<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper published in *Information and Software Technology*. This paper has been peer-reviewed but does not include the final publisher proof-corrections or journal pagination.

Citation for the original published paper (version of record):

Lindstrom, B., Offutt, J., Sundmark, D., Andler, S F., Pettersson, P. (2017)

Using mutation to design tests for aspect-oriented models.

Information and Software Technology, 81: 112-130

<http://dx.doi.org/10.1016/j.infsof.2016.04.007>

Access to the published version may require subscription.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-34018>

Using Mutation to Design Tests for Aspect-Oriented Models

Birgitta Lindström^{a,*}, Jeff Offutt^b, Daniel Sundmark^c, Sten F. Andler^a, Paul Pettersson^d

^aUniversity of Skövde, Skövde, Sweden

^bGeorge Mason University, Fairfax VA, USA

^cSwedish Institute of Computer Science, Kista, Sweden

^dMälardalen University, Västerås, Sweden

Abstract

Context: Testing for properties such as robustness or security is complicated because their concerns are often repeated in many locations and muddled with the normal code. Such “cross-cutting concerns” include things like interrupt events, exception handling, and security protocols. Aspect-oriented (AO) modeling allows developers to model the cross-cutting behavior independently of the normal behavior, thus supporting model-based testing of cross-cutting concerns. However, mutation operators defined for AO programs (source code) are usually not applicable to AO models (AOMs) and operators defined for models do not target the AO features.

Objective: We present a method to design abstract tests at the aspect-oriented model level. We define mutation operators for aspect-oriented models and evaluate the generated mutants for an example system. **Method:** AOMs are mutated with novel operators that specifically target the AO modeling features. Test traces killing these mutant models are then generated. The generated and selected traces are abstract tests that can be transformed to concrete black-box tests and run on the implementation level, to evaluate the behavior of the woven cross-cutting concerns (combined aspect and base models). **Results:** This paper is a significant extension of our paper at Mutation 2015. We present a complete fault model, additional mutation operators, and a thorough analysis of the mutants generated for an example system. **Conclusions:** The analysis shows that some mutants are stillborn (syntactically illegal) but none is equivalent (exhibiting the same behavior as the original model). Additionally, our AOM-specific mutation operators can be combined with pre-existing operators to mutate code or models without any overlap.

Keywords: Model-based testing, Aspect-oriented model, Mutation testing

1. Introduction and Background

Model-based development is gaining widespread use in the software industry. Models provide a graphical view of software behavior that developers find intuitive. In addition, certain types of models, such as state charts [23], Petri nets [47], and timed automata [5, 9] are useful for analysis and verification purposes. Such models can be used by model checkers to verify properties, e.g., to guarantee that a model is free from deadlocks, or to infer 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 [6, 48]. Consequently, developers can better

understand and analyze complex behavior by modeling software behavior.

1.1. Aspect-Oriented Modeling

One proposed approach to managing complex behavioral models is to separate cross-cutting concerns from the main behavior by using *aspect-oriented modeling* [4, 13, 22, 28, 46]. A *cross-cutting concern* applies throughout multiple locations in 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

*Corresponding author: Birgitta Lindström, University of Skövde, Box 408, 541 28 Skövde, Sweden, Tel.: +46 500 448368

Email address: birgitta.lindstrom@his.se (Birgitta Lindström)

normal behavior of the system in a *base model*, leaving the cross-cutting concerns to be described in separate *aspect models*. By modeling these concerns separately, the behavioral models become cleaner and less cluttered. This makes it easier for a tester to focus on one concern at a time and adjust the level of testing to the level of criticality for the specific concern. An AOM tool then *weaves* the base model and the aspect models together, in a predefined order, to create a complete behavioral model of the system. The woven model is complex but neither the developer nor the tester need to view it. It is generated and used by model-based tools for model-checking, transformation, and test execution.

1.2. Mutation-Based Testing

We propose to mutate aspect-oriented models (AOMs) 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* [15]. 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 exhibit different behavior from the original version, called *killing* (or detecting) 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 will kill, called 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 “gold 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 that error to an output of the program.

Mutation operators have been created for many different languages, including Fortran, Java, and C [2, 30, 31, 37]. Mutation operators have also been defined for aspect-oriented programs (source code) in AspectJ [7, 19, 40, 53]. AspectJ is an aspect-oriented programming (AOP) extension to Java and has become a *de facto* standard for AOP [11]. However, mutation operators defined for aspect-oriented source code are usually not appropriate for models such as finite state machines or state charts.

Mutation operators have also been defined for modeling languages such as finite state machines [8, 18, 26], state charts [32, 52], Petri nets [17], and timed automata

[44]. Mutation operators for models focus on the modeling elements and can do things like remove an element or change the target node for an edge. However, these model-level mutation operators do not target aspect-oriented features such as pointcut descriptors (see Section 3).

1.3. Contributions

We describe the use of mutation testing for aspect-oriented models that are expressed as extended finite state machines. Specifically, we describe a fault model for aspect-oriented models and then use the fault model to define mutation operators. We provide an example mutant for each mutation operator and then illustrate the approach using a descriptive application in the form of a video conferencing system in a timed automata implementation for UPPAAL [33].

We include and extend work published at the Mutation workshop 2015 [36]. In addition to previous work, we present an elaborated fault model, additional mutation operators and a thorough analysis of the generated mutants. We have also included a definition of timed automata, which is used in our work.

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

Our proposed mutation operators are evaluated and compared to traditional operators through a thorough analysis with respect to the generated, stillborn, equivalent, redundant, and duplicated mutants we get as we apply mutation to a small video conferencing system. This approach of evaluation is both a strength and a limitation. We do not present any mutation score for the tests we get for this system. Instead, we analyze the mutants we get for the system and show that our new mutation operators gave no overlap to traditional mutation operators, no equivalent or redundant mutants and very few duplicates compared to traditional mutation operators. We got 10% and 1% duplicates respectively for the two sets of new mutation operators compared to 37% for the traditional mutation operators. The reason for having two sets of our proposed mutation operators in the evaluation is that some of them are not meant to be used together. This will be further explained in Section 3

The remainder of this paper is organized as follows: Section 2 presents a running example of a system model and example aspects. Section 3 presents a fault model for aspect models. Section 4 introduces several mutation operators for such models. We also propose using

a large set of pre-existing mutation operators in combination with our new AOM-specific operators.

Section 5 shows examples of how the mutation operators can be used in an approach for robustness testing using timed automata in the UPPAAL tool. Section 6 presents an analysis of applying the mutation operators. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. Example AOM System

This paper uses a running example of a video conferencing system. This system has been used by Ali et al. [4], but is slightly modified here so as to better illustrate our mutation analysis approach. In addition to the four states *Idle*, *OnePart*, *NotFull* and *Full* in the state chart for the base model used by Ali et al. [4], we also have a state *TwoPart*. The extra state and the transitions to and from it do not change the semantics of the base model or the woven system. The aspect models are identical to the original, except for the notation in the illustrations, where we have chosen to use informal language for simplicity. We have also introduced an additional advice to reset the variable *number.size* after each use. The additional advice is only added to better illustrate our mutation analysis approach and has no practical implication to the system behavior since this particular variable will always get a new value before it is read.

We use *extended finite state machines* (EFSMs) 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 (here called nodes), $l_0 \in L$ is the initial node, A is a set of events, V is a set of (finite domain) integer variables. Assuming $B(V)$ is the set of Boolean combinations (or guards) of simple constraints over V and $U(V)$ is the set of arithmetic updates (or actions) over V , then $E \subseteq L \times B(V) \times A \times U(V) \times L$ is a 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 $l'.incoming$, g is a guard in $B(V)$ and u is an update in $U(V)$. 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 conferencing 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. As we have mentioned, 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* [4].

Figure 1: A base model of a video conferencing system

Figure 2: A simple example of an aspect adding a guard to a subset of edges

A *pointcut* is a set of elements where the aspect *applies*, or connects to the base model. The *pointcut descriptor* is usually a select query written in a formal language and is associated with a single element in the aspect model. The pointcut descriptor selects a non-empty 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 to any edge in the base model where there is an event “*dial()*”. Since an edge cannot be modeled without the nodes it is attached to, this aspect model has three pointcuts and the pointcut descriptors can be defined as follows:

- select vertex v where $v.outgoing$ is labeled with event “*dial()*”
- select edge e where $e.event = \text{“dial()”}$
- select vertex v where $v.incoming$ is labeled with event “*dial()*”

We have chosen to use informal language in our illustrations since the formal expressions would be tightly connected to the modeling language and weaver (the tool combining the base model with the aspect model).

