Mutating Aspect-Oriented Models to Test Cross-Cutting Concerns

Birgitta Lindström*, Sten F. Andler*, Jeff Offutt†*, Paul Pettersson‡, and Daniel Sundmark§
* University of Skövde, Skövde, Sweden, {birgitta.lindstrom,sten.f.andler}@his.se
† George Mason University, Fairfax VA, USA, offutt@gmu.edu
‡ Mälardalen University, Västerås, Sweden, paul.pettersson@mdh.se
§ Swedish Institute of Computer Science, Kista, Sweden, dsk@sics.se

***Abstract*—Aspect-oriented (AO) modeling is used to separate normal behaviors of software from specific behaviors that affect many parts of the software. These are called "cross-cutting concerns," and include things such as interrupt events, exception handling, and security protocols. AO modeling allow developers to model the behaviors of cross-cutting concerns independently of the normal behavior. Aspect-oriented models (AOM) are then transformed into code by "weaving" the aspects (modeling the cross-cutting concerns) into all locations in the code where they are needed. Testing at this level is unnecessarily complicated because the concerns are often repeated in many locations and because the concerns are muddled with the normal code. This paper presents a method to design robustness tests at the abstract, or model, level. The models are mutated with novel operators that specifically target the features of AOM, and tests are designed to kill those mutants. The tests are then run on the implementation level to evaluate the behavior of the woven cross-cutting concerns.**

***Index Terms*—Mutation analysis, aspect-oriented modeling, robustness testing**

I. Introduction and Background

Model-based development is gaining widespread use in today's software industry. Models provide an intuitive, graphical view of software behavior. In addition, certain types of models, such as state charts [1], Petri nets [2], and timed automata [3], [4] are useful for analysis and verification purposes. Such models can be used by model checkers to verify properties such as absence of deadlock, or to ensure the correct ordering of certain events. Moreover, behavioral models can be used to generate test suites that cover the software with respect to model elements or sub paths [5], [6]. Consequently, modeling software behavior can help developers understand and analyze complex behavior.

The expressiveness of a model typically depends on the level of detail in which it is modeled. However, detailed models are also generally more complex. Thus, more detailed models are harder to understand and maintain.

A. Aspect-Oriented Modeling

One proposed approach for avoiding overly complex behavioral models is to use *aspect-oriented modeling* of cross-cutting concerns [7], [8], [9], [10], [11]. A cross-cutting concern applies throughout the software, and may be crucial to the reliability, performance, security, or robustness of the system. Typical examples include events that require immediate attention, such as intrusion attempts or disturbances. Cross-cutting concerns have a tendency to clutter models, leading to complex models that are hard to analyze.

In aspect-oriented modeling, cross-cutting concerns are modeled as *aspects*, which are separated from the normal behavior, thus creating an *aspect-oriented model* (AOM). The general idea with an AOM is to model the normal behavior of the system in a *base model*, leaving the cross-cutting concerns to be described in separate aspect models. The base model and the aspect models are then *woven* together. By modeling these concerns separately and then automatically weaving them into the model later, the behavioral models become cleaner and less cluttered.

B. Mutation-Based Testing

This paper proposes the use of mutation of aspect-oriented models to test cross-cutting concerns. In mutation testing, a software artifact such as a program or a model is modified to create alternate, usually faulty, versions called *mutants* [12]. The mutants are created by systematically applying mutation operators, which are rules for changing syntactic elements. Tests are then designed to cause the mutants to have different behavior from the original version, called *killing the mutant*. Mutation operators either mimic typical programmer mistakes or make changes that encourage testers to design particularly valuable test inputs.

Test suites are run against collections of mutants to determine the percentage of mutants the tests kill, the *mutation adequacy score*. The mutation adequacy score is a coverage criterion, like statement and data flow coverage, but has been found to be stronger than other known criteria and is thus often referred to as a "golden standard" [6]. Mutation is unique among coverage criteria in that it not only requires a test to reach a location in the program (the mutated statement), but it also requires the mutated statement to create an error in the program state, and then propagate to an output of the program.

Mutation operators have been created for many different languages, including Fortran, Java, and C [13], [14], [15]. Mutation operators have also been defined for aspect-oriented software in AspectJ [16], [17], [18], [19], and applied to

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th International Workshop on Mutation Analysis (Mutation 2015)
978-1-4799-1885-0/15/$31.00

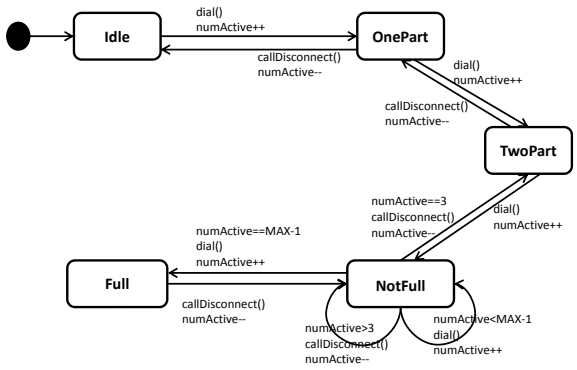

Fig. 1. A base model of a video conference system

modeling languages such as finite state machines [20], [21], [22], statecharts [23], Petri nets [24], and timed automata [25].

Mutation operators for models focus on the modeling elements and can do things like remove edges or change the target node for an edge. In this paper we show how mutation testing can be used to generate tests for aspect-oriented models, specifically targeting the cross-cutting concerns.

C. Contribution

This paper describes the use of mutation testing for aspect-oriented models, expressed as extended finite state machines. Specifically, we describe a fault model for aspect-oriented models, then use the fault model to define mutation operators. We provide an example mutant for each mutation operator, then illustrate the approach using a descriptive application of a video conference system in a timed automata implementation for UPPAAL [26].

To our knowledge, there have been no previous attempts to apply mutation testing to aspect-oriented models, or to define mutation operators targeting the special constructs that are found in such models.

The rest of the paper is organized as follows: In the next section we present a running example of a system model and example aspects. In Section III, we introduce several mutation operators for aspects-oriented models, and in Section IV we present and exemplify how these can be used in an approach for robustness testing using timed automata in the UPPAAL tool. Related work is discussed in Section V and Section VI concludes the paper.

II. EXAMPLE AOM SYSTEM

This paper uses a running example of a video conferencing system. This system has been used by Ali et al. [7], but is slightly modified here so as to better illustrate our mutation analysis approach.

