Scalable Model-based Robustness Testing: Novel Methodologies and Industrial Application

Shaukat Ali

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo

September 2011

*Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1154*

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Embedded systems, as for example communication and control systems, are being increasingly used in our daily lives and hence require thorough and systematic testing before their actual use. Many of these systems interact with their environment and, therefore, their functionality is largely dependent on this environment whose behavior can be unpredictable. Robustness testing aims at testing the behavior of a system in the presence of faulty situations in its operating environment (e.g., sensors and actuators). In such situations, the system should gracefully degrade its performance instead of abruptly stopping execution. To systematically perform robustness testing, one option is to resort to Model-Based Robustness Testing (MBRT), which is a systematic, rigorous, and automated way of conducting robustness testing. However, to successfully apply MBRT in industrial contexts, new technologies need to be developed to scale to the complexity of real industrial systems. This thesis presents a solution for MBRT on industrial systems, including scalable robustness modeling and executable test case generation.

One important contribution of this thesis is a scalable RobUstness Modeling Methodology (RUMM), which is achieved using Aspect-Oriented Modeling (AOM). It is a complete, automated, and practical methodology that covers all features of state machines and aspect concepts necessary for MBRT. Such methodology, relying on a standard (Unified Modeling Language or UML) and using the target notation as the basis to model the aspects themselves, is expected to make the practical adoption of robustness modeling easier in industrial contexts. The applicability of the methodology is demonstrated using an industrial case study. Results showed that the approach significantly reduced modeling effort (98% on average), improved separation of concerns, and eased model evolution. The approach is further empirically evaluated using two controlled experiments involving human subjects and results showed that the proposed methodology significantly improves the readability of models as compared to modeling using standard UML notations.

Another important contribution of this thesis is an efficient approach for solving constraints (written in Objects Constraint Language (OCL)) on the operating environment of a system, which is mandatory for emulating faulty situation in the environment for the purpose of MBRT. A set of novel heuristics is devised for various OCL constructs, which are required for the application of search algorithms. The heuristics have been empirically evaluated on an industrial case study for robustness testing and the results showed to be very promising and significantly better than the existing works in the literature on OCL constraint solvers.

A final contribution of the thesis is robustness test case generation from the models developed using RUMM. Test case generation also includes scripts generation for environment emulation, which is mandatory for automated robustness testing again using an industrial case study. In preliminary experiments, the execution of test cases found one critical, robustness fault in a deployed industrial system.

Acknowledgements

First of all, I would like to thank my supervisors Lionel Briand and Andrea Arcuri. Without their deep knowledge in software engineering and constant guidance throughout my PhD, this work couldn’t have been possible. Lionel is an excellent supervisor and his profound knowledge and passion for research has instilled in me a passion to conduct quality research. Andrea is an outstanding supervisor and his immense knowledge about search algorithms and statistics have provided me an opportunity to grasp these topics very quickly. I would also like to thank Erik Arisholm, who was my supervisor during earlier part of the PhD. I always had good discussions with him. I would also like to thank Simula Research Laboratory (SRL) and Simula School of Research and Innovation (SSRI) for funding my PhD and providing an excellent multicultural research environment.

I would like to thank Marius Liaaen from Cisco, whose intense passion for model-based testing enabled me to successfully apply and evaluate my model-based methodologies in Cisco. I would also like to thank Suneth Walawege from Cisco, who is also a friend, for helping in modeling Cisco's case study and running experiments.

I would like to specially thank my best friend Tao Yue, for her unconditional support in everything. Particularly, during my PhD she helped in conducting controlled experiments in Pakistan and China. Special thanks to my best friend and office mate, Aiko Yamashita for providing an excellent company in the office and outside. She has been a wonderful friend in many different ways. I would also like to thank my best friends in Simula, Rajwinder Panesar-Walawege, Zohaib Zafar, and Hadi Hemmati for all the good discussions at work and fun times in general.

Last but not least, I would also like to thank my parents, brother, and sisters for their love, support, and care.

List of papers

The following papers are included in this thesis:

Paper 1. A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation

S. Ali, L. Briand, H. Hemmati, and R. K. Panesar-Walawege

Published in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010

Paper 2. Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems

S. Ali, L. Briand, and H. Hemmati

Accepted for a publication in the Journal of Software and Systems Modeling (SOSYM), Springer, 2011.

Paper 3. Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines?

S. Ali, T. Yue, and L. Briand

Submitted to the the Journal of Software and Systems Modeling (SOSYM), Springer, 2011.

Paper 4. Solving OCL Constraints for Test Data Generation in Industrial Systems with Search Techniques

S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand.

Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), 2011

Paper 5. An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms

S. Ali, L. Briand, A. Arcuri, and S. Walawege.

In: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), ACM/IEEE, 2011.

The five papers are self-contained and thus some information might be redundant across the papers. Different abbreviations may have been used in the papers.

My contributions

For all papers except the first paper, I was the main contributor. My supervisors contributed in all phases of the work. For Paper 1, all authors equally contributed to the paper. In case of Paper 2, I was main contributor for the idea, implementation, and case study design and application. The controlled experiments reported in Paper 3 are conducted in collaboration with Tao Yue and Lionel Briand. I was responsible for creating the experiment material, experiment execution, data collection and analysis, and writing of the paper. Tao Yue and Lionel Briand were involved throughout the process. In Paper 4, I was the main contributor, but got help from supervisors (Andrea Arcuri and Lionel Briand) and collaborator Mohammad Zohaib Zafar. In the last paper, I was the main contributor for the idea, implementation, and case study design and application.

In addition, during my PhD study, I also contributed in other papers which are not included in this thesis. Paper 6 is not included since the journal version of the paper (Paper 4) is included in this thesis. Paper 7 is excluded since it is a result of an equal contribution from all authors and is a basic framework which was needed to conduct the research reported in this thesis. Paper 8 and Paper 9 are not directly related to this PhD thesis, but are preliminary extensions to the work presented in the thesis. Paper 10 is an additional controlled experiment, which was conducted to evaluate the modeling methodology presented in Paper 2. The paper was submitted to a conference at the time of submission of this thesis for evaluation.

Paper 6. A Search-based OCL Constraint Solver for Model-based Test Data Generation

S. Ali, M. Z. Iqbal, A. Arcuri, L. Briand

In: Proceedings of the 11th International Conference on Quality Software (QSIC 2011), pp. 41-50, IEEE, 2011.

Paper 7. Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies.

