Softw Syst Model (2012) 11:633–670
DOI 10.1007/s10270-011-0206-z

REGULAR PAPER

Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems

Shaukat Ali · Lionel C. Briand · Hadi Hemmati

Received: 4 December 2010 / Revised: 8 April 2011 / Accepted: 17 May 2011 / Published online: 15 June 2011

Abstract Model-based robustness testing requires precise and complete behavioral, robustness modeling. For example, state machines can be used to model software behavior when hardware (e.g., sensors) breaks down and be fed to a tool to automate test case generation. But robustness behavior is a crosscutting behavior and, if modeled directly, often results in large, complex state machines. These in practice tend to be error prone and difficult to read and understand. As a result, modeling robustness behavior in this way is not scalable for complex industrial systems. To overcome these problems, aspect-oriented modeling (AOM) can be employed to model robustness behavior as aspects in the form of state machines specifically designed to model robustness behavior. In this paper, we present a RobUstness Modeling Methodology (RUMM) that allows modeling robustness behavior as aspects. Our goal is to have a complete and practical methodology that covers all features of state machines and aspect concepts necessary for model-based robustness testing. At the core of RUMM is a UML profile (AspectSM) that allows modeling UML state machine aspects as UML state machines (aspect state machines). Such an approach, relying on a standard and using the target notation as the basis to model the aspects themselves, is expected to make the practical adoption of aspect modeling easier in industrial contexts. We have used AspectSM to model the crosscutting robustness behavior of a videoconferencing system and discuss the benefits of doing so in terms of reduced modeling effort and improved readability.

Keywords Aspect-oriented modeling · UML state machines · Robustness · UML profile · Crosscutting behavior · Robustness testing

Communicated by Dr. Jean-Michel Bruel.

S. Ali (✉) · L. C. Briand · H. Hemmati
Simula Research Laboratory, P. O. Box 134, 1325 Lysaker, Norway
e-mail: shaukat@simula.no

L. C. Briand
e-mail: briand@simula.no

H. Hemmati
e-mail: hemmati@simula.no

S. Ali · L. C. Briand · H. Hemmati
Department of Informatics, University of Oslo, Oslo, Norway

1 Introduction

Modeling software functional behavior has always been an important focus of the modeling community to support many development activities such as model-based testing (MBT) and automated code generation. Regarding model-based testing, which is the specific focus on this paper, much less attention has been given to modeling non-functional behavior such that the testing of non-functional properties (e.g., safety and robustness) can be automated. Though several UML profiles have been proposed to address the modeling of non-functional properties (including the UML profile for QoS and Fault Tolerance [1], the MARTE profile [2], and UMLSec [3]), it is not yet clear whether they can fully support test automation.

Our motivation here is to support model-based robustness testing. An IEEE Standard [4] defines robustness as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions". A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, using our industrial case study as an example, modeling such robustness behavior of a videoconferencing system (VCS) is

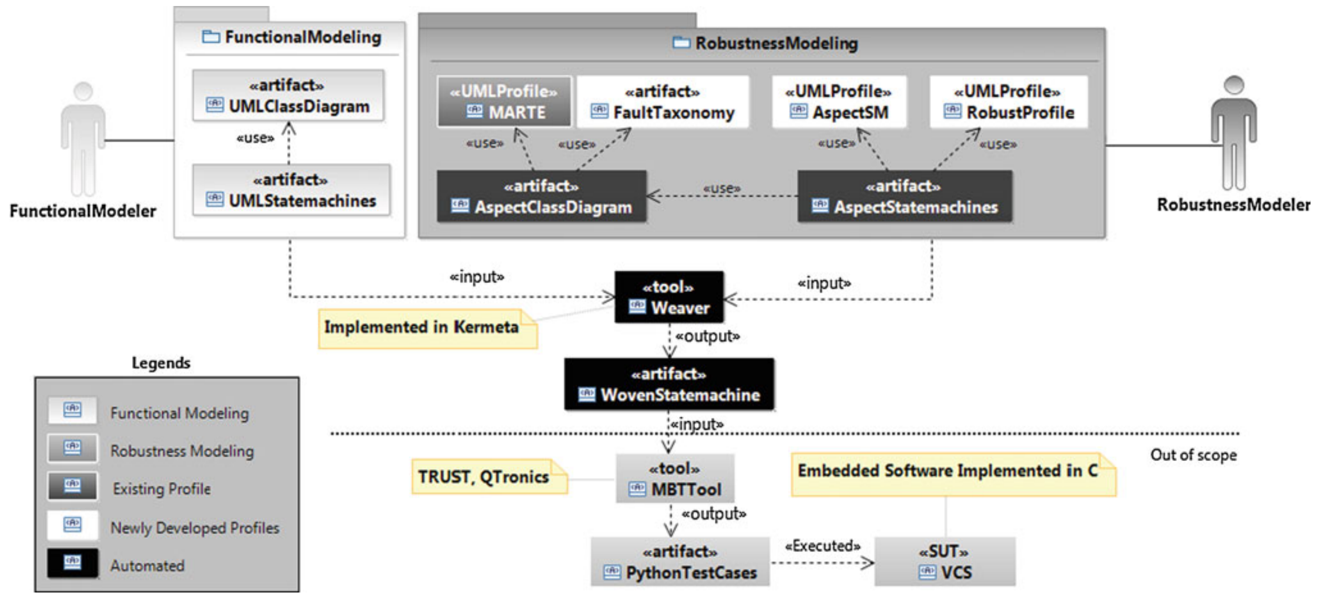


Fig. 1 An overview of RUMM

to model its behavior in the presence of hostile environment conditions (regarding the network and other communicating VCSs), such as high percentage of packet loss and high percentage of corrupt packets. The VCS should not crash, halt or restart in the presence of, for instance, a high percentage of packet loss. Furthermore, the VCS should continue to work in a degraded mode, such as continuing the videoconference with low audio and video quality. In the worst case, the VCS should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS and must be tested systematically and automatically to be scalable.

To automate such systematic testing, one can model the system robustness behavior to such events and resort to model-based testing (MBT). However, robustness behavior is typically crosscutting many parts of the system functional model and, as a result, modeling such behavior directly within the functional models is not practical since it leads to many redundancies and hence results in large, cluttered models. To cope with this issue, we decided to adopt aspect-oriented modeling (AOM) [5], which provides separation of concerns (SoC) during design modeling. Crosscutting concerns are modeled as aspect models and are woven into a primary model (base model), modeling non-crosscutting concerns. AOM can potentially offer several benefits such as: (1) enhanced modularization, (2) easier evolution of models, (3) increased reusability, (4) reduced modeling effort, and (5) improved readability [5,6].

Our goal in this paper is to provide a complete solution in terms of both aspect and state machine features necessary for model-based robustness testing. Furthermore, we want to minimize the effort involved in learning a new language over standard UML and enable automated, model-based testing. To achieve this, we present a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM and assess it on an industrial case study involving a commercial videoconferencing system. Such studies are very few in the research literature and are rarely run and reported in a satisfactory manner [7]. To the knowledge of the authors, only a few industrial applications of AOM have been reported to date [8–11] and had very different objectives than RUMM. An overview of RUMM is shown in Fig. 1. The core of RUMM is the definition of a UML state machine profile for AOM: AspectSM (shown as a white artifact in Fig. 1 in *RobustnessModeling*). We limited our profile to UML state machines as follows: (1) They are the main notation currently used for model-based test case generation [12] and are particularly useful in control and communication systems. (2) As it is often the case, our industrial case study exhibits state-based behavior so that it is natural to initially provide support for UML state machines. The profile can, however, be extended to other UML diagrams in the future, following similar principles. We rely on developing a profile instead of developing a domain-specific language since, in our case study context as in many others, minimizing extensions to UML is expected to ease practical adoption. More thorough discussions on this issue are presented in Sect. 7. Modelers of functional aspects of the system can be different from the ones specifying its robustness behavior. The latter make use of AspectSM to model aspect state machines.

Another important part of the RUMM is another UML profile (RobustProfile) shown as a white artifact in Fig. 1, based on the fault taxonomy defined by [13] and the IEEE standard classification for anomalies [14]. The profile is used by a robustness modeler to develop aspect state machines and is defined specifically to assist in defining test strategies for robustness testing. In addition, the profile helps generating test scripts based on classes of faults modeled using the profile. Once again, the profile is defined on UML state machines, as they are the main focus of this paper. We follow the widely accepted and used definitions in [13] for faults and failures. A fault is an incorrect state of a system or its environment in the presence of which the system cannot provide a correct service. Such deviation from the correct service is called a failure. A fault type is identified based on a fault taxonomy (white artifact in Fig. 1), and the UML profile MARTE is used to model it in a UML class diagram (Aspect Class Diagram, dark gray artifact in Fig. 1). In a subsequent step, aspect class diagrams are used to model actual faulty behavior as aspect state machines (AspectStatemachines) using both AspectSM and RobustProfile. Finally, robustness models comprising aspect class diagrams and aspect state machines are woven into functional models once again composed of UML class diagrams and state machines. This is performed using our weaver implemented in Kermeta [15], and the woven state machines produced by the weaver can be used in turn by a model-based testing tool, for instance the TRUST tool [16] or QTronics [17], to generate executable test cases. In our case, test cases are generated in Python, which is used as a test script language by our industry partner (Cisco, Norway). Note that this paper addresses only robustness modeling, and details on test case generation and execution are outside the scope of this paper.

The contributions of the paper can be summarized as follows: (1) A RobUstness Modeling Methodology (RUMM) that enables the systematic modeling of robustness behavior in a practical and scalable way. (2) a UML 2.0 profile (RobustProfile), which is based on a fault taxonomy in [13] and the IEEE standard classification for anomalies [14], to model faults, recovery mechanisms, and failure states. (3) The application of the MARTE profile in conjunction with RobustProfile to model faulty environment conditions. (4) A UML 2.0 profile (AspectSM) to support comprehensive aspect modeling for UML 2.0 state machines and enable automated robustness testing. AspectSM supports modeling crosscutting on all features of UML 2.0 state machines and supports all basic features of AOSD such as pointcuts, introduction, joinpoints, and advice. (5) An empirical evaluation and discussion of the benefits of modeling robustness behavior of an industrial system using RUMM and AspectSM. (6) Tool support, based on model transformations in Kermeta [15], to automatically weave AspectSM aspects into base state machines (modeling the core functional behavior of a system).

The rest of the paper is organized as follows: Sect. 2 provides a case study and a running example that we use to explain various concepts in RUMM. Section 3 provides an overview of the RUMM methodology. Section 4 describes the terminology, techniques, and tools that are required to understand and apply RUMM, including a definition and justification of the AspectSM profile (Sect. 4.2) and details of its corresponding weaver (Sect. 4.7). Section 5 demonstrates the application of the profile using a very simplified version of our industrial case study. Section 6 discusses the benefits achieved when applying RUMM to one complete subsystem of our industrial case study. Section 7 discusses existing works that are directly related to the objectives of RUMM. Finally, Sect. 8 reports on future work and conclusions.

2 Case study and running example

Our case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a videoconference system (VCS) called Saturn [16]. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. In this paper, to demonstrate the applicability of RUMM, we focused on this particularly important subsystem (Saturn) and left out the other functionalities of the VCS. We selected this subsystem because robustness testing is concerned with testing the behavior of VCS in the presence of hostile environment situations, which can only be tested when the VCS is in a conference call with other systems and which is what Saturn manages. Saturn is complex enough to demonstrate the applicability and usefulness of RUMM while still remaining manageable in the context of a case study. To provide simple running examples in the next sections, we modeled a reduced version of Saturn where one can only establish calls and cannot start or stop presentations. From now onwards, we will refer to this simplified Saturn model as S-Saturn to differentiate it from the complete case study model used in Sect. 6 to discuss the benefits of RUMM.

2.1 Functional models of S-Saturn

The functional model of S-Saturn consists of a class diagram and a state machine. The class diagram of S-Saturn is shown in Fig. 2 and is meant to capture information about APIs and system (state) variables, which are required to generate executable test cases and oracles in our application context. Saturn's API is modeled as a set of methods in the *Saturn* class such as *dial*() and *callDisconnect*(). In our case, the parameters of these methods are either

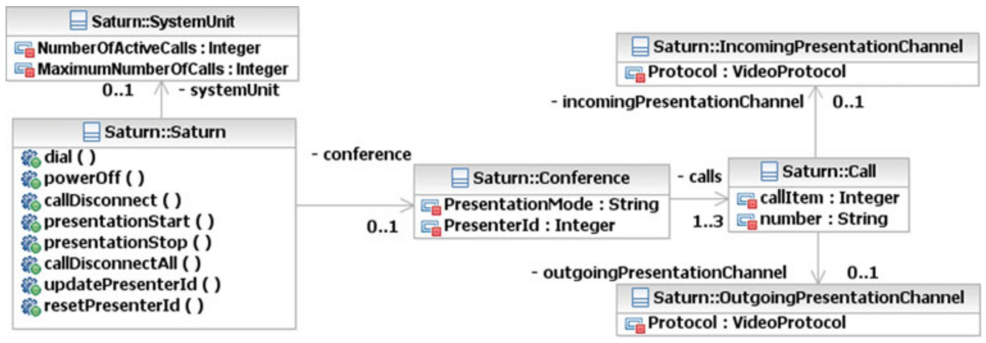


Fig. 2 Conceptual model of the S-Saturn subsystem

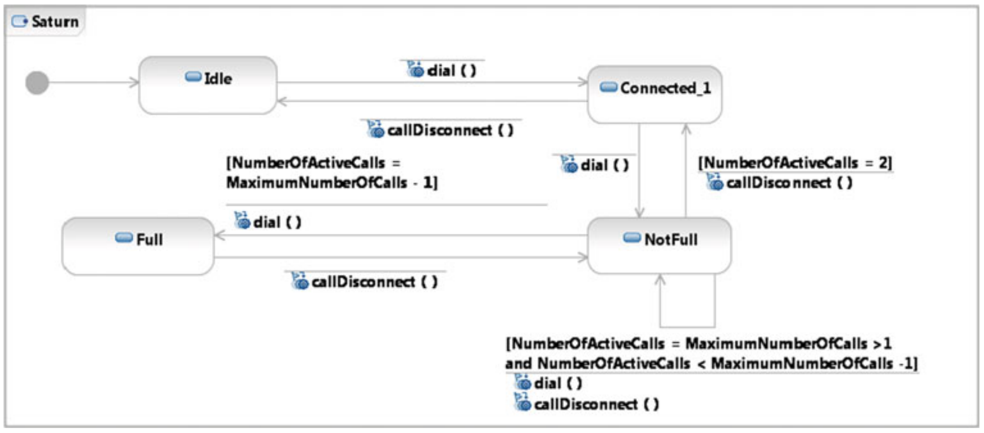

Saturn
Idle
dial ( )
Connected_1
callDisconnect ( )
[NumberOfActiveCalls = MaximumNumberOfCalls - 1]
dial ( )
[NumberOfActiveCalls = 2]
callDisconnect ( )
dial ( )
Full
NotFull
callDisconnect ( )
[NumberOfActiveCalls = MaximumNumberOfCalls >1 and NumberOfActiveCalls < MaximumNumberOfCalls -1]
dial ( )
callDisconnect ( )

Fig. 3 Base state machine for the S-Saturn subsystem

modeled as primitive data types (e.g., *String*) or as Enumeration types (e.g., *CallProtocol*).The state variables of the system are modeled as instance variables of classes in the conceptual model. For example, two system variables in the *SystemUnit* class are *NumberOfActiveCalls* and *MaximumNumberOfCalls*. *NumberOfActiveCalls* is an *Integer* which determines the number of VCS that are currently in a Saturn videoconference, whereas *MaximumNumberOfCalls* determines the maximum number of simultaneous calls supported by Saturn.

The state machine modeling the nominal functionality of S-Saturn, referred to as a base state machine, is shown in Fig. 3. It consists of four simple states. From the *Idle* state, invoking the *dial*() method of the *Saturn* class leads to the *Connected_1* state, which represents the behavior of the system when there is a conference without any presentation with one endpoint. As long as there exists one endpoint in the conference and no presentation is transmitting, S-Saturn stays in the *Connected_1* state, and when S-Saturn dials to more endpoints, it transitions to the *NotFull* state until it connects to the maximum number of endpoints it supports and transitions to the *Full* state. Each simple state has an associated state invariant based on the system variables modeled in the conceptual model. For instance, the *Idle* state has the following state invariant:

$$self.systemUnit.NumberOfActiveCalls = 0$$
$$and\ self.conference.PresentationMode = \text{‘}off\text{’}$$

2.2 Robustness behavior

To explain various activities and concepts involved in defining the profiles, we will use a crosscutting robustness behavior named ‘*MediaQualityRecovery*’. This behavior is related to the robustness behavior of a VCS in the case when media quality falls below an acceptable media quality level and tries to recover. The VCS should not crash when the media quality falls below this acceptable level, but should rather keep on operating at a lower quality level and try to recover from this situation. In the worst case, the VCS should clean up system resources and go back to the most recent safe state, in which the VCS was exhibiting normal behavior. In our current case study, an example of a safe state is the *Idle* state. Such a robust behavior is very important in a commercial VCS, as quality expectations are high regarding robustness to media quality faults. Recall that the models above are greatly simplified and that, in Sect. 6, we provide results from the complete case study and other important robustness aspects that we modeled for Saturn.

3 Robustness modeling methodology

Our goal is to devise a solution to model robustness behavior, which (1) is complete in terms of aspect and state machine features, (2) minimizes the learning curve over standard modeling skills, and (3) enables automated, model-based testing. To achieve this, we defined a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM. Recall from Sect. 1 that we follow the standard definition of robustness provided in the IEEE 610.12 standard [4]. Such robustness is considered very critical in many standards such as in the *IEEE Standard Dictionary of Measures of the Software Aspects of Dependability* [4], the ISO's *Software Quality Characteristics* standard [18], and the *Software Assurance Standard* [19] by NASA. The RUMM methodology (Fig. 4) is suitable for systems, which implement substantial robustness behavior to deal with faulty situations in the environment, such as communication and control systems. A1 and A2 activities are related to functional modeling, whereas activities A3–A6 are related to modeling robustness behavior. Activity A7 is automated and merges functional (base state machines) and robustness (aspects) models together into a complete model. Activities A1–A6 are related to modeling functional and robustness behavior and are manual. In this section, we will explain very briefly each activity. Additional, detailed information will be provided in the next sections, followed by the application of RUMM in an industrial case study.

The first activity (A1) involves developing a conceptual model [20] of an SUT using a UML 2.0 class diagram based on the domain analysis of the SUT. In this activity, we model different domain concepts of the SUT as classes and relationships between them, which are determined as the result of domain analysis. In addition, we model state variables of the SUT as attributes in the class diagram. We also model public operations of the SUT (API) and external events in the SUT environment as signal receptions. The conceptual model is then used in activity A2 for developing a behavioral model of the SUT as one or more UML state machines. Attributes defined in the conceptual model are used for various purposes such as defining state invariants and guards on transitions. The operations and signal receptions defined in the conceptual model are used as triggers on transitions of state

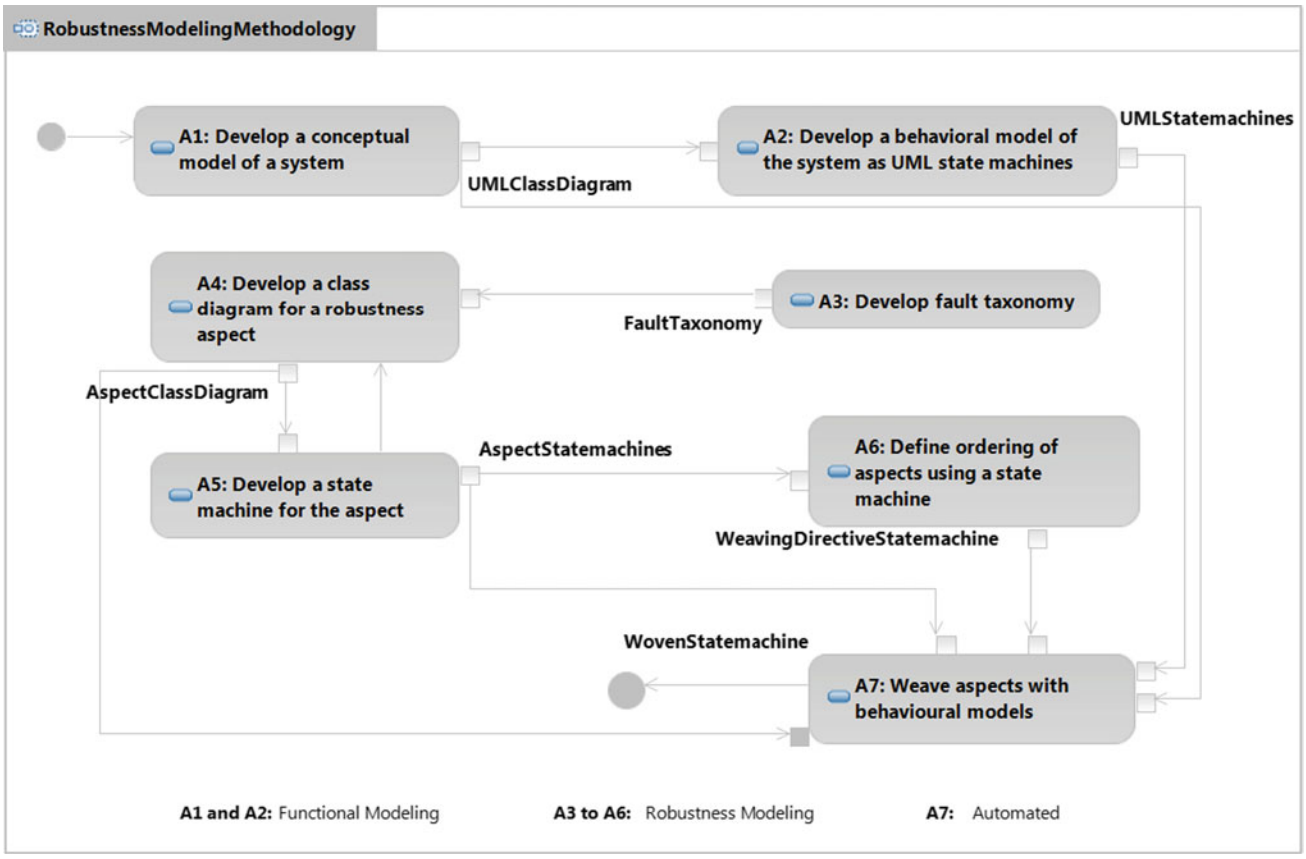

RobustnessModelingMethodology
A1: Develop a conceptual model of a system
UMLClassDiagram
A2: Develop a behavioral model of the system as UML state machines
UMLStatemachines
A4: Develop a class diagram for a robustness aspect
FaultTaxonomy
A3: Develop fault taxonomy
AspectClassDiagram
A5: Develop a state machine for the aspect
AspectStatemachines
A6: Define ordering of aspects using a state machine
WeavingDirectiveStatemachine
WovenStatemachine
A7: Weave aspects with behavioural models
A1 and A2: Functional Modeling
A3 to A6: Robustness Modeling
A7: Automated

Fig. 4 Methodology for robustness modeling (RUMM)

machines. In model-based robustness testing, one of the most important tasks is the identification and modeling of faults, in the presence of which we must test the behavior of the SUT. To systematically identify these faults, the development of fault taxonomy is required (A3) and is provided in Sect. 4.1. The application of the fault taxonomy to an industrial system is reported in Sect. 5.3. Activity A4 requires modeling different properties of the system's environment, the violations of which lead to the various types of faults identified from the fault taxonomy (A3). The guidelines for this process are defined in Sect. 5.4. Activity A5 requires modeling robustness behavior as aspect state machines. As described in Sect. 4.4, this requires the use of the AspectSM profile. The profile definition is provided in Sect. 4.2. The control flow arrow from activity A5 to A4 depicts that multiple robustness aspects can be modeled one after another. Once all robustness aspects have been modeled, we may need to define the order in which the aspects should be woven into the base state machine developed in activity A2. Guidelines for modeling the ordering of aspect state machines as a weaving-directive state machine are presented in Sect. 4.6. Finally, activity A7 weaves aspect state machines with base state machines. For this activity, we developed a tool using Kermeta [15], a well-known model transformation environment. The details of the tool are presented in Sect. 4.7, and the weaving algorithm is detailed in Appendix B.

