Runtime Monitoring based on Interface Specifications

Ivan Kurtev[1] Jozef Hooman[2,3] Mathijs Schuts[4]

[1] Altran, Eindhoven, The Netherlands
[2] Embedded Systems Innovation by TNO, Eindhoven, The Netherlands
[3] Radboud University, Nijmegen, The Netherlands
[4] Philips, Best, The Netherlands

Abstract. Unclear descriptions of software interfaces between components often lead to integration issues during development and maintenance. To address this, we have developed a framework named ComMA (Component Modeling and Analysis) that supports model-based engineering of components. ComMA is a combination of Domain Specific Languages (DSLs) for the specification of interface signatures, state machines to express the allowed interaction behaviour, and constraints on data and timing. From ComMA models a number of artefacts can be generated automatically such as proxy code, visualizations, tests, and simulation models. In this paper, the focus is on the generation of runtime monitors to check interface conformance, including the state machine behaviour and the specified data and time constraints. We report about the development of this approach in close collaboration with the development of medical applications at Philips.

1 Introduction

Precise interface descriptions are crucial in the development of complex systems with many components, including third-party components. Unclarity about interfaces is a frequent source of errors. This does not only concern the signature of messages exchanged between components, but also the expected order of messages, assumptions on timing behaviour, and constraints on the exchanged data values. During system development, proper interface definitions are essential to prevent integration issues. During later phases of the system life cycle, continuous monitoring of interfaces is important to prevent system failures due to component changes. For instance, a supplier might deliver an improved version of a component which, however, has different timing characteristics.

The focus of this paper is on the automatic generation of monitoring support from interface specifications. This is done in the context of the development of minimally-invasive interventional X-ray systems of Philips. An example of such a system is depicted in Fig. 1.

In order to support the development and usage of precise interface specifications we proposed a framework named ComMA (Component Modeling and Analysis). ComMA enables model-based engineering of high-tech systems by

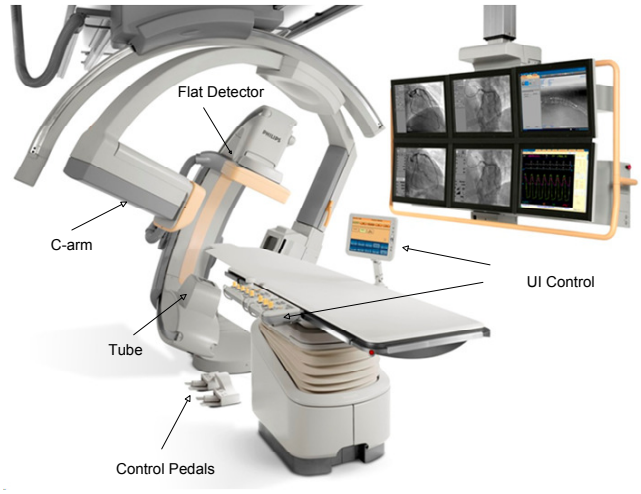

Fig. 1. Interventional X-ray System

formalizing interface specifications. It provides a family of domain-specific languages (DSLs) that integrates existing techniques from formal behavioural and time modeling and is easily extensible. The most important analysis tool in ComMA allows monitoring and checking of component executions against interface specifications. The monitoring can be performed for already existing logs with execution traces or by monitoring executions at runtime.

While developing ComMA and its monitoring capabilities we used the *industry-as-laboratory* approach [12]. This means that tools and techniques are developed in close interaction (e.g. on a daily basis) with real industrial projects. A similar approach has been applied in an earlier industry-as-lab project on runtime awareness [2].

The first version of ComMA was a basic domain-specific language and a few generators for glue code, visualisations and very basic monitoring. This provided immediate benefits for the industrial projects and created interest for further applications. It also led to a stream of feature requests which have been incorporated based on user priorities. For instance, inspired by the usefulness of monitoring timing constraints, users asked for monitoring of advanced data constraints. We also added document generation based on requests of users. By gradually increasing the number of generators and features, we incrementally add more value to the projects using the ComMA framework.

As part of our industry-as-lab approach, we had to refactor ComMA based on new insights we obtained while executing industrial cases. In addition, we refactored the framework to improve maintainability. The initial monolithic language has been split into a composition of DSLs for types, expressions, state machines, etc.

The initial component monitoring supported checking of conformance between interfaces and their implementations and checking timing constraints expressed as rules. The rules give the admissible time intervals between events in different contexts. Apart from timing behaviour, users were also interested in monitoring data values. For example, client requests for a given system mode with certain parameters have to match the parameters communicated later by the system. We developed a new small DSL for expressing only data constraints. It is known from the literature on runtime verification that time information in the form of event timestamps can be treated in the same way as the data values carried by events; they are just data fields associated to events. This observation motivated the unification of data and timing constraints into an underlying generic language which is used to generate the monitoring infrastructure. These changes in the DSLs infrastructure were executed iteratively and remained invisible for the users. Users could keep using the basic front-end notations for data and timing constraints and stayed isolated from the changes in the implementation.

The main contribution of this paper is the description of the syntax and semantics of the generic constraints language. In addition, we also present how the two existing languages for time and data constraints relate to the generic language.

This paper is organized as follows. Section 2 gives an overview of the related work. An industrial case that provided a motivation and insights for our work is presented in Section 3. Examples from the case are used to explain the DSLs in ComMA (Section 4), and the support for monitoring time and data constraints (Section 5). Section 6 presents the syntax and semantics of the generic constraints language. Section 7 concludes the paper.