Table 1: Definition of before, around and after advice [4]

Modeling element	Before Advice	Around Advice	After Advice
Node	Adding a constraint to be evaluated before entry to joinpoint nodes	Replacing the joinpoint nodes with a new node	Adding a constraint to be evaluated on leaving the joinpoint nodes
Edge	Adding a guard to joinpoint edges. If a guard already exists, the additional guard is joined to the existing guard	Replacing joinpoint edges with a new edge	Adding an effect with one or more actions to joinpoint edges
Event	Not applicable	Replacing events on joinpoint edges with a new event	Not applicable
Guard and invariant	Adding an additional constraint (conjunct) to the guards (or invariants) selected by the pointcut	Replacing one or more guards (or invariants) selected by a pointcut with a new guard (or invariant)	Same as before advice
Action	Adding an action to be executed before the joinpoint actions	Replacing the joinpoint actions by a new action	Adding an action to be executed after the joinpoint actions

An *advice* is tied to a pointcut and describes a change to be made at each joinpoint that the pointcut includes. For example, the aspect in Figure 2 adds a guard conjunct and an action appended to the selected edges. An advice can be of type *before*, *around*, or *after*. For each pointcut, there may be at most one advice of each type.

1. 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).
2. An *around advice* component replaces the selected elements with a new element.
3. An *after advice* component adds something to the selected element. For example, an extra action may be added to the selected edges to be executed *after* the transition is triggered (see Figure 2).

The effect of an advice depends on what type of advice it is (e.g., before) and what type of modeling elements (e.g., node) the pointcut includes, see Table 1.

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 from 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 from which to restart the system when a timeout occurs. Apart from the two pointcuts selecting elements

Figure 3: A simple example of an aspect adding recovery behavior

from the base model, the *introduction* adds four additional elements, three edges and one node, to the model. Introductions are not necessarily just nodes and edges. In our context however, the extra elements are typically nodes (modeling recovery or degraded service) or edges (modeling faults in the environment).

To get a complete model of the system behavior that can be analyzed by a tool, the base model (Figure 1) and the aspect models (Figure 2 and Figure 3) must then be combined by a weaver. The weaver takes as input a base model, a set of aspect models and a *weaving directive model*, which holds the information of which order to weave each aspect model to the base model. Each aspect model is woven to the base model one aspect model element (pointcut or introduction) at a time. In case the element is a pointcut, the base model is queried to identify the joinpoints to be included in the pointcut and any before, around or after advice associated with the pointcut descriptor is applied to each of these joinpoints. The output is a woven model [4]. Figure 4 agrees with the

Figure 4: A woven model combining the base model with the two aspect models. Elements that come from the aspects are highlighted in the figure by dotted lines or italics.

resulting woven model described by Ali et al. [4], with the only modification that comes with the added node *TwoPart* and resetting *number.size* to 0.

3. Fault Model for Aspect-Oriented Models

Mutation testing inevitably relies on an underlying *fault model*, detailing the assumptions made regarding potential mistakes that could be made during coding (or, in this case, modeling). The fault model thus constitutes the theoretical foundation of the formulation of mutation operators. Below, we propose a fault model tailored for aspect-oriented models.

We assume that aspects already exist in the developed system under test, or can be derived from specifications, to create an aspect-oriented 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 would normally have little opportunity to distinguish robustness code from other code. If the tester (or developer) instead focuses on the behavior and separates normal and robustness behavior into different models, it becomes easier to design the robustness tests. This can be done as a black-box approach with respect to normal behavior, 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 thus assumes that aspects can be used or developed for software testing by means of one of the three approaches below:

- A1 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 pre-existing aspects to create a test suite that properly tests these aspects.

- A2 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 the existing model as a base model and then create an aspect-oriented test model by designing aspects to be used with this base model.

- A3 The tester does not have a behavioral model to start with, and has to create the entire test model from specifications.

Worth to note here is that approaches A2 and A3 come with an overhead since the aspect model or the entire test model needs to be created. As in case of A2 only the aspect models need to be created this may be cost-effective compared to alternative approaches to define how cross-cutting concerns interact with the system. In case of A3 the overhead is likely to be high, however.

Mutation operators are often designed to mimic typical mistakes such as using the wrong operator or the wrong variable name. The types of mistakes that developers make with cross-cutting concerns vary. Mistakes such as forgetting to implement it or misinterpreting where a cross-cutting concern applies are common mistakes, which can manifest as different types of faults with respect to the pointcut descriptors in aspect-oriented programs. Such faults have therefore been used to design mutation operators for AspectJ [7, 10, 14, 53]. Given approach A1 for aspect-oriented modeling, a misinterpretation of where a cross-cutting concern applies can also manifest as different types of faults with respect to the pointcut descriptor. Moreover, even if aspect-orientation is not used (as in approach A2 or A3), such misinterpretation can manifest as the cross-cutting concern being implemented at the wrong location or left out where it should have been implemented. Given the latter approaches, an aspect-oriented test model can, therefore, emulate such faults by introducing faults to the pointcut descriptor or in other ways manipulating the set of joinpoints.

Pointcut Descriptor Fault Types. Figure 5 illustrates the four types of faults for pointcut descriptors, as defined by Lemos et al. [34]. These fault types have also been used by Wedyan and Ghosh [53], and Delamare et al. [14]. The grey areas represent the set of joinpoints that the pointcut selects while the lined areas represent the set of joinpoints that should be selected if the

Figure 5: The four types of pointcut descriptor faults (1:overlap, 2:disjoint, 3:subset, 4:superset) [34][14]

pointcut descriptor were correct. The four fault types described in Figure 5 are generic and can be caused by several different types of mistakes. For example, fault type number 3 (subset) can be caused by a pointcut descriptor that is too strong, but it can also be caused by mistaking one pointcut for another. The mutation operators that we define for pointcuts cover the four fault types.

Advice Fault Types. When it comes to advice, many of the fault types identified for source code in aspect-oriented languages such as AspectJ do not apply to the type of models we use. A typical example is the *incorrect advice type specification* suggested by Ferrari et al. [19]. Changing the type of an advice (for example, from *before* to *after*) in an AOM would in most cases lead to models that are syntactically incorrect (or equivalent to the original) and therefore not useful for the analysis. However, binding an advice to a different pointcut than intended or to implement the advice incorrectly can be applied to models as well as code [19][53]. These types of faults may be further elaborated:

1. Advice is incorrectly bound
 - Advice is bound to both the intended and unintended pointcut
 - Advice is only bound to an unintended pointcut
 - Advice is not bound to any pointcut
2. Advice is incorrectly implemented
 - Advice is implemented using the wrong operator (e.g., using '+' instead of '-')
 - Advice is referring to the wrong variable, method or synchronization event
 - The advice lacks one or more instructions

Introduction Fault Types. Finally, some fault types are related to introductions. An introduction can be ignored or it can be modeled or implemented incorrectly. For example, an edge can be attached to the wrong node or, a guard might use the wrong conditional operator. Since introduction elements are no different from other model elements, pre-existing mutation operators for models apply. The only difference from the traditional use is that we apply mutation to the aspect models instead of the entire model to test the specific cross-cutting concerns.

Given approach A1, any mistakes to the design of an advice will propagate to all parts of the woven model or the resulting code where that advice applies according to the pointcut descriptor. Moreover, any mismatch of selected and intended joinpoints would be caused by a fault in the pointcut descriptor, in case of A1. Hence, mutation operators that apply to the pointcut descriptor or to the advice and that take effect throughout the model are reasonable for approach A1. We refer to this as a pointcut approach.

However, given approach A2 or A3, where the software may not be aspect-oriented and the aspect models are created for testing purposes only, the fault model can be slightly different. It is for example, possible that the behavior modeled as an advice in an aspect by the tester is implemented incorrectly at one location in the software, but correctly at others. Hence, it makes sense to test each advice at each point in the woven model to where it applies. Moreover, a mismatch between intended and selected joinpoints would in case of approach A2 or A3 not be caused by an ill-defined pointcut descriptor (e.g., too strong), but rather to a misinterpretation of the specification or a simple implementation mistake such as forgetting to implement the behavior at one of the intended locations. Hence, mutation operators that apply to single joinpoints are reasonable for approach A2 and A3 since this will ensure that the behavior is tested at each location where the behavior should be implemented. We refer to this as a joinpoint approach.

Mutation operators that delete aspect elements such as pointcuts, introductions, and advice elements ensure that these elements are covered by tests. For example, consider a mutant that changes the leftmost pointcut in Figure 3 (the recovery nodes) 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 node labeled 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 A2 and A3.