4 Concepts, techniques, and tools required for RUMM

This section describes the concepts, techniques, and tools that are needed to apply RUMM. In addition, we provide further definitions of the terminology employed as needed.

4.1 Definitions

This section provides basic definitions required to understand the rest of the paper.

4.1.1 Faults and failures in the context of UML state machines

While modeling robustness, we model faults in the behavior of the operating environment of an SUT. Such behavior of the environment may lead the SUT into abnormal situations. In UML state machines, we model faults in the environment as either signal events or change events, on one or more transitions in the state machine of the SUT. Firing such transitions may lead the SUT to a degraded state where the SUT tries to recover from the fault while still providing some of the required service in a degraded mode. If the SUT is successful in recovering from the fault, it then goes back to a normal mode of operation. Otherwise, it may go to a failure state or the initial state.

4.1.2 Fault classification based on taxonomy

Many fault taxonomies are proposed in the literature; however, most of them are either specific to architectures, for instance service-oriented architecture (SOA) [21,22] and Component-based Systems [23], or to application domains such as aeronautics and space [24]. We chose the widely known and referenced fault taxonomy presented in [13] because it is very comprehensive and generic, and thus can be extended for specific needs as required in our case. For instance, we extended the taxonomy to accommodate for media quality faults, which are very important for a commercial VCS. The fault taxonomy for elementary fault classes provided in [13] is modeled in Fig. 5 as a class diagram. Dark gray-colored classes in Fig. 5 show the fault classes we extended for our specific needs. The taxonomy states that a fault can be categorized based on different views/perspectives such as those based on *SystemBoundary* or *Dimension*. Using *SystemBoundary*, faults can be classified into either *InternalFault* or *ExternalFault* depending on where they occur. Details on classes of faults are provided in [13]. Given our goal, we extended some fault classes in the fault taxonomy to model faults which are specific to the VCS. For instance, to provide a support for modeling media-related faults, which are important for an industrial VCS, we introduced a view *RequirementType* (Fig. 5) and defined two fault classes: *FunctionalFault* and *NonFunctionalFault*. We further classified *NonFunctionalFault* into *MediaFault* (Fig. 5), with further subclasses *Audio* and *Video*. In addition, we extended *ExternalFault*, which comprises faults in networks and external systems, into *NetworkFault* and *SystemFault* subclasses. *SystemFault* corresponds to the faults in one or more VCS communicating with the SUT. Since in robustness testing the focus is always on modeling behavior of an SUT in the presence of faults in its environment, all fault classes in the taxonomy are valid from the perspective of other VCSs communicating with the SUT. For instance, a *SoftwareFault* in a VCS communicating with the SUT can have an effect on the latter's behavior. We provide an example use of the taxonomy in Sect. 5.3 for our case study.

4.2 The AspectSM profile

Using the AspectSM profile, we model each aspect as a UML state machine with stereotypes (aspect state machine). The modeling of aspect state machines is systematically derived from a fault taxonomy (Fig. 5) categorizing different types of faults (incorrect states [13]) in a system and its environment (such as communication medium and other systems).

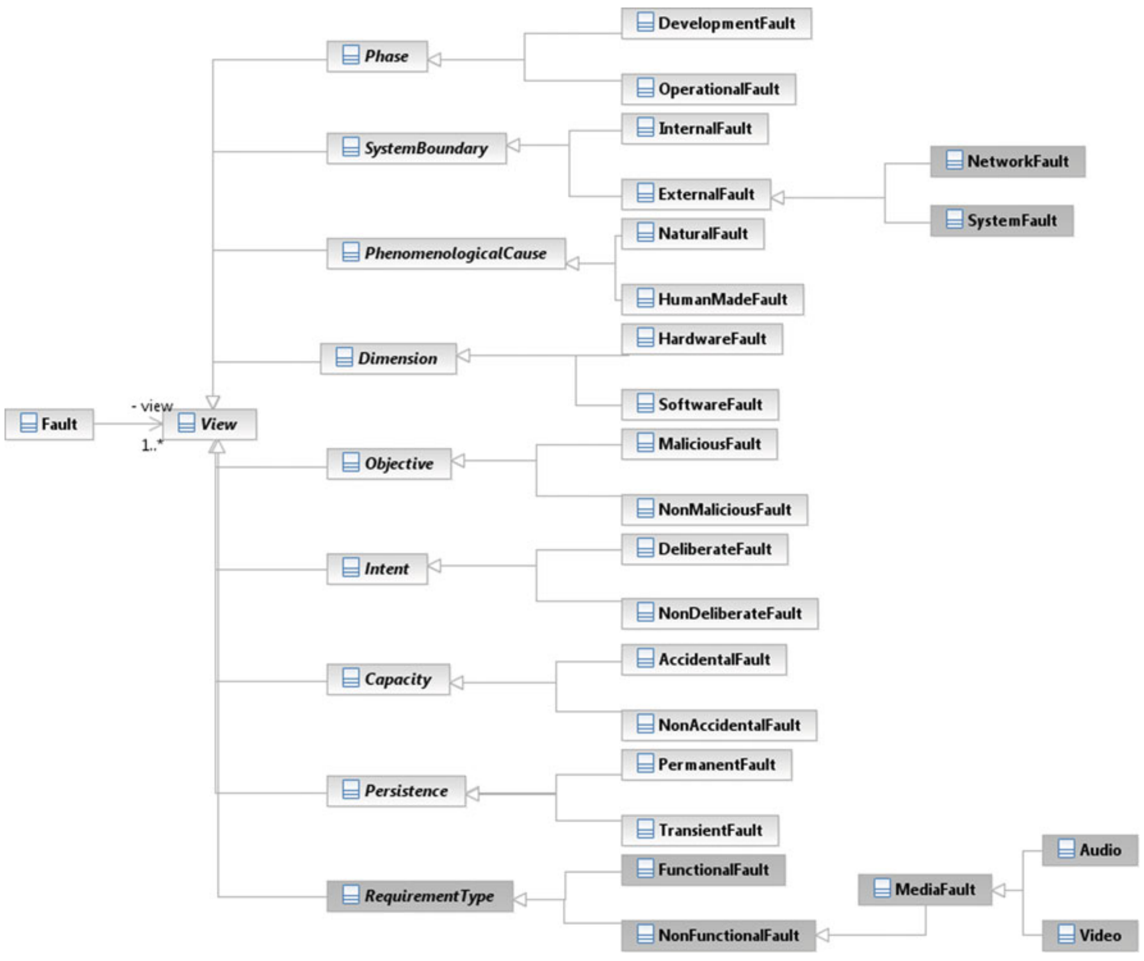


Fig. 5 High-level fault taxonomy

Such a modeling approach models each type of robustness behavior separately from the state machines modeling nominal functionality (base state machine) and hence results in enhanced separation of concerns. Furthermore, our modeling approach models crosscutting behaviors as separate aspect state machines and hence reduce modeling effort when compared with modeling robustness directly in combination with nominal behavior. The readability of models is then improved as robustness behavior that tends to be redundant when modeled directly is clearly separated out and expressed once. Following the general ideas proposed in [26,25], to model aspects using the same notations as the base model, we used UML state machines to model both aspect and base models, which is expected to facilitate practical adoption. In industrial applications of model-based testing, it is always desirable to minimize the need to learn different notations to model different testing concerns (such as security and robustness concerns). Though profiles already exist in the literature that allow modeling aspects as UML state machines [6,27–30], we decided to define our own profile to address the three following problems:

1. Crosscutting behavior can exist on any modeling element in UML 2.0 state machines, but the existing profiles and approaches do not support all features, such as state invariants and guards [6,27,30,31]. These are however crucial in the context of model-based testing, and in particular for automated test case generation [32].
2. Existing modeling approaches using profiles require, for modeling aspect features (such as pointcut and advice), to develop new diagrams that are not part of the UML 2.0 standard [29,33], thus making adoption in practical contexts more difficult. Indeed, such profiles require developing specific tool support for new diagrams and entails training users on how to build them. As a result, in practice, the use of non-standard modeling languages is discouraged.
3. Some of the existing approaches do not support *all* basic features of aspect orientation such as *Introduction*.

More details and discussions on related work are provided in Sect. 7.2

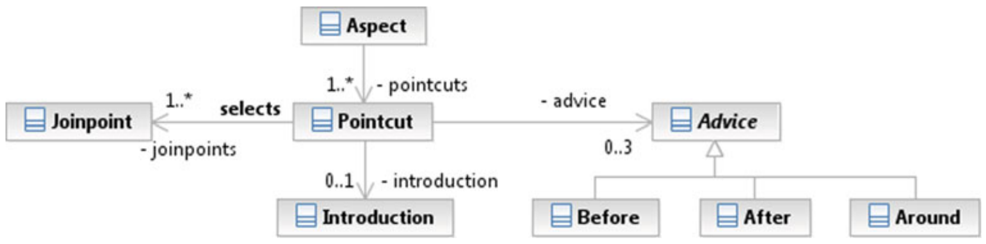


Fig. 6 Conceptual domain model of the profile

Fig. 7 Constraint on Pointcut

```
Context Pointcut inv:
    self.advice.oclIsKindOf(Before)->size()= 0 or  self.advice.oclIsKindOf(Before)->size()=1
    and  self.advice.oclIsKindOf(Around)->size()=0 or  self.advice.oclIsKindOf(Around)->size()=1
    and self.advice.oclIsKindOf(After)->size()=0 or  self.advice.oclIsKindOf(After)->size()=1
```

The AspectSM profile is the core component of RUMM because modeling robustness as aspect state machines is achieved through standard UML extension mechanisms. This profile was developed by augmenting many of the concepts in existing UML state machine profiles for AOM (Sect. 7) to achieve the specific goal of supporting automated, model-based robustness testing. Although the AspectSM profile is developed specifically for robustness testing, its application to other purposes such as for security testing should be investigated. In this section, we provide a detailed description of AspectSM.

A UML profile enables the extension of UML for different domains and platforms, while avoiding any contradiction with UML semantics. In [34], two main approaches for profile creation are discussed. The first approach directly implements a profile by defining key concepts of a target domain, such as what was done to define SysML [35]. The second approach first creates a conceptual model outlining the key concepts of a target domain followed by creating a profile for the identified concepts. This latter approach has been used for defining profiles such as the UML profile for Schedulability, Performance, and Time specification (SPT) [36], the QoS and Fault Tolerance specifications [1], and the UML Testing Profile (UTP) [37].

We used the second approach to define the AspectSM profile since it is more systematic as it separates the profile creation process into two stages. In the first stage, we develop a conceptual model, which helps identify domain concepts and their relationships. In the second stage, we identify the mapping between the main concepts and UML modeling elements and define corresponding stereotypes on UML metaclasses. Finally, the relationships between stereotypes are obtained from the relationships that were identified between the domain concepts in the first stage.

4.2.1 Domain view of the profile

The conceptual domain model for AspectSM is shown in Fig. 6 as an MOF-based [38] metamodel. The conceptual domain model defines aspect-oriented modeling concepts.

An *aspect* describes a crosscutting behavior, which in our context is the robustness behavior of a system, i.e., the behavior of the system in the presence of faults in its environment, such as packet loss and jitter for a network. Since a network can experience packet loss at any time, it crosscuts the SUT functional behavior. Since in our case study, like in many systems with state-driven behavior, the behavior of the system is modeled as UML 2.0 state machines, we also model aspects as UML 2.0 state machines to facilitate adoption in practice. Robustness behavior, for example the behavior of an SUT in the presence of packet loss or corrupt packets, is modeled using one or more state machines.

A *joinpoint* is a model element, which corresponds to a pointcut where an advice (additional behavior) can be applied [39]. All modeling elements in UML are possible joinpoints, where an advice can be applied [5]. For UML state machines, some examples of joinpoints include a state or a transition.

A *pointcut* selects one or more joinpoints with similar properties, where advices can be applied. A pointcut can have at most one before advice, one around advice, or one after advice (Fig. 7). All pointcuts are expressed with the OCL on the UML 2.0 metamodel. We decided to use the OCL to query joinpoints since it is the standard to write constraints on UML models and is also commonly used to query jointpoints (modeling elements such as states and transitions). Also, several OCL evaluators are currently available that can be used to evaluate OCL expressions such as the IBM OCL evaluator [40], OCLE 2.0 [41] and EyeOCL [42]. Furthermore, writing pointcuts as OCL expressions do not require a modeler to learn a notation that is not part of the UML standard. In the literature, several alternatives are proposed to write pointcuts [6,27–29,33], but all of them either rely on languages (mostly based on wildcard characters to select joinpoints, for instance, '*' to select all joinpoints) or diagrammatic notations, which are not standard, thus forcing modelers to learn and apply new notations or languages. Using the OCL, we can write precise pointcuts to select jointpoints with similar

```
Context uml::Transition
    self->select(trigger|trigger.event.oclIsKindOf(SignalEvent))
```

Fig. 8 A pointcut in OCL selecting all transitions with signal events

properties. We do so by selecting modeling elements (jointpoints) based on the properties of UML metaclasses. This further gives us the flexibility to specify pointcuts of varying complexities. For instance, we can specify a very complex pointcut based on all properties of a UML metaclass, e.g., a pointcut on the *Transition* metaclass, selecting a subset of transitions in a base state machine for which all properties of the *Transition* metaclass are the same. On the other hand, we can also specify a simple pointcut based on a small subset of properties of a UML metaclass. For example, a pointcut on the *Transition* metaclass selecting all those transitions from a base state machine, which have the same guards, though other properties such as triggers or effects can be different. In UML state machines, states and transitions are the most important modeling elements and all other elements are contained within them such as state invariants in states, and guards and actions in transitions. Therefore, pointcuts are defined in the context of the UML metaclass *Vertex*, to query states and apply advices on states and its composing elements such as state invariants and do, entry, and exit activities. Similarly, pointcuts are also defined in the context of the UML metaclass *Transition* to query transitions, and advices are applied on transitions and its containing elements such as *Guard* and *Actions*. The attributes for the *Vertex* and *Transition* metaclasses can be obtained from the UML specifications [43]. For example, a pointcut may select all transitions of a state machine, which have triggers with signal events. This pointcut, defined in Fig. 8, is written as an OCL expression on attributes of the UML metaclass Transition and selects all transitions that have triggers with signal events on them.

An advice is an additional behavior added at joinpoint(s) selected by a pointcut. This behavior can be added as OCL constraints or in the form of state machine modeling elements such as a guard or an effect. As most of the concepts in AOM are inspired from aspect-oriented programming (AOP) languages such as AspectJ [44], in a similar way in AOM, an advice can be of type before, after, or around. A before advice is applied before joinpoint(s), an after advice is applied after joinpoint(s), whereas an around advice replaces joinpoint(s). For example, introducing guards on all transitions of a state machine that have signal events as triggers is an example of a before advice on transitions. Table 1 summarizes the semantics of each type of advice for each UML 2.0 state machine modeling element. Examples for advice on all UML 2.0 state machine modeling elements are provided in [45].

An introduction is similar to the inter-type declaration concept in AspectJ [44] and is used in many AOM approaches [33,46–48] to introduce new modeling elements in a base model. In a similar fashion, we use introduction in our context to introduce new modeling elements in a UML state machine, e.g., a new state or a transition. In our context, we mostly use introduction to introduce transitions in a base state machine, which correspond to faults in the environment (Sect. 4.1.1). We also use introduction to introduce new states in a base state machine, which are related to a robustness behavior such as the state of a system which is operating with degraded performance (Sect. 4.1.1).

4.2.2 UML representation

In this section, we provide details on the AspectSM profile, such as details on stereotypes and their attributes.

Profile diagrams: Profile diagrams for AspectSM are presented in Figs. 9, 10, and 11. Profile diagrams show extension relationships between stereotype classes (denoted « *stereotype* ») and UML metaclasses (denoted « *metaclass* »), i.e., relationships showing which stereotypes are applied to which UML metaclasses (extension relationship). For example, Fig. 10 shows the *Introduction* stereotype applied to *Transition, Behavior, Trigger, Constraint*, and *State* metaclasses. These diagrams also show relationships between stereotype classes such as associations and generalizations. For instance, in Fig. 11, *Before, After*, and *Around* metaclasses are inherited from the *Advice* metaclass. To decrease the complexity of profile diagrams, we have not shown associations between stereotype classes. However, associations of stereotype classes are listed in Table 2. In addition, Table 2 provides information about extensions and generalizations. The extensions column in Table 2 shows which UML metaclasses a particular stereotype is applied to. For example, the *Aspect* stereotype is applied to the *uml::StateMachine* metaclass in row 2 of Table 2. The generalizations column illustrates the inheritance relationship between stereotype classes. For example, in row 5 of Table 2, the *Before* stereotype is inherited from the *Advice* stereotype.

Profile elements description: We now describe each profile element. Extensions, generalizations, and associations are shown in Table 3. The extension relationship tells on which metaclasses of UML a stereotype is applied. For instance, in Table 2, the « *Aspect* » stereotype has an extension relationship with the UML metaclass *StateMachine*. This means that the « *Aspect* » stereotype can be applied to a UML state machine. All stereotypes except « *Aspect* » are applied to all modeling elements related to UML state machines, though in Table 3 we list only the key metaclasses of UML state machines.