2 Related Work

Runtime verification [7, 9] is a technique for monitoring the behaviour of software during its execution and checking if the behaviour conforms to a specification. The literature contains a large number of methods to annotate programs with specifications and the use of these annotations for runtime checks. Examples are design by contract in Eiffel [10] and the Java Modeling Language (JML)[4]. As a unifying approach, monitor-oriented programming [6] supports runtime monitoring by integrating specifications in the program via logical annotations, where a particular specification formalism can be added as a logic plug-in. Actual monitoring code is automatically synthesized from these annotations and integrated at appropriate places in the program, according to user-defined configuration attributes of the monitor.

The approach of ComMA is independent of the hardware or software implementation of a component. In our working context, the implementation of many third-party components is not directly available. The properties are specified over traces of component executions. Traces can be obtained via sniffing the network traffic or from logs (if available).

Properties checked during monitoring can be expressed in various formalisms. Examples are logic formulas, automata, and context free grammars. The constraint languages in ComMA are inspired by constructs available in Linear Temporal Logic (LTL) and its real-time extension Metric Temporal Logic (MTL)[11]. Furthermore, our framework follows the ideas behind the languages RuleR and Eagle [1] with respect to formulating properties as a set of rules and it adapts the algorithms from the same work.

Interface signatures and behaviour can be defined in general-purpose modeling languages like UML and SysML [8] for which many commercial tools exist. Engineers usually need only a subset of these rich languages. In addition, tools require tailoring for a given problem area via profiles which is in effect a domain-specific extension. ComMA provides a set of standalone DSLs instead of extending an existing modeling language.

3 Industrial Case

With the interventional X-ray systems of Philips, X-ray movies of a patient's body can be made in real-time while executing a medical procedure. An example procedure is placing a stent into the aorta of a patient. The physician uses the system to navigate the stent through the patient's arteries to the target position. The arteries can be made visible by injecting a contrast medium. The physician positions the X-ray beam with respect to the patient in such a way that she/he can see the region of interest. This can be done by moving the table on which the patient lies, and by moving the C-arm which holds the X-ray generator and the X-ray detector. The table and C-arm can be maneuvered by means of a user interface.

A patient table has multiple axes of movement that can be controlled by a software interface. An example is the vertical movement that changes the height of the table with respect to the table base. During movements, the patient and the table can get in close proximity to the C-arm. This is controlled by safety mechanisms to prevent hitting the patient. Examples of these mechanisms are limiting the movement speed when the distance between table and C-arm is reduced and to stop all movements when the communication between table and system is lost.

Figure 1 shows an example of the system with a table developed at Philips. Besides Philips tables, the system also supports tables of third-party vendors. We applied ComMA to model a new interface between the interventional X-ray system and a third-party table. The communication between the system and the table uses an Ethernet connection and a proprietary protocol. The table has its own User Interface (UI) that can be used to change the positions. The X-ray system is treated as another UI for the table. From the perspective of the X-ray system this means that movements can originate from other sources. Thus, the system needs to observe the position of the table and to act when the distance between table and C-arm becomes too close.

Table movements are controlled by a joystick. The joystick has to be constantly leaned to a certain direction during the movement until the target position is reached. If the joystick is released the movement stops. While moving, the table notifies the system about its position and the movement status (moving, position is reached, position is not reached).

ComMA was used to model the software interface of the table by defining the signatures of the messages and the behaviour of the table by means of a state machine. In addition, several time and data constraints related to safety mechanisms were specified and checked.

In the following sections we explain the ComMA framework with a focus on support for monitoring time and data constraints. The presentation is based on simplified examples derived from the industrial case.

4 Overview of ComMA

4.1 ComMA Framework: Languages and Tools

The ComMA framework is based on a family of DSLs and allows users to specify the interface of a server towards its clients by two main ingredients:

- The interface signature, consisting of groups of synchronous & asynchronous calls and asynchronous notifications.
- The interface behaviour which is defined by:
  - State machines that describe the allowed interactions between clients and server, e.g., the allowed order of calls of clients and the allowed notifications by the server in any state.
  - Constraints on data and time, such as the allowed response time, the periodicity of notifications, and data relations between the parameters of subsequent calls.

Based on a ComMA model, the framework supports a number of generators, as shown in Fig. 2:

- Visualization and documentation. ComMA generates PlantUML[5] files that visualize state machines. In addition, constraints can be intuitively represented as annotated UML sequence diagrams. Also MS Word documents that are compliant with the company standard can be generated, based on the M2Doc framework[6]. This transformation extracts definitions and comments from models and inserts them in a document template. This process also utilizes the diagrams obtained from state machines and constraint rules.
- Interface proxy code. Interface signatures can be transformed to interface proxy code (C++ and C#) that can be incorporated in the company-specific platform for transparent component deployment.

