Innovations Syst Softw Eng (2008) 4:233–240
DOI 10.1007/s11334-008-0054-3

ORIGINAL PAPER

# Model-based requirements analysis for reactive systems with UML sequence diagrams and coloured petri nets

**Kristian Bisgaard Lassen · Simon Tjell**

Received: 1 July 2008 / Accepted: 26 July 2008 / Published online: 9 August 2008


**Abstract** In this paper, we describe a formal foundation for a specialized approach to automatically check traces against real-time requirements. The traces are obtained from simulation of coloured petri net (CPN) models of reactive systems. The real-time requirements are expressed in terms of a derivative of UML 2.0 high-level sequence diagrams. The automated requirement checking is part of a bigger tool framework in which VDM++ is applied to automatically generate initial CPN models based on problem diagrams. These models are manually enhanced to provide behavioral descriptions of the environment and the system itself.

**Keywords** Requirements engineering · VDM++ · UML 2.0 · Real-time · Coloured petri nets

## 1 Introduction

The high-level sequence diagrams of UML 2.0 (SDs) are becoming widely used for describing desired system behavior in terms of scenarios. In this paper, we describe how SDs can be used to express formal real-time requirements for reactive systems. This is done by the definition of a formal requirement language. We also present an approach to using such requirements for validating traces generated by the execution of graphical models of behavior. These models are expressed as coloured petri nets (CPNs) [7] and describe: (1) assumptions about the behavior of the physical environment of the reactive system and (2) a high-level specification of the reactive system.

As an example, we use the requirements specification for a bridge navigational watch alarm system (BNWAS), which is a ship safety device aimed at ensuring that the officer of the watch (OOW)—the person responsible for steering the ship—does not fall asleep (or leave the bridge), while the ship is traveling. This is done by installing an alarm, which will become activated if a dedicated reset button is not pressed for a predetermined period of time. The mode of the alarm is controlled by a key switch. The OOW must push this button periodically in order to signal his presence on the bridge. Furthermore, the BNWAS shall detect attempts to fool the system—e.g. by fixing the button in the pushed-in position.

The International Maritime Organization (IMO)—a United Nations agency—is currently treating an international proposal demanding all ships within some criteria related to size and purpose to be equipped with a BNWAS. The proposal is based on a set of reference requirements [10]. An example of a commercial BNWAS complying with these requirements is shown in Fig. 1. In this paper, we will examine the subproblem of developing the core functionality regarding the activation of alarms based on the reset button and key switch activity.

## 2 A framework for describing requirements

The requirements are expressed in a model-based tool framework for requirements engineering. Referring to Fig. 2, we will now introduce parts of this framework. Other parts of the framework are presented in [8]. The starting point for expressing requirements for the BNWAS is a problem diagram (PD). Problem diagrams are a central description technique to the problem frames approach [5]. A PD describes the

K. B. Lassen · S. Tjell (✉)
Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark
e-mail: tjell@daimi.au.dk

K. B. Lassen
e-mail: krell@daimi.au.dk


**Fig. 1** An example of a commercial BNWAS unit (Navitron Systems Ltd)

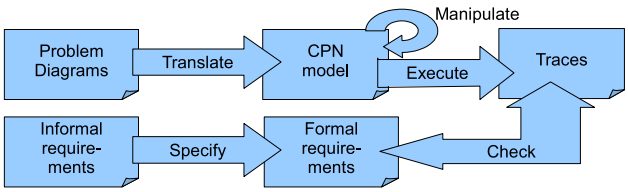


**Fig. 2** Typical workflow

structure of the context in which a (software development) problem is to be solved. In our case, the problem is to develop a design for the BNWAS that will meet our requirements and the context consists of the physical entities that the BNWAS interacts with.

The PD is found in Fig. 3. The boxes are *domains*, the lines are interfaces formed by *shared phenomena*, and the ellipse is the requirement. The leftmost domain is the BNWAS—the two vertical lines denote the *machine*. A PD always has exactly one machine. This is the only domain for which we have the power to decide the behavior. All other domains have given behavior. The purpose of the machine is to cause the context to behave in a manner that satisfies the requirement.