S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. Briand

Technical Report 2010-01, Simula Research Laboratory

Paper 8. Automated Transition from Use Cases to UML State Machines to Support State-based Testing.

T. Yue, S. Ali, and L. Briand.

In: Seventh European Conference on Modeling Foundations and Applications (ECMFA), 2011

Paper 9. An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study.

H. Hemmati, L. Briand, A. Arcuri, and S. Ali.

In: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), ed. by Gruia-Catalin Roman and André van der Hoek, ACM (ISBN: 978-1-60558-791-2), 2010.

Paper 10. Comprehensively Evaluating Conformance Error Rates of Applying Aspect State Machines for Robustness Testing,

S. Ali, T. Yue, Z. Malik,

Accepted for a Publication in International Conference on Aspect-Oriented Software Development (AOSD 2012), 2012

Contents

Summary

1 Introduction

In our daily life, many important activities are directly or indirectly dependent on embedded, control and communication systems [1]. For instance, the use of smart phones and tele-presence systems has been quickly increasing. The correct functioning of these systems largely depends on their operating environment, whose behavior is inherently unpredictable. Assuring a reasonable dependability level for these systems in the presence of dysfunctional or hostile environment conditions is very important and such behavior is commonly denoted as *robustness behavior*.

Robustness, as defined by an IEEE Standard [2], is *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions"*. Such robustness is considered very critical in many standards such as in the IEEE Standard Dictionary of Measures of the Software Aspects of Dependability [3], the ISO's Software Quality Characteristics standard [4], and the Software Assurance Standard [5] by NASA. A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, in our industrial case study (Cisco Systems, Norway), modeling such robustness behavior for a videoconferencing system (VCS) means to model its behavior in the presence of problems with the network and other communicating VCSs, such as high percentage of packet loss and high percentage of corrupt packets. The VCS should not crash, halt, or restart in the presence of such problems. Furthermore, the VCS should continue to work in a degraded mode, such as continuing the videoconference with low audio and video quality. In the worst case, the VCS should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS and must be tested systematically and automatically to be scalable.

In practice, robustness testing is often manual and hence is not rigorous and is limited in scope. To perform rigorous, systematic, and automated robustness testing, which eventually reduces the number of faults in the delivered systems, one needs to resort to testing methodologies such as model-based testing (MBT). This thesis presents a solution to Model-based Robustness Testing (MBRT), which includes the contributions discussed below:

- A RobUstness Modeling Methodology (RUMM) that enables the systematic modeling of robustness behavior in a practical and scalable way. The main contributions with respect to RUMM are:
    1) A UML 2.0 profile (RobustProfile), which is based on a fault taxonomy in [6] and the IEEE standard classification for anomalies [7], to model faults, recovery mechanisms, and failure states.
    2) The application of the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile [8] in conjunction with RobustProfile to model faulty environment conditions.
    3) A UML 2.0 profile (AspectSM) to support comprehensive aspect modeling for UML 2.0 state machines and enable automated robustness testing. AspectSM supports modeling crosscutting behavior on all modeling elements of UML 2.0 state machines and supports all basic features of AOSD such as pointcuts, introduction, joinpoints, and advice.
    4) An empirical evaluation and discussion of the benefits of modeling robustness behavior of an industrial system using RUMM and AspectSM.
    5) Tool support, based on model transformations in Kermeta [9], to automatically weave AspectSM aspects into base state machines (modeling the core functional behavior of a system).
    6) Empirical evaluation of the benefits provided by the methodology via two controlled experiments.
- Robustness behavior is modeled based on different functional and non-functional properties (e.g., bandwidth of a communication network), whose violations lead to faulty situations. Such properties can be related to the System

Under Test (SUT) or its environment such as the network and other systems interacting with the SUT. These properties are modeled as constraints (e.g., bandwidth = 0 kilo bits per second) which need to be solved to emulate faulty situations and thus automate robustness testing. In our context, such properties are modeled using the Object Constraint Language (OCL) [10], which is a standard language to express constraints on UML models based on first order logic and set theory. To solve these constraints, this thesis reports on novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), and Alternating Variable Method (AVM), to solving OCL constraints (covering the entire OCL 2.2 semantics [10]) in order to generate test data to emulate faulty situations. A search-based OCL constraint solver is implemented and evaluated on the first reported, industrial case study on this topic.

- Finally, to successfully apply MBRT in industrial contexts, new technologies need to be developed to scale to the complexity of real industrial systems. This thesis finally reports on our experience of performing MBRT on VCSs developed by Cisco Systems, Norway and discuss various techniques and tools that are developed and integrated to achieve a fully automated MBRT that is able to detect previously uncaught software faults in those systems.

This thesis has two parts:

Summary: This part of the thesis consists of the following sections: Section 2 provides the background information required to understand the summary of the thesis. Section 3 briefly presents the contributions of the thesis and Section 4 presents the research methodology that was followed during the thesis. Section 5 provides salient results from the research papers submitted as part of the thesis; Section 6 outlines future research directions, whereas Section 7 concludes the thesis.

Papers: The second part of the thesis presents the published or submitted research papers, which are included in this thesis.

2 Background

In this section, the background knowledge necessary to understand the rest of the thesis is presented. Section 2.1 introduces modeling with the Unified Modeling Language (UML), including a discussion on UML extension mechanisms, and Section 2.2 provides a brief overview of the Object Constraint Language (OCL). Section 2.3 provides an introduction to Aspect-Oriented Modeling (AOM), Section 2.4 provides details on testing topics, i.e., Model-Based Testing (MBT) and Search-Based Software Testing (SBST). Finally, Section 2.5 provides a brief introduction to various types of empirical evaluations conducted in this thesis.

2.1 UML-based Modeling

UML [11] is the de-facto standard for modeling software systems. UML provides a unified, precise, and consistent way to communicate information among different people involved in software development. The architecture of UML is standardized based on the Meta-Object Facility (MOF) [12], which provides standardized mechanisms for defining new modeling languages. MOF is a metamodel [12], which precisely defines the concepts necessary to define a modeling language. The semantics of UML are captured in its metamodel, which is an instantiation of the MOF metamodel.

UML provides a variety of diagrams for various purposes. For example, the static structure of software systems can be modeled using diagrams such as class and composite structure diagrams. Similarly, the behavior of software systems can be modeled using diagrams such as state machines and activity diagrams. In addition, UML diagrams can be defined at different levels of details. For instance, a UML model can be defined at a coarse-grained level of detail for the purpose of analysis or communication with a stakeholder (e.g., a high level domain model). Alternatively, a UML model can be at a fine-grained level of details for the purpose of automated code generation and test case generation (e.g., a state machine including action specifications).