We use *extended finite state machines* (EFSM) to model the behavior of systems. An EFSM is a tuple $\langle L, l_0, A, V, E\rangle$, where L is a set of vertices (or nodes), $l_0 \in L$ is the initial vertex, A is a set of events, V is a set of (finite domain) integer variables, $B(V)$ is the set of Boolean combinations (or guards) of simple constraints over V, $U(V)$ is a set of

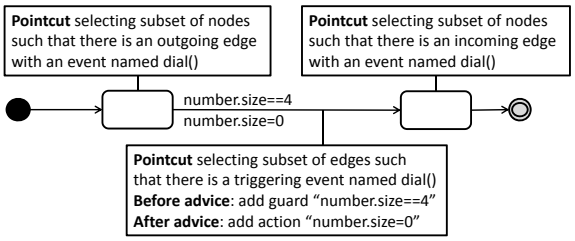

Fig. 2. A simple example of an aspect adding a guard to a subset of edges

arithmetic updates (or actions) over V, and $E \subseteq L \times B(V) \times A \times U(V) \times L$ is the set of edges. For an edge $e = \langle l, g, a, u, l'\rangle$, we use *e.event* to denote the event a, and we say that e is *l.outgoing* and *l'.incoming*. In the figures, a filled circle points at the initial vertex.

The basic operation of the video conferencing system is shown as an EFSM in the behavioral base model in Figure 1. However, a video conference system needs to be robust enough to handle disturbances during a conference session. For example, whenever the frequency of video frame loss exceeds a certain threshold, the system should recover that session. Instead of cluttering the model with recovery behavior that applies to most of its states, this behavior can best be modeled as an aspect. An aspect consists of *pointcuts*, *advice*, and *introductions* [7].

- A *pointcut* describes where the aspect *applies*, or connects to the base model. The pointcut is usually a select query that selects a set of elements such as nodes or edges, called *joinpoints* from the base model. For example, consider the aspect model in Figure 2. This aspect will add a guard (conjunct) and append an assignment to any edge in the base model where there is an event '$dial()$'. This aspect has three pointcuts, which can be defined as follows:
  - select vertex *v* where *v.outgoing* is labeled with event '$dial()$'
  - select edge *e* where *e.event* = '$dial()$'
  - select vertex *v* where *v.incoming* is labeled with event '$dial()$'
- An *advice* describes a change to be made for a pointcut. For example, the aspect in Figure 2 adds a guard and an action to the selected edges. There can be at most three advice components for each pointcut, a *before*, *around*, and *after* advice component.
  - A *before advice* component adds something to the selected elements, such as an extra guard on selected edges to be evaluated *before* traversing any of the selected edges (see Figure 2).
  - An *around advice* component replaces the selected elements with a new element.
  - An *after advice* component adds something to the selected element. For example, an extra action might be added to selected edges to be executed *after* the transition is triggered (see Figure 2).

- An *introduction* introduces a new element such as a node or an edge to the model. For example, consider the aspect model in Figure 3. This aspect represents recovery for a media failure. The leftmost pointcut selects all nodes in the base model where the event must be handled if it occurs. The rightmost pointcut selects a node to restart the system from when a timeout occurs. Apart from the two pointcuts selecting elements from the base model, there are four additional elements, three edges and one node. These additional elements are introductions and are not part of the base model.

To get a complete model of the system behavior that can be analyzed by a tool, the base model and the aspect models have to be combined by a weaver. Figure 4 shows the resulting woven model.

III. Fault Model and Mutation Operators for Aspect-Oriented Models

In this work, we assume that aspects already exist in the developed system under test or can be developed from specifications to create a test model to design tests for cross-cutting behaviors such as robustness. Robustness is an example of an emergent system property that needs to be addressed in all parts of the system. A tester who tries to design tests to cover robustness has little opportunity to distinguish robustness code from normal code. If the tester instead focuses on the behavior and separates normal and robustness behavior into different models, then it becomes easier to design the robustness tests. This can be done as a black-box approach, freeing the tester to design the models of robustness aspects based on his or her interpretation of what the system should be able to cope with in terms of disturbances and erroneous input events. This work assumes that aspects can be used or developed for software testing in one of three different approaches:

1) The software under test is already modeled as a finite state machine using aspect-orientation in a model-based development environment. The tester can apply mutation to the aspects to create a test suite that properly tests these aspects.
2) The tester only has access to a finite state machine of the normal behavior of the system. The cross-cutting behavior is handled by other parts of the system, such as the runtime system. The tester can use this model as a base model and then create a test model by designing aspects to be used together with this base model.

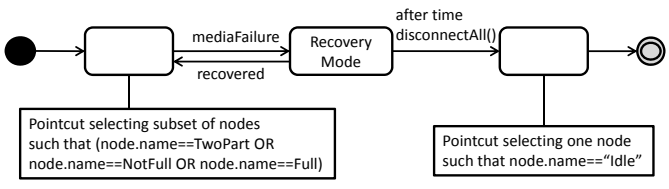

Fig. 3. A simple example of an aspect adding recovery behavior

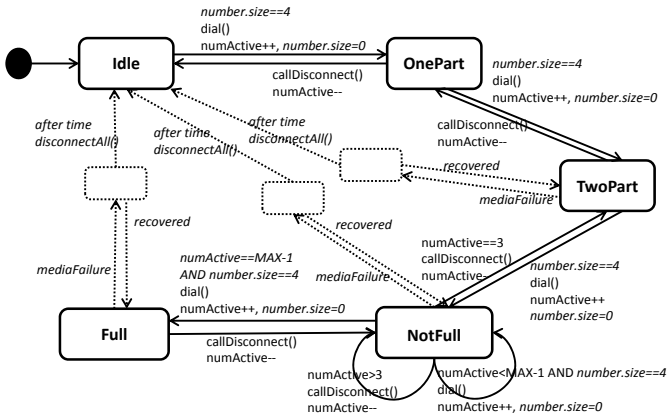

Fig. 4. The resulting model after weaving. Elements that come from the aspects are highlighted in the figure by dotted lines or italics.

3) The tester does not have a behavioral model to start with, and has to create the entire aspect-oriented test model from specifications.

Mutation operators are often designed to mimic typical mistakes such as using the wrong operator or the wrong variable name. The type of mistakes that developers make with cross-cutting concerns vary. Given approach 1, mistakes such as forgetting a pointcut, misinterpreting where a pointcut applies, are common mistakes and therefore used to design mutation operators for AspectJ [19], [27], [28], [18]. Moreover, any mistakes to the design of an aspect will propagate to all parts of the woven model or the resulting code where that aspect applies. However, the fault model is slightly different when the tester creates the aspects, as with approach 2 or 3. It is possible that the behavior that is modeled as an aspect by the tester is implemented incorrectly at one location in the software, but correctly at another. Hence, this is also addressed in our work.

Mutation operators that delete aspect elements such as pointcuts and advice elements ensure that these elements are covered by tests. For example, consider a mutant that changes the leftmost pointcut in Figure 3 so that it selects an empty set of nodes. Killing the mutant ensures that at least one test follows a path that includes a transition to a recovery mode. Again, coverage can be achieved by covering the modified element at some site where the aspect applies or by ensuring that all of the sites are covered by modifying one joinpoint at a time in different mutants. For example, mutants can remove node Full, NotFull, and TwoPart from the leftmost pointcut in Figure 3. This approach is useful with approaches 2 or 3.

