Runtime Verification of Compound Components with ComMA

Ivan Kurtev[1,2](✉) and Jozef Hooman[3]

[1] Capgemini Engineering, Eindhoven, The Netherlands
ivan.kurtev@capgemini.com
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
[3] ESI (TNO), Eindhoven, The Netherlands
jozef.hooman@tno.nl

Abstract. The ComMA language has been developed to specify interfaces of software components, including protocol state machines, time and data constraints, and constraints on relations between events of multiple interfaces. The language has been devised in close collaboration with an industrial partner where it has been used to model a large number of interfaces. Based on a ComMA model, a number of artefacts can be generated such as documentation and test cases. Important is the generation of a monitor which is used to check if an implementation conforms to the specified model. This paper describes the ComMA monitoring algorithms. They are based on runtime verification techniques which have been extended to deal with the expressive ComMA language.

Keywords: Interface modeling · Runtime Verification · Component-based development

1 Introduction

Modern high-tech systems are complex entities consisting of multiple interacting components, typically supplied by different vendors. The lack of precise and explicit specifications of component interfaces often leads to problems during the integration of components. Component updates in already deployed systems may also lead to issues caused, for example, by unexpected changes in the interaction protocol and the time behavior. To address these issues, the ComMA (Component Modeling and Analysis) method and tool have been developed to support precise modeling of components and their interfaces.

ComMA provides a number of domain-specific languages for specifying client-server interfaces and component models in which multiple interfaces are used together. The interface language allows definitions of custom types, interface signatures in terms of messages exchanged between a client and server, behavior that specifies the allowed order of messages, and constraints on timing and data parameters. The component language allows modeling of simple and compound components (containing multiple parts) that use several interfaces together.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
N. Jansen et al. (Eds.): Vaandrager Festschrift 2022, LNCS 13560, pp. 382–402, 2022.
https://doi.org/10.1007/978-3-031-15629-8_21

Figure 1 shows a simple *Control* component that provides interface *IControl* to its clients via the provided *iControlPort3* port and uses the interfaces *ITemperature*, *IVacuum* and *ISource* via its required ports shown as dashed squares. Furthermore, component models support definition of constraints on the input/output relation of a component in the view of its interfaces and their interactions.

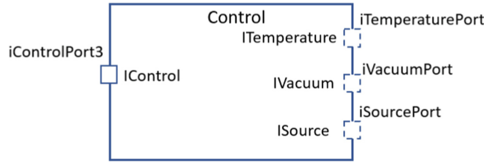

Fig. 1. Example simple ComMA component

The ComMA tool facilitates a number of engineering tasks by automatically generating artefacts from models. Figure 2 shows the main generators. Using models as a single source, the generators create UML diagrams of models, documentation based on a predefined MS Word template, monitoring infrastructure and test cases among others. The ability to monitor interfaces and components is a powerful feature of ComMA. It is used to check if an implementation conforms to an interface and/or component model. The automatically generated monitor checks if an execution trace that contains messages observed during component executions adheres to the behavior model and the time and data constraints. The output of the monitor is conveniently shown in a dashboard that summarizes the discovered issues along with other useful diagnostic information.

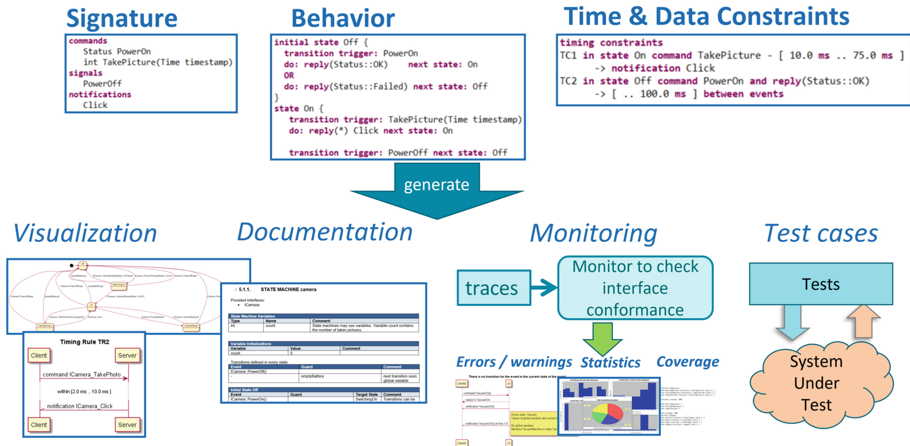

Fig. 2. Overview of ComMA generators

One of the main goals of ComMA is to allow easy application by industrial users. The modeling languages use familiar engineering notations such as state-based specification of the behavioral aspects, commonly found patterns for timing properties, and software architecture description concepts for component models. The languages have been developed iteratively respecting the requests and the feedback from the industrial users.

A number of previous publications [6,7] focused on ComMA interface specifications giving the syntax and semantics of the language, and elaborated on the interface monitoring algorithm and the check of time and data constraints. This paper builds upon these results and explains in details the component modeling language and monitoring algorithm.

The main contribution of our work is the integration of various theoretical results from the runtime verification body of knowledge into a framework that bridges the gap between the formal specification languages and the notations used by engineers, supports automation and integration of engineering tasks. The proposed specification language constructs do not introduce new logic, they focus on specifying constraints at the level of abstraction of the engineering models, handling components with multiple clients, and achieving compact specifications by refering to interface states. We are currently not aware of other component monitoring frameworks that utilize commonly used modeling notations to the degree that ComMA does.