2.1.1 UML Extension Mechanism

UML provides a standard extension mechanism that allows adding extensions to the semantics of pre-defined UML concepts using a UML profile. A UML profile enables the extension of UML for different domains and platforms, while avoiding any inconsistency with UML semantics. A profile allows defining stereotypes (annotations), on metaclasses defined in the UML metamodel, which can have attributes representing properties of the stereotype. For instance, in Figure 1, a stereotype *Motor* is defined on the *Class* metaclass of the UML metamodel. The arrow from *Motor* to *Class* is an extension arrow showing that *Motor* can be applied to an instance of the UML metaclass *Class*. The *Motor* stereotype has an attribute named *type*, which is of type *MotorType* Enumeration having two literals *Electrical* and *Mechanical*. Figure 2 shows an example of applying the *Motor* stereotype, which is applied to a *StepperMotor*, which is an instance of UML metaclass *Class*. The attribute of *Motor* is set to *Electrical* telling that the stepper motor is an electrical motor.

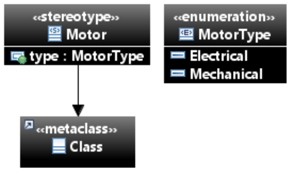


Figure 1 An example of defining a stereotype on a UML metaclass

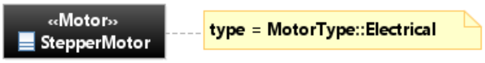


Figure 2 An example of applying a stereotype

Two main approaches for profile creation are discussed in [13]. The first approach directly implements a profile for key concepts of a target domain. Such an approach has been used to define the Systems Modeling Language (SysML) [14]. The second approach involves first creating a conceptual model outlining the key concepts of a target domain followed by creating a profile for the identified concepts. This latter approach has been used for defining profiles such as Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [8], QoS and Fault Tolerance specifications [15], and UML Testing Profile (UTP) [16]. The second approach is more systematic than the first one as it separates the profile creation process into two stages. In the first stage, we develop a conceptual model which helps identify domain concepts and their relationships. In the second stage, we identify a mapping between the main concepts and UML modeling elements and define corresponding stereotypes on UML metaclasses. Finally, the relationships between stereotypes are obtained from the relationships that were identified between the domain concepts in the first stage.

2.1.2 Standard UML Profiles

A set of UML profiles has been standardized by the Object Management Group (OMG) including QoS and Fault Tolerance specifications [15], and UML Testing Profile (UTP) [16], and Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [8]. The most relevant profile in the context of this thesis is MARTE, which is specifically defined to model real-time embedded systems. The MARTE profile has a package dedicated to modeling non-functional properties (NFP). It provides different data types, such as *NFP_Percentage* and *NFP_DataTxRate*, which are helpful to model properties of the environment, for instance *jitter* and *packet loss* in communication networks. When the built-in data types of MARTE are not sufficient, the open modeling framework of MARTE can be used to define new NFP types by either extending the existing NFPs or by defining completely new NFPs.

2.2 Object Constraint Language (OCL)

OCL [10] is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and is at a higher expressive level than Boolean predicates written in programming languages such as C and Java. The language allows modelers to write constraints at various levels of abstraction and for various types of models. It can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post conditions of operations. A basic subset of the language has been defined that can be used with metamodels defined in Meta Object Facility (MOF) [12] (which is a standard defined by Object Management Group (OMG) for defining metamodels). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles.

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger and/or state variables) to achieve full transition coverage, then it is essential to provide input values to the event that triggers the transition such that the values satisfy the guard. Another example can be to generate valid parameter values based on the pre-condition of an operation.

2.2.1 OCL Evaluator

An OCL evaluator tells whether a constraint on a class diagram satisfies an instantiation of the class diagram provided to it. Several OCL evaluators are currently available that can be used to evaluate OCL constraints such as the IBM OCL evaluator [17], OCLE 2.0 [18], EyeOCL [19], and OCL evaluation in CertifyIt by Smarttesting [20].

2.2.2 OCL Constraints Solver

The purpose of an OCL constraint solver is to verify whether the constraints can be satisfied. In other words, an OCL constraint solver checks whether correctness properties such as satisfiability or absence of contradictory constraints hold for a model with constraints written with OCL. This is achieved by searching a solution, which is simply a set of values of variables used in a constraint, which satisfies the constraint. In the context of UML and OCL, it is a valid instance of a class diagram. If any such solution exists, it means the satisfiability property holds.

A number of approaches use constraint solvers for analyzing OCL constraints for various purposes. These approaches usually translate constraints and models into a formalism (e.g., Alloy [21], temporal logic BOTL [22], FOL [23] , Prototype Verification System (PVS) [24], graph constraints [25]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [26], model checker [22], Satisfiability Modulo Theories (SMT) Solver [23], theorem prover [23], [24]).

2.3 Aspect-Oriented Modeling (AOM)

Aspect-orientation provides enhanced modularization by separating out crosscutting concerns as separate entities called aspects. Aspect-orientation is a very active field [27] of research, which has mainly focused on aspect-oriented programming (AOP), but also led to significant progress in the realms of design and modeling, denoted as aspect-oriented Modeling (AOM) [27-29]. Crosscutting concerns, for example related to robustness or security behavior, are modeled as aspect models and are subsequently woven into a primary/base model capturing nominal functional behavior. AOM is expected to yield benefits such as improved readability, enhanced modularization, easier evolution, and increased reusability of models, as well as reduced modeling effort [28, 29].

Since the standard UML notations are not enough to support AOM, several extensions have been proposed in the literature. All these extensions are broadly classified as either extensions based on defining a UML profile to support AOM [30-36] or defining a new Domain Specific Language (DSL) to support AOM [37, 38]. A DSL defines a new language, which does not follow standard notations, and thus may be difficult to transfer to some industrial contexts.

2.4 Testing

This section provides an overview of MBT (Section 2.4.1) and SBST (Section 2.4.2), which are necessary to understand the rest of the thesis.

2.4.1 Model-based Testing (MBT)

Deriving test cases from a behavioral model of a system, known as MBT [39], is not a new domain of research in software engineering. However, in recent years, the level of interest for MBT in industry and academia has been rapidly increasing. This interest can be seen from the many academic studies [40-43] and industrial projects [20, 44] on model-based testing being reported. This suggests that there is an increasing awareness of the benefits offered by MBT.