Attributes associated with the « *Aspect* » stereotype are shown in Table 4. Attributes associated with the « *Pointcut* », « *Before* », « *After* », and « *Around* » stereotypes are

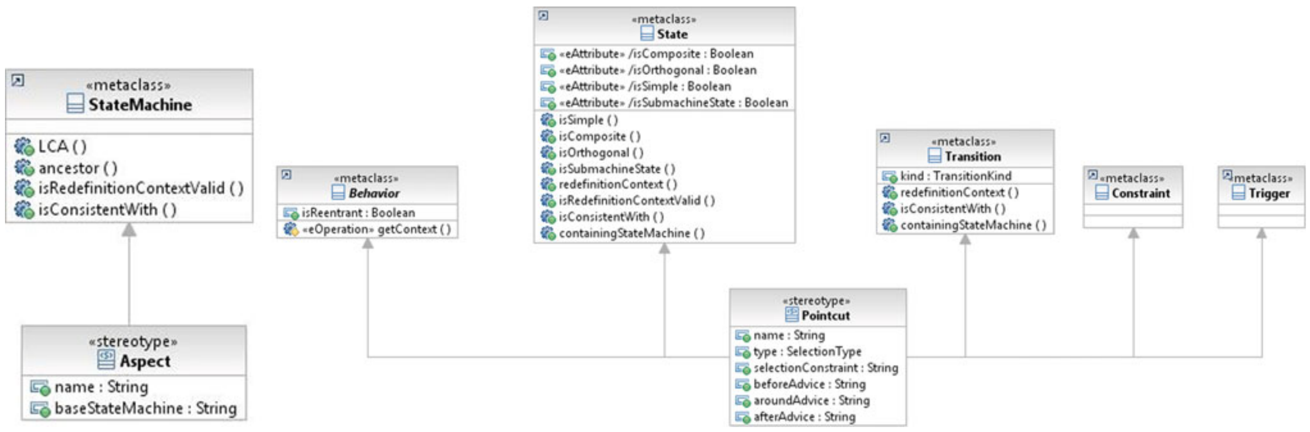


Fig. 9 « *Aspect* » stereotype applied to *StateMachine* metaclass (*left*) and « *Pointcut* » stereotype applied to various metaclasses (*right*)

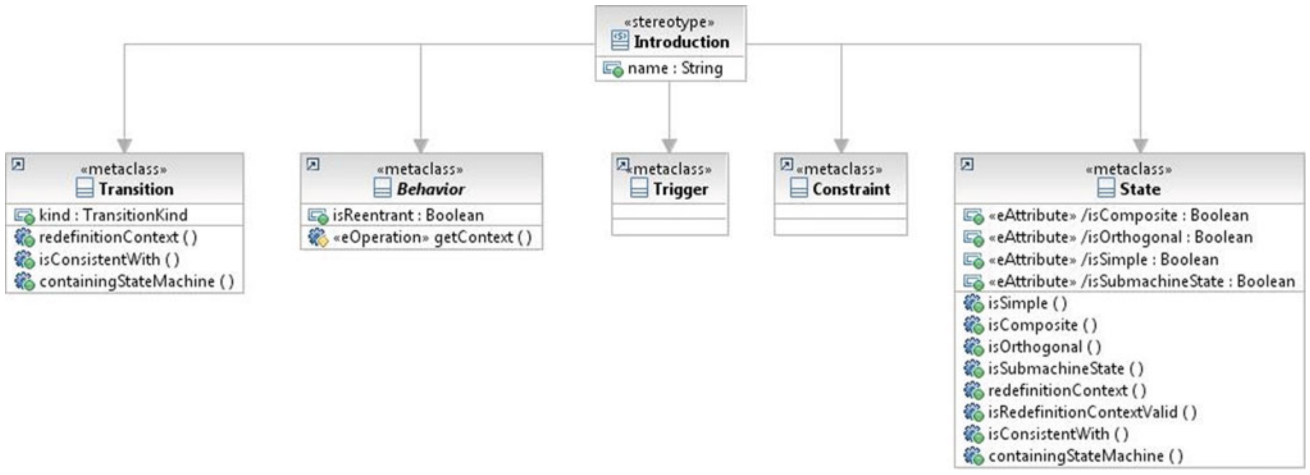


Fig. 10 The « *Introduction* » stereotype applied to various metaclasses

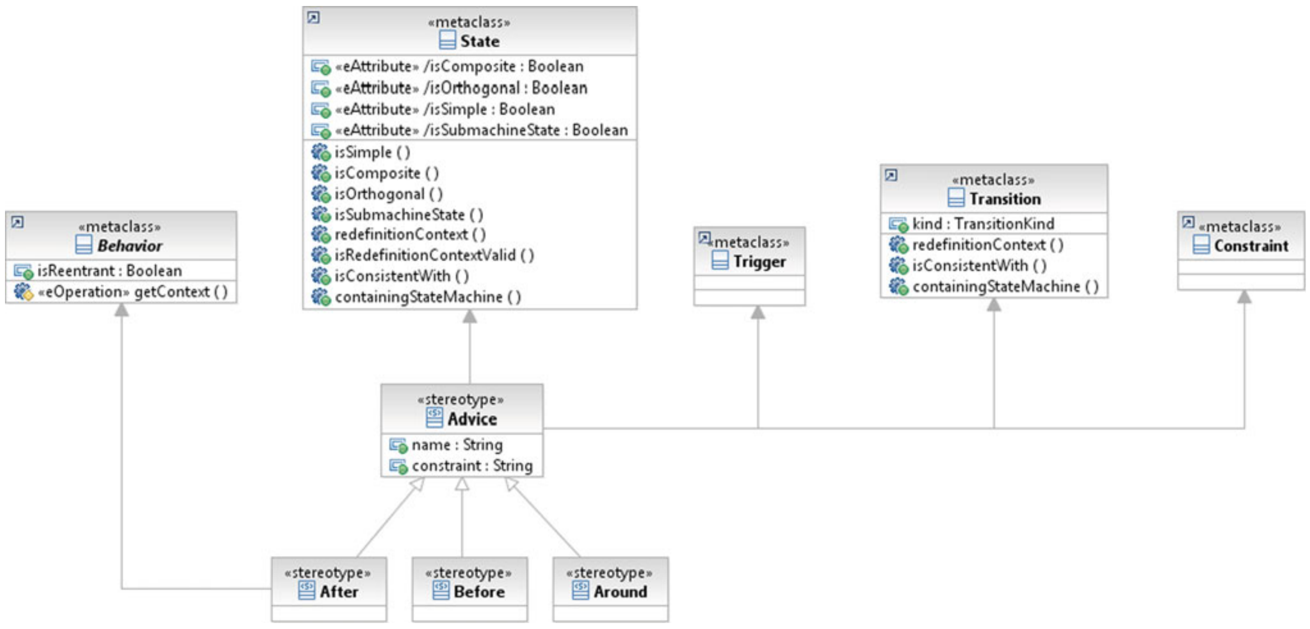


Fig. 11 The « *Advice* » stereotype applied to various metaclasses

Table 1 Definition of before, around, and after advice

State machine modeling element	Before advice	Around advice	After advice
State	Adding an OCL constraint that will be evaluated before entry to one or more states selected by a pointcut	Replacing one or more states selected by a pointcut with a new state	Adding an OCL constraint that will be evaluated on leaving one or more states selected by a pointcut
Transition	Adding a guard to one or more transitions selected by a pointcut. If a guard already exists, the additional constraint is conjuncted to the existing guard	Replacing one or more transitions selected by a pointcut with a new transition	Adding an effect with one or more actions to one or more transitions selected by a pointcut
Trigger	Not applicable	Replacing one or more triggers on transitions selected by a pointcut with new triggers	Not applicable
Effect	Adding a new behavior to the effect	Replacing one or more effects on transitions selected by a pointcut with a new effect	Same as Before advice
Guard and state invariant	Add an additional constraint (conjunct) to the guards (or state invariants) selected by a pointcut	Replacing one or more guards on transitions (or state invariants) selected by a pointcut with a new guard (or a state invariant)	Same as Before advice
Do, entry, and exit activities of a state	Adding a behavior to the activities selected by a pointcut	Replacing one or more activities in states selected by a pointcut with a new activity	Same as Before advice

Table 2 Extensions, generalizations, and associations of each stereotype

Stereotype	Extensions	Generalizations	Associations (association name[Cardinality]: Target stereotype class)
Aspect	uml::StateMachine	None	None
Pointcut	uml::State, uml::Transition, uml::Trigger, uml::Constraint, uml::Behavior	None	beforeAdvice[0..1]:Before, afterAdvice[0..1]:After, aroundAdvice[0..1]:Around, introduction[0..*]:Introduction
Advice	Same as for Pointcut	None	pointcut[1]:Pointcut
Before	Same as for Advice	Advice	Same as for Advice
After	Same as for Advice	Advice	Same as for Advice
Around	Same as for Advice	Advice	Same as for Advice
Introduction	Same as for Advice	None	pointcut[1]:Pointcut

shown in Tables 3 and 5. When applying these stereotypes, attributes must be supplied in accordance to the description in these tables. Examples are presented in [45].

4.2.3 Example of an application of AspectSM

We present next a small example of the application of AspectSM. On the *MediaQualityRecovery* aspect state machine in Fig. 12, the « *Aspect* » stereotype is described in a top-left note (labeled as "1") in the upper left part of Fig. 12. This aspect consists of one pointcut on a state: *SelectedStates*, in which attribute values are described in the note labeled as "2". The *SelectStatesPointcut* applied to the *SelectedStates* state selects all states of the base state machine (Fig. 3) except for the *Idle* state. Whenever media quality (in this case, *audioQuality*) falls below the acceptable level in any of the states selected by the *SelectStatesPointcut* pointcut, the system goes to the *RecoveryMode* state, which is stereotyped as « *Introduction* » indicating that this state will be introduced in the base state machine (Fig. 3). This is shown as a transition with the « *Introduction* » stereotypes indicating this transition will be introduced in the base state machine.

Table 3 Attributes defined for the « *Pointcut* » stereotype

Name	Type	Description
Name[1]	String	Name of the pointcut
Type[1]	SelectionType	SelectionType is an enumeration which has *All, Subset*, and *One* enumeration literals. The *All* literal means that all modeling elements of a particular type will be selected. For instance, if a pointcut of the type *All* is specified on a state in an aspect, this means that the pointcut will select all states of the base state machine. When the type of a pointcut is specified as *All*, there is no need to specify selectionConstraint. When the type of a pointcut is specified as *One*, the name of the modeling element is specified as selectionConstraint. In the case of a pointcut of type Subset, an OCL constraint is specified at the UML metamodel level to select a subset of modeling elements
SelectionConstraint	String	An OCL constraint on the UML 2.0 metamodel level to select model elements. For instance, a pointcut may select all transitions of a state machine which have triggers with signal events. (See for Fig. 8 an example)
BeforeAdvice[0..1]	String	A before advice associated with the pointcut
AfterAdvice[0..1]	String	An after advice associated with the pointcut
AroundAdvice[0..1]	String	An around advice associated with the pointcut

Table 4 Attributes defined for the « *Aspect* » stereotype

Name	Type	Description
Name[1]	String	Name of the aspect
BaseStateMachine[1..*]	uml::StateMachine	Base state machines on which an aspect is applied

Table 5 Attributes defined for the stereotypes related to advice

Name	Type	Description
Name[1]	String	Name of the advice
Constraint[0..1]	String	A constraint in OCL at the model level as a before, after, or around advice

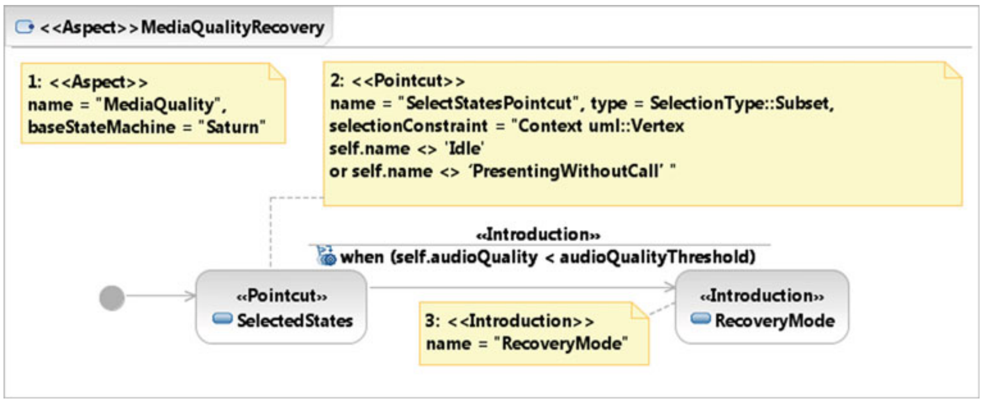

<<Aspect>>MediaQualityRecovery
1: <<Aspect>> name = "MediaQuality", baseStateMachine = "Saturn"
2: <<Pointcut>> name = "SelectStatesPointcut", type = SelectionType::Subset, selectionConstraint = "Context uml::Vertex self.name <> 'Idle' or self.name <> 'PresentingWithoutCall' "
«Introduction»
when (self.audioQuality < audioQualityThreshold)
«Pointcut» SelectedStates
3: <<Introduction>> name = "RecoveryMode"
«Introduction» RecoveryMode

Fig. 12 An example for the application of AspectSM

Table 6 Extensions and generalizations of each stereotype for FMProfile

Stereotype	Extensions	Generalizations
Fault	uml::Transition, uml::Trigger, uml::Event	None
DevelopmentFault	No Direct Extensions	Fault
OperationalFault	No Direct Extensions	Fault
InternalFault	No Direct Extensions	Fault
ExternalFault	No Direct Extensions	Fault
NaturalFault	No Direct Extensions	Fault
HumanMadeFault	No Direct Extensions	Fault
HardwareFault	No Direct Extensions	Fault
SoftwareFault	No Direct Extensions	Fault
MaliciousFault	No Direct Extensions	Fault
Non-MaliciousFault	No Direct Extensions	Fault
DeliberateFault	No Direct Extensions	Fault
NonDeliberateFault	No Direct Extensions	Fault
AccidentalFault	No Direct Extensions	Fault
IncompetenceFault	No Direct Extensions	Fault
PermanentFault	No Direct Extensions	Fault
TransientFault	No Direct Extensions	Fault
FunctionalFault	No Direct Extensions	Fault
NonFunctionalFault	No Direct Extensions	Fault
NetworkFault	No Direct Extensions	ExternalFault
SystemFault	No Direct Extensions	ExternalFault
MediaFault	No Direct Extensions	NonFunctionalFault
AudioFault	No Direct Extensions	MediaFault
VideoFault	No Direct Extensions	MediaFault

Table 7 Extensions and generalizations of each stereotype for FRProfile

Stereotype	Extensions	Generalizations
RecoveryMechanism	uml::Vertex	None
Forward	No Direct Extensions	RecoveryMechanism
Backward	No Direct Extensions	RecoveryMechanism
SystemState	uml::Vertex	None
Initial	No Direct Extensions	SystemState
Final	No Direct Extensions	SystemState
Error	No Direct Extensions	SystemState
Degraded	No Direct Extensions	SystemState
Normal	No Direct Extensions	SystemState

4.3 RobustProfile

To help with the definition of robustness test strategies, we defined a UML profile RobustProfile to model faults and their properties. In addition, the profile supports the modeling of recovery mechanisms when a fault has occurred and the modeling of states a system can transition to when it has recovered. The profile has two sub-profiles: the first sub profile, FMProfile, deals with modeling faults and their attributes. The second sub-profile, FRProfile, deals with modeling recovery mechanisms and states of a system after recovery from a failure. Below, we provide details on the definition of these sub-profiles. We reused all the concepts presented in [13] and in addition added a few more

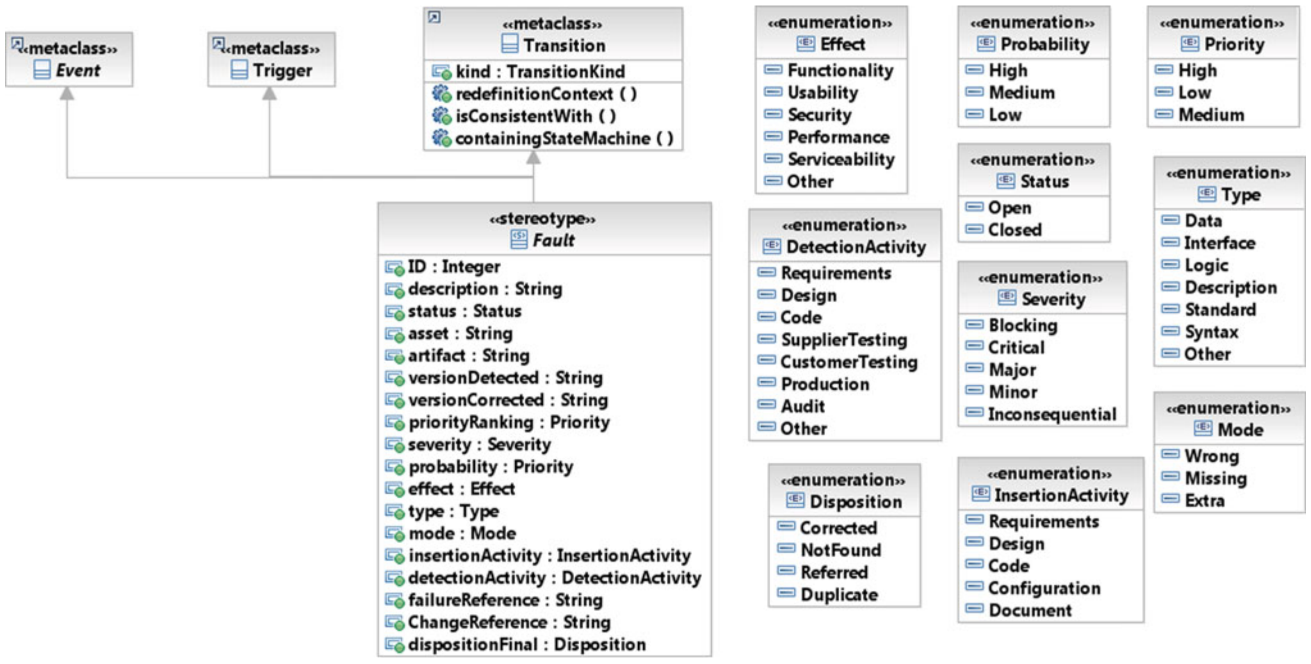


Fig. 13 Profile diagram for FMProfile

concepts presented in Sect. 4.1.2. In addition, we reused all the concepts from the IEEE standard on the classification of software anomalies as defined in [14]. All these concepts from the IEEE standard were captured in a UML profile so that the standard can be used in combination with UML models. The newly introduced concepts are italicized in Tables 6 and 7.

4.3.1 Fault Modeling Profile (FMProfile)

We used the same procedure to define FMProfile as that for AspectSM (Sect. 4.2). The domain view for FMProfile is the same as the fault taxonomy shown in Fig. 5 [13]. Below, we provide a UML representation of FMProfile, which includes profile diagrams and details on stereotypes and their attributes.

Figure 13 shows a part of the profile diagram for FMProfile that is related to the abstract « *Fault* » stereotype class, which corresponds to the *Fault* class in Fig. 5. We show different attributes of « *Fault* » and also show its extension relationships to UML metaclasses. Additional information about FMProfile is summarized in Table 6. The « *Fault* » stereotype is applied to the metaclasses *Transition, Trigger*, and *Event* because each fault in our case occurs when an event associated to trigger on a transition is fired (see Sect. 4.1). Furthermore, according to UML semantics [43], a transition can have multiple triggers, and each trigger can model different faults belonging to the same super class. For instance, a transition can model multiple external faults (*ExternalFault* in Fig. 5), and one trigger on the transition can model one fault from *NetworkFault*, while the other trigger can model one fault from *SystemFault*. This is the reason that the « *Fault* » stereotype class has an extension relationship with the *Trigger* metaclass. The attributes of « *Fault* » are obtained from the IEEE Standard in [14] where more details can be found on each attribute. Based on the values of these attributes, test strategies can be devised. For instance, the transitions that are stereotyped with « *Fault* » or any of its sub-stereotype classes with value *High* for the *severity* attribute could be given priority over other transitions modeling faults with lower severity. In addition, complex test strategies can be defined to test the robustness of an SUT in the combined presence of faults that belong to different fault classes. For example, a test strategy can be devised that can test the behavior of an SUT in the presence of one media fault and one network fault at the same time. We also defined stereotypes for all other classes shown in the taxonomy and provide detailed information about these stereotypes in Table 6. All stereotypes inherit attributes from « *Fault* ».

This profile also assists in test script generation. For instance, different stereotypes can indicate for which entity (for instance, network or other systems) in the environment, test scripts are to be generated. For example, the « *NetworkFault* » stereotype indicates that test scripts will be generated for a network emulator and the test scripts will emulate a particular fault in the emulator. The « *MediaFault* » stereotype indicates that test scripts will be

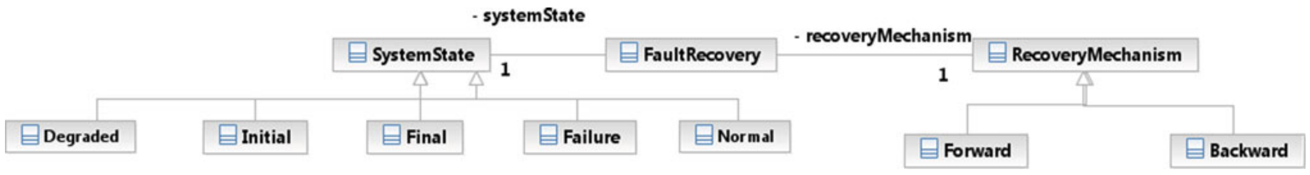


Fig. 14 Domain view of FRProfile

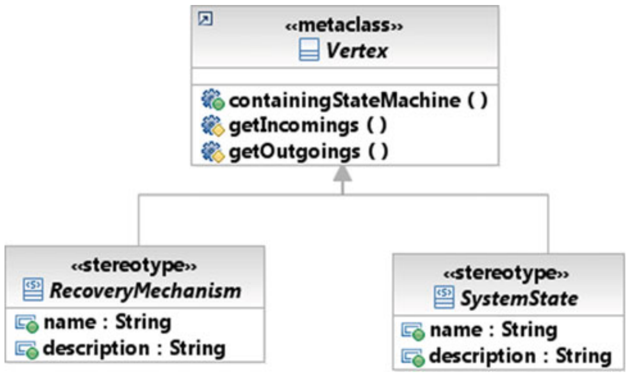


Fig. 15 Profile diagram for FRProfile

generated to introduce media faults in the VCS that communicates with the SUT. It is important to distinguish between faults for different entities in the environment because different scripting languages are normally used to control these entities. In our case study, a proprietary scripting language is used for the SUT and other VCS communicating with it, whereas Python is used to control a proprietary network emulator used by our industry partner.

4.3.2 *Fault Recovery Profile* (*FRProfile*)

FRProfile deals with modeling recovery mechanisms associated with the occurrence of a fault. The domain view of FRProfile is shown in Fig. 14. It consists of two main parts. The first part describes recovery mechanisms such as *Forward* and *Backward* [13]. The second part deals with the state of the system after a recovery mechanism is executed, which could be *Initial, Final, Failure*, or a *Degraded* state [13].