The ComMA tooling is available via the open source Eclipse CommaSuite project[1]. The example used in the paper is included in the distribution.

Section 2 is an overview of ComMA interface models and monitoring, highlighting the features that are later used for the purpose of component modeling. Section 3 introduces the component modeling language on the basis of an example of a simple component. Section 4 follows with the presentation of compound components. Section 5 discusses the purpose, challenges and implementation of component monitoring. Sections 6 and 7 discuss related work and present the concluding remarks.

2 Interface Modeling and Monitoring

In this section we briefly describe the modeling of interfaces in ComMA (Sect. 2.1) and the monitoring of interfaces (Sect. 2.2).

2.1 ComMA Interface Modeling

An interface has a signature that defines synchronous and asynchronous calls from client to server (named commands and signals respectively) and notifications which are asynchronous messages from server to client. These three together with replies to commands are the messages that can be exchanged between a client and server and will be referred to as interface events or messages. The

[1] https://www.eclipse.org/comma/.

events may carry parameters. The signature of a simple interface called *IVacuum*, for managing vacuum in a system is shown in the next listing.

```
signature IVacuum
commands
  void VacuumOn
  void VacuumOff
notifications
  VacuumOK
```

In ComMA, the allowed order of interface events is captured in an interface behavior model which is defined as a protocol state machine. In addition, an interface defines time and data constraints. Time constraints specify allowed time intervals between events. Time and data constraints have been reported in [6] and are out of scope of this paper. As an example, the state machine of interface *IVacuum* is listed.

```
interface IVacuum
machine VacuumMachine {
  initial state NoVacuum {
        transition trigger: VacuumOn
            do: reply
            next state: Evacuating
  }
  state Evacuating {
        transition
            do: VacuumOK
            next state: Vacuum
  }
  state Vacuum {
        transition trigger: VacuumOff
            do: reply
            next state: NoVacuum }
  }
```

The state machine describes a client-server interface from the viewpoint of a server, that is, transitions are triggered by client calls of a command or a signal. The *do* part of a transition contains a sequence of actions of the server, which may include assignments to variables, a reply to a command, if-then-else statements, and notification patterns. A notification pattern specifies the occurrence of notifications; a special case is the *any order* construct to specify that events may happen in any order. Moreover, the language allows non-determinism, e.g., after a client call there may be multiple possible transitions by the server, possibly leading to different responses and states.

2.2 Interface Monitoring

Interface monitoring is the process of checking if a trace of observed events between client and server conforms to the interface definition. The following is

an example of the ComMA trace format (apart from this, JSON format is also supported):

```
components
Control ctrl
Vacuum vacuum

events

command 0.0 ctrl iVacuumPort vacuum iVacuumPort
            IVacuum VacuumOn
End

reply 0.11 vacuum iVacuumPort ctrl iVacuumPort
           IVacuum VacuumOn
End

notification 1.2 vacuum iVacuumPort ctrl iVacuumPort
           IVacuum VacuumOK
End
```

A trace starts with declarations of component instances (elaborated later when the component language is explained). They interact by sending messages to each other. Each message has a timestamp, a source instance and port (ports are explained in Sect. 3), a target instance and port, and contains the event and the interface it belongs to. The first message in the example is the command *VacuumOn* with timestamp 0.0 sent from component *ctrl* and its port *iVacuumPort* to component *vacuum*.

In ComMA, an interface monitor is a Java program that is automatically generated from the interface model. The ComMA monitor starts from the initial state of the state machine and consumes the events from the trace one by one. As soon as the monitor detects that an event in the trace does not conform to the state machine, it reports an error with some diagnostic info and stops monitoring.

Interface Events Augmented with State Information. As will be described in Sect. 3, component constraints may refer to states of interface descriptions. To allow checking of such constraints, the interface monitor augments interface events with the current state of the interface model. If an event is accepted by the monitor, it is annotated with the state in which it has been observed (known as *observation state*) and with the state that will be the current state when the next event is observed (known as *post-observation state*).

Since interface models allow non-determinism, multiple transitions for an event may be possible leading to potentially different observation and post-observations states. Consider Fig. 3: after observing notification *n1*, two transitions can be taken leading to different post-observations states: *S1* and *S3*

respectively. The interface monitor explores all possible traversal paths. If for a given path, the observed event is not allowed in the current state, the path is discarded. If all traversal paths are discarded then an interface monitoring error is detected. If in our example, signal *s* is observed after *n1* then the path which contains the transition to *S2* will be discarded since there notification *n2* is expected.

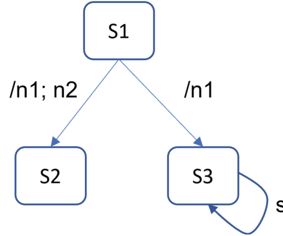

Fig. 3. Example state machine with non-determinism

The interface monitor maintains a list with *traversal path descriptions*. A description contains an identifier of the path, observation and post-observation states. At the start of monitoring only one path exists, assume its identifier is *p*. If a path leads to branching due to multiple possible transitions, each branch is uniquely numbered. The identifiers of the new paths are formed by concatenating the identifier of the parent path with the branch number. For example, if path with identifier *p122* leads to two new branches their identifiers will be *p1221* and *p1222*. A path *p* is a branch of *q* if the identifier of *q* is a prefix of the identifier of *p*.