The general process of MBT starts with modeling a SUT and making the resulting model ready for test generation. The next step is deriving abstract test cases from the test ready model according to a test strategy, which is typically defined based on a test model and coverage criteria to guide its traversal [39]. Finally, executable test cases are generated using abstract test cases and input test data.

In the context of MBT based on UML models, most techniques have focused on state-based testing (SBT). This is due to the reason that many systems, such as real-time and embedded systems [45, 46], and multimedia systems [47], exhibit state-driven behavior. UML state machines, which are extensions of traditional Finite State Machines (FSM), can be used to model such behavior. Traditional FSMs cannot model software systems with concurrent behavior. Concurrency in UML state machines is modeled using composite states with two or more regions [11]. When modeling complex software systems with FSMs, the number of states and transitions can grow exponentially with system size. This can be handled by UML state machine features for modeling submachine states.

To apply MBT on UML state machines as the input model, several test strategies are presented in the literature, such as piecewise, all transitions, all transitions k-tuples, all round-trip paths, M-length signature, and exhaustive coverage [48]. For example, for all round-trip strategy, a test tree also known as a transition tree (consisting of nodes and edges corresponding to states and transitions in a state machine) is constructed by depth-first traversal of the state machine. A node in the transition tree is a terminal node if the node already exists anywhere in the tree that has been constructed so far or is a final state in the state machine. Now, by traversing all paths in the transition tree, we cover all round trip paths and all simple paths (the paths in the state machine that begins with the initial state and ends with the final state) [48]. Another stopping criterion for the transition tree construction is proposed in [49], where a node is terminal if (i) it is a final state of the state machine or (ii) it is a node that already exists on the path that leads to the node. This stopping criterion makes the all round-trip strategy more demanding. This strategy has been experimentally evaluated to be more cost-effective than the all transitions and all transition pairs criteria [49]. Henceforth, the transition tree or all round-trip paths coverage criterion refer to the modified versions proposed in [49].

To automate testing based on UML state machines, test data must be generated to fire triggers associated with transitions, which typically require parameter values. Test data can be generated randomly from the possible set of values, or using more sophisticated techniques such as constraint solvers [50, 51], or search-based techniques (for example using Genetic Algorithms for test data generation [52, 53]) .

Constraints defined on UML state machines, such as state invariants, guards, and pre/post conditions of triggers, should be evaluated during the execution of the generated test cases. As shown by many studies, constraints are a very effective way to detect faults [54, 55], e.g., state invariants serving as oracles in state-based testing. These constraints are usually written as OCL expressions in the context of UML models.

2.4.2 Search-based Software Testing (SBST)

Several software engineering problems can be reformulated as a search problem, such as test data generation [56]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good'' solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results of using search algorithms are reported in the literature for many types of software engineering problems [52, 57, 58].

To use a search algorithm, a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search algorithms toward fitter solutions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms are the most well-known [56], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. Alternating Variable Method (AVM) is a local search algorithm, which is similar to the Hill Climbing (HC) algorithm. The algorithm performs an exploratory search on a variable, where the value of the variable is slightly modified every time. If the fitness of the modified variable value is better than the previous one, the current value becomes the optimum solution. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (RS) as a comparison baseline [56].

2.5 Empirical Evaluation

According to [59] empirical studies can be classified as controlled experiments, case studies, and surveys.

2.5.1 Case Study

A case study is an in depth observational study [59], whose purpose is to examine projects or evaluate a system under study. Case studies are mostly used to demonstrate the applicability of an approach. For example, in the context of software modeling, the goal of a case study could be to demonstrate the applicability of modeling non-functional behavior (e.g., security) of an industrial context and reporting lessons learnt during this process.

2.5.2 Survey/Systematic Reviews

A survey is an exploration of existing literature and is a descriptive research method with no control over measurements. Surveys summarize existing works in an abstract and a broader sense. In contrast, a systematic review is a type of a survey, which is more systematic in the sense that all the steps of conducting a systematic review are reported. This includes the search query used, search engines used, research questions, and so on. In other words, a systematic review is a way to collect all existing information about a particular topic or area of research in such a manner that limits bias in the gathering, critical evaluation and synthesis of all relevant studies. Initially, systematic reviews were conducted in medicine and other social sciences. Later on it was adopted in almost every field of science, including computer science. The purpose of a systematic review is to find gaps in the existing literature to place new proposals in the context of existing knowledge. It provides the foundation for proposing future research agendas and make recommendations for research [60].

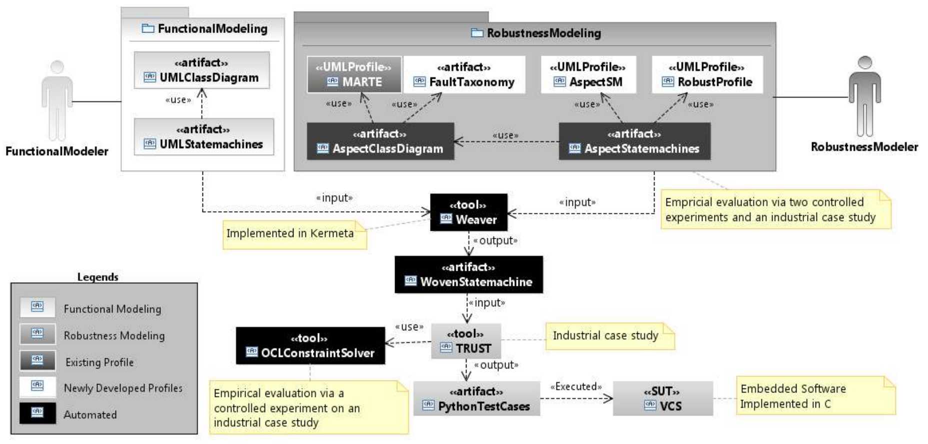
2.5.3 Controlled Experiment

A controlled experiment [59] is performed in a controlled environment with the aim to manipulate one or more variables and maintain other variables at fixed values to measure the effect of change (outcome). Experiments usually have two or more treatments, whose outcomes are to be compared. For example, in the context of software modeling, an example of a controlled experiment could be to measure effort required by two techniques, which are developed to solve a particular modeling problem, such as modeling non-functional properties.