A part of the profile diagram for FRProfile is shown in Fig. 15. Both recovery mechanisms and systems states refer to states in the SUT state machines and we therefore applied stereotypes « $RecoveryMechanism$ » and « $SystemState$ » on metaclass *Vertex*. In addition, we defined stereotypes for other classes shown in the domain view of the profile such as « $Forward$ » and « $Degraded$ ». These stereotypes inherit attributes from their corresponding super classes, e.g., « $Degraded$ » inherit attributes from « $SystemState$ ». Details on stereotypes are shown in Table 7.

4.3.3 *Example of an application of RobustProfile*

This section provides a small example of the application of RobustProfile in Fig. 16. A change event $when(notself.audioQuality < audioQualityThreshold)$ is fired from *SelectedStates* (stereotyped as « $Normal$ » from RobustProfile indicating that it is a normal state) when the audio quality in a videoconference becomes lower than the allowed threshold of audio quality. This change event is stereotyped as « $AudioFault$ » indicating that it is an audio fault (see the comment labeled C1) and its attribute values are provided in the note labeled as “1”. For instance, the *effect* attribute has value *Effect::Performance* indicating that this fault affects the performance of the system. Recall that the *effect* attribute is defined based on the IEEE standard defined in [14]. The *RecoveryMode* state in Fig. 16 is stereotyped as « $Degraded$ » from RobustProfile indicating that in this state the system functions with degraded performance.

4.4 Guidelines to model properties of an environment based on the fault taxonomy

Figure 17 shows a set of guidelines to model properties of the operating environment of an SUT in a UML class diagram, violations of which lead to faults in the environment. These properties are modeled based on a fault taxonomy such as the one presented in Sect. 4.1.2. Faults related to the environment are mostly violations of non-functional properties (NFP) such as media properties and network properties. UML does not directly support modeling NFP; therefore, we used part of the MARTE profile for modeling such properties [2]. The MARTE profile is an extension for UML 2.0 that allows modeling real-time and embedded systems. MARTE provides a generic framework to model NFP on UML models. Moreover, MARTE provides a model library that provides NFP data types for defining various NFP properties and specific applications. MARTE also provides mechanisms to extend the model library to either extend the existing NFP data types or define entirely new NFP types.

Now, we present an example to use the above guidelines (Fig. 17) to model a class diagram, which captures the properties of the environment. Figure 18 shows a partial class diagram of the *MediaQualityRecovery* robustness behavior (Sect. 2.2). For this robustness behavior, we identify that

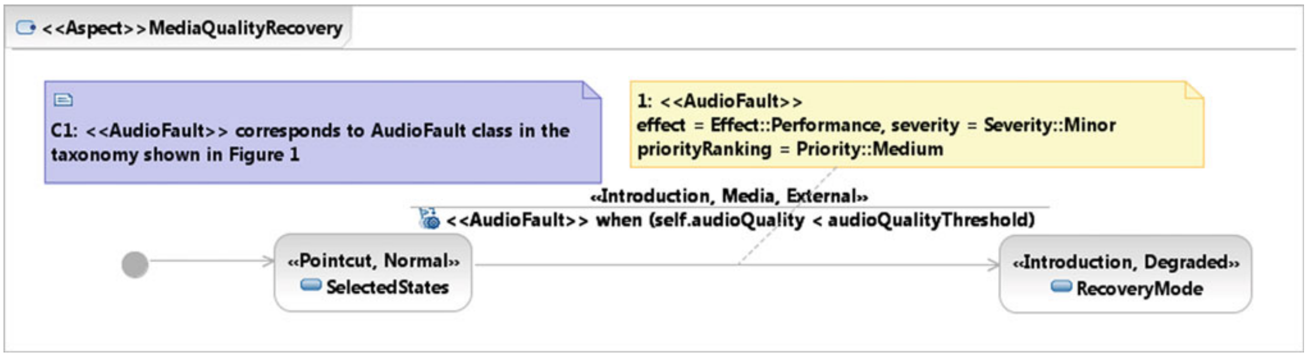

<<Aspect>>MediaQualityRecovery
C1: <<AudioFault>> corresponds to AudioFault class in the taxonomy shown in Figure 1
1: <<AudioFault>>
effect = Effect::Performance, severity = Severity::Minor
priorityRanking = Priority::Medium
«Introduction, Media, External»
<<AudioFault>> when (self.audioQuality < audioQualityThreshold)
«Pointcut, Normal»
SelectedStates
«Introduction, Degraded»
RecoveryMode

Fig. 16 Application of RobustProfile

1. For each fault class indentified in the taxonomy, model one or more faults belonging to the class.
2. For each fault of a fault class, define an attribute in the aspect class representing the property whose violation leads to the particular fault. The type of the property can be defined as:
   a. Using UML standard primitive data types such as *Integer*, *Boolean*, etc.
   b. Using the NFP_Types defined by MARTE such as NFP_Percentage
   c. Defining a new NFP_Type using the MARTE's extensibility mechanism to define new NFPs.

Fig. 17 Guidelines to model faults in aspect class diagram

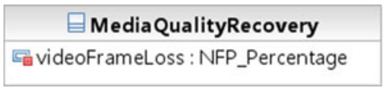


Fig. 18 An example of modeling a property of environment

the *Video* fault class from the fault taxonomy (Fig. 5) is relevant. For this fault class, video frame loss in incoming video streams to a VCS is important for robustness testing of the VCS. To model video frame loss, we model a property named *videoFrameLoss* in the *MediaQualityRecovery* class shown in Fig. 18. The *videoFrameLoss* property is modeled as *NFP_Percentage* defined in MARTE. The property holds the percentage of video frame loss in incoming video streams to the VCS.

4.5 Aspect state machine

An aspect state machine is a standard UML state machine with stereotypes from the AspectSM profile. The complete definition of an aspect state machine follows the template shown in Fig. 19.

4.6 Template for modeling weaving-directive state machine

In this paper, a robustness behavior, such as the behavior of an SUT in the presence of network faults or faults in incoming media streams to the SUT, is modeled using one or more related aspects. Each of these aspects is modeled as a separate aspect state machine. Aspect state machines should be woven into a base state machine in a specific order to ensure that the woven state machine is complete and correct. To achieve this, an ordering must be defined by a modeler/tester who instructs the weaver about the ordering of aspect state machines. This is modeled as a state machine (denoted weaving-directive state machine), containing all aspect state machines as submachine states ordered using UML state machine's control structure features such as decision, join, and fork. If the ordering does not matter, then a modeler/tester is free to specify any order. The template for the complete definition of a weaving-directive state machine is shown in Fig. 20.

4.7 Weaver

The aspect state machines are woven into the base state machine by a weaver, which reads the base state machine, aspect state machines, and a weaving-directive state machine, and produces a woven state machine. The weaving algorithm is shown in Fig. 31 in Appendix B and is based on the same weaving approach advocated in [25]. We developed a weaver for AspectSM by using Kermeta [15], which is a metamodeling language [15] that allows manipulating models by defining transformation rules at the metamodel level. We do not implement any explicit model validation, but rely on Kermeta's model validation, which partially prevents violations of UML semantics. Kermeta conforms to OMG's metamodeling language Essential Meta Object Facility (EMOF) and Ecore [38]. Figure 21 shows the architecture of the weaver by using transformations in Kermeta to weave one or more aspect state machines into a base state machine. The AspectSM profile is defined on the UML 2.0

An aspect state machine *A* is a UML 2.0 state machine stereotyped as <<*Aspect*>> consisting of the following UML 2.0 state machine elements:

1. *I*: An initial state
2. *F*: A set of one or more final states
3. *S*: A set of states, each of one of the following types
    a. A state *s* in *S* can be a new state to be introduced in the base model (stereotyped as <<*Introduction*>>)
    b. A state *s* in *S* can be a pointcut selecting *one*, *a subset*, or *all* states of a base state machine (stereotyped as <<*Pointcut*>>)
    c. A state *s* in *S* without any stereotype can be a state that has one or more new elements introduced (stereotyped <<*Introduction*>>) or as pointcuts (stereotyped as <<*Pointcut*>>) of the type state invariant, do, entry, or exit activity
4. *T*: A set of transitions connecting states in the set *S*, each transition of one of the following types
    a. A transition from an initial state to any type of state described in item 3, which doesn't have any trigger, guard, or effect
    b. A set of transitions from any state (except from the initial state) to the final state
    c. A transition *t* in *T* can be a new transition to be introduced in the base model (stereotyped as <<*Introduction*>>). This type of transition can exist on the following pairs of stereotyped states:
        i. Between a state stereotyped as <<*Introduction*>> and a state stereotyped as <<*Pointcut*>>
        ii. Between two states stereotyped as <<*Introduction*>>
        iii. Between two states stereotyped as <<*Pointcut*>>
    d. A transition *t* in *T* is a pointcut selecting *one*, a *subset*, or *all* transitions of a base state machine (stereotyped as <<*Pointcut*>>). This transition can exist on the following pairs of states:
        i. Between a state stereotyped as <<*Introduction*>> and a state stereotyped as <<*Pointcut*>>
        ii. Between two states stereotyped as <<*Introduction*>>
        iii. Between two states stereotyped as <<*Pointcut*>>
    e. A transition *t* in *T* can be the transition without any stereotype that has any contained element such as a guard, a set of triggers, and an effect as a new element introduced (stereotyped as <<*Introduction*>>) or as a pointcut stereotyped as <<*Pointcut*>>. This transition can only exist between a pair of states stereotyped as <<*Pointcut*>>

Fig. 19 Definition of an aspect state machine

A weaving directive state machine *W* is a UML 2.0 state machine having the following modeling elements:

1. An initial state *I*
2. A set of final states *F*
3. A set of submachine states *S*, where each submachine state refers to an aspect state machine
4. A set of transitions *T* that can be of any of the following types:
    a. A transition from an initial state to a submachine state, which doesn't have any trigger, guard, or effect, but can have a name.
    b. A set of transitions from submachine states (except from the initial state) to the final state.
    c. A set of transitions *T* connecting submachine states *S* using UML 2.0 state machine's features such as decision, join, and fork to show the order in which the submachine states (aspects) will be woven into the base state machine. For instance, in a very simple scenario, if there is an outgoing transition from submachine state *S* to *S'*, then *S* will be woven before *S'*.

Fig. 20 Definition of a weaving directive state machine

metamodel. An aspect state machine is defined as a UML 2.0 state machine by applying the AspectSM profile. A base state machine is a standard UML 2.0 state machine. Transformations rules in Kermeta are defined on the UML 2.0 metamodel and the AspectSM profile. Finally, the Kermeta engine uses the transformation rules that read an aspect state machine and the base state machine and weaves the aspect state machine into the base state machine. The Kermeta engine then produces a woven state machine, which is again an instance of the UML 2.0 metamodel, since the woven state machine is a standard UML 2.0 state machine. The woven state machines can then be used as input for automated model-based testing tools such as Conformiq Qtronic [17] and Smartesting Test Designer [49]. The weaver is fully automated and does not require any additional inputs from the user apart from aspect state machines and a base state machine.

The weaver is developed to support automated, model-based robustness testing, and thus aspect state machines are woven into the base state machine, which can be used for test case generation. Currently, our approach and its weaver do not support modeling and weaving interactions [6] that may occur between different aspects and may lead to conflicts between aspects during weaving. On the other hand, our weaver does support to a limited extent the handling of aspect conflicts. In [50], four classes of aspect conflicts are discussed: conflicts due to crosscutting specification,

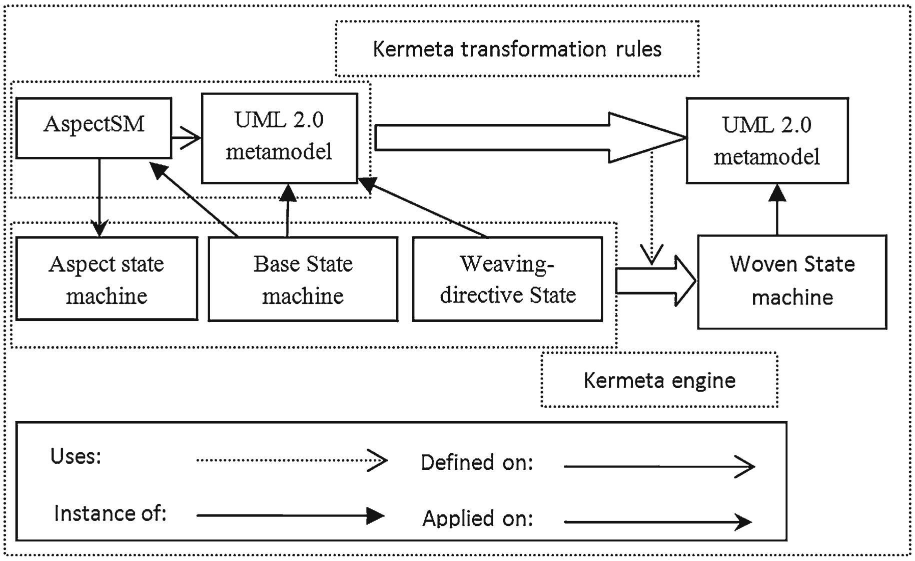


Fig. 21 Aspect weaver implemented in Kermeta

aspect–aspect conflicts, aspect–base conflicts, and concern–concern conflicts. In our application context, i.e., robustness modeling and testing, the most relevant conflicts are aspect–aspect conflicts, which are related to handling conflicts between aspects. One of the most important aspect–aspect conflicts is the ordering conflict, which is related to the order in which aspect state machines should be woven into a base state machine. Ordering conflict is most relevant in our context since, for testing purposes, we focus on modeling, weaving, and testing one or more related aspects at a time. We specify the ordering between aspect state machines in a UML state machine containing all aspect state machines as submachine states, ordered using state machine control structure features: decision, join, and fork.

The algorithm implemented in the weaver is presented in Appendix B. For the current application, we do not foresee the need to define other interactions/conflicts; however, in the future we plan to apply RUMM to other case studies and as required we will further improve the process. For testing purposes, one first has to focus on testing one concern at a time and may eventually at a later stage test several concerns together. For robustness testing, at this stage of the work, we weave faulty behavior of the environment (e.g., network) one concern at a time, as the goal is to test robustness behavior one concern at a time in order to facilitate debugging.

5 Application of RUMM to our simplified industrial case study

In this section, we illustrate the different activities in RUMM using the simplified version of our industrial case study (S-Saturn).

5.1 Activity A1: develop a conceptual model of a system

This activity involves developing a conceptual model [20] of a system using UML 2.0 class diagram based on the domain analysis of the system. As we discussed in Sect. 2, the *Saturn* subsystem deals with establishing videoconferencing calls, disconnecting calls, and starting/stopping presentation. In Sect. 2, Fig. 2 shows what we refer to as a *'conceptual model'* for the system being modeled, which is here S-Saturn.

5.2 Activity A2: develop a behavioral model of the system as UML state machines

This activity models the nominal system behavior using UML 2.0 state machines, as illustrated for S-Saturn in Fig. 3, Sect. 2. This behavioral model is referred to as the '*base state machine*' since all aspect state machines are woven into this state machine.

5.3 Activity A3: identify relevant faults from fault taxonomy

A VCS should be robust against possible faults arising in its environment, which includes users, the network, and other videoconferencing systems. A user interacts with the VCS and sends different commands such as starting a videoconferencing, stopping a videoconference, and starting a presentation. All the interactions of the VCS with other VCSs take place through the network. Therefore, the VCS should be robust against faults in the network and other VCSs communicating with it.

In our case study, we modeled *Media* faults in the VCSs communicating with the SUT, which are the ones that are related to quality of media such as audio, video, and their synchronization. From Fig. 5, we see subclasses of *Media*

Table 8 Media faults and their description

Fault class	Fault instance	Fault description
Audio Fault	No audio	This fault removes audio from a videoconference
	Loss of audio frames	This fault introduces loss in audio frames
	Low audio quality	This fault reduces audio quality in a videoconference
	Noise in audio	This fault introduces noise in audio during a videoconference
	Echo in audio	This fault introduces echo in audio
	Mixing of multiple audio	This fault mixes multiple audio during a videoconference
Video Fault	No video	This fault removes video from a videoconference
	Loss in video frames	This fault introduces loss in video frames
	Low video quality	This fault reduces video quality in a videoconference
Media Fault	Synchronization mismatch between audio and video	This fault loses synchronization between audio and video in a videoconference

Table 9 Network faults and their description

Fault	Description of the fault
Packet Loss	This fault introduces network packet loss during a videoconference
Jitter	This fault introduces delays in the packet during a videoconference
Illegal H323 packet	This fault introduces illegal/malformed H323 packets in a H323 videoconference
Illegal SIP packet	This fault introduces illegal/malformed SIP packets in a SIP videoconference
No network connection	This fault shut downs the network
Low bandwidth	This fault reduces the bandwidth of the network to less than the bandwidth required by a videoconference

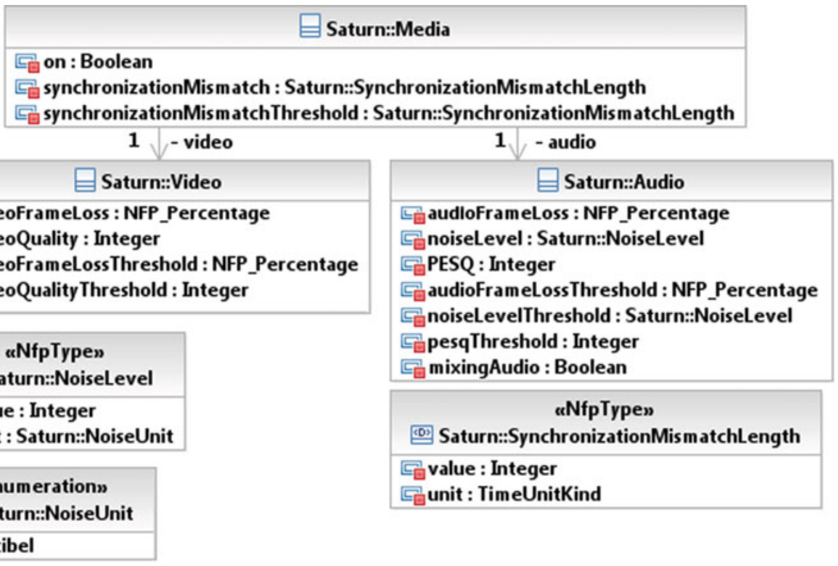

Saturn::Media
on : Boolean
synchronizationMismatch : Saturn::SynchronizationMismatchLength
synchronizationMismatchThreshold : Saturn::SynchronizationMismatchLength
1
- video
1
- audio
Saturn::Video
videoFrameLoss : NFP_Percentage
videoQuality : Integer
videoFrameLossThreshold : NFP_Percentage
videoQualityThreshold : Integer
Saturn::Audio
audioFrameLoss : NFP_Percentage
noiseLevel : Saturn::NoiseLevel
PESQ : Integer
audioFrameLossThreshold : NFP_Percentage
noiseLevelThreshold : Saturn::NoiseLevel
pesqThreshold : Integer
mixingAudio : Boolean
«NfpType»
Saturn::NoiseLevel
value : Integer
unit : Saturn::NoiseUnit
«enumeration»
Saturn::NoiseUnit
Decibel
«NfpType»
Saturn::SynchronizationMismatchLength
value : Integer
unit : TimeUnitKind

Fig. 22 Class diagram for media quality attributes

faults which are *Audio* Faults and *Video* Faults. Table 8 provides a description of *Media* faults that are relevant to our case study.

In addition, network faults (*NetworkFault*, see Fig. 5) are important for a VCS. Several types of faulty situations can happen in the network that must be dealt by the VCS. We show network faults that are relevant to our case study in Table 9.

5.4 Activity A4: develop a class diagram for a robustness aspect

As advocated by the aspect-oriented paradigm, crosscutting concerns (functional or non-functional) [29] must be modeled as aspects. Activities A3 and A4 model aspects of the robustness behavior of the system using aspect state machines and aspect class diagrams. To do so, we use the AspectSM profile using the existing UML state machine notation, as presented in Sect. 4.2.

As an example, we demonstrate how to model two representative crosscutting behaviors on S-Saturn. The first one models the behavior that checks the quality of media (audio and video) during a videoconference and in case the quality falls below a threshold value, specific procedures try to recover an acceptable quality. This is achieved by modeling three aspects: (1) First aspect updates state invariants of all states with audio quality attributes. (2) The second aspect updates state invariants of all states with video quality attributes. (3) The third aspect models the behavior that checks the quality of media (audio and video) during a videoconference and, in case the quality falls below the threshold value, triggers the above-mentioned recovery procedures (*MediaRecoveryAspect*). Such behavior is redundant in various states and hence is a crosscutting behavior. The second crosscutting behavior example factors out constraints on input parameters of a call event as an aspect, which are also scattered across many transitions in the base state machine. Details about the modeling of these two aspects are presented in Appendix A.

Each aspect state machine has an associated class diagram (aspect class diagram), which is an augmentation of the conceptual model of the *Saturn* subsystem shown in Fig. 2. This class diagram models the information about different kinds of faults in the fault taxonomy, such as audio and video related faults. Guidelines for such modeling based on a fault taxonomy (Sect. 4.1.2) are presented in Sect. 4.4. The *Audio* class defines audio quality attributes based on which different audio faults can be introduced, as shown in Fig. 22. For instance, the *on* attribute is a *Boolean* attribute that determines if the audio is present in a videoconference. The Perceptual Evaluation of Speech Quality (PESQ) [51] is a metric for measuring audio quality. The *audioFrameLoss* is an attribute that determines the current percentage of audio frames loss during a videoconference and is defined as the MARTE type *NFP_Percentage*. The *noiseLevel* attribute is defined as the *Nfp type NoiseLevel* (modeled with « NfpType » from MARTE), which has two attributes: *value* that holds current noise value, and *unit* contains a unit to measure audio noise such as “decibel”.