After checking an event, the interface monitor provides a list of descriptions of all active traversal paths. In Sect. 5, we show how these path descriptions are used in the component monitoring process.

3 Component Models

We present the language constructs to specify components with constraints in Sect. 3.1 and to capture the identity of communication partners in Sect. 3.2.

3.1 Components with Functional Constraints

Interface specifications define the allowed order of events when a client uses an interface. Multiple interfaces are usually used together in the context of a single software unit that interacts with its environment. ComMA uses component models to define the allowed order of events from multiple interfaces and from multiple clients of the same interface.

As an example, we consider the *Control* component in Fig. 1 that provides interface *IControl* to its clients and uses services from other components via three interfaces. The textual syntax of component models in ComMA is as follows:

```
component Control

provided port IControl iControlPort3
required port ITemperature iTemperaturePort
required port IVacuum iVacuumPort
required port ISource iSourcePort
```

Ports are connection points used in the communication between component instances and are always associated to an interface. We distinguish between provided and required ports. A provided port is used by the clients of the component to connect to and interact with it according to the port's interface. Multiple clients are allowed to connect to a provided port. Required ports are used by the component to connect to its environment (consisting of other component instances).

The main purpose of a component model is to define constraints on the order of events observable in the context of the model (sent to or from the component ports). The construct used to specify this order is called *functional constraint*. A functional constraint captures an aspect of the complete behavior of the component and is usually restricted to a small subset of the observable events. Component models are not intended to define the complete component behavior in terms of reactions to all possible events in different states. In other words, component models are not design specifications that are used to derive a complete component implementation. An implementation is expected to satisfy all the functional constraints defined in a component model. In addition, a component model may define time and data constraints (not covered in this paper).

Functional constraints have two forms: state-based specification (known as state-based functional constraint) and an expression that has to evaluate to true for every observed event in the component context (called predicate functional constraint). The information about the current state of the interface associated to a port can be used in functional constraints.

As an example, the Control component handles requests for image acquisition and controls the vacuum and temperature in the system. A requirement for the control logic is that image acquisition is only possible if vacuum is present, and a certain temperature is reached. In terms of allowed message sequences, the command *AcquireImage* must be observed after the notifications about the correct state of vacuum and temperature, and only then the acquisition can be started. The following (simplified) constraint captures this requirement.

```
use events
command iControlPort3::AcquireImage
command iSourcePort::StartAcquisition
```

```
initial state Ready {
        // if vacuum and temperature OK start acquisition
        command iControlPort3::AcquireImage
            where iVacuumPort in Vacuum and
                  iTemperaturePort in TemperatureSet
        command iSourcePort::StartAcquisition
        next state: Ready
}
```

The constraint uses only two events listed in the section *use events*. The *use events* sections can also specify event patterns that denote more than one event such as all commands observed at a given port, all messages at a given port and so on. The allowed order of the specified events is given in a state machine similarly to the interface protocol state machines. Events that do not belong to the set defined in *use events* are not restricted.

In the example, the machine is very simple, consisting of one state and a single transition. The transition is triggered when command *AcquireImage* is observed at port *iControlPort3*. The pattern *command iControlPort3::AcquireImage* is a subject of a condition: a Boolean expression after the *where* keyword. *iVacuumPort in Vacuum* evaluates to true if the sequence of messages at *iVacuumPort* until the observation of *AcquireImage* has led to state *Vacuum*. Here *Vacuum* is a state in the *IVacuum* interface as shown in Sect. 2.1.

The pattern match is successful only if the condition is true. In terms of our example: *AcquireImage* is allowed to occur only if the vacuum and temperature ports are in the right state. This access to interface state information of ports is extremely handy. Without it, the functional constraint needs to replicate the sequence of the messages on the two ports that lead to the indicated interface states, information that is already present in the interface specifications. This way, code duplication is avoided and the size of the constraint is reduced.

In general, transitions in functional constraints are sequences of actions where the first action is a *message pattern*: an indication that a message of a given kind is expected to be observed. The other supported actions are assignment and if-then-else. Informally, a state-based functional constraint determines a set of message traces that conform to it. A trace in this set is such that (i) for every port, the projection of the trace on this port (i.e. the trace obtained by keeping only the messages on this port) conforms to the port interface; (ii) the trace obtained by keeping all the used events conforms to the constraint state machine. Here 'conforms' means that starting from the initial state and the first event in the trace, there is at least one transition traversal path that accepts the trace.

3.2 Using the Identity of Communication Partners

In a trace, every message has a source and a target, which are identifiers of component instances, and source and target ports. The component language provides a construct to capture the identity of the communication party for

a message observed at a component port. For example, when a command is received at a provided port, the identifier of the client can be obtained. Similarly, when a command is sent from a required port of a component, the identifier of the component can be obtained. This is illustrated by an example of a shared resource with multiple clients.