3 Automated Model-based Robustness Testing

Modeling software functional behavior has always been an important focus of the modeling community to support many development activities such as model-based testing (MBT) and automated code generation. Regarding MBT, which is the specific focus of this thesis, much less attention has been given to modeling non-functional behavior such that the testing of non-functional properties (e.g., safety and robustness) can be automated. Though several UML profiles have been proposed to address the modeling of non-functional properties (including the UML profile for QoS and Fault Tolerance [15], the MARTE profile [8], and UMLSec [61]), it is not yet clear whether they can fully support test automation.

The main motivation of this thesis is to provide a complete solution to MBRT and in this respect Figure 3 provides an overview of its contributions. Overall the thesis can be divided into two main parts: robustness modeling and robustness testing.



FunctionalModeling
«artifact» UMLClassDiagram
«use»
«artifact» UMLStatemachines
FunctionalModeler
RobustnessModeling
«UMLProfile» MARTE
«artifact» FaultTaxonomy
«UMLProfile» AspectSM
«UMLProfile» RobustProfile
«use»
«use»
«use»
«use»
«artifact» AspectClassDiagram
«use»
«artifact» AspectStatemachines
RobustnessModeler
«input»
«tool» Weaver
«input»
Empricial evaluation via two controlled experiments and an industrial case study
Implemented in Kermeta
«output»
«artifact» WovenStatemachine
Legends
Functional Modeling
Robustness Modeling
Existing Profile
Newly Developed Profiles
Automated
«input»
«use»
«tool» OCLConstraintSolver
«tool» TRUST
Industrial case study
«output»
Empirical evaluation via a controlled experiment on an industrial case study
«artifact» PythonTestCases
«Executed»
«SUT» VCS
Embedded Software Implemented in C

Figure 3 Scope of the thesis

3.1 Modeling Robustness Behavior

Robustness behavior is typically crosscutting many parts of the system functional behavior and, as a result, modeling such behavior directly within the functional models is not practical since it leads to many redundancies and hence results in large, cluttered models. To cope with this issue, we decided to adopt Aspect-Oriented Modeling (AOM) [27], which provides Separation of Concerns (SoC) during design modeling. Crosscutting concerns are modeled as aspect models and are woven into a primary model (base model), modeling non-crosscutting concerns (e.g., nominal functional behavior). AOM can potentially offer several benefits such as: 1) enhanced modularization, 2) easier evolution of models, 3) increased reusability, 4) reduced modeling effort, and 5) improved readability [27, 37].

Our aim is to provide a complete solution in terms of both aspect and state machine features necessary for MBRT. Furthermore, we want to minimize the effort involved in learning a new language over standard UML and enable automated MBT. To achieve this, we developed a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM and assessed it on an industrial case study involving a commercial videoconferencing system. Such studies are rare in the research literature and are rarely run and reported in a satisfactory manner [62]. To the knowledge of the authors, only a few industrial applications of AOM have been reported to date [31, 63-65] and had very different objectives than RUMM. The core of RUMM is the definition of a UML state machine profile for AOM: AspectSM (shown as a white artifact in Figure 3 in RobustnessModeling). We limited our profile to UML state machines as: 1) They are the main notation currently used for model-based test case generation [66] and are particularly useful in control and communication systems, 2) Like it is often the case, our industrial case study exhibits state-based behavior so that it is natural to initially provide support for UML state machines. The profile can, however, be extended to other UML diagrams in the future, following similar principles. We rely on developing a profile instead of developing a domain specific language since, in our case study context as in many others, minimizing extensions to UML is expected to ease practical adoption. Modelers of functional aspects of the system can be different from the ones specifying its robustness behavior. The latter make use of AspectSM to model aspect state machines.

Additionally, we evaluated the "readability" of state machines when modeling crosscutting behavior using AspectSM. Readability is indirectly measured through defect identification and fixing rates in state machines, and the scores obtained when answering a comprehension questionnaire about the system behavior. With AspectSM, crosscutting behavior is modeled using so-called "aspect state machines". Their readability is compared with that of system state machines directly modeling crosscutting and standard behavior together. An initial controlled experiment and a much larger replication were conducted with trained graduate students, in two different institutions and countries, to achieve the above objective. We used two baselines of comparisons—standard UML state machines without hierarchical features (flat state machines) and standard state machines with hierarchical/concurrent features (hierarchical state machines).

Another important part of the RUMM is another UML profile (RobustProfile) shown as a white artifact in Figure 3, based on the fault taxonomy defined by [67] and the IEEE standard classification for anomalies [7]. The profile is used by a robustness modeler to develop aspect state machines and is defined specifically to assist in defining test strategies for robustness testing. In addition, the profile helps generating test scripts based on classes of faults modeled using the profile. Once again, the profile is defined on UML state machines, as they are the main focus of this paper. We follow the widely accepted and used definitions in [67] for faults and failures. A fault is an incorrect state of a system or its environment in the presence of which the system cannot provide a correct service. Such deviation from the correct service is called a failure. A fault type is identified based on a fault taxonomy (white artifact in Figure 3) and the UML profile MARTE is used to model it in a UML class diagram (*AspectClassDiagram*, dark grey artifact in Figure 3). In a subsequent step, aspect class diagrams are used to model actual faulty behavior as aspect state machines (*AspectStatemachines*) using both AspectSM and RobustProfile.

The aspect state machines developed with RUMM, and more specifically using AspectSM, are woven into the base state machine by a weaver. Ordering between different aspect state machines is specified using a weaving-directive state machine, which is a standard UML state machine with control flow features. The weaver reads the base state machine, aspect state machines, and a weaving-directive state machine and produces a woven state machine. The weaver is developed using Kermeta [9], which is a metamodeling language [68] that allows manipulating models by defining transformation rules at the metamodel level. Kermeta conforms to OMG's metamodeling language Essential Meta Object Facility (EMOF) and Ecore [68]. Figure 4 shows the architecture of the weaver by using transformations in Kermeta to weave one or more aspect state machines into a base state machine. The AspectSM profile is defined on the UML 2.0 metamodel. An aspect state machine is defined as a UML 2.0 state machine by applying the AspectSM profile. A base state machine is a standard UML 2.0 state machine. Transformations rules in Kermeta are defined on the UML 2.0 metamodel and the AspectSM profile. Finally, the Kermeta engine uses the transformation rules that read an aspect state machine and the base state machine and weaves the aspect state machine into the base state machine. The Kermeta engine then produces a woven state machine, which is again an instance of the UML 2.0 metamodel, since the woven state machine is a standard UML 2.0 state machine. The woven state machines can then be used as input for the TRansformation-based tool for Uml-baSed Testing (TRUST) (Section 3.2). The weaver is fully automated and does not require any additional inputs from the user apart from aspect state machines and a base state machine.