Mutants for approach 1 can often be created by making small syntactic changes to the aspect models, but creating mutants for approach 2 and 3 might require more effort. This is because a syntactic change made in an aspect applies to all sites in the woven model where the aspect is applied. To make a syntactic change at a single joinpoint, it is necessary to iterate over the joinpoints and treat them in isolation. Hence, a large change to the aspect-oriented model might result in a small syntactic change at a single element in the resulting

woven model.

A. Suggested Mutation Operators for Aspects

This section suggests several mutation operators for aspect-oriented models. The idea is to focus on the aspects and the elements (pointcuts, advice and introductions) that they may consist of. These are syntactic elements or structures that we do not find in other models and hence are not specifically targeted by other mutation-based approaches for models. We describe the semantics of each mutation operator and then show examples of the resulting mutants. The example mutants are all shown after weaving to show their affect on the woven model. Actually, however, the mutants are applied to the aspect models before weaving.

The goal with these mutation operators is not just to identify tests that detect faults related to robustness, but also to identify a test suite that covers robustness. This includes tests that reach and trigger execution of robustness code (including the code for recovery). Different mutation operators will exercise coverage in different detail.

B. Mutation Operators for Pointcuts

Mutation operators for pointcuts focus on the select queries (or pointcut descriptors [29]) that define the pointcuts.

Pointcut deletion (PCD): This operator has the semantics of not selecting any element for the pointcut. For example, consider the aspect in Figure 3. This aspect has two pointcuts, so PCD will create two mutants. Figures 5 and 6 show the result after weaving with these mutated aspects. The mutant in Figure 5 has a recovery mode that cannot be reached. This mutant will be killed by any test where a media failure occurs when a session has at least two participants. The mutant in Figure 6 has a recovery mode that can only be exited if the recovery is successful. This mutant will be killed by any test where a media failure occurs when a session has at least two participants and the system fails to recover within the given time frame.

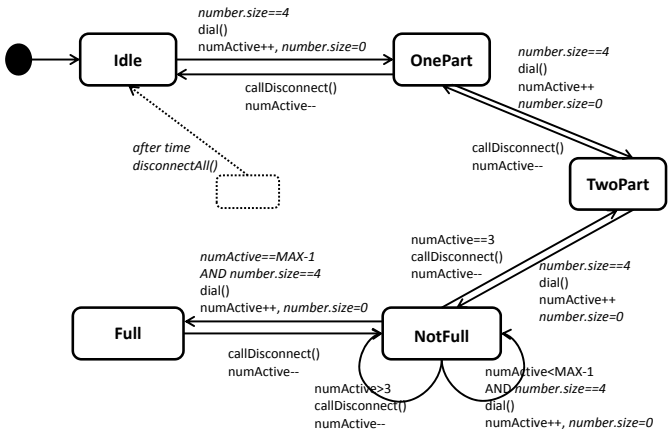

Fig. 5. The resulting woven mutant when applying PCD to the leftmost pointcut in the recovery aspect

Pointcut strengthening (PCS): We can strengthen a pointcut if the select query uses any of the operators *OR*, $\leq$, or $\geq$.

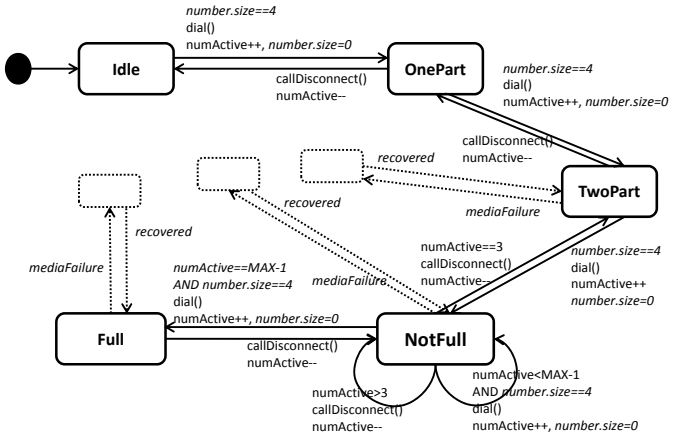

Fig. 6. The resulting woven mutant when applying PCD to the rightmost pointcut in the recovery aspect

An *OR* is replaced by an *AND*, $\leq$ is replaced by $<$, and $\geq$ is replaced by $>$. Furthermore, for each operand in an *OR*-expression there should be a mutant where that operand is left out. If elements of the correct type exist that are selected by the original pointcut but not by the mutated pointcut, this will result in a reduced set of joinpoints in the mutant. Figure 7 shows the resulting mutant when the pointcut that selects nodes where *node.name==TwoPart OR node.name==NotFull OR node.name==Full* has been mutated to select nodes where *node.name==NotFull OR node.name==Full*. This mutant will be killed by a test that triggers error handling in the original but not in the mutant, that is, when media failure occurs during a two part session.

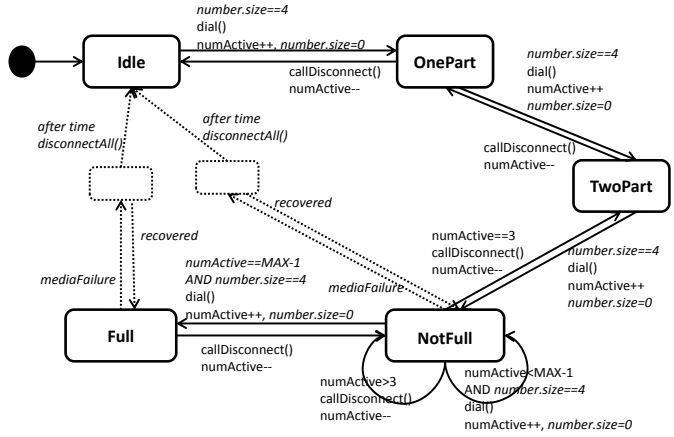

Fig. 7. Resulting woven mutant when applying PCS to the leftmost pointcut in the recovery aspect

Pointcut weakening (PCW): We can weaken a pointcut if the select query uses any of the operators *AND*, $<$, or $>$. An *AND* is replaced by an *OR*, $<$ is replaced by $\leq$, and $>$ is replaced by $\geq$. Furthermore, for each operand in an *AND*-expression there should be a mutant where that operand is deleted. Given elements of the correct type for which the original select query is false and the mutated is true, this will

result in more joinpoints in the mutant. This mutant will be killed by a test that executes the aspect due to the weaker condition in the mutant model but not in the original model.