Similarly, the following video quality properties are defined in the class diagram: The *on* attribute determines if the video is present in a videoconference. The *videoQuality* attribute is a metric for measuring video quality and *videoFrameLoss* determines the current video frame loss during a videoconference modeled as MARTE’s *NFP_Percentage*.

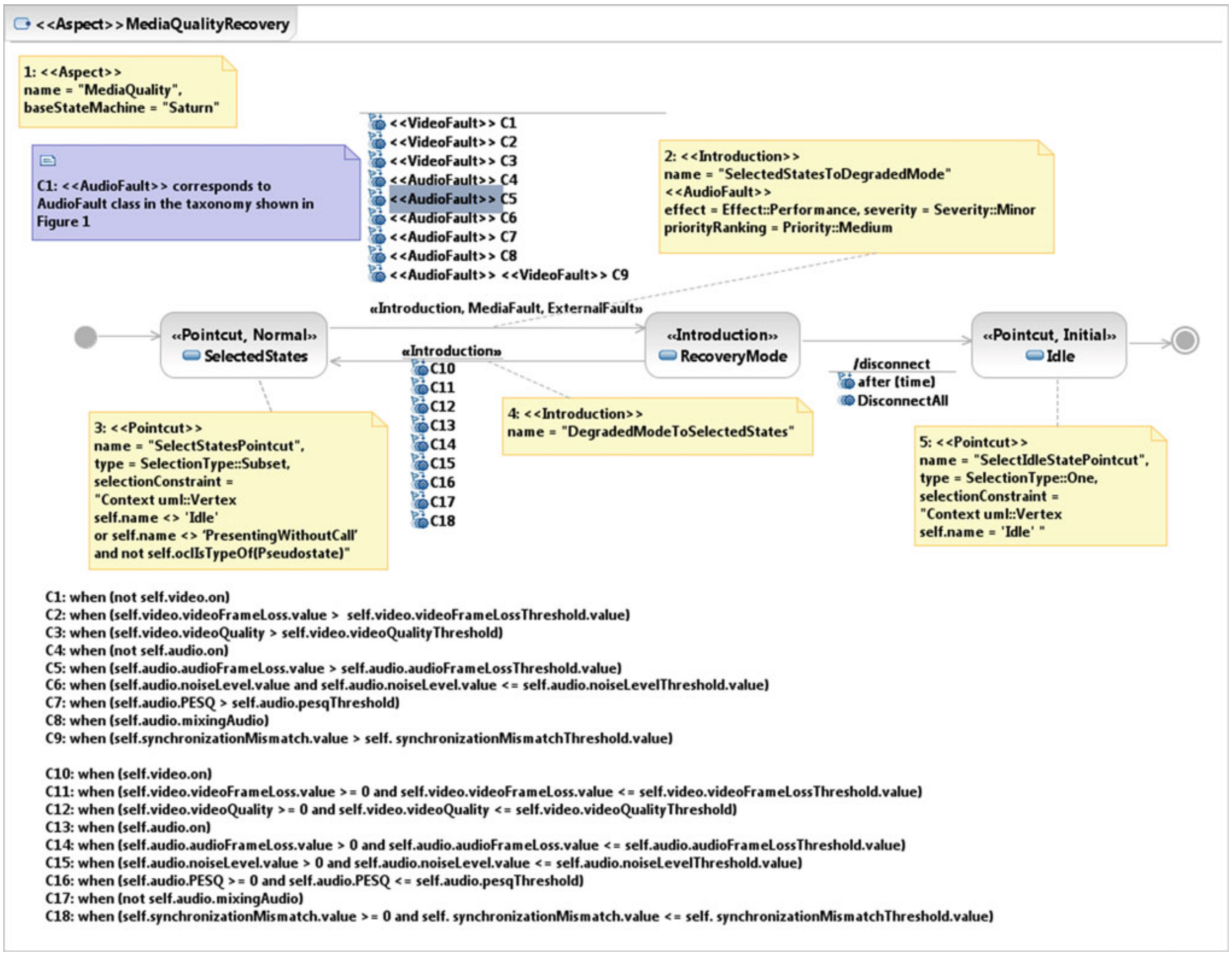
5.5 Activity A5: develop a state machine for the robustness aspect

5.5.1 Modeling recovery from media faults

Recall that each robustness aspect is modeled as a UML state machine with stereotypes from AspectSM (aspect state machine). Figure 23 shows the details of the *MediaQualityRecovery* aspect state machine. Attribute values of the various stereotypes are presented in Fig. 23 in notes. The aspect state machine models the robust behavior of a VCS in the case when media quality falls below the acceptable level and tries to return to an acceptable media quality level. In the worst case, the VCS cleans up system resources and goes back to the most recent safe state (e.g., *Idle* in our industrial case study), in which the VCS was exhibiting normal behavior. Such a robust behavior is very important in a commercial VCS, as quality expectations are high regarding robustness to media quality faults.

On the *MediaQualityRecovery* aspect state machine, the « *Aspect* » stereotype is described in the top-left note (labeled “1”) in the upper left part of Fig. 23. This aspect state machine consists of two pointcuts on states: *SelectedStates* and *Idle*, the attribute values of which are described in notes explicitly linked to each « *Pointcut* » note. Representing pointcuts as modeling elements of UML state machines (for instance, state in this case) enables the modeling of aspect state machines using standard UML notation, while keeping in line with UML semantics. The *SelectStatesPointcut* (see note 3 for attribute values) applied to the *SelectedStates* state selects all states of the base state machine (Fig. 3) except for the *Idle* state. The *SelectIdleState* pointcut (see note 5 for attribute values) on the *Idle* state selects the *Idle* state of the base state machine (Fig. 3). Whenever media quality (defined based on the quality attributes in Fig. 22) falls below the acceptable level in any of the states selected by the *SelectStatesPointcut* pointcut, the system goes to the

RecoveryMode state. This is shown as a transition with the «*Introduction*», «*MediaFault*», and «*ExternalFault*» stereotypes (indicating that this transition will be introduced in the base state machine and models media faults which are external to S-Saturn) from the *SelectedStates* state to the *RecoveryMode* state with nine change events. Each change event is defined based on one media quality attribute and determines if this attribute falls below the acceptable level and is stereotyped as either «*AudioFault*», «*VideoFault*»,



<<Aspect>>MediaQualityRecovery
1: <<Aspect>> name = "MediaQuality", baseStateMachine = "Saturn"
C1: <<AudioFault>> corresponds to AudioFault class in the taxonomy shown in Figure 1
<<VideoFault>> C1
<<VideoFault>> C2
<<VideoFault>> C3
<<AudioFault>> C4
<<AudioFault>> C5
<<AudioFault>> C6
<<AudioFault>> C7
<<AudioFault>> C8
<<AudioFault>> <<VideoFault>> C9
2: <<Introduction>> name = "SelectedStatesToDegradedMode" <<AudioFault>> effect = Effect::Performance, severity = Severity::Minor priorityRanking = Priority::Medium
«Introduction, MediaFault, ExternalFault»
«Pointcut, Normal» SelectedStates
«Introduction» RecoveryMode
«Pointcut, Initial» Idle
«Introduction» C10 C11 C12 C13 C14 C15 C16 C17 C18
/disconnect after (time) DisconnectAll
4: <<Introduction>> name = "DegradedModeToSelectedStates"
3: <<Pointcut>> name = "SelectStatesPointcut", type = SelectionType::Subset, selectionConstraint = "Context uml::Vertex self.name <> 'Idle' or self.name <> 'PresentingWithoutCall' and not self.oclIsTypeOf(Pseudostate)"
5: <<Pointcut>> name = "SelectIdleStatePointcut", type = SelectionType::One, selectionConstraint = "Context uml::Vertex self.name = 'Idle' "

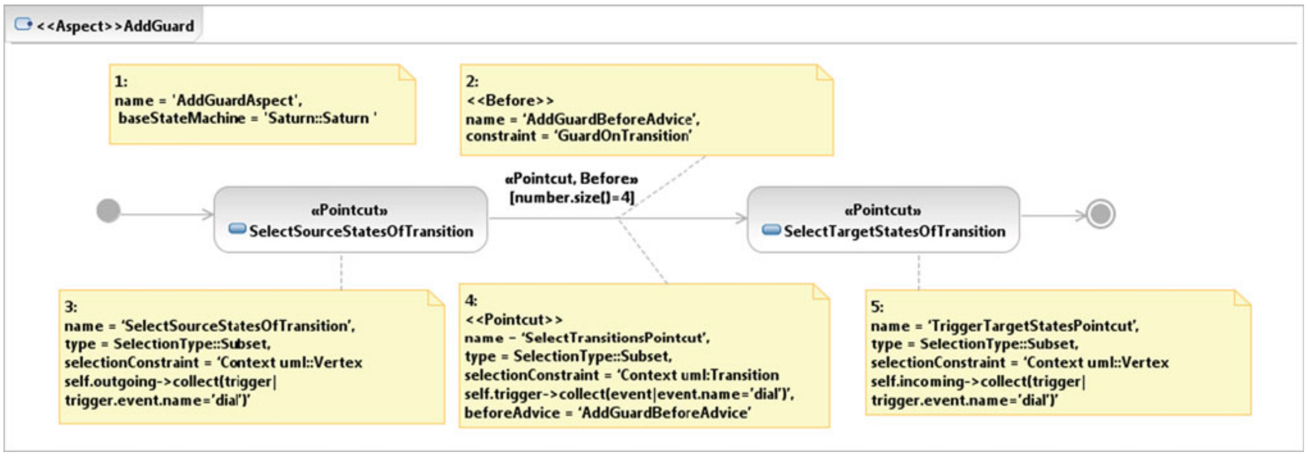
```
C1: when (not self.video.on)
C2: when (self.video.videoFrameLoss.value > self.video.videoFrameLossThreshold.value)
C3: when (self.video.videoQuality > self.video.videoQualityThreshold)
C4: when (not self.audio.on)
C5: when (self.audio.audioFrameLoss.value > self.audio.audioFrameLossThreshold.value)
C6: when (self.audio.noiseLevel.value and self.audio.noiseLevel.value <= self.audio.noiseLevelThreshold.value)
C7: when (self.audio.PESQ > self.audio.pesqThreshold)
C8: when (self.audio.mixingAudio)
C9: when (self.synchronizationMismatch.value > self. synchronizationMismatchThreshold.value)

C10: when (self.video.on)
C11: when (self.video.videoFrameLoss.value >= 0 and self.video.videoFrameLoss.value <= self.video.videoFrameLossThreshold.value)
C12: when (self.video.videoQuality >= 0 and self.video.videoQuality <= self.video.videoQualityThreshold)
C13: when (self.audio.on)
C14: when (self.audio.audioFrameLoss.value > 0 and self.audio.audioFrameLoss.value <= self.audio.audioFrameLossThreshold.value)
C15: when (self.audio.noiseLevel.value > 0 and self.audio.noiseLevel.value <= self.audio.noiseLevelThreshold.value)
C16: when (self.audio.PESQ >= 0 and self.audio.PESQ <= self.audio.pesqThreshold)
C17: when (not self.audio.mixingAudio)
C18: when (self.synchronizationMismatch.value >= 0 and self. synchronizationMismatch.value <= self. synchronizationMismatchThreshold.value)
```

Fig. 23 The *MediaQualityRecovery* aspect

Context Saturn::Media
not self.video.on
or self.video.videoFrameLoss.value > self.video.videoFrameLossThreshold.value or self.video.videoQuality > self.video.videoQualityThreshold
or not self.audio.on
or self.audio.audioFrameLoss.value > self.audio.audioFrameLossThreshold.value or self.audio.noiseLevel.value and self.audio.noise Level.value <= self.audio.noiseLevelThreshold.value
or self.audio.PESQ > self.audio.pesqThreshold or self.audio.mixingAudio or self.synchronizationMismatch.value > self.synchronizationMismatchThreshold.value

Fig. 24 State invariant for RecoveryMode

or both. For example, the change event *when(not self.audio.on)* is fired from *SelectedStates* when the audio is turned off in a videoconference and is stereotyped as « *AudioFault* » indicating that it is an audio fault (see the comment labeled C1 and note "2" for attribute values—recall that these attributes are defined based on IEEE standard classification for anomalies [14]). If the system manages to return to acceptable media quality, it goes back to the normal state shown as a transition introduced from the *RecoveryMode* state to the *Selected-States* state stereotyped as « *Normal* » (indicating that these states are normal states of S-Saturn) with again nine change events. For example, the change event *when*(*self.audio.on*) is fired from the *RecoveryMode* state when the audio is back in the videoconference. The state invariant of the *Recovery-Mode* state ensures that S-Saturn remains in *RecoveryMode* as long as any of the faults in the environment exist. This state invariant is simply the logical disjunction of all change events modeling the faults (Fig. 24). In the other case, if the system cannot recover within time *time*, it disconnects all connected VCS and goes to the *Idle* state. This is modeled



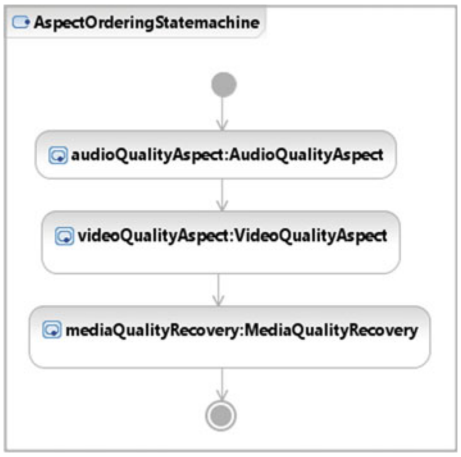
<<Aspect>>AddGuard
1:
name = 'AddGuardAspect',
baseStateMachine = 'Saturn::Saturn '
2:
<<Before>>
name = 'AddGuardBeforeAdvice',
constraint = 'GuardOnTransition'
«Pointcut, Before»
[number.size()=4]
«Pointcut»
SelectSourceStatesOfTransition
«Pointcut»
SelectTargetStatesOfTransition
3:
name = 'SelectSourceStatesOfTransition',
type = SelectionType::Subset,
selectionConstraint = 'Context uml::Vertex
self.outgoing->collect(trigger|
trigger.event.name='dial')'
4:
<<Pointcut>>
name = 'SelectTransitionsPointcut',
type = SelectionType::Subset,
selectionConstraint = 'Context uml:Transition
self.trigger->collect(event|event.name='dial')',
beforeAdvice = 'AddGuardBeforeAdvice'
5:
name = 'TriggerTargetStatesPointcut',
type = SelectionType::Subset,
selectionConstraint = 'Context uml::Vertex
self.incoming->collect(trigger|
trigger.event.name='dial')'

Fig. 25 State machine for the *AddGuard* aspect

as a transition introduced between the *RecoveryMode* state and the *Idle* state with a time event and an effect *DisconnectAll* with an opaque behavior, which is a type of behavior defined in UML to specify implementation-specific semantics. In addition, the *Idle* state is stereotyped as « *Initial* », which indicates the state of S-Saturn if it is not successful in recovering to an acceptable level of media quality. In our context, *DisconnectAll* is a call to Saturn's API in a python-based proprietary test script language. This call disconnects all connected systems to a VCS.

5.5.2 Constraining input parameter values

The second crosscutting behavior example we present is constraining parameters of events on transitions. Since many transitions in a state machine can have the same trigger and constraints on the associated event of the trigger may be the same, redundant constraints can exist in the model and hence can be factored out as an aspect. Such constraints can be used to generate test cases exercising the system robustness with illegal inputs [52]. The aspect state machine *AddGuard* shown in Fig. 25 models this crosscutting behavior. The associated class diagram for the aspect state machine is identical to Fig. 2 as we do not need to model additional properties. This aspect state machine defines two pointcuts (*SelectSourceStatesOfTransition, SelectTargetStatesOfTransition*) on two states and one pointcut *SelectTransitionsPointcut* on the transition between the two states stereotyped as « *Pointcut* ». This aspect state machine selects all transitions which have a *dial* call event and applies a before advice *AddGuardBeforeAdvice* that adds an additional constraint *"number.size()* = 4*"* to the existing guards on the selected transitions. This constraint ensures that the number parameter of the *dial* call event has exactly four digits.



AspectOrderingStatemachine
audioQualityAspect:AudioQualityAspect
videoQualityAspect:VideoQualityAspect
mediaQualityRecovery:MediaQualityRecovery

Fig. 26 A state machine describing ordering of aspects for weaving

5.6 Activity A6: define ordering of aspects using a state machine

We begin with testing a related set of aspects modeling on robustness behavior. The related set of aspects is woven into a base model in a specific order to ensure that the woven model is complete and correct. To achieve this, an ordering must be defined between the aspect state machines (activity A5). This ordering is also modeled as a state machine (denoted as weaving-directive state machine), containing all aspect state machines as submachine states ordered using UML state machine's control structure features such as decision, join, and fork. The complete template for the definition of a weaving directive state machine is shown in Sect. 4.6.

The weaving directive state machine for *MediaQualityRecovery* is shown in Fig. 26. Using such state machine, we define the ordering of aspect state machines related to media quality. By weaving the aspect state machines in this order, the woven state machine will be correct for testing. The reason is that *MediaQualityAspect* introduces the *DegradedMode* state in the base state machine and the first two aspect state machines update audio and video quality constraints in state invariants of all states of the base state machine. These constraints should not be updated in *DegradedMode* because in this state the system works with degraded performance and audio and video quality will not be as expected. If *MediaQualityAspect* is woven before *AudioQualityAspect* and *VideoQualityAspect*, the woven state machine will contain *DegradedMode* with wrong state invariants. In this paper, we aim to weave and test a set of related aspects (e.g., related to media quality) but not all aspects altogether. In the future, we will investigate how to test by weaving different aspects at the same time.

5.7 Activity A7: weave aspects with behavioral models

Finally, the aspect state machines are woven into the base state machine by the weaver, which reads the base state machine, aspect state machine(s), and a weaving-directive state machine and produces a woven state machine.

5.7.1 Modeling recovery from media faults

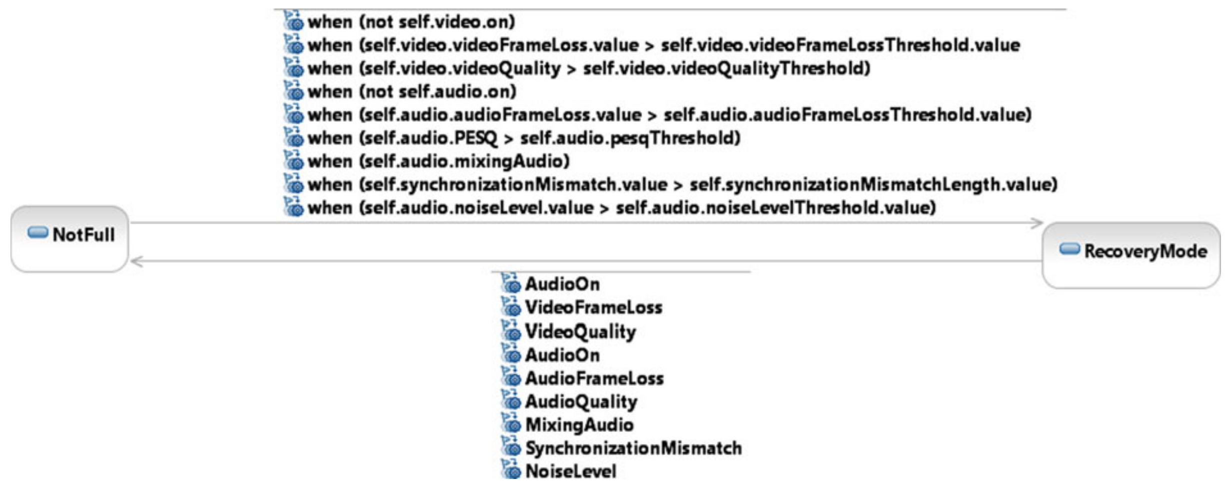
The woven state machine resulting from applying *MediaRecoveryAspect* to the Saturn base state machine is not easily comprehensible, but it is only meant to be processed by model-based testing tools. An excerpt of the woven state machine is however shown in Fig. 27, and details regarding the model complexity of woven state machines are summarized in Table 11. From all states except *Idle* and *PresentingWithoutCall*, transitions to *RecoveryMode* are added. Each of these transitions contains nine change events that can lead to the *RecoveryMode* state, such as the woven state machine in Fig. 27, which contains a new state *RecoveryMode*. From *NotFull*, a transition is added that contains nine change events that can lead to the *RecoveryMode* state such as change events “$self.video.videoFrameLoss.value > videoFrameLossThreshold.value''$ and *“not (self.audio. on)”*. The first change event is triggered when, during a videoconference, video frame loss becomes greater than the allowed frame loss (*videoFrameThreshold*), whereas the second change event is triggered when audio disappears from a videoconference. These change events are defined in the context of the conceptual class diagrams shown in Fig. 2 and the class diagram modeling media quality attributes in Fig. 22. Recall from Sect. 5.4 that both class diagrams are defined in the same package: *Saturn*. After weaving, the class diagram in Fig. 22 is merged into the conceptual class diagram in Fig. 2. Therefore, after weaving, the attributes defined in Fig. 23 have the same context: the *“Saturn”* class in Fig. 2. Similarly, six transitions from *RecoveryMode* to all states except *Idle* and *PresentingWithoutCall* have been woven into the base state machine. Each transition has nine change events that can lead the system back to the state it was in before *RecoveryMode*, e.g., in Fig. 27, a transition with six change events is added that can lead the system back to the *NotFull* state. For instance, the *VideoFrameLoss* change event in Fig. 27 specifies that when video frame loss is within the allowed frame loss and the system was in the *NotFull* state, a VCS transitions from *RecoveryMode* to *NotFull.* The change event has two parts: the first part *(self.video.videoFrameLoss.value* >= 0 *and self.video.videoFrameLoss.value* <= *videoFrameLossThreshold.value)* checks if *videoFrameLoss* is within the allowed threshold. The second part is the state invariant of the *NotFull* state, which checks that the active calls in a videoconference are more than one (*self.systemUnit.NumberOfActiveCalls > 1 and self.systemUnit.NumberOfActiveCalls< self.systemUnit.MaximumNumberOfCalls*) and S-Saturn does not send a presentation (*self.conference.PresentationMode = ’off’*). In addition, it checks that S-Saturn is not sending or receiving a presentation (*self.conference.calls->select(c:Call| c.outgoingPresentationChannel->asSequence*()*->last*()*.Protocol = VideoProtocol::off)->size*() *= 0 and self.conference.calls->select(c:Call |c.incomingPresentationChannel->asSequence* ()*->last*()*.Protocol* <> *VideoProtocol::off)->size*() = *0*).