A key point in the problem frames approach is the necessity of describing not only the behavior of the machine but also the knowledge about the behavior of the physical entities in the environment of the machine—i.e. the given

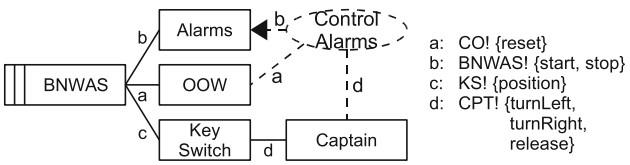


**Fig. 3** A problem diagram for the BNWAS

behavior. The context of the BNWAS is formed by the *OOW*, the *Alarms*, the *Captain*, and the *Key Switch*. The domains are connected by lines representing shared phenomena. A phenomenon is a state or an event. When a phenomenon is shared between two domains, it means that both domains are able to observe the phenomenon while only one domain controls it—i.e. causes a state to change or an event to occur. As an example, the BNWAS interacts with the key switch through a single shared phenomenon: the position. The right hand annotation specifies that the position phenomenon is controlled by the key switch (KS). The BNWAS is able to monitor the current position measurements but is not able to change it.

The requirements for the machine should be expressed about its observable effect on the physical context rather than its behavior seen in the interface between the machine and the environment [6]. This is the reason for describing the given behavior of the physical entities in the context. The dashed ellipse represents the requirement that the BNWAS should control the alarm according to some rules. The requirement is connected to the domains through shared phenomena. In this context, it means that the requirement is expressed in terms of these phenomena. As seen, one interface has an arrowhead. This shows that the start/stop phenomena are *constrained* by the requirement. The other phenomena are only *referenced*. Intuitively, the requirement constrains the observable behavior of the start/stop signals with reference to the KS and the behavior of the human actors—the OOW and the captain (through the KS). This is a natural property of the problem structure, since it would not make sense for example to express a requirement about the behavior of the human actors, who are acting spontaneously and out of free will. The KS is a given physical domain that we can assume has already been developed and therefore has known, predictable, and unchangeable behavior. We could have had more than one PD in order to decompose the problem into subproblems. In [8], we handle a problem with multiple subproblems.

Referring to the framework illustration in Fig. 2, the PD from Fig. 3 is *translated* into a CPN model. This translation is performed by an algorithm specified in VDM++. The input to this algorithm is a VDM++ object representation of the PD and the output is a VDM++ object representation of a hierarchical CPN model. The CPN model is a skeleton based on the structure of the PD: it contains one module per domain in the PD. Apart from these modules, it contains a special link module that links the domain modules together and a requirements module in which the requirements are specified. Examples of the CPN modules for another case study are found in [8]. Initially, the CPN model specifies free/spontaneous behavior for all domains. Knowledge about the domains is specified by a human modeler through *manipulation* of the model in CPN Tools—a graphical modeling/simulation environment [1]. For example, the modeler

will describe the behavior of the KS domain: when the captain turns the key, the position will change to one or the other side (*run* or *set*) and stay in that position when the key is removed. The modeler will also describe typical behavior of the human actors in the form of scenarios in which the actors interact with the BNWAS. Finally, the modeler will develop a high-level specification of the BNWAS domain. This will enable interactive execution of the model making it possible to validate the domain descriptions, the human actor scenarios and the BNWAS specification.

Another important asset of the automatically generated CPN model is that it contains a mechanism for collecting traces of phenomena activity. This mechanism is a part of the link module and is based on a proxy principle. The purpose is to record all shared phenomena activity (interaction) as time-stamped traces. These traces are later to be evaluated against real-time requirements. In this way, the requirements and the model are dynamically validated and the modeler can experiment with the model while iteratively specifying the requirements. Since the requirements are specified in a generic and formal manner, they can later by applied to tasks such as model-based testing of a real implementation.

In the lower part of Fig. 2, we illustrate the specification of requirements. Initially, the requirements are identified informally in a traditional process involving relevant stakeholders. With respect to the BNWAS, examples of such requirements are found in the proposal to the IMO:

> " 4.1.2.1 Once operational, the alarm system should remain dormant for a period of between 3 and 12 minutes ($T_d$).
> 4.1.2.2 At the end of this dormant period, the alarm system should initiate a visual indication on the bridge.
> 4.1.2.3 If not reset, the BNWAS should additionally sound a first stage audible alarm on the bridge 15 s after the visual indication is initiated.
> 4.1.3.4 A continuous activation of any reset device should not prolong the dormant period or cause a suppression of the sequence of indications and alarms.
> 4.2 Accuracy: The alarm system should be capable of achieving the timings stated in Sect. 4.1.2 with an accuracy of 5% or 5 s, whichever is less, under all environmental conditions." [10]