Joinpoint deletion (JPD): This operator has the semantics of excluding *one joinpoint at a time* from the pointcut before weaving. For example, consider the leftmost pointcut in the aspect shown in Figure 3. This pointcut selects three nodes in the base model. Hence, there will be three mutants for this specific pointcut: (i) M1, where the aspect model cannot be reached from state TwoPart, (ii) M2, where the aspect model cannot be reached from state NotFull, and (iii) M3, where the aspect cannot be reached from the state Full. M3 can only be killed by a test where media failure occurs when there is a maximum number of connected calls. M1 can only be killed by a test where media failure occurs where the current session has exactly two participants. Similarly, applying JPD to the pointcut that adds a guard and an action in the aspect shown in Figure 2, will give five mutants that all miss the guard and action on one of their edges. This type of mutation operator is useful for cases where the software under test is not already modeled with aspect orientation, so the implementation may differ at the various joinpoints (cf. JPI and JPR).

Joinpoint introduction (JPI): This mutation operator adds extra joinpoints to the pointcut. It applies to all elements of the same type as the joinpoints included in the original pointcut. For example, the original pointcut that selects all edges such that there is a trigger named *dial()*, selects five of the ten edges in the base model (see Figure 1). Five mutants are created, one for each edge that are not selected by the original pointcut. For example, there will be a mutant where the guard *number.size==4* is added to the edge from state *OnePart* to state *Idle* as well as to all edges included in the original pointcut.

Joinpoint replacement (JPR): This mutation operator combines JPD and JPI by creating all pair-wise combinations with respect to elements that are selected by a pointcut and the rest of the elements that are of the same type. Each mutant differs from the original by having one joinpoint replaced by an element of the same type. For example, the leftmost pointcut in the recovery aspect (Fig. 3) selects three of the five nodes. Hence, there will be six JPR mutants for this pointcut. Figure 8 shows an example of a JPR mutant where the node OnePart is selected instead of node TwoPart.

C. Mutation Operators for Advice

We have two approaches for designing advice mutation operators: (*i*) the advice is mutated at all its sites in a single mutant, and (*ii*) one mutant is created for each place where the advice applies. If a pointcut has more than one piece of advice, for example, a *before* advice and *after* advice, these will be mutated separately regardless of the approach. When the first approach is used, at most three mutants will be created for each pointcut (one per advice) and the mutation operator will apply at all elements pointed out by the pointcut. With the second approach, the advice should only be mutated at one of the sites pointed out by the pointcut at a time. Given

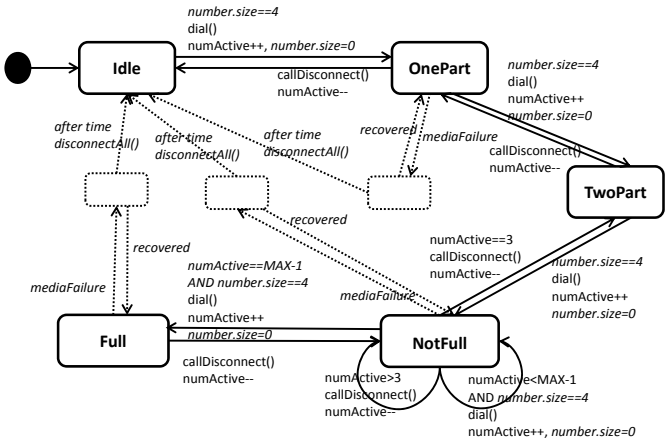

Fig. 8. One of the JPR woven mutants

J joinpoints, the second approach means that there will be at most 3**J* mutants.

Advice deletion at pointcut (ADP): Consider Figure 2. The pointcut with advice selects five edges and applying ADP to this aspect model will create two mutants, one for the before advice and one for the after advice. The first will not add the guard *number.size==4* to any of the five edges. A test can kill this mutant if any of the calls tries to connect with an incorrect number. The second mutant will not add the assignment *number.size=0* to any of the five edges. This mutant is trivial since it will be killed by any test that visit any of these edges.

Advice deletion at joinpoint (ADJ): Consider Figure 2 again. Applying ADJ to this aspect model will give ten mutants. Five mutants will delete the guard *number.size==4* on one of their edges and five mutants will delete the assignment *number.size=0* on one of their edges. For example, one mutant will delete the guard on the edge (TwoPart, NotFull). This mutant can be killed by a test if any of the calls tries to connect with an incorrect number when the session has exactly two participants.

Advice introduction at pointcut (AIP): Consider Figure 2 again. This is the only aspect model in our example that has a pointcut with advice. It has a before and an after advice. It is also the only pointcut that selects a set of edges. Assume that there is a second pointcut that also selects a set of edges and has no before or after advice. AIP would then create one mutant where the before advice is copied to the second pointcut and one mutant where the after advice is copied to the second pointcut. AIP applies to any pair of pointcuts that are of the same type (i.e., selecting the same type of model elements), in the same scope, and with a type of advice (e.g., before) that originally only existed in one of them.

Advice introduction at joinpoint (AIJ): AIJ is the same as AIP except it applies to a single joinpoint in each mutant.

Advice replacement at pointcut (ARP): It is possible to replace a before advice by another before advice. The replacing advice should be an existing advice of the same type