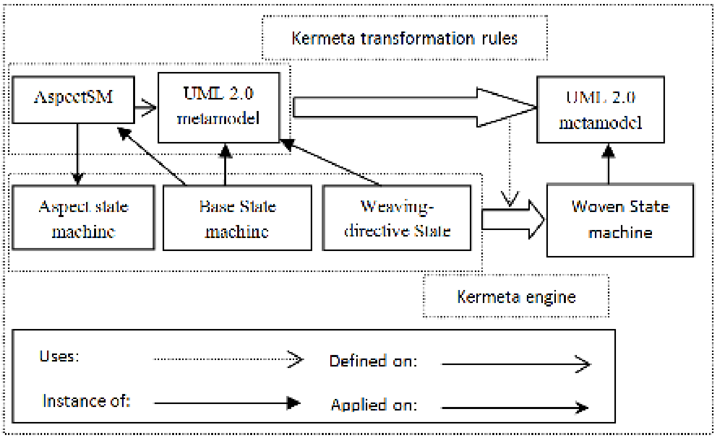


Figure 4 Aspect weaver implemented in Kermeta

Finally the woven state machines produced by the weaver are in turn used by the TRUST tool [69] to generate executable test cases. Other MBT tools such as Conformiq Qtronic [44] and CertifyIt [15] cannot be used in our context for robustness test case generation since the woven state machines contain concepts from RobustProfile (Section 3.1), which facilitates defining specialized robustness coverage criteria for robustness test case generation.

3.2 Model-based Robustness Testing

The second main contribution of this thesis is Model-Based Robustness Testing (MBRT). As it can be seen in Figure 3, test cases are generated with TRUST. It is an MBT tool, which was initially developed for functional model-based testing. The tool was developed by using a software architecture and implementation strategy that can easily facilitate its customization to different contexts by supporting extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages (Figure 5). For example, the tool is extensible with respect to coverage criteria if it lets the user select among several coverage criteria such as all transitions and all round trip path coverage criteria [48].

To generate robustness test cases with TRUST, we needed to solve complex OCL constraints on the properties of the environment emulating faulty situations. These constraints must be solved during test case generation to set the environment properties in such a way as to trigger faulty situations. To efficiently solve these constraints, we developed a search-based OCL constraint solver [70], since current OCL solvers were not able to handle the complexity of our model's constraints within reasonable time. Figure 6 shows the architecture diagram for our Search-based Constraint solver. The tool is developed in Java, and it interacts with an existing library, an OCL evaluator called the EyeOCL Software (EOS) [19]. EOS is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our constraint solver implements the heuristics (e.g., branch distance) that were defined for various expressions in OCL. To calculate the branch distance for an OCL expression, the expression is sent to EyeOCL for parsing, which returns a parse tree of the expression. The parse tree is further manipulated such that a branch distance can be calculated, which heuristically tells how far input data are from satisfying a constraint. The manipulated tree is reconstructed as an OCL expression. The reconstructed expression is sent to EyeOCL with the current set of values for variables in the expression and a branch distance is obtained. The search algorithms were also implemented in Java and includes Genetic Algorithms, (1+1) Evolutionary Algorithm, and Alternating Variable Method [70].

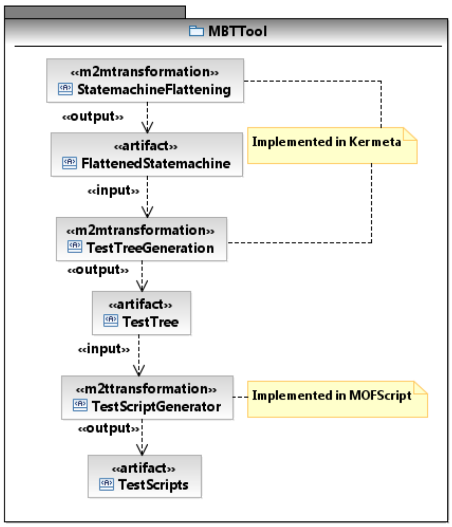


Figure 5 Details on the TRUST tool

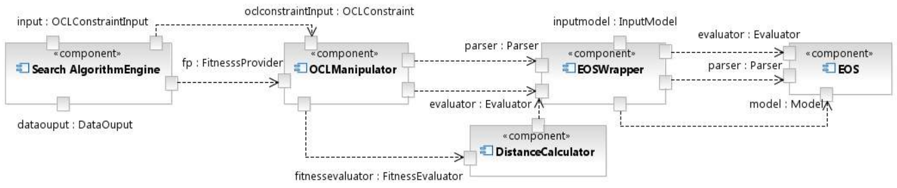


Figure 6 Architecture of OCL constraint solver

Finally, TRUST generates test cases in Python, which is used as a test script language by our industry partner (Cisco Systems, Norway). To execute generated robustness test cases, a test setup is needed to execute them. Figure 7 shows the test execution setup for executing robustness test cases generated by TRUST. The current setup involves *Saturn*, which is the system under test (*SUT*) and three video conferences systems (*VCSs*). Since the execution of test cases requires emulating faulty situations in the environment, we needed a network emulator. For this purpose, a software-based emulation facility (netem [71]) is used. The setup of network emulator requires setting up a PC with three network interface cards (*NICs*). All communication to/from *Saturn* (*SUT* in Figure 7) passes through *NetworkEmulator*. *Saturn* is connected to *NIC3* of *NetworkEmulator* and all incoming and outgoing traffic from *Network* comes through *NIC1*. *NIC1* is bridged to *NIC3* and hence all the traffic goes to *Saturn* via *NIC3*. Our test case execution system is directly connected to *NIC2* of *NetworkEmulator* and through this *NIC* all faulty situations in the network are introduced by test scripts. All other communication from the test execution system to *SUT* and *VCSs* takes place through *NIC2* of *NetworkEmulator*. This separation was necessary because if the faulty situations are introduced via the same *NIC* as other communication flows, we might end up affecting the commands that introduce faulty situations. Thus, we may end up not introducing faulty situations at all.