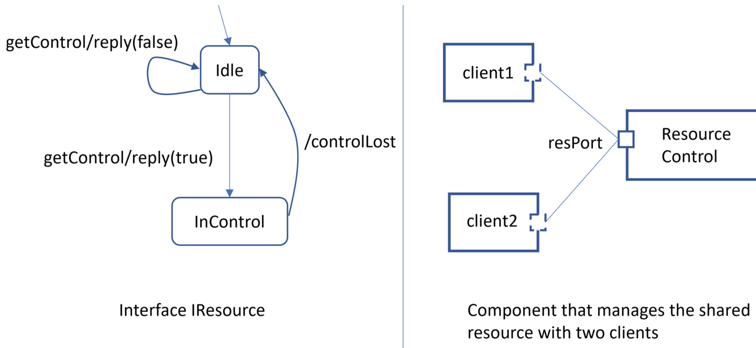

Fig. 4. Example interface and component models for managing shared resource

Assume that a component is providing access to a shared resource via a single provided port named *resPort* associated to interface *IResource* (see Fig. 4). Multiple clients can request control over this resource and the component is responsible for the policy of sharing it. The requirement is that at most one client is allowed to control the resource at a given moment. The following is a snippet from the corresponding functional constraint.

```
use events
resPort::reply to command getControl
notification resPort::controlLost

variables
id c
id c1

initial state ResourceFree {
    <c> resPort::reply(true) to command getControl
    next state: ResourceTaken

    resPort::reply(false) to command getControl
    next state: ResourceFree
}

state ResourceTaken {
    notification <c1>resPort::controlLost where c == c1
```

```
    next state: ResourceFree

    resPort::reply(false) to command getControl
    next state: ResourceTaken
}
```

If in the initial state, called *ResourceFree*, a reply to command *getControl* with argument *true* is observed then the identifier of the client who receives the reply is bound to the variable *c*. Note that the type of the variable is *id*. This is a predefined primitive type that allows only identity comparison operations. If a client receives a positive reply to *getControl*, the state *ResourceTaken* becomes the current state. In *ResourceTaken* no more positive replies to control requests are allowed. The control over the resource can be released only if the component decides to send notification *controlLost* to the client which currently has the control. Observe the usage of the variable *c1* that takes as value the identifier of the receiver of the notification. It is used in the condition that ensures the client which is currently in control receives the notification.

This example can be formulated more compactly as a *predicate* functional constraint. A predicate constraint is an expression preceded with the keyword *always*:

```
always [0-1] connections at portRes in InControl
```

The constraint states that at most one client connected to *portRes* is in interface state *InControl*. The expression uses a quantifier over the port connections (zero or one connection satisfies a condition). Note that for every client/connection of a provided port, a separate instance of the interface state machine is created, each with its own current state. This example shows how component functional constraints can restrict the order of events over the connections of a single port. In contrast, the first example (about control, vacuum and temperature) involves multiple ports and interfaces.

The interface state of the client at the moment of observing a message may be different from the state assumed after the observation (recall the difference between observation and post-observation states explained in Sect. 2.2). As an illustration, consider two consecutive actions in a functional constraint in the context of our current example:

```
<c> resPort::reply(true) to command getControl
   where c at resPort in Idle

b := (c at resPort in Idle)
```

The first action is a pattern that matches replies to *getControl* with parameter true. If it matches the currently observed message, variable *c* takes a value (the identifier of the receiver of the reply) and then the *where* clause is evaluated. Assume that its value is true (indeed, such a reply can only be observed in state *Idle*).

In the second action, however, the same expression evaluates to false since after observing the reply, the transition to *InControl* is taken and the interface

changes its state (see Fig. 4). Variable b will be assigned with false. This subtlety affects how the expressions that use the current interface state of a connection are evaluated. If they are used in the context of a message pattern, the state at the moment of observing the message is used for the corresponding connection, otherwise the post-observation state for the last observed message for this connection is used.

4 Compound Components

Component models may also define the internal component structure: its subcomponents (parts) and their interactions.