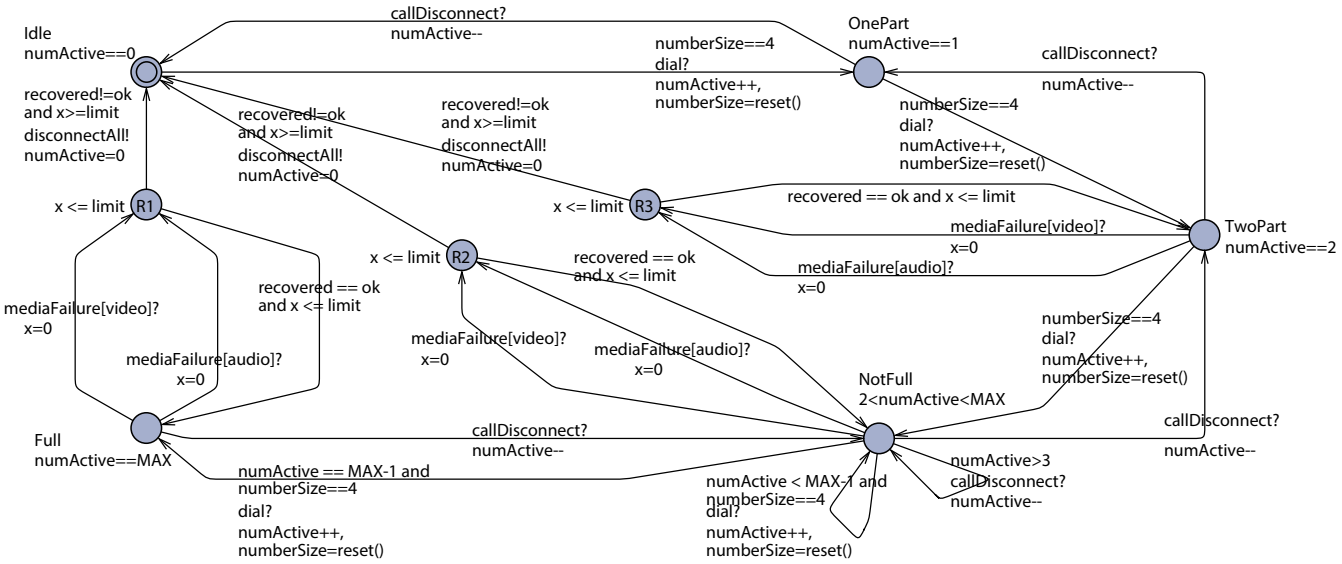

Fig. 9. A timed automata model of the woven system

as the advice being replaced. For example, a before advice for a pointcut selecting a set of nodes is replaced by the before advice for another pointcut that selects sets of nodes, has a before advice, and is in the same scope.

Advice replacement at joinpoint (ARJ): ARJ is the same as ARP except it applies to a single joinpoint in each mutant.

Traditional operators: Several traditional mutation operators could also apply to advice elements. Three likely candidates are (i) ROR, which replaces relational operators in constraints and guards by other relational operators, (ii) COR, which replaces logic operators in constraints and guards by other logic operators, and (iii) AOR, which replaces arithmetic operators in actions by other arithmetic operators. We do not define these mutation operators here since they already exist and would apply to an aspect model in exactly the same way as to existing languages. Ferrari et al. [30] defined some mutation operators for source code generated from aspect-oriented source code. However, the syntax as well as the semantics of an advice in an aspect-oriented model differ depending on what type of model element it is applied to, and whether the advice is a before, around or after advice. Hence, these mutation operators would usually generate mutants that are syntactically incorrect.

D. Mutation Operators for Introductions

Introduction deletion (IDL): An IDL mutant deletes each introduction element in the aspect model in turn. IDL mutants are killed by any test that visits the deleted element. Our aspect-oriented model has four introductions–three edges and one node. Hence, IDL will give four mutants. For example, there will be one mutant where the only edges leading from a recovery mode lead to a timeout.

Discussion: Just as advice elements, introductions can come with constraints, guards, actions etc. containing relational, arithmetic or logic expressions. The same traditional mutation

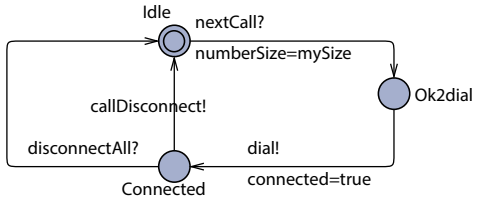

Fig. 10. A timed automata model of a conference participant

operators that we suggest for advice therefore, also apply to introductions. Since there is no overlap between introductions and pointcuts, there is no redundancy between applying e.g., ROR both to advice and to introductions.

We have discussed the possibility of applying mutation operators to single joinpoints rather than the pointcut. This is fairly straightforward since a pointcut describes a set of joinpoints and each advice is mapped to a specific pointcut. It is therefore possible to iterate over a set of joinpoints and treat them differently by modifying the aspect models. However, introductions are elements in the aspect model that have no corresponding elements in the base model, so there is no set to iterate over before the weaving process. A more fine-grained mutation approach for introductions therefore requires integration with or at least control of the weaver and is not addressed here. Mutating an introduction will therefore affect all parts of the woven model where that element is introduced.

IV. Application to Robustness Testing

This section discusses the use of our suggested approach for robustness testing of a system modeled in timed automata. Timed automata are finite state machines that are extended with clocks. Timed automata models can be executed and therefore tested as well as verified by model checkers. We

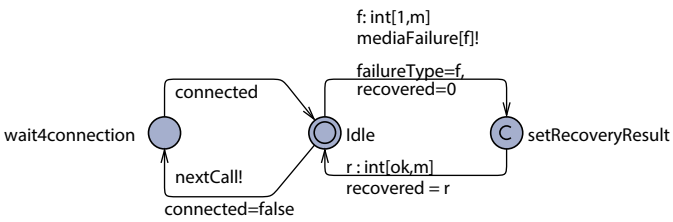

Fig. 11. A timed automata model of a driver triggering new calls and generating disturbances

translated the video conference system to timed automata for UPPAAL (Figure 9) and used the UPPAAL model checker to verify its behavior [26]. In addition to the woven system, we also have models that implement the system's environment, including calls and disturbances that cause media failures (Figures 10 and 11). We used the UPPAAL simulator to execute the test scenarios and to generate the traces that we discuss in our examples.

Robustness is defined as *"The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions"* [31]. Systems can be stressed in many different ways. Examples of such disturbances are frame loss, noise, synchronization mismatches and lost connections [7]. Each type should of course be identified and addressed by the aspect models. A major difference between the previous examples and the timed automata model used here is, therefore, that the timed automata model distinguishes between two types of failures: audio and video failure. This example, with two types of failures, is used to illustrate the approach. With all types of media failures included, a fully woven model would be too cluttered to show in a figure.

A. Example System

Figure 9 shows a timed automata model of the woven video conference system. Figure 10 shows a timed automata model that describes the behavior of participants in the video conference. A participant connects to the system by taking the transition labeled *dial!* from *Ok2dial* to *Connected*. This transition is synchronized with a transition labeled *dial?* in the system shown in Figure 9. In the same way, a transition from *Connected* to *Idle* is triggered by a synchronization on *callDisconnect*. A *disconnectAll* is a broadcast signal triggered by the system. All participants that are connected when this broadcast occurs will take a transition to their *Idle* state.