Mutants for approach A1 can often be created by making small syntactic changes to the aspect models, but creating mutants for approach A2 and A3 may require more effort. This is because a syntactic change made in an aspect in approach A1 applies to all sites in the woven model where that aspect is applied and that particular change makes a difference. To make a syntactic change at a single joinpoint, it is necessary for the mutation tool to iterate over the set of joinpoints and treat them in isolation. Joinpoints can often be treated in isolation by the mutation tool, simply by identifying the joinpoints that the pointcut in focus will select, in the same order as the weaver. The tool has to take the base model and also the relevant pointcuts and advice in previously processed aspects into consideration when identifying the current joinpoints. When the set of joinpoints is identified, the mutation tool can manipulate the pointcut descriptor to include, exclude or replace these joinpoints one at a time. Sometimes, however, a bigger change to the AOM is required to instruct the weaver to produce the desired woven mutant model. This would be the case if the joinpoint that is subject for mutation is an element, which has been introduced by a preceding aspect and hence, is not part of the base model. This means that some mutants might be very different from the original before weaving if introductions were to be treated with the level of detail as is the case with the joinpoint approach. However, automating such large changes is not trivial and we currently only consider our pointcut approach for introductions.

4. Mutation Operators for Aspect-Oriented Models

Based on the fault model presented in the previous section, we propose several mutation operators for aspect-oriented models. We focus on the aspects and the elements (pointcuts, advice and introductions) that they may consist of, as these are syntactic structures that we do not find in other models and hence are not already 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 illustrated example mutants are all shown as woven models for two reasons: i) to better illustrate the mutation operator's effect on the models, which are later used for the analysis and ii) to better focus on the semantics of the mutation operators rather

than their implementation. In practice, the mutation operators apply to the aspect models before weaving, as described in the previous section. The mutant models that we get from applying the mutation operators can then be used by a model-checker to generate traces with which these mutants are killed.

4.1. Mutation Operators for Pointcuts

Mutation operators for pointcuts focus on the pointcut descriptors [34] that define the pointcuts. These AOM-specific mutation operators cover the fault types for pointcut descriptors described in our fault model (Figure 5).

Figure 6: The resulting woven mutant when applying PCD to the descriptor for the leftmost pointcut in the recovery aspect

Pointcut deletion (PCD): This operator changes the pointcut descriptor so that no element is selected. It covers the scenario in our fault model where the pointcut is ignored. For example, consider the aspect in Figure 3. This aspect has two pointcuts, so PCD will create two mutants. Figures 6 and 7 show the results after weaving with these mutated aspects.

Figure 7: The resulting woven mutant when applying PCD to the descriptor for the rightmost pointcut in the recovery aspect

The mutant in Figure 6 has a recovery node 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 7 has recovery nodes 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.

Figure 8: Resulting woven mutant when applying PCS to the descriptor for the leftmost pointcut in the recovery aspect

Pointcut strengthening (PCS):

By strengthening the pointcut descriptor, we create faults of type 3 (subset) in Figure 5 since the set of joinpoints selected by the pointcut in the mutant model will be a subset of the set of joinpoints that are selected by the pointcut in the original model. We can strengthen the pointcut descriptor if the select query uses any of the operators *OR*, $\leq$, or $\geq$. 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 and corresponding operator is omitted. If elements 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 8 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.

Pointcut weakening (PCW): By weakening the pointcut descriptor, we create faults of type 4 (superset) in Figure 5 since the set of joinpoints that are selected by the pointcut in the original model will be a subset of the set of joinpoints selected by the pointcut in the

mutant model. We can weaken a pointcut descriptor 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 and corresponding operator is deleted. Given elements 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.

Pointcut replacement (PCR): This mutation operator creates faults of type 1 or 2 (overlap, disjoint) in Figure 5. It may also create faults of type 3 and 4 (subset, superset) if one of the pointcuts selects a subset of the joinpoints selected by another pointcut. Each pointcut is replaced by every other pointcut, where elements are of the same type. In our example system, there are four pointcuts of the type that selects node elements; two in Figure 3 and two in Figure 2. PCR will yield 12 mutants, where each mutant differs from the original with respect to one of the four pointcuts, which is replaced by one of the other three. For example, PCR will yield one mutant where the leftmost pointcut in Figure 3 selects nodes such that there is an outgoing edge with an event *dial()*. The set of nodes from which a recovery node can be reached in this particular mutant is therefore changed from $\{TwoPart, NotFull, Full\}$ to $\{Idle, OnePart, TwoPart, NotFull\}$.

Joinpoint deletion (JPD): Just as PCS (PointCut Strengthening), JPD creates faults of type 3 (subset) in Figure 5. The difference is that while PCS focuses on faults in the pointcut descriptor, JPD excludes *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 recovery node cannot be reached from state *TwoPart*, (ii) M2, where the recovery node cannot be reached from state *NotFull*, and (iii) M3, where the recovery node 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 when 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 yield 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 aspects (approach A2 or A3), so the implementation may

differ at the various joinpoints (cf. JPI and JPR below).

Joinpoint introduction (JPI): This mutation operator adds extra joinpoints to the pointcut. Just as PCW (PointCut Weakening), JPI creates faults of type 4 (superset) in Figure 5. The difference is that while PCW focuses on faults in the pointcut descriptor, JPI includes *one joinpoint at a time* to the pointcut before weaving. 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 is 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.

Figure 9: One of the JPR woven mutants

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. This mutation operator creates faults of type 1 in Figure 5 (or 2 if the pointcut has exactly one joinpoint). The difference from PCR is that PCR mimics a fault in the pointcut descriptor and the overlap between intended and matched joinpoints may be small or empty, whereas JPR gives an overlap between intended and matched joinpoints that is almost complete. 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 (Figure 3) selects three of the five nodes. Hence, there will be six JPR mutants for this pointcut. Figure 9 shows an example of a JPR mutant where the node *OnePart* is selected instead of node *TwoPart*.

4.2. Mutation Operators for Advice

We have two approaches for designing advice mutation operators: (i) the advice is mutated at all of 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 descriptor. With the second approach, the advice should only be mutated at one of the sites pointed out by the pointcut at a time. Given J joinpoints, the second approach means that there will be at most $3 * J$ mutants.

4.2.1. AOM-Specific Operators for Advice

The AOM-specific mutation operators address the part of our fault model where an advice is ignored or bound to an incorrect pointcut.

Advice deletion at pointcut (ADP): Consider Figure 2. The center pointcut with advice selects five edges but applying ADP to this aspect model will create only 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 visits any of these edges.

Advice deletion at joinpoint (ADJ): Consider Figure 2 again. Applying ADJ to this aspect model will yield 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 call 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 descriptor 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 which joinpoints are of the same type (that

is, selects 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 as the advice being replaced. For example, a before advice for a pointcut of a set of nodes is replaced by the before advice for every other pointcut that is a set 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.

4.2.2. Code-Level Mutation Operators

Mutation has traditionally been applied to code [16, 39], where individual statements in a program are mutated. We call these *code-level* mutation operators and emphasize that they were not specifically designed for aspects in aspect-oriented software. In our research, these operators are not actually applied to code, but to the model, however we do not wish to introduce a term and follow the established convention of calling them code-level.

The code-level mutation operators address the part of our fault model where an advice is implemented incorrectly. There are several traditional mutation operators that apply to advice elements, such as the ROR operator, which replaces relational operators by other relational operators plus two mutation-specific operators [38, 39].

- ROR replaces relational operators in the aspect models by other relational operators plus falseOp, and trueOp, which replace the expression with true and false respectively. For example, $m \leq n$ is replaced by $m < n$, $m == n$, $m > n$, $m >= n$, $m! = n$, *true* and *false*.
- RORJ is the same as ROR but is applied to the advice at one joinpoint in each mutant
- COR replaces logical operators by other logical operators plus leftOp, rightOp, trueOp, and falseOp, where leftOp and rightOp replace the expression with the left and right operand respectively. For example, $m \& n$ is replaced by $m || n$, $m \& n$, $m | n$, $m \wedge n$, m , n , *true* and *false*.
- CORJ is the same as COR but is applied to the advice at one joinpoint in each mutant

- AOR replaces arithmetic operators by other arithmetic operators plus leftOp, rightOp, and mod. For example, $m + n$ is replaced by $m - n$, $m * n$, m / n , $m ** n$, m , n , and $m \% n$.
- AORJ is the same as AOR but is applied to the advice at one joinpoint in each mutant
- SVR replaces each variable reference by every other variable of appropriate type declared in current scope. For example, $x = m + n$ is replaced by $x = m + m$, $m = m + n$, $x = n + n$, $n = m + n$, $x = x + n$ and $x = m + x$.
- SVRJ is the same as SVR but is applied to the advice at one joinpoint in each mutant
- ASR replaces each assignment operator by other assignment operators. For example, $m += 3$ is replaced by $m -= 3$, $m *= 3$, $m /= 3$, $m \% = 3$, $m \& = 3$, $m | = 3$, $m \wedge = 3$, $m < < = 3$, $m > > = 3$ and $m > > > = 3$.
- ASRJ is the same as ASR but is applied to the advice at one joinpoint in each mutant

Ferrari et al. [20] defined some additional mutation operators for advice in source code. We have not used these in our work since the syntax as well as the semantics of advice in an aspect-oriented model differs 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 if applied to an AOM.