The specification activity seen in Fig. 2 covers the task of expressing the requirements above in terms of SDs. In Sect. 3, we will introduce a formal semantics for real-time requirements expressed as SDs. Fig. 4 shows SD1, which is the formal representation of requirements 4.1.2.1, 4.1.3.4, and 4.2. Some of these requirements may also be represented partly in other SDs. For example, the accuracy requirement (4.2) will be traceable in all requirements containing timing constraints.

The SD depicts domains interacting through shared phenomena. The first phenomenon represents the OOW pushing in the reset button. We have chosen to represent the button as a state in order to be able to express requirement

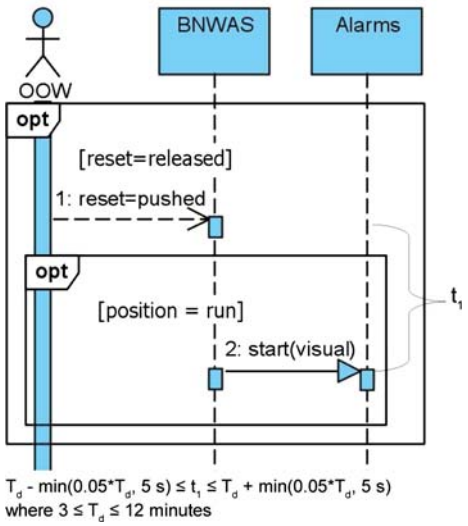


**Fig. 4** SD1

4.1.3.4 easily. In this SD, we only consider the state changing event of the button being pushed (and not released). We also see that the scenario will only become activated if the button is released. The *reset=pushed* message represents a state change phenomenon. Such a phenomenon occurs when a predicate becomes true. The inner *opt* block is relevant if the KS is in the *run* position. While the captain is not explicitly shown as an actor, his behavior affects the scenario—since he is the only one able to change the *position* state by turning the KS. This means that if the KS is in the run position, we require the state change phenomenon (*reset=pushed*) to be followed by a *start*(*visual*) phenomenon. The latter represents the starting of the visual indication as described in requirement 4.1.2.2. On the right-hand side of the SD, we see the indication of a time span ($t_1$). A constraint on $t_1$ is found in the bottom of the SD. In this case, the constraint expresses the requirement to the timing of the starting of the visual indication. We see, that the requirement to accuracy and the parameterization through $T_d$ is taken into account.

Another important thing to notice is the use of dashed and solid lines resembling the arcs of the PD in Fig. 3. The *reset=pushed* is a referenced phenomenon while *start visual* is constrained. We specify this at phenomenon occurrence level, meaning that a phenomenon, which is constrained in the PD may be used for constrained or referenced phenomenon occurrences in SDs. A phenomenon which is only referenced in the PD is only allowed to be used for referenced phenomenon occurrences in SDs. The distinction between referenced and constrained phenomenon occurrences has influence on the interpretation of requirements checking as we will describe in Sect. 3.4.

Figure 5 shows SD2. This SD specifies requirement 4.1.2.3 conforming with the accuracy requirement (4.2). Here, we

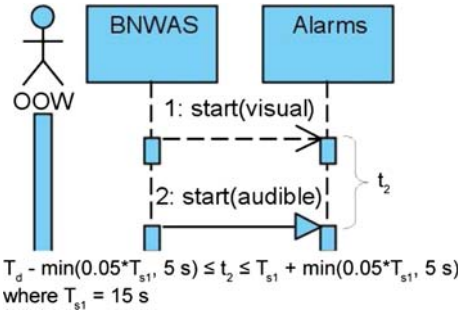


**Fig. 5** SD2

see that the *start*(*visual*) is now specified as a reference phenomenon occurrence (as opposed to in SD1). This technique is used to link together the two SDs and to express what could be informally interpreted as: if *start*(*visual*) occurs, then it must be followed by *start*(*audible*) with in some time constraint—unless the reset button is pushed. In that case, the requirement of this scenario would be abandoned since the scenario would no longer match the trace. The IMO proposal contains two similar requirements (4.1.2.4 and 4.1.2.5) that concern the activation of third and fourth stage alarms. These are similar to the requirement expressed in SD2.