5.7.2 Constraining input parameter values

An excerpt of the woven state machine is shown in Fig. 28. On the transitions with the *dial*() trigger, where there were no guards, *“number.size*() = 4*”* was added, such as on the transition with the *dial*() trigger from *Connected_1* to *NotFull* in Fig. 28. For the transitions with the *dial*() trigger, where there were guards already present in the base state machine, *“number.size*() = 4*”* was conjuncted to the existing guards, such as on the self-transition on *NotFull* in Fig. 28.

6 Results from the complete industrial case study

In this section, we present results and discussions from the entire industrial case study. This is based on an augmented and complete version of the simplified case study presented in Sect. 5. Our goal is to assess whether RUMM addresses practical needs when modeling the robustness behavior of a realistic system and whether it has the potential to provide significant benefits in terms of reducing modeling effort and error proneness.


when (not self.video.on)
when (self.video.videoFrameLoss.value > self.video.videoFrameLossThreshold.value
when (self.video.videoQuality > self.video.videoQualityThreshold)
when (not self.audio.on)
when (self.audio.audioFrameLoss.value > self.audio.audioFrameLossThreshold.value)
when (self.audio.PESQ > self.audio.pesqThreshold)
when (self.audio.mixingAudio)
when (self.synchronizationMismatch.value > self.synchronizationMismatchLength.value)
when (self.audio.noiseLevel.value > self.audio.noiseLevelThreshold.value)
NotFull
RecoveryMode
AudioOn
VideoFrameLoss
VideoQuality
AudioOn
AudioFrameLoss
AudioQuality
MixingAudio
SynchronizationMismatch
NoiseLevel

VideoFrameLoss = when (self.video.videoFrameLoss.value >= 0 and self.video.videoFrameLoss.value <= videoFrameLossThreshold.value) and (self.systemUnit.NumberOfActiveCalls > 1 and self.systemUnit.NumberOfActiveCalls < self.systemUnit.MaximumNumberOfCalls) and self.conference.PresentationMode = 'off' and self.conference.calls->select(c:Call| c.outgoingPresentationChannel->asSequence()->last().Protocol = VideoProtocol::off)->size() = 0 and self.conference.calls->select(c:Call | c.incomingPresentationChannel->asSequence()->last().Protocol <> VideoProtocol::off)->size() = 0)

Fig. 27 An excerpt of woven state machine obtained after applying the *MediaQualityRecovery* aspect

Table 10 Complexity of Saturn state machines

Subsystem	Number of states		Number of transitions
	States	Submachine states	
1	15	4	56
2	6	0	20
3	2	0	2
4	2	0	5
5	2	0	2
6	22	7	63
7	2	0	2
8	5	0	2
9	2	0	2
10	2	0	2
11	3	0	2
12	4	0	7
13	6	0	8
14	2	0	3
15	2	0	3
16	2	0	2
17	3	0	2
18	4	0	10
19	2	0	2
20	4	0	20

Table 11 Modeling tasks when using and not using AspectSM

Crosscutting behavior	Using aspects			Without aspects			Effort saved (%)		
	States (Added)	Transition (Added)	Trigger (Added)	States (Modified/ Added)	Tran-sitions (Modified/ Added)	Trigger (Added)	States	Transi-tions	Trigger
Updating audio constraints	1	–	–	86 (Modified)	–	–	99%	–	–
Updating video constraints	1	–		86 (Modified)	–	–	99%	–	–
Media quality recovery	3	3	19	20 (Added)	178	1604	–	98%	99%
Network communication	3	3	13	20 (Added)	178	1082	–	98%	99%
Add Guard	2	1	–	0	22 (Modified)	–	–	95%	–

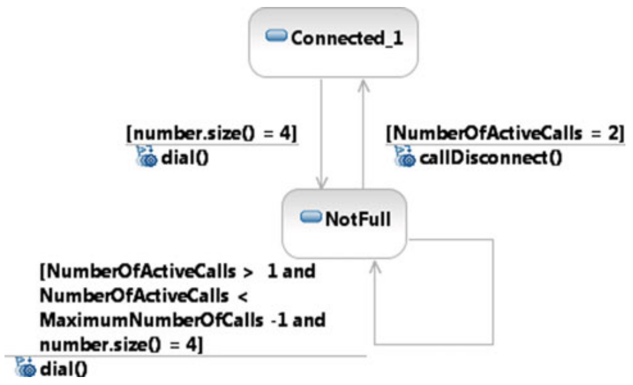


Fig. 28 An excerpt of woven state machine obtained after applying the *AddGuard* aspect

6.1 Behavioral models of Saturn

Saturn consists of 20 subsystems. Each subsystem can work in parallel to the S-Saturn subsystem shown in Fig. 3. For each subsystem, we modeled a class diagram to capture APIs and state variables. In addition, we modeled one or more state machines to model the behavior of each subsystem. Due to confidentiality restrictions, we do not provide names and details of the subsystems. For one subsystem (subsystem no 1), which is described in Sect. 2, we provided a conceptual model in Fig. 2. The behavioral model of the subsystem number 1 in Table 10 consists of 15 states: 4 of them are modeled as submachine states to reduce model complexity. The state machines of this subsystem are presented in [45]. For other subsystems, we do not provide class diagrams and state machines, but their complexity is summarized in Table 10. It is important to note though the complexity of an individual subsystem may not look high in terms of number of states and transitions; all subsystems work in parallel to each other and, therefore, the overall complexity is enormous after combining them. Saturn's implementation consists of more than 3 million lines of C code.

6.2 Modeling robustness behavior

We modeled three crosscutting behaviors on *Saturn*. The first two are the same as presented in Sect. 5.4 and Sect. 5.5. In addition, we modeled the behavior of Saturn in the presence of different network communication faults (*Network-Communication*) such as packet loss, jitter, and illegal packets in videoconference protocols. The *NetworkCommunication* aspect is presented in Appendix C.

6.3 Results and discussion

In this section, drawing lessons learned from our case study, we discuss the benefits achieved by applying RUMM to model the robustness crosscutting behavior of Saturn.

6.3.1 Reduced modeling effort

Modeling effort can be measured in different ways. One way, which is part of our future research plans, is to conduct a controlled experiment that can compare the modeling effort of applying aspect state machines with standard UML state machines. An alternate, much less expensive way is to estimate modeling effort through a surrogate measure, the number of modeling elements required to be modeled. This number can then be compared in aspect state machines and standard UML state machines when modeling the same crosscutting behaviors. Table 11 summarizes the modeling tasks involved when using and not using aspect state machines for modeling the above-mentioned crosscutting behaviors. The first two crosscutting concerns are related to updating audio and video constraints (Appendix A) in 86 states of Saturn. Using our profile we need to model one state in the aspect state machine, whereas 86 states of Saturn need to be changed if one is modeling this behavior directly. This means a reduction of approximately 99% of the number of elements involved in the change.

The third crosscutting behavior is for modeling media quality recovery. When using AspectSM, we need to model three states and three transitions in the aspect state machine (Fig. 23). Two transitions have nine triggers, each with change events, and one transition has one trigger with a time event. On the other hand, without aspect state machines, we need to model one new state and 178 new transitions with 1604 triggers (1603 with change events and one with a time event) in the base state machines of *Saturn*. This means that, assuming modeling effort is roughly proportional to the number of modeling elements, there is a 99% effort reduction in modeling triggers and a 98% effort reduction in modeling transitions. However, since the use of aspect state machines requires modeling three extra states with the « *Pointcut* » stereotype, there will only be a benefit if modeling 1604 triggers on a state machine is more time-consuming than modeling three pointcuts. Though this seems to be likely, it would need to be confirmed via controlled experiments involving human designers to determine the actual percentage of modeling effort saved when using aspect state machines. Similar results were obtained for the *Network Communication* aspect. Results from the last crosscutting behavior in Table 11 (*Add Guard*) indicate that when using aspect state machines, we need to model two states and one transition, whereas without aspect state machines we need to change 22 transitions in the base state machine of one of the subsystems of Saturn.

Overall, the results of this industrial case study seem to suggest that the modeling effort can be significantly reduced when using aspect state machines for modeling crosscutting behavior using AspectSM. Such industrial case studies showing the practical advantage of aspect modeling are unfortunately still too rare in the research literature and we are therefore not in a position to make comparisons with previous works.

6.3.2 Enhanced separation of concerns

Modeling crosscutting behavior in the UML state machines provides enhanced separation of concerns. For instance, the *AddGuard* aspect state machine models constraints on input parameters of the call event *"dial"* separately from the base state machine. In addition, the *MediaQualityRecovery* aspect state machine (Fig. 23) models a complex media quality crosscutting behavior separately from the base state machines and other aspect state machines. This means that a modeler, or several of them with possibly different expertise, can focus on each crosscutting concern separately and therefore model them separately from the core functionality and other crosscutting concerns. This is very important for our industrial partner since they have separate groups for different kinds of testing activities including functional testing, video testing, audio testing, and network testing. Using our methodology, each group can model aspects which are related to their expertise and our tool can then be used to automatically weave these aspects with the behavioral base models (models developed by the functional testing group).

6.3.3 Improved readability

Modeling crosscutting behavior as aspect state machines keeps the base state machine less cluttered and hence easier to read. For instance, the woven state machine after applying *MediaQualityRecovery* on the *Saturn* base state machine results in a highly complex, cluttered state machine, which is difficult to read: 20 states and 178 new transitions with 1604 triggers are added into the base state machines. Our experience is that modeling such complex state machines without aspect state machines is difficult to understand for practitioners and is error prone. Using aspect state machines, the base state machine and aspect state machines are separate and are less complex in isolation. To confirm this, we recently conducted a controlled experiment to measure the readability of aspect state machines using AspectSM [53]. Readability was measured based on the identification of defects seeded in state machines (modeled with and without AspectSM) and the score obtained when answering a comprehension questionnaire about the system behavior. The results of the experiment showed that readability with AspectSM is significantly better than that with both flat and hierarchical state machines measured in terms of inspecting models to identify seeded defect. In terms of the comprehension questionnaire, the AspectSM scores were better than flat state machines, but worse than hierarchical state machines. However, there were no significant differences between aspect and hierarchical state machines. In addition, no significant differences were observed in terms of the effort required to inspect models and detect defects.

6.3.4 Easier model evolution

Model evolution is also expected to be easier when using aspect state machines. For instance, *AudioQualityAspect* and *VideoQualityAspect* presented in Appendix A change the state invariants of 86 states in the base state machines. In future, more media quality measures will likely be introduced, and constraints specific to these measures will be required. Using our profile, they will be added only in the aspect state machines we defined. Otherwise, with regular state machine modeling, the new constraints would need to be added to all nine states of the base model. In systems with hundreds of states, changing the state invariants of all states is cumbersome and error prone, which makes model evolution difficult. This will be further investigated with controlled experiments in the future.

6.3.5 Systematic fault modeling

Using RUMM, we can systematically identify possible classes of faults for a specific SUT based on the proposed fault taxonomy. Furthermore, we can then instantiate specific fault types from the identified classes, which are considered critical in the SUT environment. We then model them using an aspect class diagram according to our guidelines (Sect. 5.4) and aspect state machines based on RobustProfile (Sect. 4.3). The entire process follows systematic steps to identify and model faults (Fig. 4).

6.4 Limitations

RUMM is a modeling methodology specifically developed for modeling robustness behavior to facilitate automated model-based testing. While developing the methodology, we took into consideration only those issues which were relevant for modeling the behavior of a system in the presence of faulty situations in the environment. We have not investigated whether other non-functional crosscutting concerns such as security and dependability can be successfully modeled using RUMM or an adapted version of it. The reason is that RUMM starts with modeling faults based on fault taxonomy for the system environment, which may not be necessary, for instance, when modeling security concerns such as logging. In addition, since RUMM was developed for model-based testing, we only considered issues which were important to support automated testing. For instance, we focused on UML state machines, which are often used for the automated testing in control and communication systems, which typically exhibit state-driven behavior. We also focused on modeling crosscutting behavior on those modeling elements of state machines that are mandatory to support test automation such as states (including state invariants, entry, exit, and do activities) and transitions (including guard, trigger, and effect). In AspectSM, we write pointcuts as OCL queries, and we have not yet empirically evaluated and compared their expressiveness when using other related languages and notations such as the one presented in [6]. We used OCL to write pointcuts as it is the only standard for writing constraints in UML models, an important advantage in industrial contexts. Last, our work for defining interactions and ordering between different aspect state machines still requires further investigation.

7 Related work

This section discusses existing works that are directly, but often partially, related to the objectives of RUMM. We analyze and compare published work on robustness modeling methodologies and AOM profiles for UML state machines, generic AOM weavers, and testing based on AOM.

7.1 Robustness modeling methodologies

Most of the work related to robustness modeling does not make use of AOM and focus only on modeling the behavior of a system when invalid inputs are given to the system, or on modeling exceptions in the SUT in a similar fashion to programming languages. For instance, Pintér and Majzik [54] reports on the modeling of exceptions in statecharts in a similar fashion to Java mechanisms for writing exceptions (*try catch* blocks). Exceptions are modeled as events on transitions in statecharts. Such statecharts are subsequently used for model checking. Jiang et al. [55] proposed a generic framework to model self-healing software, i.e., software which tries to recover from faults during their execution. The framework supports modeling faults (such as related to invalid inputs to a system), their detection, and their resolution with the help of different patterns defined for these purposes. Self-healing is modeled as separate, which is then combined into the functional model. Lei et al. [56] provides a methodology to check the robustness of component-based systems in the case of invalid inputs. Test cases are then generated for invalid inputs at various states and the robustness of the system is checked. Nebut et al. [57] provides an automatic test generation approach based on use cases extended with contracts, after transforming them into a transition system. Their approach supports both functional and robustness test generation. Robustness test cases are generated by calling use cases when their preconditions are false. Entwisle et al. [58] proposed a framework for modeling various domain-specific exception types such as network exceptions, database exceptions, and Web service exceptions using use cases. This approach generates exception policy configurations from application models using model transformation and finally generates code in Java for exceptions management, such as how to catch a particular exception.

The work (RUMM) presented in this paper is different from the existing work in robustness modeling in one or more of the following ways: (1) It provides a robustness modeling methodology to model system robustness in the presence of faults in its environment; this aspect has received little attention in the literature. In contrast, most of the existing works focus only on modeling the behavior of a system when invalid inputs are given to them [54–57]. (2) It is aimed at performing automated model-based robustness testing based on the robustness models for industrial systems. In contrast to the work presented in [57], our work is based on UML state machines, which are the main notation currently used for model-based test case generation [12]; (3) It relies on modeling standards, in this case UML state machines and the MARTE profile [2], to model faulty situations of the environment. (4) It uses AOM to model robustness behavior separately from the core, functional behavior, hence, decreasing modeling effort by avoiding clutter in models, making them easier to read and decreasing chances of modeling errors. (5) We use standard UML extension mechanism, i.e., profile, to support robustness modeling as aspects using standard UML state machines, thus eliminating the need to adopt new notations and consequently facilitating the practical adoption of RUMM in industry. (6) RUMM is driven by defining a fault taxonomy, thus leading to the more systematic modeling of robustness behavior. The process of defining the taxonomy helps in developing a clear and thorough understanding of the different kinds of faults that may occur in the environment against which system robustness must be tested.

7.2 AOM profiles for UML state machines

Several UML profiles for AOM have been proposed in the literature [59–62] for different UML diagrams. Since we have defined a profile for aspects on state machines, we only assess the existing AOM work focusing on state machines. We do so along three dimensions: (1) Features of UML state machines supported by a profile such as state, state invariant, do activity, entry activity, exit activity, transition, guard, trigger, and effect; (2) Features of aspect-orientation supported by a profile or a modeling approach such as pointcut, advice, and inter-type declaration (a programming construct in AspectJ [44] used to introduce new variables in a base class); (3) Representation used for the aspect-orientation features. Based on the above selection criteria, we found five related works in the literature [27–29,33,63]. Tables 12 and 13 characterize these works with respect to their coverage of important UML state machine modeling elements including state, transition, and their contained elements, e.g., state invariant in state and guard in transition. For instance, in Tables 12 and 13, the approach presented in [27] only supports modeling crosscutting behavior in states and transitions (indicated by a + sign), but not in other modeling elements (indicated by a − sign). Certain features of UML state machines which are mandatory for performing automated, model-based testing are not supported by any of the existing works. This includes state invariants and guards which, as discussed above, are essential to generating automated oracles and automated test data, respectively.

Table 14 assesses existing works with respect to the features of aspect-orientation they support such as types of advice. In light of these comparisons, one of our profile (AspectSM) contributions is that it supports all UML state machines and aspect-orientation features. Table 15 provides information on the notations used by each approach for modeling aspect-oriented features, whether UML diagrams or other non-standard notations. Table 15 suggests that no existing profile is exclusively based on standard UML notation and OCL, thus requiring the learning of additional, non-standard notations or languages, and therefore making it difficult to reuse open source and commercial technology. This is, as discussed earlier, highly important in most industrial contexts and strongly affects the adoption of modeling technologies. In conclusion, based on the information provided in Tables 12, 13, 14, and 15, we conclude that our approach supports all necessary features of UML state machines and aspect-orientation, which are all required for model-based robustness testing and do so based exclusively on standard modeling notations. In addition, our profile is developed with minimum extensions to the UML standard and hence eases adoption by our industrial partner.

7.3 Comparisons with generic AOM weavers

A generic weaver, GeKo, is presented in [26], but the current implementation of the weaver is not complete (e.g., it does not support state machines) and its use requires many manual steps such as specifying mappings from pointcuts to the base model. Metamodels for pointcut and advice are defined by relaxing the UML 2.0 metamodel and are generated automatically from it using a transformation. However, there is no support for modeling pointcuts and advice based on the generated metamodels. It therefore requires developing a new diagrammatic support for these metamodels, which will not be standard, and consequently will not be supported by UML modeling tools, making the practical adoption of the weaver difficult. Another similar generic weaver, SmartAdapter, is presented in [64]. The only major difference between GeKo and SmartAdapter is that SmartAdapter requires manually writing composition rules for aspect and base models, whereas this is not required by GeKo.

An aspect composition language (SDMATA/MATA) is presented in [6,65], which allows modeling and composing aspects on UML state machines using patterns. The selection of modeling elements of a UML state machine (concept similar to pointcuts) is performed using *state diagram patterns*. With state diagram patterns, modeling elements are selected using regular expressions defined on diagrammatic notations that ‘resemble’ UML state machines (defined based on the extension of UML state machine metamodel). In AspectSM, we write pointcuts as OCL expressions to query modeling elements of a base state machine. To compare the expressiveness of OCL expressions for writing pointcuts with regular expressions, a controlled experiment is required, which will be conducted in the future. The tool support for modeling patterns in SDMATA, however, is still under development.

Table 12 Comparison of supported modeling elements related to a state

Reference	State	State invariant	Entry activity	Do activity	Exit activity
[27]	+	–	–	–	–
[28]	+	–	–	–	–
[29]	+	–	–	–	–
[33,66]	+	–	–	–	–
[63]	+	–	–	–	–

Table 13 Comparison of supported modeling elements related to a transition

Reference	Transition	Guard	Trigger	Effect
[27]	+	–	–	–
[28]	+	–	–	–
[29]	+	+	+	–
[33,66]	+	–	–	–
[63]	+	–	+	+

Table 14 Comparison of supported features of aspect-orientation

Reference	Before advice	Around advice	After advice	Pointcut	Introduction
[27]	+	–	+	+	–
[28]	+	–	+	+	–
[29]	+	–	+	+	+
[33,66,26]	+	+	+	+	+
[63]	–	+	–	+	–

Table 15 Comparison of the representation of aspect-orientation features

Reference	Aspect	Advice	Pointcut	Introduction
[27]	State machine	State machine elements	Non-Standard	Not supported
[28]	State machine	Non-Standard	Non-Standard	Not supported
[29]	State machine	Non-Standard	Non-Standard	Non-Standard
[33,66]	State machine	Non-Standard	Non-Standard	Non-Standard
[63]	Class	Activity diagram	Non-Standard	Not supported
AspectSM	State machine	State machine elements and OCL	OCL	State machine elements

SDMATA requires defining *composition operators* (concept similar to advice) using a language based on graph transformations. As for other approaches in the literature, applying SDMATA to industrial contexts, requires learning additional, non-standard notations such as state diagram patterns.