4.3. Mutation Operators for Introductions

Introduction deletion (IDL): An IDL mutant deletes each introduction element in the aspect model in turn. IDL addresses the part of our fault model where introductions are ignored. IDL mutants are killed by any test that visits the deleted element. Our example aspect-oriented model has four introductions; three edges and one node. Hence, IDL will yield four mutants. For example, one mutant will have only one outgoing edge from the recovery node leading to a timeout.

Apart from IDL, there are several mutation operators defined for FSMs and source code that are not specific for aspect-oriented models but also apply to introduction elements. We describe these in Sections 4.3.1 and 4.3.2

4.3.1. Model-Level Mutation Operators

Several mutation operators for models address the part of our fault model where introductions are modeled incorrectly. We call these *model-level* mutation operators and emphasize that they target modeling elements but are not specifically designed for introductions in aspect-oriented models.

An introduction is a model element, thus mutation operators defined for model elements apply to introductions. Some of these mutation operators were proposed by others and some are new to this paper.

The pre-existing operators that we use are 1, 2, 3, 4, and 5 below. Items 8 and 9 are variants of pre-existing operators and items 6, 7, 10 and 11 are new to this research. All the pre-existing operators were originally defined for general FSMs, not aspect models.

1. RTN replaces each target node for edge introductions by other nodes [26, 35, 50]
2. RSN replaces each starting node for edge introductions by other nodes [35, 50]
3. RSI replaces each synchronization event at edge introductions by other synchronization events [18, 29, 50]
4. DSI deletes each synchronization event at edge introductions [18, 50]
5. DGI deletes each guard at edge introductions [29]
6. RGI replaces each guard at edge introductions by other guards
7. SSI replaces each send with a receive and each receive with a send at synchronization events on edge introductions
8. DAI deletes each action at edge introductions [32]
9. RAI replaces each action at edge introductions by other actions [32]
10. DIN deletes each invariant at node introductions
11. RIN replaces each invariant at node introductions by other invariants

4.3.2. Code-Level Mutation Operators

The code-level operators address the part of our fault model where introductions are implemented incorrectly. Just as advice elements, introductions can come with constraints, guards, actions etc. containing relational, arithmetic or logic expressions. The traditional code-level mutation operators of ROR, COR, AOR, SVR, and

ASR that we suggest for advice also apply to introductions. Since there is no overlap between introductions and pointcuts, there is no redundancy between applying these operators both to advice and to introductions.

We have discussed the possibility of applying mutation operators to single joinpoints rather than pointcuts. This is fairly straightforward since a pointcut describes a set of joinpoints and each advice is mapped to a 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, there is no set to iterate over before the weaving process. A fine-grained mutation approach for introductions may therefore require 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.

4.4. Choice of Mutation Operators

The mutation operators that we have defined in previous sections are in most cases defined for two different approaches. For example: PCD vs JPD, ADP vs ADJ and ROR vs RORJ. Given the approach, the tester should choose to use the set of mutation operators that works at a pointcut/aspect level or at a joinpoint level. The choice of which set of mutation operators to select depends on the fault model and this in turn depends on whether the software is aspect-oriented to begin with (approach A1) or whether the aspects are something that is created for a test model (approaches A2 and A3). In the first case, it makes sense to assume that a fault in an aspect propagates to all sites in the woven model to which the faulty element applies during weaving. For example, if an advice to add a guard at some edge is missed, this guard will be missed at all edges selected by the pointcut descriptor. Hence, a mutation operator such as ADP would be sufficient to detect this fault. On the other hand, if the aspects are only used as a test model it would make sense to apply ADJ in order to test that the guard is present at each edge selected by the pointcut descriptor.

5. Application to Robustness Testing

This section describes an example use of our proposed approach for robustness testing of systems modeled in timed automata (TA). TA is used by engineers to specify and verify real-time systems.

The primary focus for this work is to use model-checking algorithms to kill mutants, not to verify timeliness or performance. We are particularly interested

in embedded real-time systems, where there often is a timing aspect associated with robustness. For example, recovery of a subsystem might have an associated deadline. For this reason, we selected TA as a suitable model for our approach.

Applying the previously defined mutation operators to the TA modeling language will change the scope or domain of some operator slightly. E.g., the domain of SVR is extended to include clock variables used in TA, and the scope of the mutation operator COR is extended to encompass some additional conditional operators used in TA. This is described in Section 6.2.

5.1. Timed automata

This section provides a brief but necessary introduction to TA. For more details on these concepts, see Bengtsson and Yi [9] or Hessel et al. [25].

In a timed automaton, **clocks** are represented by a finite set of real-valued variables C and **events** are represented by a finite alphabet Σ . Let $\mathcal{B}(C)$ denote the set of Boolean combinations of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in C$, n is a natural number and $\sim$ represents one of the relational operators $\{<, \leq, =, \geq, >\}$.

A **timed automaton** ($\mathcal{A}$) over events and clocks (Σ, C) is a tuple $\langle N, l_0, E, I \rangle$ where:

- N is a finite set of vertices (here called nodes)
- $l_0 \in N$ is the initial node
- $E \subseteq N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ is a set of edges
- $I : N \rightarrow \mathcal{B}(C)$ assigns invariants to nodes

The **semantics of a timed automaton** is a timed transition system over states of the form $\langle l, u \rangle$, where $l \in N$ and u is a clock assignment of all clocks in C to non-negative real-numbers. The initial state is $\langle l_0, u_0 \rangle$, where u_0 is the clock assignment that assigns all clocks in C to 0. Transitions are defined by two rules:

1. (discrete transitions) $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ if $\langle l, g, a, r, l' \rangle \in E$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$
2. (delay transitions) $\langle l, u \rangle \xrightarrow{d} \langle l, u \oplus d \rangle$ if $u \in I(l)$ and $(u \oplus d) \in I(l)$ for a non-negative real $d \in \mathbb{R}_+$

where $u \oplus d$ denotes the clock assignment that maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ is the clock assignment u with each clock in r reset to zero.

Timed automata have two kinds of transitions, thus two transition rules: the discrete transition, which is an instant move from one node to another that is enabled when the clocks satisfy the guard on the edge, and the delay transition, which increments the clocks but does not include a move to another node.

A **run** of a timed automaton $\mathcal{A} = \langle N, l_0, E, I \rangle$ with initial state $\langle l_0, u_0 \rangle$ over a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3) \dots$ is a sequence of transitions:

$$\langle l_0, u_0 \rangle \xrightarrow{d_1 a_1} \langle l_1, u_1 \rangle \xrightarrow{d_2 a_2} \langle l_2, u_2 \rangle \xrightarrow{d_3 a_3} \langle l_3, u_3 \rangle \dots$$

satisfying the condition $t_1 = d_1$ and $t_i = t_{i-1} + d_i$ for all $i \geq 1$. The **timed language** $L(\mathcal{A})$ is the set of all timed traces ξ for which there exists a run of $\mathcal{A}$ over ξ .

A **network of timed automata** $\mathcal{A}_1 || \dots || \mathcal{A}_n$ over (Σ, C) is the parallel composition of n timed automata over (Σ, C) , where components are required to synchronize on delay transitions and discrete transitions are required to be synchronized on complementary actions. An action $a?$ is complementary to $a!$.

5.2. Example System

We manually translated the woven video conferencing system to a timed automaton for UPPAAL (Figure 10). Mücke and Huhn [41] describe how to transform a UML state chart to UPPAAL. We then used the UPPAAL model checker to verify its behavior [33]. In UPPAAL, a double circle denotes the initial node. 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 11 and 12). We used the UPPAAL simulator to execute the test scenarios and to generate the traces that we discuss in our examples.