Figure 11 shows a timed automata model of a simple driver for the video conference system. The driver triggers new calls as well as media failures of different types. *failureType* is set by selecting a value between 1 and m, where m is the number of failure types. Each possible value of *failureType* can be mapped to a specific type of disturbance that this system should handle. As mentioned, our example system has only two types of disturbances, audio and video. When the driver triggers a media failure, it immediately continues by setting the variable *recovered* to a value between 0 and m, where

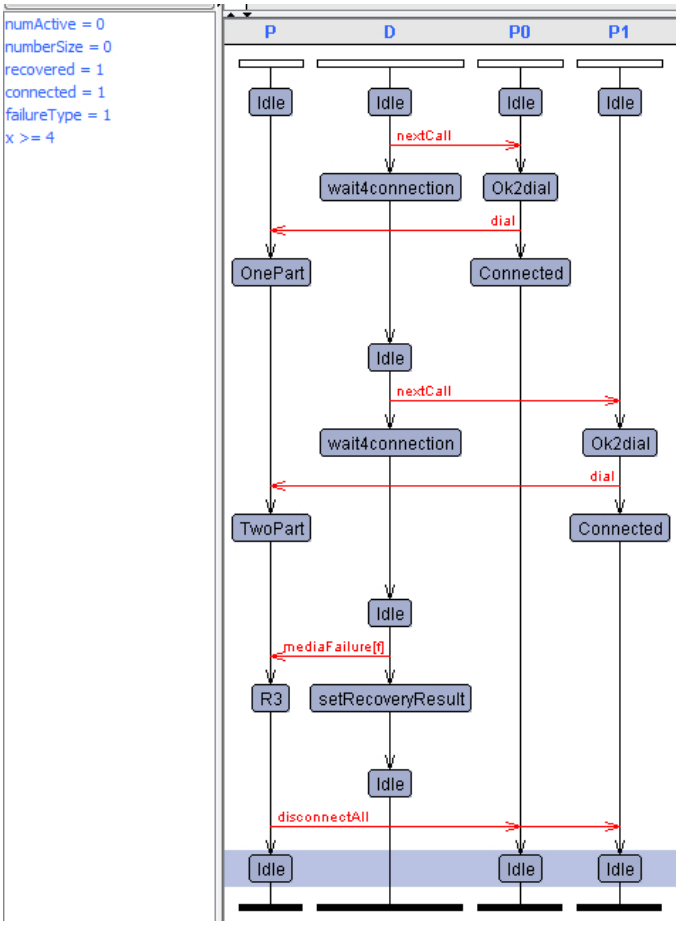

Fig. 12. A graphical view of a trace

$0 == ok$. A trace where *failureType* is set to 1 maps to a trace with an audio failure. A trace where *recovered* is set to 1 maps to a test where the audio failure persists to verify that the system implemented a timeout and can handle it by resetting the system. A trace where *failureType* is set to 2 and *recovered* is set to 1 maps to a test where there is a video failure followed by an audio failure that is persistent.

B. Example Mutants and Tests

Here, we show three example robustness mutants and discuss their use. The mutants can be used to create tests or to evaluate a set of tests with respect to their mutation score. In both approaches, it is the trace from the environment (driver and participants) that are used. Consider the trace in Figure 12. This is a graphical view of a trace of an execution of the processes P, D, P0 and P1, where P is the system shown in Figure 9 and D is the driver shown in Figure 11. P0 and P1 are instances of the template participant shown in Figure 10. The boxes in Figure 12 show states, the vertical arrows show transitions, and the horizontal arrows show synchronization between processes. In the upper left corner of the figure is a list of variables and their values after the last transition, where x is a clock variable that models the timer. The time limit is set to four. This specific trace shows two connections followed

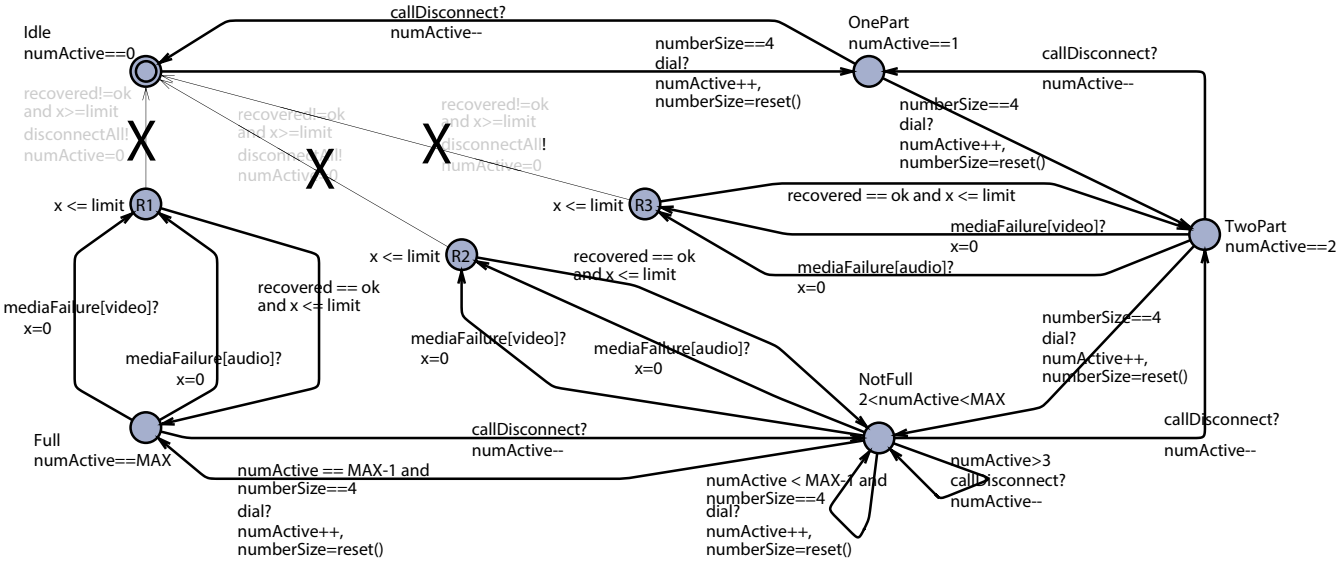

Fig. 13. Timed automata mutant where a pointcut is deleted (PCD)