## 3 Definitions

In this section, we turn to the more formal side of phenomena, traces, requirements and how a trace satisfies a requirement.

### 3.1 Phenomena

As we already mentioned, a phenomenon is either a state or an event. In the following we use the notation $P_\mathrm{s}$ to denote the set of state phenomena and $P_\mathrm{e}$ for the set of event phenomena, and for the set of all phenomena we write $P = P_\mathrm{s} \cup P_\mathrm{e}$.

A phenomenon occurrence is when a state changes or an event happens. Each phenomenon occurrence is associated with a value and a time stamp. We denote the set of values as $V$ and the set of time stamps as the total ordered set $T$. The set of phenomena occurrences, therefore, becomes $\mathrm{PO} \subseteq P \times V \times T$, with the restriction that $(p, t, v), (p', t, v') \in \mathrm{PO} \Rightarrow p \notin P_\mathrm{s} \vee p \neq p'$—i.e. the same state phenomenon cannot occur at the same time (with possibly different values). However, we do not restrict the occurrences of events. We believe this restriction on phenomena occurrences is reasonable since a state phenomenon cannot have multiple values at the same point in time, while it is possible for several occurrences of the same event phenomena at the same point in time. Later on we will explain how this restriction can be removed, but for now we will keep it, as we consider it sensible and it will make our ideas more clear.

### 3.2 Transition system and traces

Before we can define a trace, we need to introduce the transition system in Definition 1, that simplifies the definition of a trace, as well as, of how we check whether a trace satisfies a requirement.

The basic idea of the transition system is that we look at each phenomenon occurrence as a transition in the system, the states are the last observed state phenomena and the time stamp of the previous phenomena occurrences. The definition is split in two, so we first consider a total ordered transition system $(S, \triangleright_t)$ where each state in the system is succeeded by another if the next state is at the same, or at a later, point in time. This system is not enough though, since arbitrary cycles within the same time unit is allowed and we can end up in situations with no progress with respect to the time stamps. This is where the second transition system $(S, \triangleright)$ come into play.

It is similar to the first in that the set of states are the same, but we now consider the set $\mathcal{P}(P \times V) \times T$ for the type of transitions. In this system we recursively go trough each possible permutation of phenomena at some point in time before we move on to states with a bigger time stamp. This second transition system enumerates all possible sequences of phenomena occurrences.

**Definition 1** (Transition system) We introduce the total ordered transition system $(S, \triangleright_t)$, where $S = (P_\mathrm{s} \to V) \times T$, and $\triangleright_t \subseteq S \times S$. The input alphabet is PO. We define the transition relation $\delta_t : S \times \mathrm{PO} \to \mathcal{P}(S)$ as

(i) $(s, t) \overset{(p,v,t')}{\triangleright_t} (s', t'')$; *where* $p \in P_\mathrm{e} \wedge s = s' \wedge t'' = t' \geq t$.

(ii) $(s, t) \overset{(p,v,t')}{\triangleright_t} (s', t'')$; *where* $p \in P_\mathrm{s} \wedge s' = s[p \mapsto v] \wedge t'' = t' \geq t$.

Next we introduce the transition system $(S, \triangleright)$, where $\triangleright \subseteq S \times S$. The input alphabet is $\widetilde{\mathrm{PO}} \subseteq \mathcal{P}(P \times V) \times T$. We define the transition relation $\delta : S \times \widetilde{\mathrm{PO}} \to \mathcal{P}(S)$ as

(iii) $(s, t) \overset{(\emptyset,t')}{\triangleright} (s', t'')$; *where* $s = s' \wedge t'' = \min\{t''' \in T | t''' > t\}$.

(iv) $(s, t) \overset{(\{(p,v)\} \cup P \times V, t')}{\triangleright} (s', t'')$; *where* $(s, t) \overset{(p,v,t')}{\triangleright_t} (s', t'')$. □

Definition 1 describes a transition system which gives all paths through the set of phenomena occurrences, but when we check a requirement against the set of phenomena occurrences, we are interested in a particular path (or trace) leading to a failure or a completion of the scenario. In Definition 2 we introduce the concept of a trace. A trace is simply a path given by the transition system.