Figure 13 gives an overview of the AOM mutation process. Mutation operators are applied to the aspect models before weaving, creating a set of aspect mutants. The weaving process creates a set of woven mutants by using aspect mutants instead of the original aspect model. An analysis is then performed to identify traces that can be traversed when using the original version of the woven model but not in the mutant version or vice versa. This can be viewed as a form of weak mutation since the mutant is killed based on its internal state. Identified traces can then be transformed to test cases.

The process described in Figure 13 is currently semi-automated, as both mutation and weaving have been conducted manually in our study. There is a lack of automated tool support when it comes to AOM. Weavers for AOM exist but are typically in-house built or implemented within a testing framework and not available, [12, 21, 27, 45, 49, 54].

Figure 10: A timed automata model of the woven system

Figure 11: A timed automata model of a conference participant

Figure 12: A timed automata model of a driver triggering new calls and generating disturbances

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*” [1]. Systems can be stressed in many different ways, such as frame loss, noise, synchronization mismatches and lost connections [4]. 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. 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

Figure 13: Overview of the AOM mutation process

be too cluttered to show in a single figure.

Figure 10 shows a timed automata model of the woven video conferencing system. Figure 11 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 10. 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* node.

Figure 12 shows a timed automata model of a simple driver for the video conferencing system. The driver triggers new calls as well as media failures of different types. The *failureType* variable 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 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 0 == *ok* (i.e., successfully recovered) and a higher number indicates a persisting failure of the specified type (1- m). For example:

- A trace where *failureType* is set to 1 and *recovered* is set to 0, maps to a test with an audio failure that is successfully recovered.
- A trace where *failureType* is set to 1 and *recovered* is set to 1, maps to a test where an audio failure persists and can be used 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. Such test is useful to verify that a persisting failure is handled by a timeout and reset, independent of whether the persistent failure is of the same type that caused the transition to the recovery mode in the first place.

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. Both approaches employ the trace from the environment (driver and participants).

Consider the trace in Figure 14. 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 10 and D is the driver shown in Figure 12. P0 and P1 are instances of the template participant shown in Figure 11. The boxes in Figure 14 show nodes, the vertical arrows show transitions, and the horizontal arrows show synchronization between automata. 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

Figure 14: A graphical view of a trace

models the timer. The time limit is set to four. This specific trace shows two connections followed by an audio failure (*failureType*=1) that persists (*recovered*=1) and leads to a timeout ($x \geq 4$) and a *disconnectAll*.

5.3. Example Mutants and Tests

The trace in Figure 14 is a result of executing a test scenario in the UPPAAL simulator using the original version of the system. Consider the PCD (PointCut Deletion) robustness mutant in Figure 15. The crossed and grayed out portions in the figure have been 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 is an abstract test that kills this mutant. A corresponding test scenario, to be executed on the real system, would focus on the sequence of interactions between the system (P) and its environment, in our case the driver and the participants. The interactions can be translated to real input events and used to test the software system, assuming that the events (calls and disturbances) can be produced and controlled with respect to the order in which they start and stop, and to the type and persistence of the disturbance (exceeding the time limit). Such concrete tests can preferably be realized as a scenario played by an

Figure 15: Timed automata mutant where a pointcut is deleted (PCD)

Figure 16: Timed automata mutant where an advice is deleted at a single joinpoint (ADJ)

environment simulator that interacts with the software system.

Consider the ADJ robustness mutant in Figure 16. The mutant does not have the guard $numberSize==4$ (crossed and grayed out) on the edge between *NotFull* and *Full*. Hence, it is possible to get a trace in the mutant to a state where $numberSize!=4$ and a transition from *NotFull* to *Full* is enabled. This transition is not possible 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.

We can translate candidate test scenarios to candidate test traces as a sequence of discrete transitions that Upaal can accept. By enforcing these traces on the original as well as the mutant models, we can identify which candidates tests are effective and which mutants they de-

Figure 17: Identifying the detecting traces.

tect. This way we can either create a strong test suite based on the traces that are effective or evaluate an ex-

isting test suite with respect to its mutation adequacy score (detected/all), see Figure 17.

Candidate test scenarios can be: (i) an existing test suite, (ii) defined by the tester (iteratively), (iii) randomly generated as candidate traces by simulation of the model, or (iv) defined by the model-checker (traces to error states associated with the mutation). Candidate test traces that detect mutants are included in the test suite, while others are discarded. We currently use approach (iv) and search for error states (mainly deadlocks) in both original and mutant versions. However, the process described in Figures 13 and 17 is independent on how the candidates are generated. Another option to generate candidate test traces is to instrument the aspect models and then use the model checker to identify traces to such instrumented locations (or locations where an instrumentation is missing) in the woven models.

Finally, consider the IDL robustness mutant in Figure 18. This mutant has no edge to the recovery node for the failure type *video* (again showed in figure as crossed and grayed out). Hence, this mutant is killed by a test with a media failure of this type when at least two participants are active.

6. Analysis of Mutating an Example System

This section shows how the mutation operators defined in Section 4 can be applied to a working system. We first demonstrate the application of our novel AOM-specific operators, then existing mutation operators for code, and finally existing mutation operators for models. Subsection 6.4 presents an analysis of the mutants created for our example system.

Not all generated mutants are useful when mutation is applied to software, independent of whether mutation is applied to source code or to a model. Some mutants are syntactically illegal and thus, in our case, are not accepted by the model checker. We refer to such mutants as *stillborn*. Some mutants are *equivalent* to the original software, meaning that for each possible input the original and the mutant version will show the exact same behavior. Hence, such mutants cannot be distinguished from the original by any test. Some mutants are equivalent to each other but not to the original and are referred to as *redundant*. If two mutants are redundant, then any test that detects one of them is guaranteed to also detect the other. Finally, two mutants can be syntactically identical to each other. We refer to such mutants as *duplicates*.

Table 2: Overview of the number of mutants generated for each AOM-specific operator

Mutants from the Pointcut Approach				
Operator	Generated	Equivalent	Stillborn	Remaining
PCD	5	0	2	3
PCS	5	0	0	5
PCW	0	0	0	0
PCR	12	0	6	6
ADP	2	0	0	2
Mutants from the Joinpoint Approach				
Operator	Generated	Equivalent	Stillborn	Remaining
JPD	17	0	8	9
JPI	13	0	2	11
JPR	43	0	18	25
ADJ	10	0	0	10
Mutants Relevant to Both Approaches				
Operator	Generated	Equivalent	Stillborn	Remaining
IDL	5	0	0	5
Sum	112 (29 or 88)	0	36 (8 or 28)	76 (21 or 60)
				Sum Unique 68 (18 or 59)

6.1. AOM-Specific Mutation Operators

For our example system, the novel mutation operators that specifically targets AO features yield a total of 112 mutants (see Table 2).

However, the mutation operators should not all be used together. Depending on whether there is a need to test all the joinpoints individually or not, two different sets of mutation operators should be used (approach A1 versus A2 or A3). Only IDL applies to both sets. We give the numbers for both the pointcut approach and the joinpoint approach in Table 2. 36 of the generated mutants are syntactically illegal (*stillborn*), thus cannot be used. All the stillborn mutants involve a change to the set of joinpoints selected by the pointcuts in the add guard aspect. The three pointcuts in this aspect depend on each other, so changing the set of joinpoints for one pointcut in this aspect while keeping the other pointcuts unchanged sometimes lead to stillborn mutants.

None of the mutants are equivalent to the original so 76 of the generated mutants can be analyzed. Eight of the 76 are duplicates. However, since we use two different sets of mutants depending on whether we employ a pointcut approach or a joinpoint approach, only four of these are true duplicates and can be removed, three for the pointcut approach and one for the joinpoint approach (see Table 3). The mutants that cannot be removed are shown in parentheses in Table 3. We found seven types of duplicates:

1. **PCD and PCS:** Two PCS mutants have an OR replaced by an AND in the leftmost pointcut in the

Figure 18: Timed automata mutant where an introduction is deleted

Table 3: Overview of the number of duplicates

	PCD	PCS	PCR	JPD	JPI	JPR	ADP	ADJ	IDL
PCD		2	0	(1)	0	0	0	0	1
PCS	2		0	(3)	0	0	0	0	0
PCR	0	0		0	(1)	0	0	0	0
JPD	(1)	(3)	0		0	0	0	0	1
JPI	0	0	(1)	0		0	0	0	0
JPR	0	0	0	0	0		0	0	0
ADP	0	0	0	0	0	0		0	0
ADJ	0	0	0	0	0	0	0		0
IDL	1	0	0	1	0	0	0	0	

recovery aspect. In both cases, no joinpoint is selected for the pointcut and these two PCS mutants are therefore identical to one of the PCD mutants.