[5] http://plantuml.com/
[6] https://github.com/ObeoNetwork/M2Doc

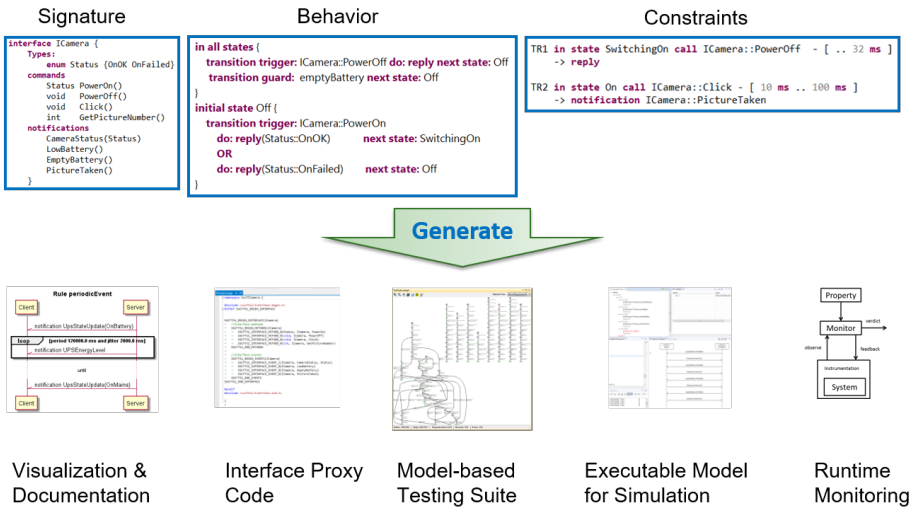

Fig. 2. Overview of ComMA and generators

- Model-based testing. Based on the state machines, models can be generated for various model-based testing tools such as SpecExplorer[7]. This allows test generation and on-line testing.
- Simulation. Simulation of a model helps in receiving an early feedback and detecting errors. State machine models are transformed to POOSL programs (Parallel Object Oriented Specification Language) [14]. Engineers can use the step-by-step visual execution facilities of POOSL[8].
- Runtime monitoring. A modified version of the transformation to POOSL produces an executable monitor for runtime verification. This feature is explained in details in Section 5.

4.2 Specifying Component Interfaces with ComMA

ComMA provides a DSL for defining interface signatures. Here we present a simplified version of the interface of the operating table.

```
interface ITable{
   types
   enum Status {PosReached PosNotReached InMove}

   commands
   bool start
   void stop
```

[7] https://www.microsoft.com/en-us/research/project/model-based-testing-with-specexplorer/
[8] http://poosl.esi.nl/

```
    signals
    alive
    moveVertical(int moveId, int pos)

    notifications
    verticalPosition(int pos, Status moveStatus)
}
```

We distinguish between two types of calls: *commands* that may be called synchronously and always reply a result and asynchronous *signals*. Notifications are asynchronous messages sent from a server to the clients. Commands, signals, replies and notifications are referred to as *events*.

The interface above defines commands for starting and stopping the table operational mode and a signal for moving the table in the vertical axis. Every movement has an unique identifier (parameter *moveId*) and a target position (parameter *pos*). The table can notify the system about its current position in the vertical axis and the movement status. The status is encoded as a value of an enumeration and denotes if the table is moving, has reached the target position or the movement is interrupted and the target position is not reached. Once the table is operational, the X-ray system has to send periodic signals to it to indicate that the client side is alive. A signal is either a move request or an alive signal (if no move is needed).

In ComMA, the behaviour of interfaces is specified by state machines. A state machine is associated with at least one provided interface. Commands and signals are triggers for state transitions. The machines have disjoint sets of transition triggers and may share variables. Only one transition can be fired at a given moment across all machines. The DSL allows only flat machines: nested states are forbidden. All state transitions must be observable: either a transition is triggered by a command/signal or the transition effect is observable, for example, by sending a notification.

The following listing is a simplified specification of the externally visible behaviour of the table interface.

```
machine Main provides ITable {

variables
int currentMoveId

init
currentMoveId := 0

initial state Inactive {
  transition trigger: start
    do: reply(true) next state: PositionReached
    OR
```

```
    do: reply(false) next state: Inactive
}

state PositionReached {
  transition trigger: moveVertical(int moveId, int target)
    guard: moveId != currentMoveId
    do: currentMoveId := moveId
  next state: Moving

  transition trigger: moveVertical(int moveId, int target)
    guard: moveId == currentMoveId
  next state: PositionReached

  transition trigger: alive next state: PositionReached

  transition
    do: verticalPosition(*, Status::PosReached)
  next state: PositionReached

  transition trigger: stop do: reply
  next state: Inactive
}

state PositionNotReached {
...
}

state Moving {
  transition trigger: moveVertical(int moveId, int target)
    do: currentMoveId := moveId
  next state: Moving

  transition
    do: verticalPosition(*, Status::PosReached)
  next state: PositionReached

  transition
    do: verticalPosition(*, Status::PosNotReached)
  next state: PositionNotReached

  transition
    do: verticalPosition(*, Status::InMove)
  next state: Moving
}
}
```

The command *start* tries to activate the table. The result is indicated in the return value. If the activation is successful, the table assumes a reached position state. It can receive a move request with a given identifier (signal *moveVertical* with a positive integer as identifier and a target position). If it is a new move request, the table starts moving (represented by state *Moving*). The table is moving as long as it receives move requests. The movement status is continuously reported via notifications *verticalPosition*. The listing shows three different transitions that send *verticalPosition* as a notification: one for each possible status. The notation '*' denotes a value that is unknown in the state machine. The state *PositionNotReached* is not elaborated. It is similar to state *PositionReached*.

If the table stops receiving the signal *moveVertical*, the movement is interrupted and a notification for 'position not reached' status is sent. The machine then moves to *PositionNotReached* state. The machine above does not capture this logic. It just states that at any moment a transition to a non-moving state is possible. The described behaviour is captured in a timing constraint explained in the next section.

5 Monitoring of Time and Data Constraints

Issues at system level are often traced back to issues related to the conformance of components (possibly supplied by a third party) to their interface specifications. Many issues of this kind are manifested during the interaction of several components and it is difficult to detect them if a component is tested in isolation. Monitoring and checking component interactions can reveal the problems at an earlier phase and help in analyzing logs harvested from systems in the field. We applied available runtime verification techniques (mainly inspired by [1]) to support specification and monitoring of interface behaviour and constraints on timing and data.

5.1 General Scheme for Component Monitoring

Generally, runtime verification is a technique for checking system behaviour against a property during the execution of the system. The general scheme [7] is given in Fig. 3.