**Definition 2** (Trace) A trace $\sigma$ in the set of phenomena occurrences PO is a path of length $n$ given by Definition 1. We write the trace as $\sigma = (S_0, t_0) \xrightarrow{(p_1,v_1,t_1)} (S_1, t_1) \xrightarrow{(p_2,v_2,t_2)} \cdots \xrightarrow{(p_n,v_n,t_n)} (S_n, t_n)$. □

### 3.3 Requirement language

Until now, we have only considered the structure of the phenomena, their occurrences and how we extract a trace from a set of phenomena occurrences. Now we turn to how we express requirements for traces in Definition 3.

**Definition 3** (Requirement language grammar) The BNF of our the requirement language is the following:

```
Scenario    ::= ϵ
              | Phenomenon
              | Sequence(Scenario,Scenario)
              | Choice(BoolExp,Scenario,Scenario)
              | Parallel(Scenario,Scenario)
              | Iteration(BoolExp,Scenario)
              | StartTimer(TimerID)
              | CheckTimer(TimerID,TimeExp)
Phenomenon  ::= EventANY(EventName,Restriction)
              | EventValue(EventName,Value,
                           Restriction)
              | EventVariable(EventName,
                              VariableName,
                              Restriction)
              | EventPredicate(EventName,
                               Predicate,
                               Restriction)
              | StateChange(StateName,BoolExp,
                            BoolExp,Restriction)
TimeExp     ::= TimeGTE Integer
              | TimeLTE Integer
              | TimeTest TimedPredicate
Restriction ::= Constrained
              | Referenced
```

BoolExp is a boolean expression; TimerID is an identifier; EventName is an element in $P_e$; StateName is an element in $P_s$; Value is an element in $V$; VariableName is an identifier; Predicate is a predicate function; TimeExp is a simple data type expressing either a greater-than-or-equal or a less-than-or-equal time constraint; TimedPredicate is a function to express any time constrain; Constrained and Referenced are constants. □

Every requirement is a scenario, which in turn can be the one of the following: a phenomenon, a sequence, a choice, a parallel, an iteration, or a timing construct—or the empty scenario ($\epsilon$). A phenomenon is either an event or a state change occurrence. If we only want to require an event of some name to be observed we use EventANY. To bind the value of an event to a variable, which is referable in later expressions, we use EventVariable. To test the value associated with an event, we use EventPredicate. State change occurrences are tested by using StateChange, which is associated with two predicate functions pre and post, where pre must hold before the state is changed and post must hold after. The structured constructs (sequence, choice, parallel and iteration) have the usual UML 2.0 semantics. Finally, the two timing construct are used in conjunction for measuring a time span: StartTimer with some id is added before the first phenomenon occurrence in the span and CheckTimer after the last phenomenon occurrence. CheckTimer is associated with an expression to test if the measured time is above or below some value.

Each phenomenon is associated with a restriction that can either be constrained or referenced. If a phenomenon is constrained, we require that it occurs at the point in the scenario where it is specified, and we interpret it as an error if something else occurs. Conversely, if it is referenced we do not require it to happen. If a referenced phenomenon is not a part of the trace, we then assume that the wrong scenario is being checked. Nevertheless, whether it is a constrained or references phenomenon, if it does occur we continue the check.

By using this distinction on phenomena, we can express that some behavior that does not match a requirement is permissible as long as the requirement does not fail matching a constrained phenomenon. For example, a requirement may be that after a certain sequence of phenomena, some state change must occur. This can be modeled in our language by letting the initial phenomena be referenced and the state change be constrained. If only a prefix of that sequence occurs and something else happens, the requirement will not fail since the sequence of phenomena leading to the state change are all referenced.

### 3.4 Testing traces against requirements

With the definition of traces and requirements, we can now define the meaning of a trace satisfying a requirement. We show how $(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}')$ models some scenario $s$, where $\sigma$ is a trace, $\mathcal{V}, \mathcal{V}'$ is the environment of variables[1] before and after the $s$ is checked, and $\mathcal{T}, \mathcal{T}'$ the environment of timers[2] as they are before and after $s$ is checked. To express this formally, we write $(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s$, which is defined in Definition 4. After the definition we explain the different rules.

**Definition 4** (Satisfiability of requirements) In the following $\sigma$ refers to a trace, $\mathcal{V}$ to the environment which maps variable names to values, and $\mathcal{T}$ to the environment which maps timer

[1] Variables are bound in $\mathcal{V}$ with the EventVariable construct.