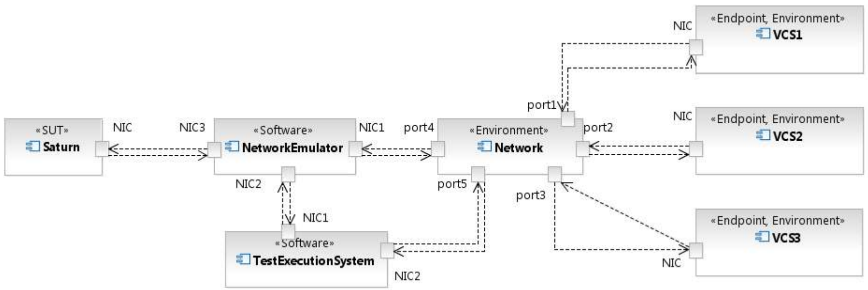
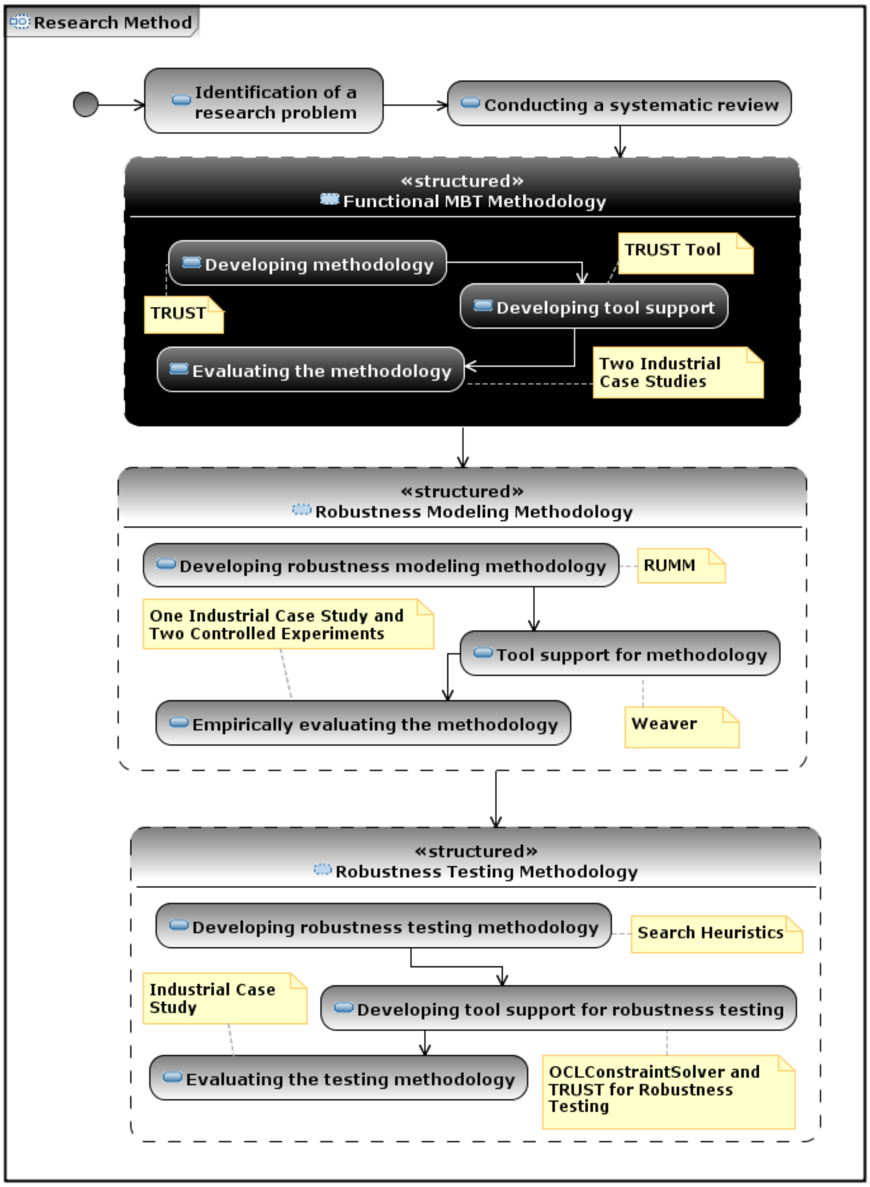


Figure 7 Test execution setup for robustness test execution

4 Research Methodology

This section of the thesis presents the research methodology that was followed in this thesis. The methodology is explained with the help of different research activities which are shown in Figure 8.



Research Method
Identification of a research problem
Conducting a systematic review
«structured»
Functional MBT Methodology
Developing methodology
TRUST Tool
TRUST
Developing tool support
Evaluating the methodology
Two Industrial Case Studies
«structured»
Robustness Modeling Methodology
Developing robustness modeling methodology
RUMM
One Industrial Case Study and Two Controlled Experiments
Tool support for methodology
Empirically evaluating the methodology
Weaver
«structured»
Robustness Testing Methodology
Developing robustness testing methodology
Search Heuristics
Industrial Case Study
Developing tool support for robustness testing
Evaluating the testing methodology
OCLConstraintSolver and TRUST for Robustness Testing

Figure 8. Research activities conducted in the thesis

4.1 Identification of a Research Problem

To ensure practical relevance, the thesis initiated with the identification of a research problem in an industrial setting, where in our case the industrial partner was Cisco Systems, Norway. More specifically, it involved the division of Cisco that develops video conferencing systems. The process included several discussions with a test manager and testing team members at Cisco. The current testing activities at Cisco were discussed along with the problems and challenges faced, while conducting testing. After several detailed discussions, it was decided that robustness testing was a high impact problem and that providing support for MBT was a priority, with a focus on MBRT in this thesis.

4.2 Conducting a Systematic Review

After identification of the research problem with our industry partner, we decided to conduct a systematic review on SBST. We decided so because MBT, although making testing very systematic and thorough, nevertheless requires sophisticated automation techniques. For instance, to automate test script generation, test data are required to be generated such that test cases can be executed without any human intervention. More specifically, in the context of MBRT, constraints on the environment of a SUT must be solved to automate faulty situations in the environment. The application of search-based heuristics for MBT has received significant attention recently (e.g., [53], [72]) and proved to be very beneficial in automating many parts of MBT such as test data generation [52, 70, 73] and test case selection [74, 75]. The systematic review on SBST helped us to understand the existing literature on SBST and helped us to identify how to properly conduct and report experiments involving search algorithms.