The property may be given in a formal specification language (automata, logic formula, grammar), as a set of rules or a program. A monitor is derived from a set of properties. The task of the monitor is to observe the execution of the system and to produce a verdict, that is, a statement if the observation satisfies the properties. The observation may be a series of system states or a series of input and output events. Monitoring is executed either step by step along with the system execution or over a log that contains the observations.

Fig. 4 shows how this general approach is applied in ComMA. The behavioral model of the interfaces (state machines, timing and data rules) plays the role of properties. The monitor processes events observed during component executions. Currently the events are obtained in two ways: by logging during executions or

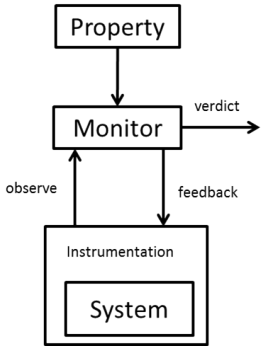

Fig. 3. General scheme of runtime verification

by monitoring network traffic when the component is deployed on the company-specific middleware. It should be noted that currently the execution trace is checked after it is finished, that is, the check does not happen at runtime. The implementation of the monitor, however, is agnostic about the exact moment when events are supplied (during or after component execution). Monitoring at runtime can be performed if instrumentation is applied to components or to the middleware layer.

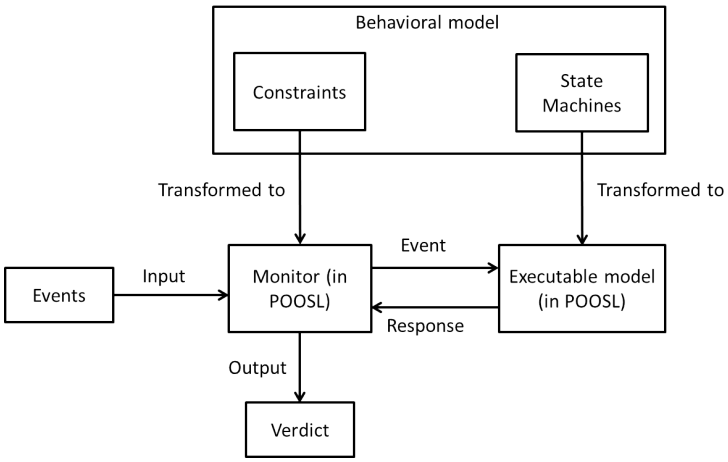

Fig. 4. Monitoring in ComMA

The monitor is a POOSL program that is partially synthesized from the constraint rules. It receives the events in the execution trace and sends them to an executable model (also a POOSL program) derived from the state machines. The state machine responds with events that are compared to the events in the trace. The monitor also checks if the constraints hold for the trace. Verdicts can be errors and warnings. Errors are violations of the state machine logic.

Warnings are violations of constraints. Errors stop the monitoring process, after a warning the monitoring continues.

In the following subsections we elaborate on the support for specifying and monitoring constraints.

5.2 Timing Constraints

The first type of constraints are timing constraints defined as timing rules. They give the admissible intervals between events in different contexts. There are four rule types.

Interval rules constrain the allowed time interval for observing an event if a given preceding event was observed. The example rule named *timeForReply* states that if the command *stop* is observed then the reply must be observed between 10 and 20 ms after the command. The rule is checked on the first occurrence of reply after the command. Before checking timing rules, static checks ensure that every command is properly matched by a reply. Pairs of commands/replies are reconstructed in the order of their handling by the component.

```
timeForReply
command stop -[10.0 ms .. 20.0 ms] -> reply
```

The second rule type is called conditional interval. It states that if two events are observed without observing the first event in between then the interval between them lies within a certain boundary. The next example states that the interval between *start* and a positive reply is 30 ms. The rule will not be applied if the reply is false.

```
intervalBetweenEvents
command start
and
reply(true)
-> [ .. 30.0 ms] between events
```

The third rule type allows specifications of periodic events. The following rule states that after starting the system the connection should be kept alive by sending signals every 100 ms. It is also possible to specify a jitter for the expected period.

```
continousCommunication
reply(true) to command start then
any signal with period 100.0 ms jitter 10.0 ms
until
command stop
```

The fourth rule type allows stating that if a certain event is observed then another event must be absent during a given interval after the observation. The four types of rules can be combined in groups that form a scenario. Scenarios and the rule for absence of an event are illustrated in the following example. It is based on the safety mechanisms implemented in the table control. One of them

states that if a move request is delayed for more than 200 ms during movement, the table must stop moving. The time for stopping after detecting the delay is also constrained. The scenario specifies this constraint. The first rule acts as a precondition: it detects if a move request is absent within 200 ms after receiving the previous move request. If this happens then the table must stop within 1 second after the delay is detected. The stop is manifested by the notification *verticalPosition* with the corresponding status value.

```
group intervalForStopping
in state Moving signal moveVertical
-> absent signal moveVertical in [.. 200.0 ms]
- [.. 1000.0 ms] ->
notification verticalPosition(*, Status::PosNotReached)
end group
```

This rule illustrates also the possibility to specify state information in timing rules. The rule will be triggered only if the signals are received in state *Moving*. This is possible because the rules are evaluated over events that are already accepted by the state machine, that is, state information is available.

5.3 Data Constraints

Apart from detecting delayed requests, other safety mechanisms in our industrial case check the distance that the table is allowed to move after detecting the absence of move requests. The distance should not exceed a certain value. This is schematically given below by showing the expected sequence of events when the table stops if move requests are no longer received:

```
verticalPosition(X, InMove)
moveVertical //The last moveVertical
... no more moveVertical signals ...
...
verticalPosition(Y, PosNotReached)
```

In order to check the distance we need to identify the last reported vertical position (say X) in the *verticalPosition* notification just before the last *moveVertical* signal. The fact that the table stops is indicated by a notification *verticalPosition* with status PosNotReached and position Y. The absolute value of Y-X must not exceed a certain limit.