[2] Timers are bound in $\mathcal{T}$ with the StartTimer constructs.

identifiers to natural numbers. In the following we show when $(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s$. We use the notation $X$ and $X'$ to denote the value of $X$ before and after a check.

$$\frac{}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \epsilon} \tag{1}$$

$$\frac{\sigma = (S, t) \xrightarrow{(p,v,t')} \sigma'}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{EventANY}(p, r)} \tag{2}$$

$$\frac{\sigma = (S, t) \xrightarrow{(p,v,t')} \sigma'}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{EventValue}(p, v, r)} \tag{3}$$

$$\frac{\sigma = (S, t) \xrightarrow{(p,v,t')} \sigma' : (\sigma', \mathcal{V}[x \mapsto v], \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(\text{EventVariable}(p, x, r), s)} \tag{4}$$

$$\sigma = (S, t) \xrightarrow{(p,v,t')} \sigma' : \frac{f(p, v, t', S(s_1), \ldots, S(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k))}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{EventPredicate}(p, r, f)} \tag{5}$$

$$\sigma = (S_1, t) \xrightarrow{(p,v,t')} (S_2, t') \longrightarrow \sigma' : \frac{\text{pre}(S_1(s_1), \ldots, S_1(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k)) \wedge \text{post}(S_2(s_1), \ldots, S_2(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k))}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{StateChange}(p, \text{pre}, \text{post}, r)} \tag{6}$$

$$\frac{\sigma = \tau\rho : (\tau, \mathcal{V}, \mathcal{V}'', \mathcal{T}, \mathcal{T}'') \models s_1 \wedge (\rho, \mathcal{V}'', \mathcal{V}', \mathcal{T}'', \mathcal{T}') \models s_2}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(s_1, s_2)} \tag{7}$$

$$\frac{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(s_1, \text{Sequence}(s_2, s_3))}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(\text{Sequence}(s_1, s_2), s_3)} \tag{8}$$

$$\sigma = (S, t) \longrightarrow \sigma' : \frac{\begin{array}{l}(\text{cond}(S(s_1), \ldots, S(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k)) \wedge \\ (\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s_1) \vee \\ (\neg\text{cond}(S(s_1), \ldots, S(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k)) \wedge \\ (\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s_2)\end{array}}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Choice}(\text{cond}, s_1, s_2)} \tag{9}$$

$$\frac{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Parallel}(s_2, s_1)}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Parallel}(s_1, s_2)} \tag{10}$$

$$\frac{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(s_1, \text{Parallel}(s_2, s_3))}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Parallel}(\text{Sequence}(s_1, s_2), s_3)} \tag{11}$$

$$\frac{\sigma = \tau\rho : (\tau, \mathcal{V}, \mathcal{V}'', \mathcal{T}, \mathcal{T}'') \models s_1 \wedge (\rho, \mathcal{V}'', \mathcal{V}', \mathcal{T}'', \mathcal{T}') \models s_2}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Parallel}(s_1, s_2)} \tag{12}$$

$$\sigma = (S, t) \longrightarrow \sigma' : \frac{\begin{array}{l}(\text{cond}(S(s_1), \ldots, S(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k)) \wedge \\ (\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(s, \text{Iteration}(\text{cond}, s))) \vee \\ \neg\text{cond}(S, \mathcal{V}, \mathcal{T})\end{array}}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Iteration}(\text{cond}, s)} \tag{13}$$

$$\frac{\sigma = (S, t) \xrightarrow{(p,v,t')} \sigma' : (\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}[id \mapsto t'], \mathcal{T}') \models s}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{Sequence}(\text{StartTimer}(\text{id}), s)} \tag{14}$$

$$\frac{\sigma = (S, t) \longrightarrow \sigma' : \mathcal{T}(\text{id}) - t \geq n}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{CheckTimer}(\text{id}, \text{TimeGTE}(n))} \tag{15}$$

$$\frac{\sigma = (S, t) \longrightarrow \sigma' : \mathcal{T}(\text{id}) - t \leq n}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{CheckTimer}(\text{id}, \text{TimeLTE}(n))} \tag{16}$$

$$\sigma = (S, t) \longrightarrow \sigma' : \frac{\text{tpred}(S(s_1), \ldots, S(s_i), \mathcal{V}(v_1), \ldots, \mathcal{V}_j, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_k), t)}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models \text{CheckTimer}(\text{id}, \text{TimeTest}(\text{tpred}))} \tag{17}$$

$$\frac{\nexists \tau : \sigma = \tau\rho : (\tau, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s}{(\sigma, \mathcal{V}, \mathcal{V}', \mathcal{T}, \mathcal{T}') \models s} \tag{18}$$