Kermeta [15] is a model-to-model transformation language, which provides the facility to write transformation code in aspect-oriented style. Using such facility, aspects can be introduced at runtime on metaclasses (e.g., UML *Statemachine* metaclass) for introducing new attributes and operations on metaclasses or for providing definitions of existing operations in metaclasses. However, applying Kermeta for our purpose in the industrial setting requires understanding not only details of the UML metamodel, but also requires learning a new language for writing aspects. Using AspectSM, we only need simple stereotypes with a few attributes, thus reducing the learning curve and improving applicability. In other words, achieving a similar objective in Kermeta may require writing hundreds of lines of complex transformation code.

These generic weavers, being applicable to a wide range of modeling languages, are of course potentially usable in our context. On the other hand, such flexibility is possible only at the expense of additional, significant cost to provide modeling support for the defined AOM concepts. This mostly stems from the fact that no standard notation (e.g., UML) and metamodel can be used, as described above. This is why, to facilitate adoption in practice, we decided to rely on a dedicated UML profile (AspectSM) to define aspect state machines, thus relying on standard modeling environments.

7.4 Testing based on aspect-orientation modeling

There are also works in the literature that deal with testing aspect-oriented programs using UML-based models such as state machines [66–68]. The focus of our work is different since we do not focus on testing implementation, which is coded in an aspect-oriented programming (AOP) language such as AspectJ [44]. For instance, in our industrial system, we target system level testing of an embedded software of a VCS developed by Cisco, Norway, which is implemented in a subset of C language. In addition, a few approaches such as those presented in [69,70] focus on testing components using AOM to specify their behavior as state machines. The aspects are also specified as state machines to be consistent with the notation of the core behaviors (components). The composition rules are specified in their own developed language (not following any standard), which specify how to weave aspects into the core behavior. These works focus on modeling and testing components when wrong inputs are provided to them by their users. Our purpose is also different from these approaches, since we focus on modeling faulty environment (network and other VCSs) conditions of the system under test using aspect state machines and test the behavior of the VCS in the presence of these conditions.

8 Conclusion

Model-based testing, and in particular automated testing based on state machines, is a very popular approach to testing, which is supported by an increasing number of open source and commercial tools. However, for such testing to be effective, one must not only model nominal behavior but also robustness behavior. For example, in control systems, one must model how the system should react to the breakdown of sensors or actuators. In communication systems, in a similar way, one must model how the system reacts to network problems. Modeling the robustness behavior of systems in state machines is often a major source of complexity, thus leading to very large, error-prone models.

To systematically model robustness behavior for model-based testing and to alleviate its complexity, this paper presents a RobUstness Modeling Methodology (RUMM) that uses a UML 2.0 profile to support the modeling of robustness behavior as aspects in UML state machines (aspect state machines). This profile was developed by augmenting many of the concepts in existing UML state machine profiles for AOM to achieve the specific goal of supporting automated, model-based robustness testing. Furthermore, to make our approach more practical in industrial contexts, aspect state machines and their features are modeled using the UML state machine notation and the Object Constraint Language (OCL), and therefore does not require that modelers learn new diagrammatic notations or languages.

Another very important contribution of the paper is that we performed and report on an industrial case study that suggests that using our methodology and profile may result in significantly reduced modeling effort. Such case studies are indeed very rare and, to the knowledge of the authors, none is reported on aspect state machines. Results show that modeling crosscutting behavior as a separate model (aspect state machine) leads to the modeling of significantly less states, less transitions and also less changes to constraints such as state invariants. Modeling both standard and crosscutting behavior—in our case robustness behavior—in one state machine would lead to many redundant modeling elements and yield cluttered models that are difficult to understand. As an example, for one of the aspect state machines in our case study, we avoided the modeling of 1586 extra triggers on 178 transitions (98% reduction) by using our profile. However, this came at the cost of modeling three pointcuts for that aspect state machine, which is clearly an additional overhead, but which should be minimized by the fact that they are modeled as a UML state machine. It is however expected that the modeling effort required to model three pointcuts is significantly less than modeling 1586 triggers. In addition, the results of a recent controlled experiment [53] showed that readability of aspect state machines was significantly better than standard UML state machines, though there was no significant difference in the effort to inspect both types of state machines. Readability was measured based on the identification of defects seeded in state machines (modeled with and without AspectSM) and the score obtained when answering a comprehension questionnaire about the system behavior.

We also developed a weaver using the model transformation tool Kermeta [15] to automatically produce woven state machines. These can in turn be used for different applications, in our case model-based testing using state machines in input based on technologies such as Conformiq QTronic [17] and SmartTesting Test Designer [49]. In the future, we are planning to integrate the woven state machines produced by our weaver with our model-based testing tool TRUST [16] to automatically generate robustness test cases. TRUST [16] has already been used for generating executable functional test cases at Cisco, Norway. In the future, we will investigate to what extent our profile is applicable for other types of crosscutting behaviors to be modeled as state machines. In addition, we need to investigate the effort required by developers and testers to learn and apply RUMM. A series of controlled experiments and case studies are required for this purpose, which we are planning to conduct in the future. Our work on modeling interactions and ordering between various aspects still needs further investigation and evaluation.

Acknowledgments We would like to thank Robert B. France and Jacques Klein for their insightful comments and feedback on this paper. We would also like to thank Marius C. Liaaen and Suneth Walawege from Cisco Systems, Norway for helping us with modeling the case study.

9 Appendix A: Aspects for Updating state invariants

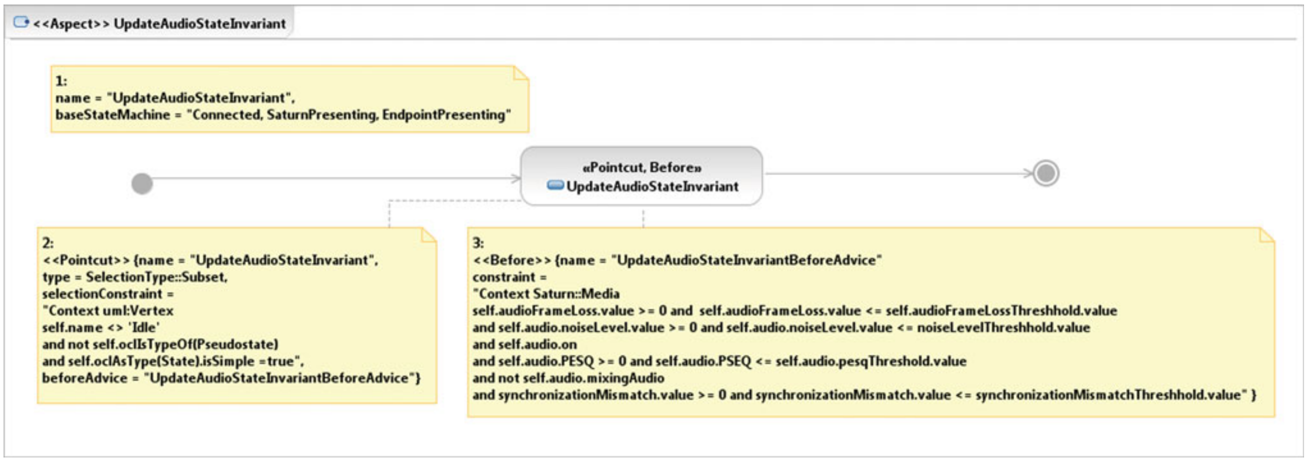
In this section, we present the details of *AudioQualityAspect* and *VideoQualityAspect*. These aspects update state invariants in the base state machine (Fig. 3) with audio and video quality constraints.

9.1 Updating state invariants with audio quality attributes

The aspect in Fig. 29 updates state invariants for all simple states where the system is in a videoconference. In Fig. 29, the « *Aspect* » stereotype is applied on the state machine, the attributes of which show that this aspect is applied to the base state machine (Saturn::Saturn) in this case. A « *Pointcut* » stereotype is applied on the state invariant of the state *UpdateStateInvariantsWithAudioQuality*. This pointcut applies a before advice on all states selected by the pointcut and this results in adding an additional constraint (see note 3). The woven state machine looks the same as the base state machines except that the state invariants of the selected states are updated.

9.2 Updating state invariants with video quality attributes

The aspect in Fig. 30 updates the state invariants of states selected in the base state machine by the « *Pointcut* » stereotype applied on the state invariant of the state *UpdateStateInvariantsWithVideoQuality* in Fig. 30 according to the before

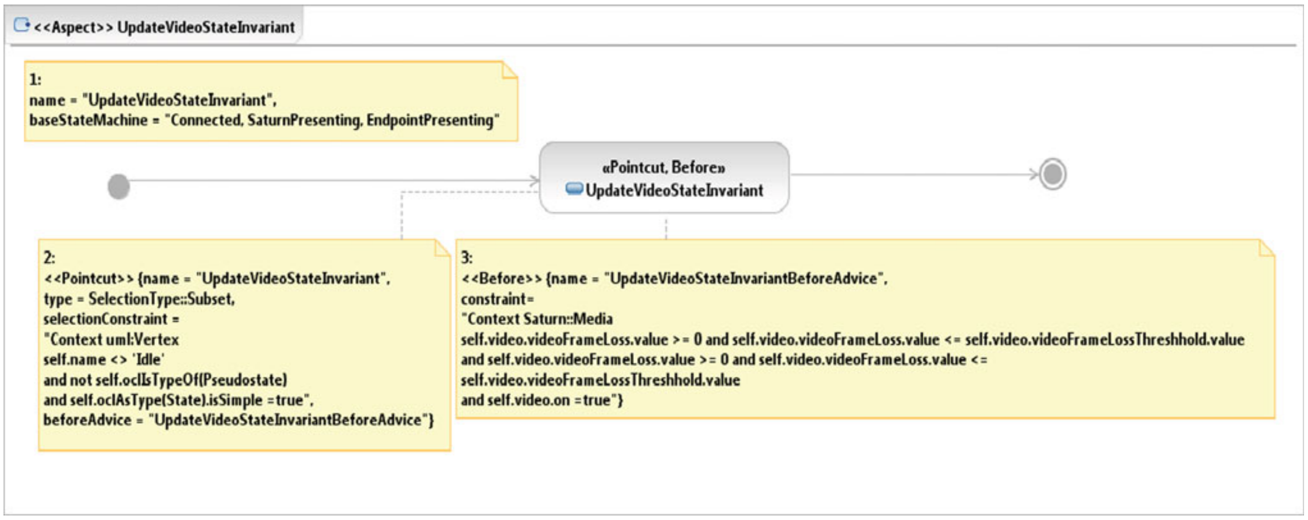

<<Aspect>> UpdateAudioStateInvariant
«Pointcut, Before»
UpdateAudioStateInvariant

```
1:
name = "UpdateAudioStateInvariant",
baseStateMachine = "Connected, SaturnPresenting, EndpointPresenting"
```

```
2:
<<Pointcut>> {name = "UpdateAudioStateInvariant",
type = SelectionType::Subset,
selectionConstraint =
"Context uml:Vertex
self.name <> 'Idle'
and not self.oclIsTypeOf(Pseudostate)
and self.oclAsType(State).isSimple =true",
beforeAdvice = "UpdateAudioStateInvariantBeforeAdvice"}
```

```
3:
<<Before>> {name = "UpdateAudioStateInvariantBeforeAdvice"
constraint =
"Context Saturn::Media
self.audioFrameLoss.value >= 0 and self.audioFrameLoss.value <= self.audioFrameLossThreshhold.value
and self.audio.noiseLevel.value >= 0 and self.audio.noiseLevel.value <= noiseLevelThreshhold.value
and self.audio.on
and self.audio.PESQ >= 0 and self.audio.PSEQ <= self.audio.pesqThreshold.value
and not self.audio.mixingAudio
and synchronizationMismatch.value >= 0 and synchronizationMismatch.value <= synchronizationMismatchThreshhold.value" }
```

Fig. 29 State machine for *AudioQualityAspect*


<<Aspect>> UpdateVideoStateInvariant
«Pointcut, Before»
UpdateVideoStateInvariant

```
1:
name = "UpdateVideoStateInvariant",
baseStateMachine = "Connected, SaturnPresenting, EndpointPresenting"
```

```
2:
<<Pointcut>> {name = "UpdateVideoStateInvariant",
type = SelectionType::Subset,
selectionConstraint =
"Context uml:Vertex
self.name <> 'Idle'
and not self.oclIsTypeOf(Pseudostate)
and self.oclAsType(State).isSimple =true",
beforeAdvice = "UpdateVideoStateInvariantBeforeAdvice"}
```

```
3:
<<Before>> {name = "UpdateVideoStateInvariantBeforeAdvice",
constraint=
"Context Saturn::Media
self.video.videoFrameLoss.value >= 0 and self.video.videoFrameLoss.value <= self.video.videoFrameLossThreshhold.value
and self.video.videoFrameLoss.value >= 0 and self.video.videoFrameLoss.value <=
self.video.videoFrameLossThreshhold.value
and self.video.on =true"}
```

Fig. 30 State machine for the V*ideoQualityAspect*

```
WeaveStateMahine (b: StateMachine, A: Set(StateMachine), w:StateMachine):StateMachine
/*
        This algorithm takes in input a base state machine b, a set of aspect state machines, and a weaving-directive state
        machine and outputs a woven state machine. All inputs and the output are instances of UML 2.0 State machine
        metaclass.
*/
Inputs:
        b: A base state machine, which is a UML 2.0 state machine.
        A:  A set of aspect state machines. Each aspect state machine is a UML 2.0 state machine.
        w: A weaving directive state machine, which consists of a set of submachine states A'. Each submachine state a'
        in A' corresponds to the an aspect state machine in the set A. w is also a UML 2.0 state machine.
Output:
        o: A woven state machine, which is a UML 2.0 state machine.
Algorithm:
    1.  Traverse sub machines states (aspects) according to the order specified in w
        a.  For each sub machine state a' in A' do
            i.  Start with the initial state and go to the first state s in a'
                1.  For each t in s.outgoing   /* For every outgoing transition of s */
                    a.  If (s.stereotype = '<<Pointcut>>')
                        i.  Call WeavePointcut(s)
                    b.  Else If (s.stereotype = '<<Introduction>>')
                        i.  Call WeaveIntroduction(s)
                    c.  Else
                        i.  Call WeaveNoStereotype(s)
```

Fig. 31 Weaving algorithm

```
Function WeavePointcut(s:State)
/*
        This function takes input a state with the stereotype <<Pointcut>> and queries the base state machine with the pointcut
        expression and calls other functions to apply advices on the base s

*/
    1.  For each t in s.outgoing
        a.  If t.target.stereotype = '<<Pointcut>>'
            i.  If t.stereotype = ''
                1.  Check which model elements (such as guard, trigger, or effect) related to the transition that
                    has a stereotype (<<Introduction>> or <<Pointcut>>)
                2.  If the model element has a stereotype  <<Pointcut>>
                    a.  Query the base model b with the selectionConstraint attribute of the pointcut
                    b.  Apply before, after, or around advice /introduction on the modeling elements
                        selected by the pointcut
                    c.  Call RepeatComposition(t.target)
            ii. Else If t.stereotype  = '<<Pointcut>>'
                1.  Call WeavePointcutOnState(s)
                2.  Call WeavePointcutOnTransition(t)
                3.  Call WeavePointcutOnState(t.target)
                4.  Call RepeatComposition(t.target)
            iii. Else
                1.  Call WeavePointcutOnState(s)
                2.  Call WeavePointcutOnState(t.target)
                3.  Add the new transition t as specified in the aspect between the states selected by above
                    two steps
                4.  Call RepeatComposition(t.target)
        b.  Else If t.target.stereotype = '<<Introduction>>'
            i.  If t.stereotype = ''
                1.  Not allowed
            ii. Else If t.stereotype='<<Introduction>>'
                1.  Call WeavePointcutOnState(s)
                2.  Call WeavePointcutOnTransition(t)
                3.  Introduce the state t.target as specified in the aspect
                4.  Call RepeatComposition(t.target)
            iii. Else
                1.  Call WeavePointcutOnState(s)
                2.  Introduce the state t.target as specified in the aspect
                3.  Add the new transition t as specified in the aspect between the states selected by above
                    two steps
                4.  Call RepeatComposition(t.target)
        c.  Else
            i.  Not allowed
```

Fig. 32 The WeavePointcut() function

```
Function Introduction(s:State)
/*
        This function takes input a state with the stereotype <<Introduction>> and introduces the new elements in the base model
        as specified by the <<Introduction>> stereotype.
*/
    1.  For each t in s.outgoing
        a.  If t.target.stereotype = '<<Pointcut>>'
                i.  If t.stereotype = ''
                        1.  Not allowed
                ii. Else If t.stereotype = '<<Pointcut>>'
                        1.  Introduce the state s as specified in the aspect
                        2.  Call WeavePointcutOnState(t.target)
                        3.  Call WeavePointcutOnTransition(t)
                        4.  Call RepeatComposition(t.target)
                iii. Else
                        1.  Introduce the state s as specified in the aspect
                        2.  Call WeavePointcutOnState(t.target)
                        3.  Add the new transition t as specified in the aspect between the states selected by above
                            two steps
                        4.  Call RepeatComposition(t.target)
        b.  Else If t.target.stereotype = '<<Introduction>>'
                i.  If t.stereotype = ''
                        1.  Not allowed
                ii. Else If t.stereotype='<<Introduction>>'
                        1.  Introduce the state s as specified in the aspect
                        2.  Introduce the state t.target as specified in the aspect
                        3.  Call WeavePointcutOnTransition(t)
                        4.  Call RepeatComposition(t.target)
                iii. Else
                        1.  Introduce the state s as specified in the aspect
                        2.  Introduce the state t.target as specified in the aspect
                        3.  Add the new transition t as specified in the aspect between the states selected by above
                            two steps
                        4.  Call RepeatComposition(t.target)
        c.  Else
                i.  Not allowed
```

Fig. 33 The Introduction() function

advice defined based on the video quality attributes modeled in Fig. 26. The *on* attribute is a *Boolean* attribute that determines whether the video is present in a videoconference. The *videoQuality* is a video quality metric for measuring video quality and is defined as an *Integer*. The *videoFrameLoss* is an *Integer* attribute that determines the current video frame loss during a videoconference.

The « *Before* » stereotype applied on the state invariant of the state *UpdateStateInvariantsWithVideoQuality* in Fig. 30 adds an additional conjunct to state invariants of all selected states (see note 3 for attribute values). The woven state machines look exactly the same as the base state machines, as only state invariants changed in this case.

10 Appendix B: Weaver algorithm

See Figs. 31, 32, 33, 34 and 35

11 Appendix C: Network communication aspect

11.1 Description of the aspect

The purpose of this aspect is to model the behavior of a system in the presence of various network faults. A system is supposed to work even under the presence of faults and unwanted conditions (degraded mode). By degraded mode, we mean that the system should continue to behave as in the non-faulty situation, except that the quality (such as audio and video) or the performance is degraded such as slow speed of running applications on a videoconference system. The system must try to recover from the degraded mode and go back to the normal mode of operation. In the worst case, the system must return to the safe state.

11.2 Network robustness (NR) aspect (aspect class diagram)

Figure 36 shows a class diagram that models the robust behavior of the system in the presence of different network faults defined based on the fault taxonomy (Fig. 5) such as jitter, packet loss, low bandwidth, illegal packets for videoconferencing protocols (SIP and H323), and in the case of no network connection. Six network properties are modeled in the class diagram that models different faulty situations. Five network properties are modeled as non-functional (NF) types using the MARTE profile [2]: packet loss, jitter, bandwidth, and percentage of illegal packets for H323 and SIP

(a)

```
Function WeaveNoStereotype(s:State)
/*
          This function takes input a state without any stereotype from an aspect state machine and applies advice/introduction on the
          base state machine as specified in the modeling elements contained within the state.
*/
     1.   For each t in s.outgoing  /* for each transition going out of s */
               a.   If t.target.stereotype = '<<Pointcut>>'
                         i.   Not allowed
               b.   Else If t.target.stereotype ='<<Introduction>>'
                         i.   Not allowed
               c.   Else
                         i.   Check which model elements (such as state invariant, do, entry, or exit activity) related to the state s
                              that has a stereotype (<<Introduction>> or <<pointcut>>)
                        ii.   If the model element has a stereotype  <<pointcut>>
                                   1.   Query the base model b with the selectionConstraint attribute of the pointcut
                                   2.   Apply before, after, or around advice /introduction on the modeling elements selected by
                                        the pointcut
                       iii.   Repeat steps i and ii for the state t.target
                        iv.   Call RepeatComposition(t.target)
```

(b)

```
Function RepeatComposition(s:State)
/*
          This function traverses the aspect state machine and calls appropriate functions to evaluate pointcut and introduction
*/
     1.   If (s.isFinal !=true) /* checks if s is a final state */
               a.   If s.stereotype = '<<Pointcut>>'
                         i.   Call WeavePointcut (s)
               b.   Else If s.stereotype = '<<Introduction>>'
                         i.   Call WeaveIntroduction (s)
               c.   Else
                         i.   Call WeaveNoStereotype (s)
```

(c)

```
Function WeavePointcutOnState(s:State)
/*
          This functions queries the base state machine according to the query expression specified in the pointcut and applies the
          advice as specified by the pointcut
*/
     1.   Query the base model b according to the query specified in the selectionConstraint attribute of the pointcut on state s.
     2.   Apply after, before, and/or around advices as specified on stereotypes <<After>>, <<Before>>, and <<Around >> to the
          model elements selected by the selectionConstraint in step 1.
```

Fig. 34 **a** The WeaveNoStereotype() function. **b** The *RepeatCompostion*() function. **c** The *PointCutOnState*() function

```
Function WeavePointcutOnTransition(t)
/*
          This function queries the base model according to the query expression specified in the pointcut and applies the advice as
          specified by the pointcut
*/
     1.   Query the base model b according to the query specified in the selectionConstraint attribute of the pointcut on state s.
     2.   Apply after, before, and/or around advices as specified on stereotypes <<After>>, <<Before>>, and <<Around >> to the
          model elements selected by the selectionConstraint in step 1.
```

Fig. 35 The *PointcutOnTransition*() function

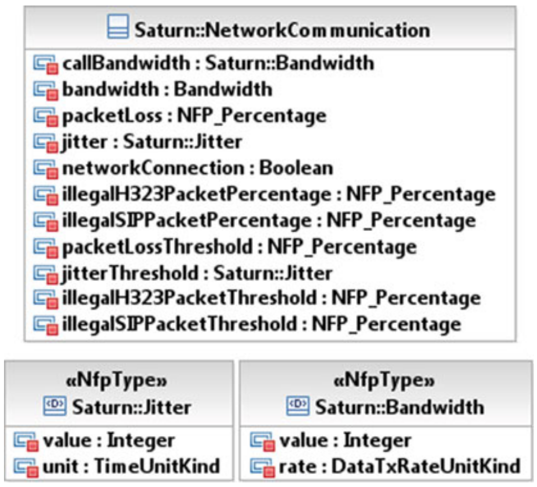

Saturn::NetworkCommunication
callBandwidth : Saturn::Bandwidth
bandwidth : Bandwidth
packetLoss : NFP_Percentage
jitter : Saturn::Jitter
networkConnection : Boolean
illegalH323PacketPercentage : NFP_Percentage
illegalSIPPacketPercentage : NFP_Percentage
packetLossThreshold : NFP_Percentage
jitterThreshold : Saturn::Jitter
illegalH323PacketThreshold : NFP_Percentage
illegalSIPPacketThreshold : NFP_Percentage
«NfpType»
Saturn::Jitter
value : Integer
unit : TimeUnitKind
«NfpType»
Saturn::Bandwidth
value : Integer
rate : DataTxRateUnitKind

Fig. 36 Class diagram for the NR aspect

protocols. The network connection is modeled as a *Boolean* attribute.