We introduced a simple language for specifying data constraints. At an intuitive level, the meaning of a data constraint rule is: if a certain sequence of events is observed then the observed data values must satisfy a given condition. The monitoring algorithm should allow the observed data values to be bound to variables and then used in conditions. The specification of an expected event sequence can be done by using a regular expression-like notation. It should be possible to assert both the presence and the absence of events.

The following example shows the constraint expressed in the data constraints language.

```
stoppingDistance //name of the constraint
notification verticalPosition(X, Status::InMove);
in state Moving signal moveVertical;
no [signal moveVertical]
until
notification verticalPosition(Y, Status::PosNotReached)
where abs(X-Y) <= 100
```

The rule is checked as follows. Given a sequence of observed events, the following sub-sequence is searched:

- notification *verticalPosition* with status InMove. The reported position (as the first parameter) is kept as value of variable X;
- signal *moveVertical* that follows immediately after the previously observed notification;
- a sequence of events that do not contain *moveVertical* and the event specified in the **until** clause. In this way we capture the fact that move requests are no longer received;
- notification *verticalPosition* with status PosNotReached. This event marks the end of the sequence to be matched. The position is kept in variable Y.

If such a sequence in the trace is detected then the condition in the **where** clause is checked. It states that the distance should be less than 100 units. If the sequence is not detected then the rule precondition is not fulfilled and the rule holds for the given trace.

Considering the implementation options for checking the data constraints, several factors played a role. First, we were aware that time and data constraints can be treated uniformly: timestamps are just data fields associated to events. This brought the option to replace the existing dedicated engine for checking time rules with a more generic engine applicable to both types of constraints. Second, there exist formalisms for expressing properties used in runtime monitoring. However, these formalisms are complex and it is preferable to shield the engineers from directly using complex notations.

We decided to define a generic constraints language and engine while keeping two separate 'surface' languages: for time rules and for data constraints. Specifications in these languages are transparently translated to the underlying generic language. In this way, the surface languages can be kept simple and easily extensible if needed. Extensions would require only a syntactical translation to the more expressive generic language and no changes in the engine.

The language for generic constraints is the main contribution of this paper and is explained in the following section.

6 Language for Generic Constraints

The main construct in our solution allows specifying patterns for sequences of timestamped events observed in execution traces. Sequence patterns (called here

just *sequence*) are used to construct formulas which are evaluated during monitoring.

A sequence is a concatenation of steps. Each step matches one or more observed events in a trace. There are three kinds of steps: event selector, a disjunction of event selectors, and two until operators (weak and strong) inspired by the similar constructs in LTL. Informally, the matching process starts from the first step in a sequence and tries to match it with the first event in a given trace. If successful, the process continues with the next steps that are matched against the remaining events in the trace. During this process free variables in the pattern are bound to matched values and become available in the next steps. The following example shows the formulation of the *stoppingDistance* data constraint rule of Section 5.3 in the generic language.

```
stoppingDistance
<t1, in state Moving
     notification verticalPosition(X, InMove)>;
<t2, in state Moving signal moveVertical>;
<t3, not [signal moveVertical]>
until
<t4, notification verticalPosition(Y, PosNotReached)>
where
abs(X-Y) <= 100
```

This rule has a name and a formula. The formula specifies a sequence (the part before the **where** keyword) and a condition that uses variables in the sequence (the part after **where**). The main difference with the syntax of the same rule expressed in the data constraints language is the presence of timestamp variables ($t1$, $t2$, ...).

The first step in the example sequence is an event selector. It has a variable named $t1$ that is bound to the event timestamp. The **until** construct with a general form $Selector1$ **until** $Selector2$ matches a sequence of events in which the last event matches $Selector2$ and all the preceding events match $Selector1$. As can be seen in the example, event patterns can be negated. Selectors may also have a boolean condition that is evaluated if the event pattern matches (this is shown in the following examples). The remaining part of this section defines the syntax and semantics of the generic constraints language.

6.1 Language Syntax

The syntax rules of the language are given in Table 1. In these rules some nonterminals are left undefined: S is a set of states, *Cond* is a Boolean expression defined by the ComMA grammar, *Var* denotes a variable, and $\bar{P}$ (event parameters) is a vector of variables and constants of types supported by ComMA. States, conditions and parameters can be omitted. For simplicity, the type of the event (command, signal, etc.) is skipped and **in state** is abbreviated to **in**. We

Table 1. Syntax of the Generic Constraints Language

Formula	$F ::= Seq \mid Seq$ **and** $Cond \mid Seq$ **cf** $F \mid$ **not** $F \mid F$ **or** F
Sequence	$Seq ::= Step \mid Step$ **until** $Step \mid Step$ **wuntil** $Step \mid Seq\ ;\ Seq$
Step	$Step ::= ES \mid ES$ **or** ... **or** ES
Event Selector	$ES ::= \langle Var, E, Cond\rangle \mid \langle Var, Var, E, Cond\rangle$
Event Pattern	$E ::=$ **in** $S\ EvDes(\bar{P}) \mid$ **not** [**in** $S\ EvDes(\bar{P})$]
Event Designator	$EvDes ::= eventName \mid *$

assume that every variable appears at most once in all event selectors. The usage of variables in conditions has to be well-formed: no forward variable references are allowed.

From the rules it can be seen that event selectors have two forms. The first one was already explained by the example rule *stoppingDistance*. The second one has an extra variable called *occurrence counter*. The value of the variable is incremented every time the event selector is successfully matched in a sequence. Consider the following example that uses an occurrence counter:

```
<t1, i, in state s A>
or
<t2, not[in state s A]>
until
<t3, B>
```

If this sequence is evaluated against a trace, the counter i will be incremented every time an event A is observed in state s until event B is observed. The usage of counters is exemplified further in the context of periodic time rules.