□

In the following we explain the rules of the satisfiability check:

1. Any trace satisfy the empty scenario.
2. A trace starting with a transition with the same phenomenon name as referred to EventANY satisfies the requirement.
3. Same as above, but with requirements to the value.
4. A variable bound in an EventVariable is added to the environment of variables and the rest of the requirement is checked with this update. Obviously the name of the phenomenon must match that of the EventVariable.
5. If the name of the EventPredicate matches the next transition, we check if the predicate holds. As input it is given the phenomenon name, value, the time stamp, and also any state, variable, and time stamp necessary.
6. A trace satisfies the StateChange construct if the name matches the next transition, and if the state before satisfies pre, and the state after satisfies the post predicate.
7. A trace satisfies a sequence if the trace can be split in two $\tau$, $\rho$ where $\tau$ satisfies $s_1$ and $\rho$ satisfies $s_2$. The two environments $\mathcal{V}''$, $\mathcal{T}''$ found after checking $s_1$, are used as initial environments when checking $s_2$.

8. This is a simple rewrite rule added for technical reasons—to ensure that the first element in a sequence is never a sequence.
9. If the condition of the state $S$ holds, we check if the first branch is satisfied, otherwise we check the other.
10. A rewrite rule stating that we can change the order of scenarios in a parallel construct.
11. Here, we unfold one of the possible interleavings of a parallel construct where the first element is a sequence. By combining this rule with rules 10 and 12 we obtain all the possible permutations of phenomena occurrences in the two branches of a parallel construct.
12. This check is similar to that of the sequence construct. We check if there is a way to divide the trace into two parts, where the first part satisfies the first branch and the rest the second. Obviously, if this rule was the only one describing how we check parallel constructs, it would indeed just be a sequence semantics we describe, but because we have rules 10 and 11, we are able to test any interleaving of the phenomena involved in $s_1$ and $s_2$.
13. An iteration is checked by checking the condition and then unfolding it one step at a time each time it holds.
14. The StartTimer construct is used to associate an id with the time stamp of the next transition in the trace. The environment of timers is updated with this new association and the rest of the requirement is checked.
15. The CheckTimer is used in conjunction with the start timer. It comes in two variants where this one checks if the time from when the timer started until the time in the trace when this CheckTimer is greater-than-or-equal to some value $n$.
16. Same as the above rule but tests less-than-or-equal instead.
17. A general rule to express any time constraint.
18. If a trace $\sigma$ matches a scenario $s$, we cannot divide $\sigma$ into $\tau$, $\rho$ and use $\tau$ to satisfy $s$. □

The restriction value (Definition 3) does not affect whether a trace satisfies a requirement, but it is relevant for the interpretation of non-satisfying traces as we will describe in Definition 5.

**Definition 5** (Trace and requirement relation) A trace $\sigma$ with at least two transitions has one of these four relations to a requirement $R$: *Completed* if $\sigma$ satisfies $R$; *Aborted* if $\sigma$ fails to satisfy $R$, where no prefix of $\sigma$ satisfies or fails to satisfy $R$, and all expected phenomena in $R$ are referenced; *Failed* if $\sigma$ fails to satisfy $R$, where no prefix of $\sigma$ satisfies or fails to satisfy $R$, and one or more of the expected phenomena in $R$ are constrained; *Undetermined* if $\sigma$ is not long enough to either satisfy or fail to satisfy $R$. If a requirement does not have any of these four kinds of relations to a trace we say that the requirement is *Irrelevant*. □