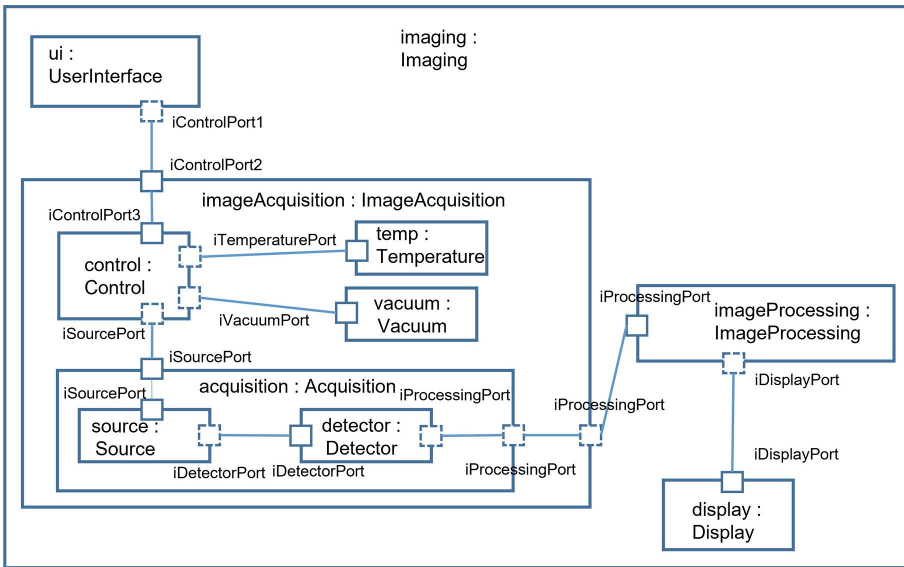

Fig. 5. Example compound component model

Figure 5 shows an example of a non-trivial component model called *Imaging*. It represents a system that captures images of some specimen. The model has four parts that are instances of other component models: *ui*, *imageAcquisiton*, *imageProcessing* and *display*. These component models may have their own internal structure as can be seen from the figure. The process of image acquisition requires vacuum in the system and a certain temperature level. The image is produced by a beam, generated by a source, going through the material and captured by a detector. The detector sends the data for further processing, storage and possibly visualization at a display. The *imageAcquisition.control* part is responsible for orchestrating the process: first ensuring vacuum and correct

temperature and then starting the acquisition process. The type of *imageAcquisition.control* is the *Control* component introduced in the previous section.

Components are connected via their ports; in Fig. 5 provided ports are shown as a solid square and required ports as a dashed square. Messages originating from the required port *ui.iControlPort1* are transmitted via a connection to the provided port *imageAcquisition.iControlPort2*. The connection between the latter and *control.iControlPort3* means that the messages will be further transmitted to the part *control*. In this way, a chain of connections defines a full path for message transmission. A more complex path can be observed between *detector.iProcessingPort* and *imageProcessing.iProcessingPort*.

As an example of the textual syntax of compound components, the next listing shows the specification of the *ImageAcquisition* component.

```
component ImageAcquisition

provided port IControl iControlPort2
required port IProcessing iProcessingPort

parts
Control control
Temperature temperature
Vacuum vacuum
Acquisition acquisition

connections
iControlPort2 <-> control::iControlPort3
control::iTemperaturePort <-> temperature::iTemperaturePort
control::iVacuumPort <-> vacuum::iVacuumPort
control::iSourcePort <-> acquisition::iSourcePort
acquisition::iProcessingPort <-> iProcessingPort
```

Note that a compound component contains *parts* which are named instances of component models. The parts have all the ports defined in their model. We will call such ports *part ports* and will refer to the ports defined by the component model as *boundary ports*. Within a component model, parts can interact with other parts by using connections between their ports. A connection of this kind is defined between a pair of provided and required ports of the same interface. Furthermore, a boundary port may be connected to a part port with the same kind (provided or required) and of the same interface. A connection indicates a channel for transmitting messages between the ports. For example, a message observed at a boundary port is redirected to the connected port of one of its parts. The ports do not perform any computation. A connection between a boundary port and a part port is just an indication of a path to the message's destination. The boundary port does not create a new message that is forwarded over the connection.

For a compound component, functional constraints can be used to relate events of interfaces of different components. For instance, such a constraint may express that an *AcquireImage* event on port *iControlPort1* of component *ui*

alternates with event *DisplayImage* on port *iDisplayPort* of component *display*. Moreover, also end-to-end time constraints can be expressed, e.g. to express that the *DisplayImage* event should happen within a certain amount of time after the *AcquireImage* event.

5 Component Monitoring

Component monitoring is performed for a given trace and component model. It checks if the trace satisfies: (i) the functional constraints in the model; (ii) the interface models associated to the component ports. Furthermore, if the component model has parts, the trace is checked against their models too.

The monitor for a trace has the following logical structure: *trace processor* that reads the trace, identifies the component instances to be monitored, and invokes *component and interface monitors*. A component monitor contains *functional constraint monitors*.

We first briefly explain how component instances are specified in the traces, an elaboration of the information previously given in Sect. 2.2.

5.1 Traces with Messages Between Component Instances

A trace starts with a declaration of all component instances and their models. The part-whole relation between the instances is encoded in their identifiers.

```
Imaging imaging
UserInterface imaging.ui
ImageAcquisition imaging.imageAcquisition
```

In this declaration, there is one instance of the *Imaging* model, called *imaging*, and two parts of *imaging* which are named with a compound name where the prefix is the name of the containing instance and the last segment is the simple name given in the component model.

Messages cannot cross the boundaries of the containing components for its source and target. For example, a message from *imaging.ui* can only be sent to the parts at the same level of nesting, that is, to *imaging.imageAcquisition*. Observe that *imageAcquisition.iControlPort2* is connected to *control.iControlPort3* (Fig. 5) so the messages received at the former will be further directed to the latter port. Regardless of the connection, it is not allowed to specify a message from *imaging.ui.iControlPort1* to *imaging.imageAcquisition.control.iControlPort3* because it crosses the boundary of the enclosing *imageAcquisition* component.

5.2 Algorithm for Monitoring a Trace

For a given trace and a component model, all instances of the component model are monitored. Note that a trace may have instances of different models. Only the instances of the given model are considered. An instance behaves according

to its model if the sequence of the messages relevant to this instance satisfies the constraints in the model and in the models of its direct or indirect parts. A direct part is contained immediately in the instance, an indirect part is contained further down in the containment tree induced by the part-whole relation among components. A message is relevant for a component instance if it is observed at one of its boundary ports or is exchanged between two of its direct or indirect parts. When a component instance is monitored all its direct and indirect parts are monitored too.