Operator **cf** stands for *conditional follow* and expresses a common case in which the match of a sequence is a precondition for checking a formula over the remaining part of the trace. For example, the formula:

```
<t1, A>;
<t2, not[in state s B]>
until
<t3, C>
cf
<t4, D>
```

is used to check if all sequences of events that start with A, end with C and do not contain event B occurring in state s, are immediately followed by event D.

Some useful logical operations are defined as the following abbreviations:

- F1 **and** F2 ≡ **not** (**not** F1 **or not** F2)
- F1 **implies** F2 ≡ **not** F1 **or** F2
- Seq **where** Cond ≡ **not** Seq **or** (Seq **and** Cond)

6.2 Language Semantics

Formulas are evaluated on traces of events. An event Ev is a tuple $\langle t, s, e(\bar{D})\rangle$ where

- t is a non-negative real number denoting the timestamp of the event. Timestamps form an increasing sequence;
- s is the state in which the event occurs. The event occurs in exactly one state due to the constraints on the state machine syntax and semantics;
- e is event name and $\bar{D} = (d_1, ..., d_n)$ is a possibly empty vector of constants (event parameters).

A trace is obtained from a monitored sequence of timestamped events that satisfies the state machine behaviour. The process of monitoring adds state information to the events. Trace T is a sequence of events $Ev_0, Ev_1, ..., Ev_i, ...$ For an integer i ≥ 0, we denote $T^i = Ev_i, Ev_{i+1}, ...$ and $T(i) = Ev_i = \langle t_i, s_i, e_i(\bar{D}_i)\rangle$.

Bindings of variables in event selectors are captured in environments. We define an environment $\Gamma = \{[v_1 \mapsto d_1], ...\}$ as a set of *mappings* from variables to values. $\Gamma[\Gamma']$ is the familiar operation of updating Γ with the mappings in Γ' and $\Gamma(v)$ gives the value of v in Γ.

For an environment Γ and a boolean expression $Cond$, we denote $\Gamma \models Cond$ if $Cond$ evaluates to true for the valuations in Γ.

When a sequence is matched in a trace, the environment with bound variables and the remaining part of the trace are propagated to the possible next steps. This is formalized as a partial function $Cont : Trace \times Env \times Seq \to Trace \times Env$. Env is a set of environments and $Trace$ is a set of traces.

We define a satisfaction relation between events, environments and event patterns as follows:

- $(\langle t, s, e(\bar{D})\rangle, \Gamma) \models \textbf{in}\ S\ EvDes(\bar{P})$ iff $EvDes = e$ or $EvDes = *$, $s \in S$, for every constant c_i in $\bar{P}$, $c_i = d_i$ and for the list of variables $v_1 \dots v_k$ in $\bar{P}$ we have $\Gamma = \{[v_1 \mapsto d_1], \dots, [v_k \mapsto d_k]\}$
- $(\langle t, s, e(\bar{D})\rangle, \varnothing) \models \textbf{not}[\textbf{in}\ S\ EvDes(\bar{P})]$ iff for all Γ, $(\langle t, s, e(\bar{D})\rangle, \Gamma) \not\models \textbf{in}\ S\ EvDes(\bar{P})$

If the set of states S and parameters $\bar{P}$ are not used in the event pattern, the corresponding checks are skipped.

The semantics of formulas is defined as satisfaction relation between formulas, traces and environments. We start with the semantics of sequences.

- $(T, \Gamma) \models \langle Var, E, Cond\rangle$ iff $(T(0), \Gamma_m) \models E$ and $\Gamma' \models Cond$, where $\Gamma' = \Gamma[\Gamma_m][Var \mapsto t_0]$

  $Cont(T, \Gamma, \langle Var, E, Cond\rangle) = \langle T^1, \Gamma'\rangle$
- $(T, \Gamma) \models \langle Var_1, Var_2, E, Cond\rangle$ iff $(T, \Gamma) \models \langle Var_1, E, Cond\rangle$

  $Cont(T, \Gamma, \langle Var_1, Var_2, E, Cond\rangle) = \langle T^1, \Gamma'[Var_2 \mapsto \Gamma(Var_2) + 1]\rangle$ where $Cont(T, \Gamma, \langle Var_1, E, Cond\rangle) = \langle T^1, \Gamma'\rangle$. Every occurrence counter takes initial value 0 before a formula is evaluated on a trace.

– $(T, \Gamma) \models ES_1$ **or**...**or** ES_n iff there exist i such that $1 \leq i \leq n$, $(T, \Gamma) \models ES_i$
$Cont(T, \Gamma, ES_1$ **or**...**or** $ES_n) = \langle T^1, \Gamma[\bigcup_k (\Gamma_k \setminus \Gamma)]\rangle$, for all k such that
$(T, \Gamma) \models ES_k$ and $Cont(T, \Gamma, ES_k) = \langle T^1, \Gamma_k\rangle$

It should be noted that $\bigcup_k (\Gamma_k \setminus \Gamma)$ cannot contain two different bindings for the same variable because a variable can occur at most once in all ES_k.