11.2.1 PacketLoss

This property is defined to introduce packets loss during communication and is measured in terms of percentage. This property is defined to be of the MARTE type *NFP_Percentage* because packet loss is always measured in percentage and the *NFP_Percentage* is defined in the MARTE profile for this purpose.

11.2.2 Jitter

This property introduces delay between network packets. This delay is introduced in the unit of millisecond (ms) and checks robustness of a videoconferencing system in the presence of delayed network packets. This property has two attributes: *value* of type *Integer* and *unit* of the MARTE type *TimeUnitKind*. The type *TimeUnitKind* of the MARTE profile is used to define units for time values such as millisecond and microsecond. We chose this data type so that a modeler can choose the appropriate unit to measure unit. We set the default value of the *unit* attribute to millisecond (ms).

11.2.3 Bandwidth

This property is used to change the bandwidth of the network and is measured in terms of Kilobytespersecond (Kbps) and checks robustness of a videoconferencing system in the presence of low bandwidth than required by a videoconference. This property has two attributes: *value* of type *Integer* and rate of the MARTE type *DataTxRateUnitKind*. The type *DataTxRateUnitKind* is used to define units for data transmission such as KiloBytesPerSecond (Kbps) and MegaBytesPerSecond (Mbps). We chose this data type because it allows a modeler to change the unit of data transmission as required. We set the default value of the *rate* attribute to KiloBytesPerSecond (Kbps).

11.2.4 IllegalH323PacketPercent

This property is used to add illegal packets for the H323 videoconferencing protocol during a videoconference to see how a VCS behaves. This property is of type *NFP_Percentage*.

11.2.5 IllegalSIPPacketPercent

This property is used to add illegal packets for the SIP videoconferencing protocol during a videoconference to see how a VCS behaves. This property is of type *NFP_Percentage*.

11.3 Aspect state machine for NR

The aspect state machine for the NR aspect is shown in Fig. 37. The *'NetworkCommunication'* state machine is stereotyped as *'Aspect'* and the attributes associated with the stereotype are shown in the note labeled 1. The first attribute *name* specifies the name of the aspect, which is *NetworkCommunication* in this case. The second attribute *baseStateMachine* specifies the base state machine on which the aspect will be woven, which is Saturn (Fig. 3) in this case.

A pointcut named *'SelectStatesPointcut'* on the state *'SelectedStates'* is shown in Fig. 37 (see note 3), which selects all states of the base state machine except for the *Idle* and *PresentingWithoutCall* states. New transitions modeling robust behavior of the system from all states selected by the '*SelectStatesPointcut*' pointcut to a new state '*DegradedMode*' stereotyped with the « *Introduction* » *and* « *ExternalFault* » stereotypes are introduced. These robustness transitions are modeled as UML change events and stereotyped with the « *NetworkFault* » stereotype, which indicates that this event is modeling a network fault. For instance, when '*when (not self.networkConnection)*' in any of the states selected by the pointcut, the system goes to the state '*DegradedMode*', which is stereotyped as « *Introduction* » indicating that this state will be introduced in the base state machine. In this state, the system tries to recover the network connection. If the system is successful in recovering the network connection, the transition with the change event '*when(self.networkConnection)*' takes the system back to the original state, which is one of the states selected by *SelectedStates* state stereotyped « *Normal* » to indicate that this is a normal state of the system. If the system cannot recover within time *t*, then the system disconnects all the systems and goes to the '*Idle*' state stereotyped as « *Initial* » indicating that this is the initial state of the system. This is modeled as a new transition from the

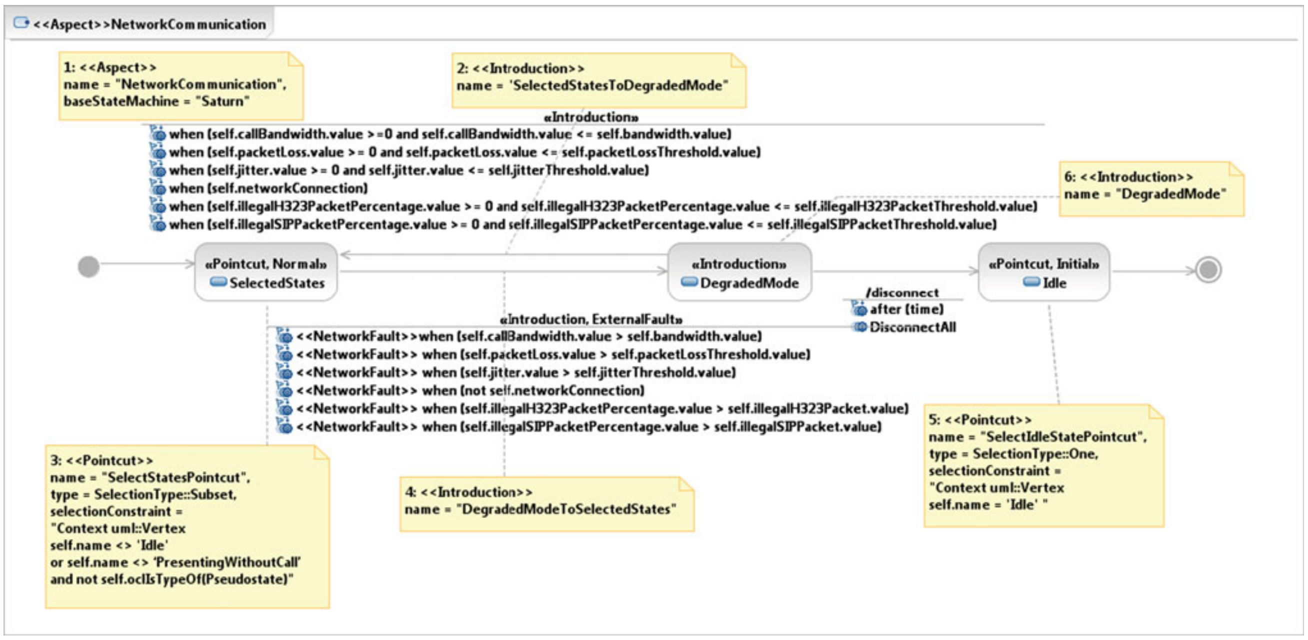

<<Aspect>>NetworkCommunication
1: <<Aspect>>
name = "NetworkCommunication",
baseStateMachine = "Saturn"
2: <<Introduction>>
name = 'SelectedStatesToDegradedMode"
«Introduction»
when (self.callBandwidth.value >=0 and self.callBandwidth.value <= self.bandwidth.value)
when (self.packetLoss.value >= 0 and self.packetLoss.value <= self.packetLossThreshold.value)
when (self.jitter.value >= 0 and self.jitter.value <= self.jitterThreshold.value)
when (self.networkConnection)
when (self.illegalH323PacketPercentage.value >= 0 and self.illegalH323PacketPercentage.value <= self.illegalH323PacketThreshold.value)
when (self.illegalSIPPacketPercentage.value >= 0 and self.illegalSIPPacketPercentage.value <= self.illegalSIPPacketThreshold.value)
6: <<Introduction>>
name = "DegradedMode"
«Pointcut, Normal»
SelectedStates
«Introduction»
DegradedMode
«Pointcut, Initial»
Idle
/disconnect
after (time)
DisconnectAll
«Introduction, ExternalFault»
<<NetworkFault>> when (self.callBandwidth.value > self.bandwidth.value)
<<NetworkFault>> when (self.packetLoss.value > self.packetLossThreshold.value)
<<NetworkFault>> when (self.jitter.value > self.jitterThreshold.value)
<<NetworkFault>> when (not self.networkConnection)
<<NetworkFault>> when (self.illegalH323PacketPercentage.value > self.illegalH323Packet.value)
<<NetworkFault>> when (self.illegalSIPPacketPercentage.value > self.illegalSIPPacket.value)
5: <<Pointcut>>
name = "SelectIdleStatePointcut",
type = SelectionType::One,
selectionConstraint =
"Context uml::Vertex
self.name = 'Idle' "
3: <<Pointcut>>
name = "SelectStatesPointcut",
type = SelectionType::Subset,
selectionConstraint =
"Context uml::Vertex
self.name <> 'Idle'
or self.name <> 'PresentingWithoutCall'
and not self.oclIsTypeOf(Pseudostate)"
4: <<Introduction>>
name = "DegradedModeToSelectedStates"

Fig. 37 State machine for the 'NetworkCommunication' aspect

'*DegradedMode*' state to the '*Idle*' state, with a time event *after(t)*, and a new effect '*DisconnectAll*' with an opaque action '*disconnect*', which disconnects all the systems connected to the system.

References

1. UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms. http://www.omg.org/spec/QFTP/1.1/ (2010)
2. Modeling and Analysis of Real-time and Embedded systems (MARTE). http://www.omgmarte.org/ (2010)
3. Jürjens, J.: UMLsec: extending UML for secure systems development. In: Proceedings of the 5th International Conference on the Unified Modeling Language. Springer, Berlin (2002)
4. IEEE Standard Glossary of Software Engineering Terminology. IEEE, IEEE Std 610.12-1990 (1990)
5. Yedduladoddi, R.: Aspect oriented software development: an approach to composing UML design models. VDM Verlag Dr. Müller, Saarbrücken (2009)
6. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., Rabbi, R.: An expressive aspect composition language for UML state diagrams (2007)
7. Runeson, H., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Eng. **14**(2I), 131–164 (2009)
8. Aldini, A., Gorrieri, R., Martinelli, F., Jürjens, J.: Model-based security engineering with UML. Springer, Berlin/Heidelberg (2005)
9. Péreza, J., Ali, N., Carsı'b, J.A., Ramosb, I., Álvarezc, B., Sanchezc, P., Pastorc, J.A.: Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Inf. Softw. Technol. **50**(9-10I), 969–990 (2008)
10. Cottenier, T., Berg, A.v.d., Elrad, T.: The Motorola WEAVR: model weaving in a large industrial context. In: Proceedings of the Aspect Oriented Software Development (AOSD) (2007)
11. Cottenier, T., Berg, A.v.d., Elrad, T.: Stateful aspects: the case for aspect-oriented modeling. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling. ACM, Vancouver (2007)
12. Shafique, M., Labiche, Y.A.: Systematic review of model based testing tools. Carleton University, Department of Systems and Computer Engineering. Technical Report (SCE-10-04) (2010)
13. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1I), 11–33 (2004)
14. IEEE Standard Classification for Software Anomalies. IEEE, IEEE Std 1044-2009 (2009)
15. Kermeta-Breathe Life into Your Metamodels, IRISA and INRIA. http://www.kermeta.org/ (2010)
16. Ali, S., Hemmati, H., Holt, N.E., Arisholm, E., Briand, L.C.: Model Transformations as a strategy to automate model-based testing—a tool and industrial case studies. Simula Research Laboratory, Technical Report (2010-01) (2010)
17. QTRONIC, CONFORMIQ. http://www.conformiq.com/qtronic.php (2010)
18. Standard for Software Quality Characteristics. International Organization for Standardization, ISO-9126-3 (2003)
19. Software Assurance Standard. NASA Technical Standard, NASA-STD-8739.8 (2005)
20. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, Englewood Cliffs (2004)
21. Bruning, S., Weissleder, S., Malek, M.: A fault taxonomy for service-oriented architecture. In: Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium. IEEE Computer Society (2007)
22. Chan, K.S., Bishop, J., Steyn, J., Baresi, L., Guinea, S.: A Fault Taxonomy for Web Service Composition. Springer, Berlin (2009)
23. Mariani, L.: A fault taxonomy for component-based software. Electron. Notes Theor. Comput. Sci. **82**(6I), 55–65 (2003)
24. Hayes, J.H.: Building a requirement fault taxonomy: experiences from a NASA verification and validation research project. In: Proceedings of the 14th International Symposium on Software Reliability Engineering. IEEE Computer Society (2003)

25. Ho, W.-M., Jézéquel, J.-M., Pennaneac'h, F., Plouzeau, N.: A toolkit for weaving aspect oriented UML designs. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development. ACM, Enschede (2002)
26. Kienzle, J., Abed, W.A., Klein, J.: Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development. ACM, Charlottesville (2009)
27. Zhang, G.: Towards aspect-oriented state machines. In: Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development (AOASIA'06), Tokyo (2006)
28. Zhang, G., Hölzl, M.: HiLA: High-Level Aspects for UML-state machines. In: Proceedings of the 14th Workshop on Aspect-Oriented Modeling (AOM@MoDELS'09) (2009)
29. Zhang, G., Hölzl, M.M., Knapp, A.: Enhancing UML State Machines with Aspects (2007)
30. Pazzi, L.: Explicit aspect composition by part-whole state charts. In: Proceedings of the Workshop on Object-Oriented Technology. Springer, Berlin (1999)
31. France, R., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modelling. IEEE Softw. **151**(4I) (2004)
32. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Reading (1999)
33. Xu, D., Xu, W., Nygard, K.: A state-based approach to testing aspect-oriented programs. In: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, Taiwan (2005)
34. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition (2008)
35. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Tim Weilkiens, Hamburg (2008)
36. UML Profile for Schedulability, Performance and Time. http://www.omg.org/technology/documents/profile_catalog.htm (2010)
37. Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Berlin (2007)
38. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, Reading (2008)
39. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley Professional, Reading (2004)
40. IBM OCL Parser, IBM. http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html (2010)
41. OCLE. http://lci.cs.ubbcluj.ro/ocle/ (2010)
42. EyeOCL Software. http://maude.sip.ucm.es/eos/ (2010)
43. Pender, T.: UML Bible. Wiley, New York (2003)
44. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, Greenwich (2003)
45. Ali, S., Briand, L.C., Hemmati, H.: Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. Simula Research Laboratory, Technical Report (2010-03) (2010)
46. Stein, D., Hanenberg, S., Unland, R.: A UML-based aspect-oriented design notation for AspectJ. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development. ACM, Enschede (2002)
47. Clarke, S., Walker, R.J.: Composition patterns: an approach to designing reusable aspects. In: Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, Toronto (2001)
48. Stein, D., Hanenberg, S., Unland, R.: Designing aspect-oriented crosscutting in UML. In: Proceedings of the In AOSD-UML Workshop at AOSD '02 (2002)
49. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Fransisco (2007)
50. Tessier, F., Badri, L., Badri, M.: Towards a formal detection of semantic conflicts between aspects: a model-based approach. In: Proceedings of the The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004 (2004)
51. Perceptual Evaluation of Speech Quality (PESQ). http://en.wikipedia.org/wiki/PESQ (2010)
52. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C.: A search-based OCL constraint solver for model-based test data generation. In: Proceedings of the 11th International Conference on Quality Software (QSIC 2011) (2011)
53. Ali, S., Yue, T., Briand, L.C., Malik, Z.I.: Does aspect-oriented modeling help improve the readability of UML state machines? Simula Reserach Laboratory, Technical Report (2010–11) (2010)
54. Pintér, G., Majzik, I.: Modeling and Analysis of Exception Handling by Using UML Statecharts (2005)
55. Jiang, M., Zhang, J., Raymer, D., Strassner, J.: A modeling framework for self-healing software systems. In: Proceedings of the Models@run.time in conjunction with MoDELS/UML (2007)
56. Lei, B., Liu, Z., Morisset, C., Li, X.: State based robustness testing for components. Electron. Notes Theor. Comput. Sci. **260**, 173–188 (2010)
57. Nebut, C., Fleurey, F., Traon, Y.L., Jezequel, J.-M.: Automatic test generation: a use case driven approach. IEEE Trans. Softw. Eng. **32**(3I), 140–155 (2006)
58. Entwisle, S., Schmidt, H., Peake, I., Kendall, E.: A model driven exception management framework for developing reliable software systems. In: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference. IEEE Computer Society (2006)
59. Jingjun, Z.: Modeling Aspect-Oriented Programming with UML Profile (2009)
60. Júnior, J.U., Camargo, V.V., Chavez, C.V.F.: UML-AOF: a profile for modeling aspect-oriented frameworks. In: Proceedings of the 13th Workshop on Aspect-Oriented Modeling. ACM, Charlottesville (2009)
61. Aldawud, O., Elrad, T., Bader, A.: UML profile for aspect-oriented software development. In: Proceedings of the The Third International Workshop on Aspect Oriented Modeling (2003)
62. Evermann, J.: A meta-level specification and profile for AspectJ in UML. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling. ACM, Vancouver (2007)
63. Zhang, J., Cottenier, T., Berg, A.V.D., Gray, J.: Aspect composition in the motorola aspect-oriented modeling weaver. J. Object Technol. **6**, 7I (2007)
64. Petriu, D., Rouquette, N., Haugen, Ø., Morin, B., Klein, J., Kienzle, J., Jézéquel, J.-M.: Flexible Model Element Introduction Policies for Aspect-Oriented Modeling. Springer, Berlin (2010)
65. Whittle, J., Jayaraman, P.: MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. Springer, Berlin (2008)
66. Xu, D., Xu, W.: State-based incremental testing of aspect-oriented programs. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development. ACM, Bonn (2006)
67. Xu, D., Xu, W., Nygard, K.: A State-based approach to testing aspect-oriented programs. In: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (2005)
68. Xu, W., Xu, D.: A model-based approach to test generation for aspect-oriented programs. In: Proceedings of the First Workshop on Testing Aspect-Oriented Programs (2005)
69. Bruel, J.-M., Araújo, J., Moreira, A., Royer, A.: Using aspects to develop built-in tests for components. In: Proceedings of the In AOSD Modeling with UML Workshop. 6th International Conference on the Unified Modeling Language (2003)

70. Bruel, J.M., Moreira, A., Araújo, J.: Adding Behavior Description Support to COTS Components through the Use of Aspects. In: Proceedings of the 2nd Workshop on Models for Non-functional Aspects of Component-Based Software (2005)

Author Biographies

Shaukat Ali received his master's degree in systems and software engineering from Mohammad Ali Jinnah University, Islamabad, Pakistan. He is currently working for his PhD degree at the Simula Research Laboratory and the Department of Informatics, University of Oslo, Norway. He is a former member of the following research groups: Center for Software Dependability (CSD), Islamabad, Pakistan; Verification and Testing (VT) Group, the University of Sheffield, UK; Software Quality Engineering Laboratory (SQUALL), Carleton University, Canada. His research interests include modeling software systems using UML and its various extensions and model-based testing of software systems. He is a student member of the IEEE.

Lionel C. Briand is heading software verification and validation activities at Simula Research Laboratory, where he is leading research projects with industrial partners. He is also a professor at the University of Oslo (Norway). Before that, he was on the faculty of the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was full professor and held the Canada Research Chair (Tier I) in Software Quality Engineering. He has also been the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland, USA. Lionel has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the Coeditor-in-Chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification and Reliability (Wiley). He was on the board of IEEE Transactions on Software Engineering from 2000 to 2004. Lionel was elevated to the grade of IEEE Fellow for his work on the testing of object-oriented systems. His research interests include: model-driven development, testing and verification, search-based software engineering and empirical software engineering.

Hadi Hemmati is a postdoctoral fellow at Queen's University, Kingston, Canada. He received his PhD degree in software engineering from Simula Research Laboratory and the Department of Informatics, University of Oslo, Norway. He has some years of industrial experience as a system analyst and software engineer in the telecommunication domain. His research interests include model-based testing, search-based software engineering, mining software repositories, software quality assurance, ubiquities and autonomic systems.