## 4 Implementation

We have implemented the check in Definitions 4 and 5 in Standard ML (SML) [9]. By implementing in SML, we can use the checker directly in CPN Tools where we develop models for the domains in the PDs, but we can also generate a command line program to check traces generated from physical systems and not just models of these. The requirements are expressed using an SML data type, which is exactly like the language in Definition 3. We can express the first example in Fig. 4 as:

```
fun test (env,vars,timers,t) =
let val t1 = timers(1)   val REAL td = env(Td)
in td - Real.min(0.05 * td,5) <= t - t1 andalso
   t - t1 <= td + Real.min(0.05 * td,5)
end

val req =
Choice(reset == released,
  Sequence(StartTimer(1),
    Sequence(StateChanged(reset,TRUE,reset == pushed,REF),
      Choice(position == run,
        Sequence(EventValue(start,visual,CONSTR),
          CheckTimer(0,TimeTest test)),
            empty))),
      empty)
```

Note that the function test is used in the CheckTimer TimeTest expression and that the requirement is the value req. The checker generates all non-trivial traces and assigns a status according to Definition 5. For each reported non-trivial trace, the state of the environment before and after is shown.

An example of a trace with status *Completed* (cf. Definition 5) for the requirement above is:

[(State reset, pushed, 45), (Event start, visual, 60)]

where the environment before has the phenomena reset=released and position=run. This is a simple example of a trace generated during the execution of the CPN model. In this case, the reset state phenomenon is changed to the value pushed at 45 time units followed by a start event phenomenon with the value visual at 60 time units. Here, enumerated types are used, but the value could be of arbitrarily complex data types.

## 5 Related work

Linear temporal logic (LTL) [11] is a well-known formalism to express modal temporal logic. The language contains primitives to express that a condition will eventually be true, a condition will be true until another condition become true,

and similar things. For checking a trace against LTL expressions, it is possible to use SPIN [4]; an efficient model checker. It could be possible to express the semantics of our requirement language in LTL and use SPIN to check them. The main reason for not doing so are the high-level expressions we wish to use in conditions of choices and iterations. Since the requirement language is centered around the use of SML (e.g. the use of predicate functions), we would have to translate SML to Promola [3] (the expression language in SPIN), which is not a trivial task. Another important issue is that most LTL model checkers cannot check unbounded properties such as integers, so expressing a requirement such as a sequence of event $x$ with value $v$ followed by event $y$ with value $v + 1$, would mean we would have to arbitrarily bind $v$.

In [2], we used CPNs to express requirements for a trace. We expressed requirements by higher-order functions by modal logic operators such as for all, $\forall$, and exists, $\exists$, and these operated on the trace. A major drawback of this approach is that it is relatively far from the way an engineer would think about requirements. We believe that SD are more familiar. Furthermore, the generality of modal logic means that it can be difficult to express structural properties such as parallelism and iteration.

## 6 Conclusions

In this paper, we have introduced a requirement language aimed at real-time requirements for reactive systems expressed as UML 2.0 SDs. The language is part of a requirement modeling framework, where CPN models derived from PDs are used for describing and validating both given and desired behavior. We believe that the combination of SDs and CPN may help closing the gap between the natural informality of real-world requirements and the necessary formality of safety-critical real-time requirements. We plan to further validate the approach by applying it to more Real-world problems.

## References

1. CPN Tools. http://www.daimi.au.dk/CPNtools
2. Fitzgerald JS, Tjell S, Larsen PG, Verhoef M (2007) Validation support for distributed real-time embedded systems in VDM++. In: Proceedings of HASE. IEEE Computer Society, USA
3. Holzmann GJ using SPIN. http://plan9.bell-labs.com/sys/doc/spin.pdf
4. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
5. Jackson M (2001) Problem frames—analyzing and structuring software development problems. Addison-Wesley, Reading
6. Jackson M (2002) Some basic tenets of description. Softw Syst Model 1(1):5–9
7. Jensen K, Kristensen LM, Wells L (2007) Coloured petri nets and CPN tools for modelling and validation of concurrent systems. STTT 9(3–4):213–254
8. Lassen KB, Tjell S (2008) Developing tool support for problem diagrams with CPN and VDM++. In: Proceedings of OVERTURE 2008. Newcastle University
9. Milner R, Tofte M, Harper R, MacQueen D (1997) The definition of standard ML. MIT Press, Cambridge
10. Performance Standards for a Bridge Navigational Watch Alarm System (BNWAS) (2002) Resolution MSC 128(75). http://www.imo.org/includes/blastData.asp/doc_id=6850/128(75).pdf
11. Pnueli A (1977) The temporal logic of programs. In: Proceedings of FOCS 1977. IEEE Computer Society