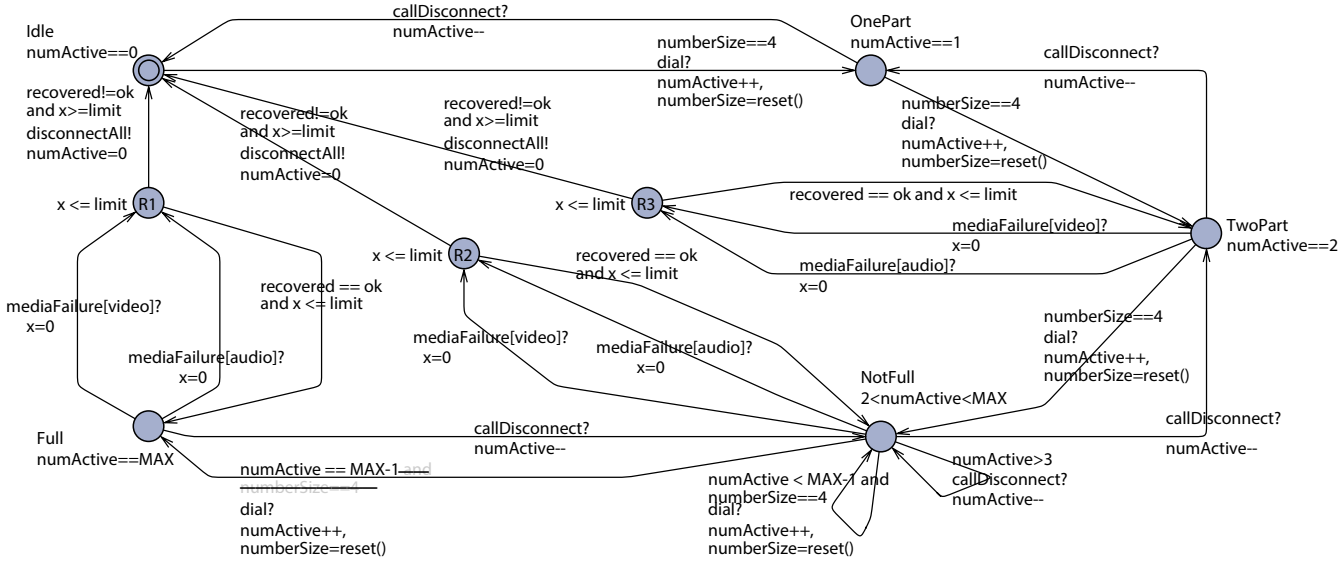

Fig. 14. Timed automata mutant where an advice is deleted at a single joinpoint (ADJ)

by an audio failure (*failureType*=1) that persists (*recovered*=1) and leads to a timeout ($x >= 4$) and a *disconnectAll*.

The trace in Figure 12 is a result of executing a test scenario in the UPPAAL simulator. Consider the PCD robustness mutant in Figure 13. The crossed and grayed out portions in the figure are deleted in the mutant. Enforcing the same trace in the mutant model will lead to a deadlock since it is not possible to take the last transition (P:R3,Idle) in the mutant model so neither P0 or P1 can receive the broadcast signal and take the transition to their Idle states. Hence, this trace represents a test that kills this mutant.

Consider the ADJ robustness mutant in Figure 14. The mutant does not have the guard *numberSize==4* (grayed out and struck over) on the edge between *NotFull* and *Full*. Hence, it is possible to get a trace to a state where the system is at node *Full* and *numberSize!=4* when executing the mutant. This state is not reachable in the original model and the test scenario would actually lead to a deadlock state. On the other hand, the mutant cannot reach this deadlock state. By translating candidate test scenarios to traces and enforcing these traces on the original and mutant models, we can identify which traces kill which mutants and create a strong test suite or evaluate an existing test suite with respect to its mutation adequacy score.

Finally, consider the IDL robustness mutant in Figure 15. This mutant does not have an edge to recovery mode for the failure type video (again showed in figure as grayed out and marked). Hence, this mutant is killed by a test with a media failure of this type when at least two participants are active.

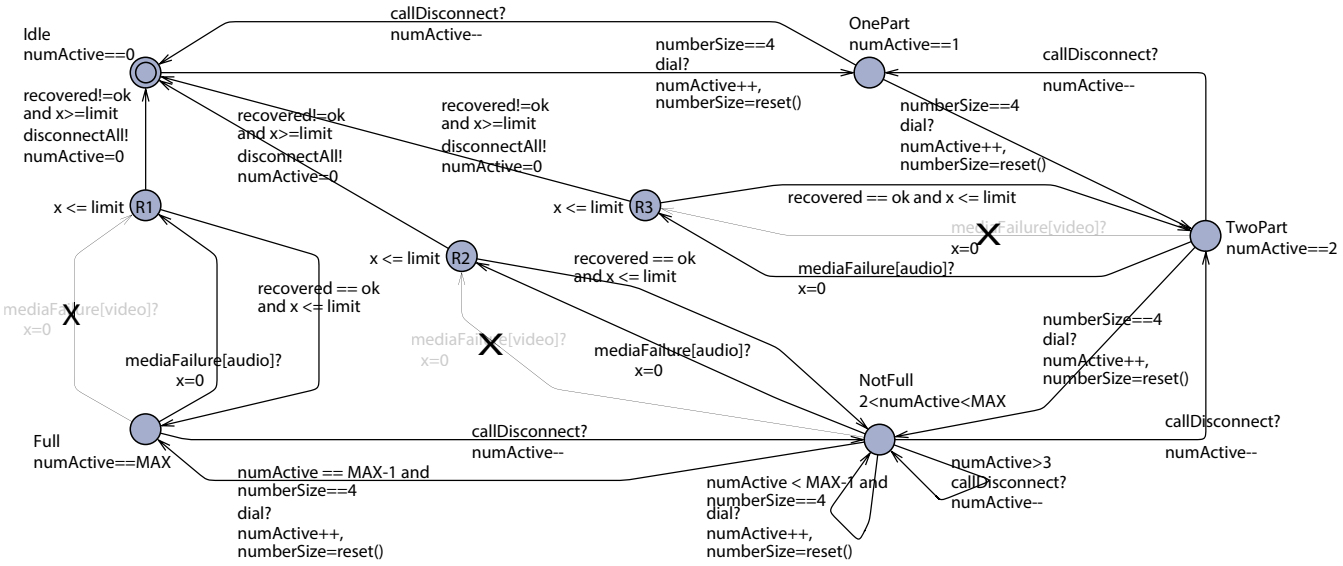

Fig. 15. Timed automata mutant where an introduction is deleted

V. Related Work

Lemos et al. [29] discuss testing for joinpoints that are unintended or neglected, which is part of what we do. However, their strategy is based on a step-wise integration of the joinpoints at the source-code level while we are focusing on aspect models before they are integrated to the base model.

Fault models for aspect-oriented programs have been suggested [28], [27], [19] and mutation operators have been defined for AspectJ [17], [18], [16]. The major difference between our work and theirs is that we define mutation operators fit to use at a model-level and which can be applied in a black-box manner without knowledge of the actual implementation.

Ali et al. [7], applies aspect-oriented modeling to reduce the complexity of robustness modeling in the context of UML state machines. A RobUstness Modeling Methodology (RUMM) for modeling of robustness behavior as aspects is presented. Ali et al. [32] used the methodology together with search algorithms in an industrial case study. We adopt their aspect-oriented methodology for robustness modeling but we put our focus on suitable mutation operators for testing of robustness in a mutation-based testing framework.

Related work on model-based testing of finite state and timed automata include Springintveld et al. [33], that showed the problem of timed trace equivalence testing of (deterministic) timed automata to be theoretically possible. In [34], Nielsen and Skou proposed a model-based test technique for event-recording automata and in [35] Hessel et al., present a model-based technique for generating optimal tests coverage criteria using the Uppaal tool. Our work is related to these works as we use timed automata and Uppaal-generated traces for test generation, but different in the sense that we generate test based on mutants rather than coverage criteria.