Due to the possibility of connections between ports, a given message can be checked against more than one component model. One of the tasks of the monitoring algorithm is for a given message to determine a sequence of checks against the relevant component models. Assume that we monitor the instance *imaging* (see the example in previous section) and *imaging.imageAcquisition* receives a message at *iControlPort2* from *imaging.ui*. Let's denote this message as *(ui, iControlPort1, imageAcquisition, iControlPort2)* abstracting away the message kind and possible parameters. This is a message between parts of the component being monitored, it is visible in the context of component model *Imaging* and therefore it has to be checked against the *Imaging* constraints. Furthermore, the message is observed at the boundary ports of two parts thus posing the need to check it against their models (*UserInterface* and *ImageAcquisition*).

When the message is received at *imageAcquisition.iControlPort2*, the connection to *control.iControlPort3* is followed and the message is ultimately received by the *control* part. The message needs to be checked against the *Control* model as well. In summary, the considered message will be checked against the following component models: *Imaging*, *UserInterface*, *ImageAcquisition*, and *Control*.

The constraints in *Control* model will refer to messages observed at *Control* instances and their boundary ports. Because of this, before checking the message against the constraints defined in *Control*, the destination of the message is changed to *(ui, iControlPort1, imageAcquisition.control, iControlPort3)*.

In summary, for every message relevant to the monitored instance, a list of (component instance, port) pairs is determined, where the presence of port is optional. For every element in the list, the message will be checked by the component monitor for the instance. Both the instance and the port will be used if one of the message ends needs renaming. In our example the list of pairs is *(imaging, _), (userInterface, _), (imageAcquisition, _), (control, iControlPort3)*.

Generally, the list is formed in the following way: (i) for a message at component boundary port, the instances are the ones reachable following the chain of port connections towards the component parts; (ii) for a message between parts, the instances are the ones reachable following the connections from the source and target message ports plus the immediate parent of the parts. Our example falls under the second case.

Before the check of functional constraints, interface monitoring is performed. Every relevant message is exchanged in a pair of client and server components and an interface monitor will be created for this pair. In our example, an interface monitor will be created for the connection between

ui.iControlPort1 and *imageAcquisition.iControlPort2*. It will provide interface state information shared among three ports: the two mentioned above and *control.iControlPort3* (note the connection between *imageAcquisition.iControlPort2* and *control.iControlPort3*).

A sketch of the algorithm that processes and monitors a trace is given in procedure *MonitorComponentInstances* that takes as input a component model *cModel* and a trace. Recall that the trace contains information about the component instances and the messages among them. Monitoring is performed on all messages from the trace that are relevant for the instances of *cModel*. All instances of *cModel* can be obtained from the component declarations part in trace (line 2). As explained before, a relevant message is observed at a boundary port or between two (direct or indirect) parts of some instance (lines 9–11). If a relevant message is found, it is first checked by its interface monitor (created for the connection between the message's sender and receiver). The interface monitor is treated as an object: it can be created, stored and it has behavior and internal state (line 18). Interface monitoring of a relevant message is always performed as long as no interface error has been detected for this connection. If at least one interface error is found for a given component instance on some of its ports, the check against the component model is not performed anymore. The map *interfaceErrorStatus* (line 6) keeps track if an interface error has been observed for an instance (see lines 24 and 42 where the map is used and updated). If the interface monitor accepts the message the next step is to perform the check against the relevant component models (which further leads to checking of their functional constraints). Traversal path descriptions that will be used in functional constraint checks are obtained from the interface monitor (line 27). The pairs of component instance and port (as explained previously) are determined and then iterated (lines 31–40). For every component instance in the pairs, a component monitor is obtained (line 36, created once on demand, then stored and used later when the same instance is monitored for another message).

```
1  MonitorComponentInstances(cModel, trace)
2    instances <- instances of cModel from trace
3
4    //indicates if interface monitoring error occurred
5    //for an instance; initialized with false
6    interfaceErrorStatus <- map from instances to Boolean
7    While trace has unprocessed messages Do
8        msg <- read next message from trace
9        If (msg at boundary port of some i in instances)
10           Or
11           (msg between parts of some i in instances)
12       Then
13           i <- the instance that satisfies the
14                condition in lines 9-11
15           //interface monitor is instantiated once on
16           //demand for each pair (client, server),
17           //stored and used when needed
```

```
18            interfaceMonitor <- obtain interface monitor
19                               for msg
20            If interfaceMonitor already gave error Then
21               Continue
22            End If
23            If msg is accepted by interfaceMonitor Then
24               If interfaceErrorStatus at i is true Then
25                  Continue
26               End If
27               pathDescriptions <- obtain traversal path
28                        descriptions from interfaceMonitor
29               pairsInstancePort <- list of (instance, port)
30                                           for cModel and i
31               For Each pair in pairsInstancePort Do
32                  change the relevant message end for pair
33
34                  //component monitor instantiated on
35                  //demand and stored
36                  componentMonitor <- obtain component
37                        monitor for pair.instance
38                  MonitorComponentInstance(componentMonitor,
39                           msg, pathDescriptions)
40               End For
41            Else
42               set interfaceErrorStatus at i to true
43            End If
44         End If
45     End While
46     collect and print results from all interface and
47     component monitors
48  End
```

The monitoring of a component instance is sketched in procedure *MonitorComponentInstance*. A component monitor contains a list of functional constraint monitors (line 4, *componentMonitor.fcMonitors*). The input message is checked by every functional constraint monitor for which no error has previously been detected.

```
1   MonitorComponentInstance(componentMonitor,
2                            msg,
3                            pathDescriptions)
4       For Each fcMonitor in componentMonitor.fcMonitors Do
5           If fcMonitor has not previously detected error
6           Then
7              MonitorFunctionalConstraint(fcMonitor, msg,
8                                          pathDescriptions)
9           End If
10      End For
11  End
```

The implementation of functional constraint monitors and the usage of the traversal path descriptions is explained in the next section.

