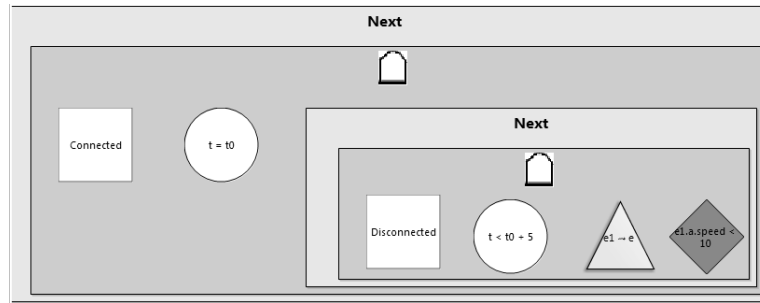


Figure 9. An example property constructed in the tool using the concrete syntax

The above mentioned categories of generic patterns as well as user-defined patterns (as extensions) can be collected into a *pattern store* from which the patterns can be copied to the graphical editor and then configured (parameterized by giving the concrete names of events, properties, context fragments, timing constants etc.). The rules of composing and parameterizing patterns are determined by the syntax of the language.

There is a mapping from complex requirements (composed using the patterns and represented internally in the tool using the pattern language) to expressions of CaTL. This mapping is relatively straightforward as the pattern language followed the semantics of CaTL.

The textual CaTL representation belonging to the example presented in Figure 9 is the following:

$$X(\text{connected and } (t0=t) \text{ and } X(\text{disconnected and } (t < t0+5) \text{ and } (a.\text{speed} < 10) \text{ and } e1 \sim e)))$$

The steps of using the tool are summarized in Figure 10. Note that the requirements that are composed from basic elements and patterns from the repository can be stored as user-defined patterns for further use. The output of the tool is the CaTL expression that can be used for synthesis of monitors or for the verification of designs.

The graphical interface of the tool shall contain the following main areas:

- The *graphical editing* of patterns: The elements displayed in this area are separated into three layers (CaTL layer, Context layer, and Store layer) that can be turned on and off in runtime.
- The *property area*. Here the properties of the model elements can be set or modified (e.g., the name of an element, the corresponding reference, the parameters of expressions, etc.).
- The *object structure* belonging to the requirement (objects, properties, representation files).
- The *palette* area: From this area elements and patterns can be copied to the editor area by drag-and-drop operations. The five groups of elements are the following:
 - Basic elements (atomic formulas as timing constraint, propositions, etc.);
 - Temporal logic elements (next, globally, etc.);
 - Boolean operators (And, Or, etc.);
 - Context elements (context fragment, node, connection);
 - Patterns (Absence, Existence etc.).

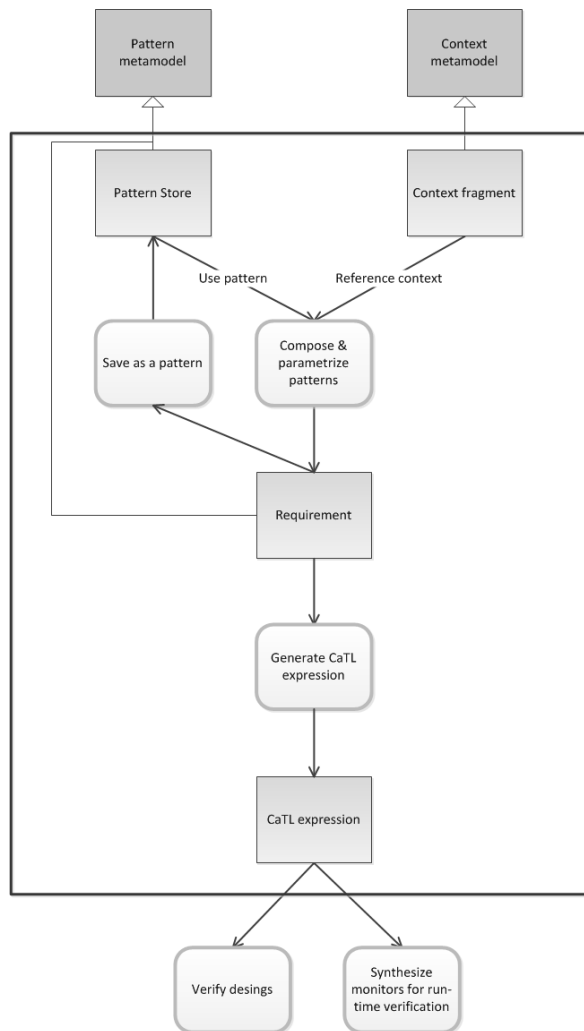


Figure 10. The steps of using the pattern composition tool

4 Conclusions

This report aimed at the description of an approach to capture requirements for runtime monitoring using a pattern library. This approach allows the formalization of safety rules and temporal or trace-based reference behaviour.

On the basis of a new temporal logic variant, the *Context-aware Timed Propositional Linear Temporal Logic* (CaTL), patterns for the typical safety and liveness properties were defined and then the related tool was specified. The implementation of this tool based on a student project is in progress.

5 References

- [1] Pnueli, A: The temporal logic of programs. Foundations of Computer Science, 18th Annual Symposium, pages 46–57, 1977.
- [2] Meyers, B., Wimmer, M., Vangheluwe, H., and Denil, J.: Towards Domain-Specific Property Languages: The ProMoBox Approach. In Proc. International Dependency and Structure Modelling Conference (DSM 13), Indianapolis, USA, pp 39-44, 2013.
- [3] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C.: Property Specification Patterns for Finite-state Verification. In Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP), pp 7-15. ACM, 1998.
- [4] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C.: Patterns in Property Specifications for Finite-State Verification. In Proc. International Conference on Software Engineering (ICSE 1999), pp 411-420, 1999.
- [5] About Specification Patterns. <http://patterns.projects.cis.ksu.edu/> (accessed on January 6, 2015).
- [6] Bitsch, F. Safety Patterns - The Key to Formal Specification of safety requirements. In Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP), pp 176-189. Springer-Verlag, 2001.
- [7] Campos, J. C., Machado, J., and Seabra, E.: Property Patterns for the Formal Verification of Automated Production Systems. In Proceedings of the 17th IFAC World Congress, pp 5107-5112. IFAC, 2008.
- [8] Campos, J. C., Machado, J.: Specification Patterns System for Discrete Event Systems Analysis. International Journal of Advanced Robotic Systems, 10(315), 2013.
- [9] Preusse, S., and Hanisch, H.-M.: Specification of technical plant behavior with a safety-oriented technical language. In Proceedings of the 7th IEEE International Conference on Industrial Informatics (INDIN), pp 632-637. IEEE, 2009.