VI. Conclusions

This paper presents a novel technique to test software developed with aspect-oriented models. The technique works by mutating cross-cutting concerns at the model level. Thirteen novel mutation operators are defined that modify the model. The operators are based on the unique features of aspect-oriented modeling and for the most part induce changes that mirror mistakes that AO modelers make. Tests are designed at the abstract level to kill the mutants, then transformed to concrete tests to run on the software. These tests evaluate both the modeling of the cross-cutting concerns, and the weaving process that creates the resulting implementation. This paper defines the mutation operators and illustrates how they are used to design tests.

This technique is novel and useful for the same reason that AO modeling is effective: it allows the tester to design tests to evaluate the cross-cutting concerns independently of the rest of the software. Instead of designing tests for the software, and mixing the evaluation of the cross-cutting concerns with the evaluation of the primary behavior, the tester can focus on one thing at a time. It is therefore also possible for the tester to adjust the test effort for different aspects, depending on the criticality of the cross-cutting concern it addresses. Thus, this is a classic divide and conquer strategy. Additionally, even though the cross-cutting behaviors are repeated potentially many times in the implementation, the tester only has to design tests once. Tests that are designed based on one single point in an aspect model can test multiple points distributed in the software.

A. Future Work

This paper proposes an approach to test cross-cutting concerns by mutating aspect-oriented models. This paper illustrates the approach to robustness testing by applying it to a

video conferencing system. The approach and the mutation operators need to be valided empirically, so we plan a larger-scale industrial case study to evaluate the mutation operators with respect to their effectiveness and efficiency for robustness testing. Based on prior work [27], [19], we already plan one additional operator to replace one pointcut with another.

Another question that we plan to address empirically is whether each AspectJ mutation operator should result in multiple code changes or in just one code change. For example, the PCD mutant in Figure 13 deletes one pointcut in the model, which windes up removing three edges in the graph. Such large scale changes might result in easy to kill mutants (*trivial*), so we plan experiments to decide whether to turn this into three separate mutants. This could be done by applying the mutation operators during the weaving.

In addition, the mutation operators for aspect-oriented models defined in this paper will be evaluated with respect to feasibility for test generation using the UPPAAL model checker. This process is currently semi-automatic. To scale to larger systems, a fully automated tool to weave the mutants and generate the mutation-based tests is required.

VII. ACKNOWLEDGMENT

This work was funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

REFERENCES

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[2] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[3] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, apr 1994.

[4] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," in *Lecture Notes on Concurrency and Petri Nets*, W. Reisig and G. Rozenberg, Eds. Springer-Verlag, 2004.

[5] S. Pimont and J. Rault, "A software reliability assessment based on a structural behavioral analysis of programs," in *Proceedings of the Second International Conference on Software Engineering*, 1976, pp. 486–491.

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[7] S. Ali, L. Briand, and H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems," *Softw. Syst. Model*, vol. 11, no. 4, pp. 633–670, 2012.

[8] A. Aldini, R. Garrieri, F. Martellini, and J. Jürjens, "Model-based security engineering with UML," Springer, 2005.

[9] J. Péreza, N. Ali, J. Carsi'b, I. Ramosb, B. Álvarezc, P. Sanchez, and J. Pastorc, "Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems," *Inf. Softw. Technol.*, vol. 50, no. 9-101, pp. 969–990, 2008.

[10] T. Cottenier, A. Berg, and T. Elrad, "Stateful aspects: The case for aspect-oriented modeling," in *10th International Workshop on Aspect-Oriented Modeling*. ACM, 2007.

[11] K. Hameed, R. Williams, and J. Smith, "Separation of fault tolerance and non-functional concerns: Aspect oriented patterns and evaluation," *Journal of Software Engineering and Applications*, vol. 2, no. 4, pp. 303–311, 2010.

[12] R. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[13] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Software Practice & Experience*, vol. 21, no. 7, pp. 685–718, jun 1991.

[14] Y. S. Ma, Y. R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2002, pp. 352–363.

[15] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proceedings of ObjectDays*, 2000, pp. 9–12.

[16] F. Ferrari, J. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *1st International Conference on Software Testing, Verification and Validation*, April 2008, pp. 52–61.

[17] M. Mortensen and R. T. Alexander, "An approach for adequate testing of AspectJ programs," in *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, 2005.

[18] P. Anbalagan and T. Xie, "Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs," in *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION06)*, 2006, pp. 3–8.

[19] F. Wedyan and S. Ghosh, "On generating mutants for AspectJ programs," *Information and Software Technology*, vol. 54, no. 8, pp. 900–914, 2012.

[20] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, "Mutation analysis testing for finite state machines," in *Fifth International Symposium on Software Reliability*, November 1994, pp. 220–229.

[21] S. Batth, E. Vieira, A. Cavalli, and M. Uyar, "Specification of timed EFSM fault models in SDL," in *27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, June 2007, pp. 50–65.

[22] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1804–1818, 2009.

[23] M. Trakhtenbrot, "New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models," in *Third IEEE Workshop on Mutation Analysis (Mutation 2007)*. IEEE Computer Society, 2007, pp. 151–160.

[24] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong, "Mutation testing applied to validate specifications based on Petri nets," in *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*, 1996, pp. 329–337.

[25] R. Nilsson, J. Offutt, and S. F. Andler, "Mutation-based testing criteria for timeliness," in *Proceedings of the 28th International Conference on Computer Software and Applications (COMPSAC)*. IEEE, 2004, pp. 306–311.

[26] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, Oct. 1997.

[27] R. Delamare, B. Baudry, S. Ghosh, and Y. Le Traon, "A test-driven approach to developing pointcut descriptors in AspectJ," in *International Conference on Software Testing, Verification, and Validation (ICST 2009)*. IEEE, 2009, pp. 376–385.

[28] J. Bsekken and R. T. Alexander, "A candidate fault model for AspectJ pointcuts," in *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*. IEEE, 2006, pp. 169–178.

[29] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes, "Testing aspect-oriented programming pointcut descriptors," in *Proceedings of the 2nd workshop on testing aspect-oriented programs*. ACM, 2006, pp. 33–38.

[30] F. Ferrari, A. Rashid, and J. Maldonado, "Design of mutant operators for the AspectJ language," Technical report Version 1.0, University of Sao Carlos, Sao Carlos, 2011.

[31] "IEEE Standard glossary of software engineering terminology," IEEE Std 610.12-1990, 1990.

[32] S. Ali, L. C. Briand, A. Arcuri, and S. Walawege, "An industrial application of robustness testing using aspect-oriented modeling, UML/MARTE, and search algorithms," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 108–122.

[33] J. Springintveld, F. Vaandrager, and P. R. D'Argenio, "Testing timed automata," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 225–257, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(99)00134-6

[34] B. Nielsen and A. Skou, "Testing timed automata," *Software Tools for Technology Transfer*, vol. 5, no. 1, pp. 59–77, 2003.

[35] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Formal methods and testing," R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Testing Real-time Systems Using UPPAAL, pp. 77–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806209.1806212