– $(T, \Gamma) \models Step_1$ **until** $Step_2$ iff there exist i such that $i \geq 0$, $(T^i, \Gamma^i) \models Step_2$ and for each k, $0 \leq k<i$, $(T^k, \Gamma^k) \models Step_1$ and $(T^k, \Gamma^k) \not\models Step_2$ where environments are defined as:
  - $\Gamma^0 = \Gamma$
  - $Cont(T^k, \Gamma^k, Step_1) = \langle T^{k+1}, \Gamma^{k+1}\rangle$ for all k, $0 \leq k<i$

  $Cont(T, \Gamma, Step_1$ **until** $Step_2) = Cont(T^i, \Gamma^i, Step_2)$

– $(T, \Gamma) \models Step_1$ **wuntil** $Step_2$ iff:
  - $(T, \Gamma) \models Step_1$ **until** $Step_2$
    $Cont(T, \Gamma, Step_1$ **wuntil** $Step_2) = Cont(T, \Gamma, Step_1$ **until** $Step_2)$

  or
  - $(T^i, \Gamma^i) \models Step_1$, for all $i \geq 0$ and Γ^i defined as in the case of **until**. $Cont$ is undefined

– $(T, \Gamma) \models Seq_1; Seq_2$ iff $(T, \Gamma) \models Seq_1$, $Cont(T, \Gamma, Seq_1)$ is defined and has value $\langle T^i, \Gamma'\rangle$ for some $i \geq 1$, and $(T^i, \Gamma') \models Seq_2$
$Cont(T, \Gamma, Seq_1; Seq_2) = Cont(T^i, \Gamma', Seq_2)$

– $(T, \Gamma) \models Seq$ **and** $Cond$ iff $(T, \Gamma) \models Seq$, $Cont(T, \Gamma, Seq)$ is defined and has value $\langle T^i, \Gamma'\rangle$ and $\Gamma' \models Cond$

– $(T, \Gamma) \models Seq$ **cf** F iff
  - $(T, \Gamma) \models$ **not** Seq

  or
  - $(T, \Gamma) \models Seq$, $Cont(T, \Gamma, Seq) = \langle T^i, \Gamma'\rangle$ is defined and $(T^i, \Gamma') \models F$

– $(T, \Gamma) \models$ **not** F if $(T, \Gamma) \not\models F$

– $(T, \Gamma) \models F_1$ **or** F_2 if $(T, \Gamma) \models F_1$ or $(T, \Gamma) \models F_2$

We state that a formula F **holds** in a trace T and an initial environment Γ if for every i ≥ 0, $(T^i, \Gamma) \models F$. The initial environment gives 0 as a value of all occurrence counter variables. For the other variables the user can supply an initial value or a default value is assumed.

During monitoring, time and data constraints are translated to formulas in the presented generic language. The translation is automatic and transparent to the users. Hence, the users do not need to work directly with the generic formulas which are often more verbose and more difficult to grasp than the source constraints. We first show how time rules are translated.

6.3 Translation of Timing Constraints

In this section we show how different types of timing rules are translated into formulas in the generic constraints language.

Timing rules use a simplified event selectors of the form $\mathbf{in}\, S\, e(\bar{P})$, where $\bar{P}$ is a possibly empty vector of constants. The set of states S and the parameters can be omitted. The translation of event selectors in timing rules to the selectors in the generic language is trivial and will not be discussed. Selectors will be given in capital letters A, B,...

Interval Rule. The general form is:

```
A -[p..q] -> B
```

A and B are selectors, [p..q] denotes a time interval with an obvious constraint $0 \leq p < q$. q may be infinity. The interval rule is translated to the following formula:

```
<t1, A>
cf
<t2, not[B]>
until
<t3, B, (t3-t1) in [p..q]>
```

The formula states that if a match of A is observed then there must be an occurrence of event that matches B and the first such occurrence is in the interval $[p, q]$.

Conditional Interval Rule. The rule gives two events as a premise of the rule and an expected interval.

```
A and B -> [p..q] between events
```

The rule is translated to:

```
<t1, A>; <t2, not[A]> until <t3, B>
where (t3-t1) in [p..q]
```

Absence of Event. The rule specifies an event that is a condition for not observing a follow up event in a certain interval.

```
A -> absent B in [p..q]
```

The corresponding formula is:

```
not
(<t1, A>;
<t2, *, t2-t1 <= q>
until
<t3, B, (t3-t1) in [p..q]>)
```

Periodic event rule. The rule specifies a triggering event A as a condition for a periodic observation of B with a period p and jitter j until a stop event C is observed.

```
A then B with period p and jitter j until C
```

The meaning of the rule is that if an event A is observed at time t then the i-th occurrence of event B after A and before C must be in the time interval $[t + i * p - j,\ t + i * p + j]$. The formula for this rule is:

```
<t, A>
cf
(<t1, i, B, t1 in [t + (i + 1)*p -j, t + (i + 1)*p + j]>
or
<t2, not[B], t2 <= t + (i + 1)*p + j>)
wuntil
<t3, C, t3 <= t + (i + 1)*p + j>
```

The rule uses an event selector with occurrence counter i. If an event A is observed then we check if the formula after 'cf' is satisfied for the events following A. The timestamp of A is bound to the variable t. There are three cases:

- we observe event B with timestamp $t1$. The condition of the first step in the disjunction checks if $t1$ is in the allowed interval. If it is not, the formula is not satisfied. If the condition is true (i.e. the occurrence is in the expected interval) the value of i is incremented and used to calculate the time interval of the eventual next occurrence of B;
- we observe an event different from B and C. In this case, the second component of the disjunction matches the event and we check the condition. If the condition is false this means that after the last occurrence of B we have not observed an event B and we have just observed an event that is already after the allowed interval. Therefore the formula is not satisfied;
- we observe event C. The condition checks if C is observed within the expected time upper bound for the event B. If the condition is false we have the situation in the previous case: B is not found in the expected interval and we have an event after this interval.

The semantics of the rule admits the case when C is never observed. **wuntil** is used to handle this.

The translation of periodic time rules illustrates that the resulting generic formulas may be more complex and more difficult to read than the original time constraint. We recall that users do not work with generic formulas directly. They use the more compact syntax of the surface languages.