5.3 Checking Functional Constraints

We will first discuss how expressions that use interface states at ports are evaluated using the information in the traversal path descriptions, and then will outline the implementation of functional constraint monitors.

Different traversal paths in an interface state machine may lead to different states. In functional constraints, the expressions that refer to port states may produce different results for different paths ultimately causing a constraint to fail for some paths and succeed for others. Furthermore, multiple ports with multiple interface monitors can exist in the context of a component instance, each monitor possibly having multiple traversal paths. This means that for a given component instance, all combinations of traversal paths from all monitors on all ports need to be formed and functional constraints have to be evaluated for every combination. When forming the combinations we take into account that connected ports share an interface monitor and therefore share traversal paths in a single combination.

In the following explanation we assume that for each functional constraint there is an implementation in some programming language. Such an implementation may be based on some of the well known ways to implement state machine specifications in a general purpose programming language. We also assume that the implementation is parameterized with a constraint execution context. The context contains the current state of the machine, the values of the variables, and the states of the interface monitors associated to the component ports (as explained previously). The implementation can be configured with a given context and provides a function called *consume* that receives a message as input. This function, based on the current machine state, searches for transitions that match the message. For such a transition, all actions are executed thus leading to new execution context. If more than one transition exists, all are explored leading to multiple new execution contexts. The function returns the list of these new execution contexts. If the list is empty, then the functional constraint fails for the observed message in the given execution context.

At conceptual level, a functional constraint monitor is a data structure that contains the implementation of the constraint, a set of tuples with the traversal path descriptions per interface monitor on the ports (referred to as *portsStates*), and for every such tuple a set of constraint execution contexts.

The next procedure sketches the check of a functional constraint on a given message. The idea here is to check the functional constraint for every tuple with ports states. The functional constraint monitor is responsible for initializing and updating the set with these tuples. Since a functional constraint may have multiple execution contexts, a given tuple with ports states is associated to a set of execution contexts.

```
1  MonitorFunctionalConstraint(fcMonitor,
2                              msg,
3                              pathDescriptions)
4      update fcMonitor.portsStates for the given msg
5          with info from pathDescriptions
6      If fcMonitor.portsStates is empty Then
7          register functional constraint error
8          Return
9      End If
10     If msg not used by the functional constraint Then
11         Return
12     End If
13     newPortsStates <- empty list
14     For Each portsStateTuple in fcMonitor.portsStates Do
15         set current ports states in fcMonitor
16         to portsStateTuple
17         newFCExecutionContexts <- empty list
18         For Each fcContext in portsStateTuple.fcContexts
19         Do
20             set current context in fcMonitor to fcContext
21             newContexts <- fcMonitor.consume(msg)
22             add newContexts to newFCExecutionContexts
23         End For
24         If newFCExecutionContexts is not empty Then
25             portsStateTuple.fcContexts <-
26                newFCExecutionContexts
27             add portsStateTuple to newPortsStates
28         End If
29     End For
30     If newPortsStates is empty Then
31         register functional constraint error
32         Return
33     Else
34         fcMonitor.portsStates <- newPortsStates
35     End If
36 End
```

The first operation in the procedure updates the tuples of ports states with the info from the path descriptions provided by the interface monitor (line 4). If in a given tuple the path identifier at the port on which the message is observed is not a prefix for any path in *pathDescriptions* this means that the path has been discarded by the interface monitor after the check of the message. The tuple is discarded as well. If the path identifier is a prefix of some paths in *pathDescriptions*, the tuple is replicated for every such path and state information is updated. It is possible that after this update, all tuples are discarded. This is treated as functional constraint violation: there are no traversal paths in the interface monitor that satisfy the functional constraint (lines 6–9). Observe also that this update is done for every message even if it is not used by the functional constraint (not listed in *use events* section).

After the update of the ports states, and the check if the message is used by the constraint (lines 10–12), the logic is straightforward: the outer loop (starting in line 14) iterates over each tuple of ports states, the inner loop (line 18–23) iterates over the functional constraint execution contexts for this tuple. The functional constraint instance is configured with a pair of ports states and context and then function *consume* is called. The result is a list of new execution contexts (line 21). If for a given tuple and all its contexts no transition in the functional constraint is found for the message (manifested by empty list of new contexts) the tuple is discarded. If all tuples are discarded, a functional constraint error is registered (lines 30–32).

The set of tuples with port states may become very large. In practice this hardly happens: multiple traversal paths in an interface are usually reduced to one after observing a few events.

The monitoring algorithms presented here serve as a base for the implementation of the component monitoring tool (done in Java and available in the Eclipse CommaSuite project). The trace processor, component and functional constraint monitors, and the functional constraints are completely generated from component and interface models.