2. **PCD and JPD:** There is a pointcut with a single joinpoint in the recovery aspect. Hence, applying JPD to this pointcut has the same effect as applying PCD. However, PCD and JPD should not be used in combination since a test suite that kills all JPD mutants would also kill the PCD mutants. The reason is that PCD results in an empty set of joinpoints so the joinpoint that is missing in the JPD mutant is also missing in the PCD mutant.
3. **PCD and IDL:** Deleting the pointcut that selects node Idle as its only joinpoint is effectively the same as deleting the edge to this pointcut.
4. **JPD and IDL:** Deleting the joinpoint Idle from the recovery aspect has the same effect as deleting the edge to Idle in the recovery aspect
5. **PCS and JPD:** Dropping one OR operand at a time (PCS) in the leftmost pointcut in the recovery

aspect and dropping one joinpoint at a time for the same pointcut (JPD) results in a duplicate mutant for each joinpoint. However, PCS should not be combined with JPD since dropping one joinpoint at a time will effectively test that the pointcut descriptor is not too strong.

6. **PCR and JPI:** Replacing the leftmost pointcut in the recovery aspect by the rightmost pointcut in the guard aspect (PCR) has the same effect as inserting the node OnePart as an extra joinpoint to the leftmost pointcut in the recovery aspect. However, PCR should not be combined with JPI in a joinpoint approach. The reason is that unless the PCR mutant is an equivalent mutant, its set of joinpoints will differ from the original and the combination of JPD and JPI will create a set of mutants to test that all intended but no unintended joinpoints are included in the set selected by the original pointcut descriptor.

It is important to note that identifying that two models are identical is much easier than showing that two

different models are equivalent. After removing duplicates from the entire set, 68 unique mutants remain, but removing duplicates from the mutants generated for a pointcut leaves 18 unique mutants, and removing duplicates from the joinpoints leaves 59 unique mutants.

Five of the mutation operators do not generate any mutant for the system. PCW does not apply since the model has no pointcut descriptor that includes an “and,” “<,” or “>”. AIP, AIJ, ARP, and ARJ also do not generate any mutants in our model. The reason is that there is only one pointcut that has an advice and this pointcut is the only one that selects a set of edges. Hence, neither the before nor the after advice can be inserted at another pointcut or be replaced by another advice.

6.2. Code-Level Mutation Operators

Code-level mutation operators date to the 1980s [16] and are stable and traditional. They were designed to test at the statement level, but do not target model or aspect-oriented features.

Table 4: Overview of the mutants generated for code-level mutation operators

Number of Mutants Pointcut Approach				
Operator	Generated	Equivalent	Stillborn	Remaining
ROR	42	2	0	40
COR	20	4	0	16
SVR	118	1	15	102
Sum	180	7	15	158
After removal of redundant				143
Number of Mutants Joinpoint Approach				
Operator	Generated	Equivalent	Stillborn	Remaining
RORJ	70	2	0	68
CORJ	20	4	0	16
SVRJ	198	1	39	158
Sum	288	7	39	242
After removal of redundant				223

We apply these mutation operators to the advice and introductions in the aspect models before weaving. Although we mutate the aspect models only, the entire model is within scope for SVR and SVRJ. Table 4 shows the number of mutants generated for our example system by applying the code-level mutation operators for Java code to introductions and advice. None of the aspects have any assignment or arithmetic operators, thus no AOR or ASR mutants are generated.

The mutation operator COR covers conditional operators used in Java but are adapted for UPPAAL to encompass the extra conditional operators “and,” “or,” and “imply”. Ten COR mutants are generated for each occurrence of a conditional operator in an advice or introduction. Note that an introduction contributes with the same amount of mutants irrespective of the approach.

Table 5: Overview of the equivalent mutants

ROR / RORJ	
Original	<i>recovered != ok</i>
Mutant	<i>recovered > ok</i>
Original	<i>recovered == ok</i>
Mutant	<i>recovered <= ok</i>
Sum ROR / RORJ	
2	
COR / CORJ	
Original	<i>recovered == ok and x <= limit</i>
Mutant	<i>recovered == ok & x <= limit</i>
Mutant	<i>recovered == ok && x <= limit</i>
Original	<i>recovered != ok and x >= limit</i>
Mutant	<i>recovered != ok & x >= limit</i>
Mutant	<i>recovered != ok && x >= limit</i>
SUM COR / CORJ	
4	
SVR / SVRJ	
Original	<i>mediaFailure[video]?</i>
Mutant	<i>mediaFailure[m]?</i>
SUM SVR / SVRJ	
1	
Sum Equivalent	
7	

The total number of generated mutants with the code-level mutation operators is 180 or 288, depending on whether pointcuts or joinpoints are mutated. Seven of these mutants are equivalent and 15 or 39 are stillborn. Table 5 presents an overview of the equivalent mutants.

The code-level mutation operators do not produce any duplicated mutants but we have identified some redundancy. When two mutants are equivalent to each other (but not the original), one is *redundant*. We found 15 such redundant mutants when applying the mutation operators to pointcuts and four when we applied them to joinpoints. An overview of the redundant mutants is shown in Table 6. After removing the redundant mutants, 143 or 223 mutants were left for analysis, depending on the approach.

None of the code-level mutants duplicate any of the AOM-specific mutants, that is, there is no overlap. This is because the code-level mutation operators are only applied to advice and introductions. None of the AOM-specific operators for advice or introduction target variables, assignments, conditions or relational operators in advice or assignments.

6.3. Model-Level Mutation Operators

Model-level mutation operators were first defined in 1994 [18] and were subsequently extended [26, 29]. They were designed to test at the model level, and do not target aspect-oriented features.

Hence, when it comes to testing aspect models, these mutation operators apply to introductions only. The number of mutants generated from the model-level mutation operators are shown in Table 7.

Table 6: Mutants in the same row are equivalent to each other but not to the original. Hence, 15 are redundant and can be skipped.

Op	Original	Mutant 1	Mutant 2	Mutant 3
ROR	recovered!=ok	recovered<ok	falseOp	
	recovered==ok	recovered<ok	falseOp	
COR	recovered==ok and x<=limit	recovered==ok or x<=limit	recovered==ok x<=limit	recovered==ok x<=limit
	recovered!=ok and x>=limit	recovered!=ok or x<=limit	recovered!=ok x<=limit	recovered!=ok x<=limit
SVR	numberSize==4	video==4	m==4	
	recovered!=ok and x>=limit	video!=ok and x>=limit	m!=ok and x>=limit	
	recovered!=ok and x>=limit	recovered!=video and x>=limit	recovered!=m and x>=limit	
	recovered!=ok and x>=limit	recovered!=ok and x>=video	recovered!=ok and x>=m	
	x<=limit	x<= 4video	x<= 4m	
	recovered==ok and x<=limit	video==ok and x<=limit	m==ok and x<=limit	
	recovered==ok and x<=limit	recovered==video and x<=limit	recovered==m and x<=limit	
	recovered==ok and x<=limit	recovered==ok and x<=video	recovered==ok and x<=m	
	mediaFailure[audio]?	mediaFailure[video]?	mediaFailure[m]?	

Table 7: Overview of the mutants generated from model-level mutation operators

Operator	Generated	Number of Mutants		
		Equivalent	Stillborn	Remaining
RTN	8	0	0	8
RSN	8	0	0	8
DGI	2	0	0	2
RGI	4	0	0	4
DSI	3	0	0	3
RSI	12	0	0	12
SSI	3	0	0	3
DAI	3	0	0	3
RAI	51	2	0	49 (12)
DIN	1	0	0	1
RIN	5	0	0	5
Sum	100	2	0	98
				Sum Unique 61

100 mutants are generated using the model-level mutation operators. None are stillborn and only two are equivalent to the original model. The equivalence is because the action $x=0$ occurs on two introductions. Replacing one with the other creates identical mutants.

The figure that stands out in Table 7 is the high number of duplicates generated by RAI. Only 12 of the 49 mutants were unique. The main reason is that the base model has three actions (*numberSize=reset()*, *numActive-* and *numActive++*) and each occurs in five different places. For each action at any of the introductions (e.g., *numActive=0*) we therefore get 15 mutants. Only three of these 15 are useful, the others are duplicates. Since there are three actions on the introduction elements, this alone gives 36 duplicates. There is also one duplicate where the action *numActive=0* is replaced by $x=0$, which occurs on two introductions.

None of the mutants generated by any of the model-level mutation operators duplicate any of the AOM-specific mutants. This is expected since the model-level

operators apply to introductions only and deletion of elements is not listed among the model-level operators that we suggest. The reason to not include deletion of elements in the list is that the overlap with IDL will **always** be 100%.