Group Time Rule. This rule type allows specifying a rule that is a precondition for a series of interval rules thus allowing a scenario of several events. We will only show the case when an absence of event may be followed by other events with given time intervals. An example was shown in the previous section. The general form is:

```
group
A -> absent B in [0 .. p]
- [p1 .. q1] -> C
- [p2 .. q2] -> D
...
end group
```

This rule is translated to the formula:

```
<t1, A>;
<t2, not[B], (t2-t1) <= p>
wuntil
<t3, *, (t3-t1) > p>

implies

<t4, A>; <t5, *> until <t6, C, t6-t4-p in [p1..q1]>;
<t7, not[D]> until <t8, D, t8-t6 in [p2..q2]>;
....
```

6.4 Translation of Data Constraints

The grammar for data constraints rules is in Table 2. This language is as a subset of the generic constraints language following the same semantics.

Table 2. Syntax of Data Constraints Language

Data Constraint	$DConstraint$	$::= Seq$ **where** $Cond$
Sequence	Seq	$::= Step \mid Step$ **until** $Step \mid Seq\ ;\ Seq$
Step	$Step$	$::=$ **in** $S\ EvDes(\bar{P}) \mid$ **not** [**in** $S\ EvDes(\bar{P})$]
Event Designator	$EvDes$	$::= eventName \mid *$

6.5 Implementation Considerations

The definition of semantics for the generic constraints allows a proof that the initial semantics of time rules is preserved by the translation to the generic language. Generally, the development of the formalization enabled better understanding of the subtle details and greatly supported the software implementation.

An important aspect of the implementation is the fact that in a practical setting we deal with finite traces whereas the semantics of the formulas is given over infinite traces. This affects the evaluation of formulas. Consider an interval timing rule. In the trace we may observe the first event and according to the rule we must observe the second event within certain period of time. If the trace ends

before passing this period and no event is observed the rule evaluates to false. However, we cannot conclude if the second event will never appear because the information is not complete (monitoring has stopped). For situations like this we do not give a yes/no verdict for the rule. Instead, a warning is produced that states the rule has not been fully evaluated due to the termination of monitoring. As an alternative, the semantics can be defined for finite traces. This is a subject of future investigation.

6.6 Application of Monitoring on the Industrial Case

Component monitoring was applied during the development of the client software for the operating table. The examples shown here are simplifications of the actual models. The real model and constraints are more complex and take into account the complete interface and its behaviour. Several issues were revealed. For instance, movement requests with negative identifiers were sent by the client and accepted by the component. This was detected as a violation of the model and corrections were implemented in the software. The availability of explicit timing constraints allowed to experiment with different values for the allowed delays. The experiments revealed situations in which some events occur earlier than expected.

Generally, the process of modeling the intended behaviour of the interface based on textual documentation supported the engineers to explore cases in which the documentation was missing or the interpretation of the information was not clear. We also faced situations when the data constraints language was not expressive enough. In these cases, the constraints were successfully expressed in the generic language.

7 Concluding Remarks

The availability of precise component interface specifications enables early detection of defects and ultimately supports the development of software with higher quality. In this paper we presented ComMA, a framework for interface behaviour specification and focused on the support for runtime monitoring of timing and data constraints. The DSLs in ComMA integrate techniques and results from different research areas and provide a single entry point for engineers to specify and develop component interfaces.

The development of ComMA follows the industry-as-laboratory approach. DSLs are based on the concrete needs of the engineers and evolve following these needs. The developed languages are not business-specific and are not restricted to the medical domain. They are aimed at problems that are found in other domains as well and utilize general techniques thus making the framework easily generalizable.

Acknowledgements. The anonymous reviewers are thanked for useful suggestions for improvement. We would like to thank Dirk-Jan Swagerman for his support and the collaborating teams at Philips for constructive feedback.

The second author is grateful to Ed Brinksma for the very pleasant collaboration when Ed was the scientific director of the Embedded Systems Institute (currently TNO-ESI). With his very broad knowledge he was able to discuss any topic with experts and he created an excellent environment for productive industry-as-lab projects. Moreover, Ed is thanked for the stimulating role in the career development of the second author.

References

1. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In Sokolsky and Tasiran [13], pages 111–125.
2. Ed Brinksma and Jozef Hooman. Dependability for high–tech systems: an industry–as–laboratory approach. In *Design, Automation & Test in Europe (DATE'08)*, pages 1226–1231. European Design and Automation Association (EDAA), 2008.
3. Manfred Broy, Doron A. Peled, and Georg Kalus, editors. *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*. IOS Press, 2013.
4. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
5. Franck Cassez and Claude Jard, editors. *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, volume 5215 of *Lecture Notes in Computer Science*. Springer, 2008.
6. Feng Chen, Marcelo D'Amorim, and Grigore Rosu. A formal monitoring-based framework for software development and analysis. In *Proceedings ICFEM 2004*, volume 3308 of *LNCS*, pages 357–372. Springer-Verlag, 2004.
7. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Broy et al. [3], pages 141–175.
8. Hongman Kim, David Fried, Peter Menegay, Grant Soremekun, and Christopher Oster. Application of integrated modeling and analysis to development of complex systems. *Procedia Computer Science*, 16:98 – 107, 2013.
9. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
10. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.
11. Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. In Cassez and Jard [5], pages 1–13.
12. Colin Potts. Software-engineering research revisited. *IEEE Software*, 19(9):19–28, 1993.
13. Oleg Sokolsky and Serdar Tasiran, editors. *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*. Springer, 2007.
14. Bart Theelen, Oana Florescu, Marc Geilen, Jinfeng Huang, Piet van der Putten, and Jeroen Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *Proc. of MEMOCODE'07*, pages 139–148. IEEE, 2007.