6 Related Work

The component modeling approach presented in this paper uses the concepts of component, interface, port and connector. They are known from software architecture and system modeling languages such as UML, SysML, AADL among others. This is a conscious choice based on the observation that these concepts are familiar to the practitioners. Monitoring of systems with complex component architecture has been addressed in [4,13]. Falcone et al. [4] propose a runtime verification framework for component-based systems modeled with similar constructs where behavior is modeled with finite state machines. The ports in this approach accept simple values whereas in ComMA, ports are associated to interfaces with signatures that may use complex structures. Stockmann et al. [13] execute monitoring on traces obtained from simulating a software architecture specified in an executable modeling language. In our approach, traces are observations on the implementation.

There is a large variety of languages for specifying properties to be monitored. Their theoretical underpinnings are usually in formal logics. Dwyer et al. [3] identify patterns for properties observed in practice. These patterns have been used in property specification languages to achieve more compact and intuitive syntax. ParTraP [2] is a recent work based on this idea. Time and data constraints in ComMA are derived from common patterns observed in industrial practice, like periodicity, response time and others. We have considered using the patterns identified in [3] for functional constraints but opted for state-based specifications, already used in the interface definition language.

RML [1] is a domain-specific language for runtime verification. ComMA constructs like *any order* and event patterns have their counterparts in this language.

An interesting possibility is to treat ComMA specifications as syntactic sugar and investigate how they can be translated to RML primitives.

There exist approaches that weave the monitor's code into the system under monitoring (e.g. the language LIME and its monitoring infrastructure [5]). In our work the monitor is executed separately from the monitored system, often in offline mode after collecting the observations. This is beneficial in cases where the system implementation cannot be altered and instrumented.

ComMA compound components may represent distributed systems. This opens the possibility for distributed monitoring. Currently, the component monitor is monolithic, executed on a single node and working with traces that unite all (possibly distributed) observations. Performing distributed monitoring is a possible future direction, whose importance is recognized in a recent survey [10].

7 Conclusions

We presented the ComMA language that allows modeling component-based systems and specifying properties that are monitored during system execution. Monitors are automatically generated from component specifications. This work extends our previous work on specification and monitoring of component interfaces in industrial context. It brings a new application scope by allowing multiple interfaces to be used together in a single component and specifying interacting components at system level.

As mentioned in earlier publications [6,7,11], ComMA has been developed driven by user needs in close collaboration with Philips, following the industry-as-laboratory approach [8]. Hence, the languages use concepts and notations that are familiar to engineers and aim at rapid industrial adoption. Currently, the ComMA tooling is actively used at Philips to model and monitor of a number of industrial components, see for instance [9,12]. Future work will focus on applications in more complex cases where monitoring the order and timing of component interaction is a primary focus.

Acknowledgements. We would like to thanks our colleague Dennis Dams and the anonymous reviewers for many useful suggestions for improvements.

References

1. Ancona, D., Franceschini, L., Ferrando, A., Mascardi, V.: RML: theory and practice of a domain specific language for runtime verification. Sci. Comput. Program. **205**, 102610 (2021). https://doi.org/10.1016/j.scico.2021.102610
2. Blein, Y.: ParTraP: a language for the specification and runtime verification of parametric properties. (ParTraP: Un langage pour la spécification et vérification à l'exécution de propriétés paramétriques). Ph.D. thesis, Grenoble Alpes University, France (2019). https://tel.archives-ouvertes.fr/tel-02269062
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999, Los

Angeles, CA, USA, 16–22 May 1999, pp. 411–420. ACM (1999). https://doi.org/10.1145/302405.302672
4. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Softw. Syst. Model. **14**(1), 173–199 (2013). https://doi.org/10.1007/s10270-013-0323-y
5. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The LIME interface specification language and runtime monitoring tool. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 93–100. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_7
6. Kurtev, I., Hooman, J., Schuts, M.: Runtime monitoring based on interface specifications. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 335–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_17
7. Kurtev, I., Schuts, M., Hooman, J., Swagerman, D.J.: Integrating interface modeling and analysis in an industrial setting. In: Proceedings of 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pp. 345–352 (2017)
8. Potts, C.: Software-engineering research revisited. IEEE Softw. **19**(9), 19–28 (1993)
9. Roos, N.: ComMA interfaces open the door to reliable high-tech systems. Bits & Chips, 8 September 2020. https://bits-chips.nl/artikel/comma-interfaces-open-the-door-to-reliable-high-tech-systems/
10. Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
11. Schuts, M., Hooman, J., Kurtev, I., Swagerman, D.J.: Reverse engineering of legacy software interfaces to a model-based approach. In: Proceedings of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS 2018). Annals of Computer Science and Information Systems (ACSIS), vol. 15, pp. 867–876 (2018)
12. Schuts, M., Swagerman, D.J., Kurtev, I., Hooman, J.: Improving interface specifications with ComMA. Bits & Chips, 14 September 2017. https://bits-chips.nl/artikel/improving-interface-specifications-with-comma/
13. Stockmann, L., Laux, S., Bodden, E.: Architectural runtime verification. In: IEEE International Conference on Software Architecture Companion, ICSA Companion 2019, Hamburg, Germany, 25–26 March 2019, pp. 77–84. IEEE (2019). https://doi.org/10.1109/ICSA-C.2019.00021