6.4. Summary of Results and Discussion

Table 8 shows an overview of the results presented above. One thing that stands out is that we did not get any equivalent AOM-specific mutants. This is positive because identifying equivalent mutants can be expensive. Moreover, we did not get any redundant AOM-specific mutants (equivalent to each other). Again, this is interesting because creating and analyzing redundant mutants waste resources and obscures the results.

We got a couple of duplicates for the AOM-specific operators. It is important to note that it is much easier to recognize that two models are syntactically identical than it is to recognize that two syntactically different models have identical semantic behavior. Also, even though our AOM-specific mutation operators generated some duplicates, these were very few in comparison with the number of duplicates that were generated when using model-level mutation operators (10% and 1% vs. 37%).

Another result that stands out is that the AOM-specific operators yielded many stillborn mutants compared with the code-level and model-level operators. The high percentage of AOM-specific stillborn mutants (approximately 30%) has to do with the fact that one pointcut (selecting edges) cannot be woven to the base model unless each starting and target node for all the joinpoints in the base model are also selected by pointcuts for the starting and target node in the aspect model. This result suggests that mutation operators that target pointcut descriptors for AOM could create more effective live mutants if neighbor pointcuts are considered.

Table 8: Overview of the results

Operators	Generated	Stillborn	Pointcut Approach			
			Equivalent	Redundant	Duplicate	Remaining
AOM spec.	29	8 (28%)	0	0	3 (10%)	18 (62%)
Code-lev.	180	15 (8%)	7 (4%)	15 (8%)	0	143 (79%)
Model-lev.	100	0	2 (2%)	0	37 (37%)	61 (61%)
Sum	309	23 (7%)	9 (3%)	15 (5%)	40 (13%)	219 (72%)

Operators	Generated	Stillborn	Joinpoint Approach			
			Equivalent	Redundant	Duplicate	Remaining
AOM spec.	88	28 (32%)	0	0	1 (1%)	59 (67%)
Code-lev.	288	39 (14%)	7 (2%)	19 (7%)	0	223 (77%)
Model-lev.	100	0	2 (2%)	0	37 (37%)	61 (61%)
Sum	476	67 (14%)	9 (2%)	19 (4%)	38 (8%)	343 (72%)

It is a reasonable assumption that a developer who misinterprets the specification regarding where the guard “numberSize==4” applies will make the same mistake for all three pointcuts in that aspect.

The percentage of stillborn mutants doubled when we focused on individual joinpoints rather than the pointcuts (14% vs 7%). This increase was mainly due to the code-level operators and not the AOM-specific operators (32% vs 28%). Stillborn mutants are easy to identify and do not have to be analyzed, but they still represent some waste of resources since they must be created. Moreover, a stillborn mutant is created for a purpose (e.g., coverage) that might not be fulfilled due to the fact that the mutant is stillborn.

7. Related Work

Xu et al. [55] present a framework to test aspect-oriented programs and, just as ours, this framework includes the use of AOM, mutants and a model-checker to create tests. The mutation operators that they use are further described in Xu et al. [56]. Xu et al. [55, 56] have, however, not a major focus on mutation operators for their models. As a consequence, their mutation operators do not cover all type of model elements and the faults that can come with them. Two of the four mutation operators that they use (PCS and PCW) are also used by us.

Lemos et al. [34] discussed testing for joinpoints that are unintended or neglected, which is tested by our mutation operators. However, their strategy is based on a step-wise integration of the joinpoints at the source-code level while we focus on aspect models before they are integrated to the base model.

Fault models for aspect-oriented programs have been suggested [10, 14, 53] and mutation operators have been defined at the source-code level for AspectJ [7, 19, 40, 53]. The major difference between our work

and theirs is that we define mutation operators, which can be used at the model-level and can be applied in a black-box manner without knowledge of the implementation. The overlap between our new mutation operators and previous operators is therefore small. Pointcut weakening and pointcut strengthening have been proposed for source code but the focus has mainly been on the use of wildcards (*). Delamare et al. [14], however, proposed to replace && by || and vice versa in pointcut descriptors, which is part of what our PCS and PCW operators do. Wedyan and Ghosh [53] proposed to remove advice and to change the advice binding. Our operator ADP removes (deletes) advice from its pointcut. We also propose two new mutation operators, AIP and ARP, which both change the advice bindings but not in the way that Wedyan and Ghosh proposed. Wedyan and Ghosh proposed to bind the advice to an incorrect pointcut instead of the correct, while our AIP operator binds the advice to both correct and incorrect pointcut and ARP replaces an advice by another advice of the same type.

Ali et al. [4] applied 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 was presented. Ali et al. [3] used the methodology combined 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.

Di Nardo et al. [42] proposed a fault model and a set of mutation operators that target communication faults such as missing packages and incorrect checksums. Their technique uses a UML class diagram annotated with stereotypes and constraints. However, their approach does not target the specific AOM features that we target with our new mutation operators.

Related work on model-based testing of finite state and timed automata include Springintveld et al. [51], who showed the problem of timed trace equivalence testing of (deterministic) timed automata to be theoretically possible. Nielsen and Skou [43] proposed a model-based test technique for event-recording automata, and Hessel et al. [24] presented a model-based technique to generate optimal tests coverage criteria using the UPPAAL tool. Our research is related to these projects in that we use timed automata and UPPAAL-generated traces to generate tests, but different in the sense that we generate tests directly to kill mutants.

8. Conclusions

We present a novel technique to test software developed with aspect-oriented models. The technique works by mutating cross-cutting concerns at the model level. Fourteen novel mutation operators are defined that modify the aspect model. We have shown how these AOM-specific operators can be used with traditional mutation operators without any overlap in mutants created.

The AOM-specific operators are based on the unique features of aspect-oriented modeling and most induce changes that mirror mistakes that AO modelers typically make. Tests are designed at the abstract level to kill the mutants. These abstract tests can then be 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.

We define the mutation operators, illustrate how they are used to design abstract tests and evaluate the mutation operators with respect to (i) generated mutants, (ii) stillborn mutants, (iii) equivalent mutants, (iv) duplicate mutants, and (v) redundant mutants for our example system. As discussed in Section 6.4, the evaluation shows that there was no overlap to traditional mutation operators and that the new mutation operators produced no equivalent or redundant mutants. Moreover, the number of duplicates was very low for the new mutation operators compared to the traditional mutation operators for models.

This technique is novel and useful for the same reason that AO modeling is effective: it allows the tester to work at a higher level of abstraction. That is, to design tests that 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 possible for the tester to adjust the

test effort for different aspects, depending on the criticality of the cross-cutting concern that each aspect 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.

8.1. Future Work

We have proposed an approach to test cross-cutting concerns by mutating aspect-oriented models and illustrated the approach to robustness testing by applying it to a video conferencing system. The mutation operators for aspect-oriented models were evaluated in terms of the number of generated mutants and their properties. There are however, some limitations to our work that need to be addressed by future work.

The video conferencing system used in our study is a small example so the approach needs to be further validated empirically. We are, therefore, currently preparing a larger-scale industrial case study to evaluate the mutation operators with respect to their effectiveness and efficiency for robustness testing. The extra overhead for approaches A2 and A3 should also be considered in this study. The planned study will include the generation of a complete test suite and an analysis of the mutation score. Moreover, the industrial system we use will not only produce a larger set of mutants for the evaluation but also allow us to make use of and evaluate a larger subset of the mutation operators.

One question that we plan to address is whether it is possible to reduce the number of stillborn mutants from the mutation operators that target pointcut descriptors. Many such mutants are stillborn because there is a dependency between pointcuts. We conjecture that this problem can be solved by taking neighboring pointcuts into consideration.

Another question that we plan to address empirically is whether each mutation operator should result in multiple code changes or in just one code change when applied to introductions. For example, applying IDL to one edge in the recovery aspect winds up removing three edges in the woven graph. Such large scale changes may result in several easy-to-kill mutants (*trivial mutants*), so we plan experiments to decide whether to turn this into three separate mutants in the same way as we have done for pointcuts and advice. This could be done by applying the mutation operators while weaving.

In addition, the mutation operators for aspect-oriented models defined in this paper should be thor-

oughly 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.

9. Acknowledgment

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

References

- [1] *IEEE Standard glossary of software engineering terminology*, 1990.
- [2] H. Agrawal, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, *Design of mutant operators for the C programming language*, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.
- [3] 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*, Model Driven Engineering Languages and Systems, 2011, pp. 108–122.
- [4] S. Ali, L. C. Briand, and H. Hemmati, *Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems*, Softw. Syst. Model **11** (2012), no. 4, 633–670.
- [5] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), no. 2, 183–235.
- [6] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.
- [7] P. Anbalagan and T. Xie, *Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs*, Proceedings of the 2nd Workshop on Mutation Analysis (Mutation’06), 2006, pp. 3–8.
- [8] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, *Specification of timed EFSM fault models in SDL*, 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, 2007, pp. 50–65.
- [9] J. Bengtsson and W. Yi, *Timed Automata: Semantics, Algorithms and Tools*, Lecture Notes on Concurrency and Petri Nets, 2004.
- [10] J. S. Bsekkén and R. T. Alexander, *A candidate fault model for AspectJ pointcuts*, Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE’06.), pages=169-178, year=2006, organization=IEEE.
- [11] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: Aspect-oriented programming with AspectJ and the Eclipse AspectJ development tools*, Addison-Wesley Professional, 2004.
- [12] T. Cottenier, A. van den Berg, and T. Elrad, *Motorola weavr: Aspect orientation and model-driven engineering*, Journal of Object Technology **6** (2007), no. 7, 51–88.
- [13] ———, *Stateful aspects: The case for aspect-oriented modeling*, 10th International Workshop on Aspect-Oriented Modeling, 2007.
- [14] R. Delamare, B. Baudry, S. Ghosh, and Y. Le Traon, *A test-driven approach to developing pointcut descriptors in AspectJ*, International Conference on Software Testing, Verification, and Validation (ICST’09), 2009, pp. 376–385.
- [15] R. DeMillo, R. J. Lipton, and F. G. Sayward, *Hints on test data selection: Help for the practicing programmer*, IEEE Computer **11** (1978), no. 4, 34–41.
- [16] R. A. DeMillo and J. Offutt, *Constraint-based automatic test data generation*, IEEE Transactions on Software Engineering **17** (1991), no. 9, 900–910.
- [17] 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*, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII, 1996, pp. 329–337.
- [18] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, *Mutation analysis testing for finite state machines*, Fifth International Symposium on Software Reliability, 1994, pp. 220–229.
- [19] F. C. Ferrari, J. C. Maldonado, and A. Rashid, *Mutation testing for aspect-oriented programs*, 1st International Conference on Software Testing, Verification and Validation (ICST’08), 2008, pp. 52–61.
- [20] F. C. Ferrari, A. Rashid, and J. C. Maldonado, *Design of mutant operators for the AspectJ language*, University of Sao Carlos, Sao Carlos, 2011.
- [21] L. Fuentes and P. Sánchez, *Designing and weaving aspect-oriented executable uml models.*, Journal of Object Technology **6** (2007), no. 7, 109–136.
- [22] 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 **2** (2010), no. 4, 303–311.
- [23] D. Harel, *Statecharts: A visual formalism for complex systems*, Science of computer programming **8** (1987), no. 3, 231–274.
- [24] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, *Formal Methods and Testing*, 2008, pp. 77–117.
- [25] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou, *Time-Optimal Real-Time Test Case Generation using Uppaal*, The 3rd International Workshop on Formal Approaches to Testing of Software, 2003, pp. 136–151.
- [26] R. M. Hierons and M. G. Merayo, *Mutation testing from probabilistic and stochastic finite state machines*, Journal of Systems and Software **82** (2009), no. 11, 1804–1818.
- [27] A. Jackson, J. Klein, B. Baudry, and S. Clarke, *Kerthème: Testing aspect oriented models*, Proceedings of IMDT workshop in conjunction with ECMDA’06, 2006.
- [28] Jan Jürjens, *Model-based security engineering with uml*, Foundations of Security Analysis and Design III, 2005, pp. 42–77.
- [29] S. Kansomkeat and W. Rivepiboon, *Automated-generating test case using UML statechart diagrams*, Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology (SAICSIT) ’03, 2003, pp. 296–300.
- [30] S. Kim, J. A. Clark, and J. A. McDermid, *Class mutation: Mutation testing for object-oriented programs*, Proceedings of ObjectDays, 2000, pp. 9–12.

- [31] K. N. King and A. J. Offutt, *A Fortran language system for mutation-based software testing*, Software Practice & Experience **21** (1991), no. 7, 685–718.
- [32] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl, *Momut:: UML model-based mutation testing for UML*, 8th International Conference on Software Testing, Verification and Validation (ICST'15), 2015, pp. 1–8.
- [33] K. G. Larsen, P. Pettersson, and W. Yi, *UPPAAL in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), no. 1-2, 134–152.
- [34] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes, *Testing aspect-oriented programming pointcut descriptors*, Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs, 2006, pp. 33–38.
- [35] J. Li, G. Dai, and H. Li, *Mutation analysis for testing finite state machines*, Second International Symposium on Electronic Commerce and Security (ISECS'09), 2009, pp. 620–624.
- [36] B. Lindström, S. F. Andler, J. Offutt, P. Pettersson, and D. Sundmark, *Mutating aspect-oriented models to test cross-cutting concerns*, Eleventh IEEE workshop on mutation analysis (Mutation'15), 2015, pp. 1–10.
- [37] Y. S. Ma, Y. R. Kwon, and J. Offutt, *Inter-class mutation operators for Java*, Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), 2002, pp. 352–363.
- [38] ———, *Inter-class mutation operators for Java*, Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), 2002, pp. 352–363.
- [39] Y. S. Ma, J. Offutt, and Y. R. Kwon, *MuJava : An automated class mutation system*, Software Testing, Verification, and Reliability, Wiley **15** (2005), no. 2, 97–133.
- [40] M. Mortensen and R. T. Alexander, *An approach for adequate testing of AspectJ programs*, Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD'05), 2005.
- [41] T. Mücke and M. Huhn, *Generation of optimized testsuites for uml statecharts with time*, Testing of Communicating Systems, pages=128–143, year=2004, publisher=Springer.
- [42] D. Di Nardo, F. Pastore, and L. Briand, *Generating complex and faulty test data through model-based mutation analysis*, 8th International Conference on Software Testing, Verification and Validation (ICST'15), 2015, pp. 1–10.
- [43] B. Nielsen and A. Skou, *Testing timed automata*, Software Tools for Technology Transfer **5** (2003), no. 1, 59–77.
- [44] R. Nilsson, J. Offutt, and S. F. Andler, *Mutation-based testing criteria for timeliness*, Proceedings of the 28th International Conference on Computer Software and Applications (COMP-SAC'04), 2004, pp. 306–311.
- [45] M. Nouh, R. Ziarati, D. Mouheb, D. Alhadidi, M. Debbabi, L. Wang, and M. Pourzandi, *Aspect weaver: A model transformation approach for uml models*, Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, 2010, pp. 139–153.
- [46] J. Pérezza, N. Ali, J. A. Carsi'b, I. Ramosb, B. Álvarez, P. Sanchez, and J. A. Pastor, *Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems*, Information and Software Technology **50** (2008), no. 9-101, 969–990.
- [47] J. L. Peterson, *Petri net theory and the modeling of systems*, Prentice Hall PTR, 1981.
- [48] S. Pimont and J. C. Rault, *A software reliability assessment based on a structural behavioral analysis of programs*, Proceedings of the Second International Conference on Software Engineering, 1976, pp. 486–491.
- [49] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg, *Directives for composing aspect-oriented design class models*, Transactions on Aspect-Oriented Software Development I, 2006, pp. 75–105.
- [50] S. D. R. S. De Souza, J. C. Maldonado, S. C.P.F. Fabbri, and W. L. De Souza, *Mutation testing applied to Estelle specifications*.
- [51] J. Springintveld, F. Vaandrager, and P. R. D'Argenio, *Testing timed automata*, Theor. Comput. Sci. **254** (2001), no. 1-2, 225–257.
- [52] M. Trakhtenbrot, *New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models*, Third IEEE Workshop on Mutation Analysis (Mutation 2007), 2007, pp. 151–160.
- [53] F. Wedyan and S. Ghosh, *On generating mutants for AspectJ programs*, Information and Software Technology **54** (2012), no. 8, 900–914.
- [54] J. Whittle and P. Jayaraman, *Mata: A tool for aspect-oriented modeling based on graph transformation*, Models in Software Engineering, 2008, pp. 16–27.
- [55] D. Xu, O. El-Ariss, W. Xu, and L. Wang, *Testing aspect-oriented programs with finite state machines*, Software Testing, Verification, and Reliability, Wiley **22** (2012), no. 4, 267–293.
- [56] D. Xu, W. Xu, and W. E. Wong, *Testing aspect-oriented programs with uml design models*, International Journal of Software Engineering and Knowledge Engineering **18**, no. 03, 413–437.