4.3 Functional MBT

This activity is the foundation for MBRT since robustness testing is not possible without conducting functional MBT. Robustness testing is indeed concerned with testing the behavior of a system, when something unexpected happens in its operating environment. This activity was performed in collaboration with two other PhD students and is not included as a contribution of this thesis. The activity involves three steps, which are discussed below:

4.3.1 Development of Functional MBT Methodology

We proposed a methodology for developing MBT tools, TRansformation-based tool for Uml-baSed Testing (TRUST). The methodology facilitates its customization to different contexts by supporting extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages. For example, the tool is extensible with respect to coverage criteria if it lets the user easily incorporate new coverage criteria (e.g., all transitions and all round trip path coverage criteria [48]) with minimum changes to the existing tool .

The approach that we have taken for implementing TRUST (Figure 9) is based on model transformations. The idea (inspired by Model-Driven Architecture (MDA) [12] concepts), is to generate a test model using a series of horizontal (endogenous and exogenous) model–to-model transformations on an input design model, modeling the Platform Independent Model (PIM) [12] of a system. Then, a vertical, exogenous model–to-text transformation is used to generate test scripts.

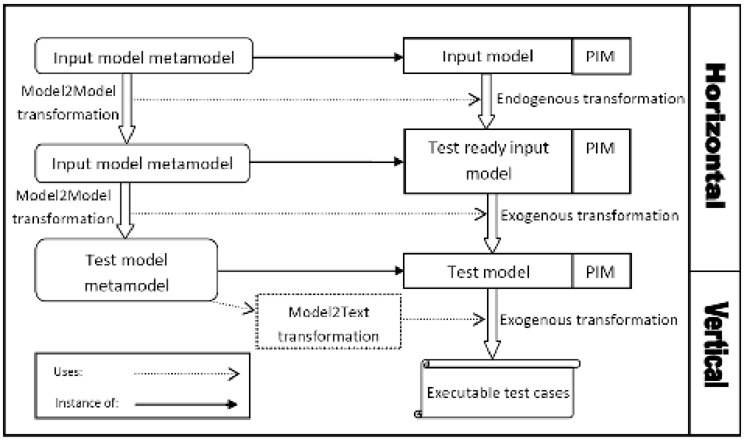


Figure 9 Model transformation-based approach for test case generation

4.3.2 *Tool Support*

We instantiated TRUST for UML state machines as 1) they are the main notation currently used for model-based test case generation [47] and are particularly useful in control and communication systems, 2) our industrial case study exhibits state-based behavior so that it is natural to provide support for UML state machines. The tool can, however, be instantiated to other UML diagrams in the future, following similar principles. The tool accepts UML 2.0 state machines with a support for concurrency and hierarchy. Constraints on state machines are written in OCL because it is an OMG standard for writing constraints on UML diagrams. Furthermore, the general model transformation-based approach, given in Figure 9, is instantiated on UML 2.0 state machines, as shown in Figure 10.

4.3.3 Evaluation of TRUST

As the aim of our approach is to provide a framework that allows instantiating new, context specific MBT tools by extending TRUST with customized features, such as input models, test models, coverage criteria, test data generation strategies, and test script languages. To demonstrate the extensibility of TRUST, we instantiated two tools for two industry partners, by extending TRUST with different test models, coverage criteria, and test scripting languages. On the basis of these case studies, we also evaluated the costs, challenges, and likely benefits of using TRUST in industrial case studies.

The companies where the case studies took place are international leaders in their respective fields. In both case studies, the models represent the state behavior of real world systems and the generated test cases are executable on the companies' testing platforms. The first company is Cisco Systems, Norway, whereas the name of the second company is not disclosed due to confidentiality restrictions. Both state machines are complemented by constraints specifying state invariants, which will be useful to derive automated test oracles. The first case study is the core subsystem of a video conferencing system, whereas the second case study is a safety monitoring component in a safety-critical control system. Both cases are suitable choices since these systems exhibit a complex state-based behavior that can be modeled as UML state machines.

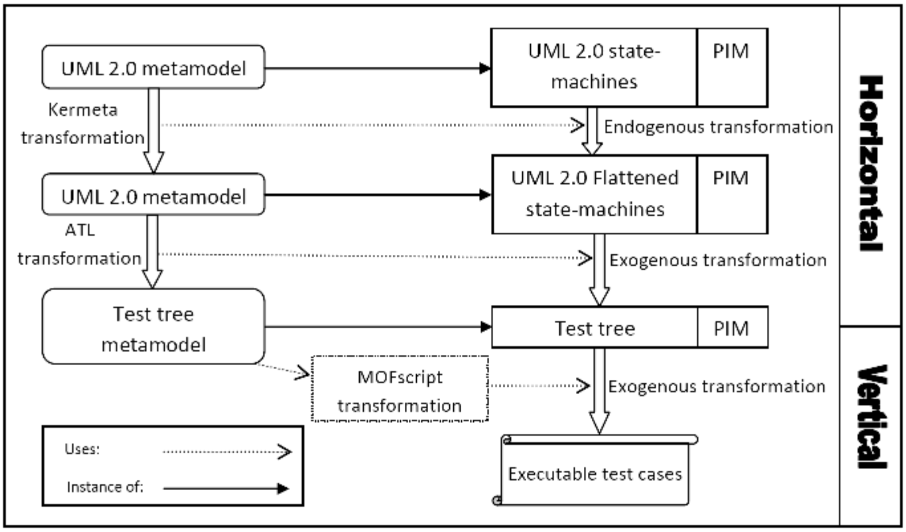


Figure 10 Model transformation-based approach for TRUST when configured for state machines

4.4 Robustness Modeling

This section describes the contributions of this thesis for robustness modeling. First, a brief overview of methodology is presented in Section 4.4.1, followed by automation in Section 4.3.2, and finally empirical evaluation in Section 4.4.3.

4.4.1 Methodology

To model robustness behavior, a novel solution is devised, i.e., RobUstness Modeling Methodology (RUMM) which (1) is complete in terms of aspect and state machine features, (2) minimizes the learning curve over standard modeling skills, and (3) enables automated MBT. The RUMM methodology (Section 3.1) is suitable for systems which implement substantial robustness behavior to deal with faulty situations in the environment, such as communication and control systems.

4.4.2 Tool Support

To automatically weave aspect state machines modeled using AspectSM for automated test case generation with TRUST, we developed a weaver whose details are provided in Section 3.1.

4.4.3 Empirical Evaluation

The empirical evaluation of robustness modeling is conducted based on an industrial case study and two controlled experiments, which are briefly discussed below.

Industrial case study: RUMM is applied to an industrial case study, which is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn [76]. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. To demonstrate the applicability of RUMM, this particularly important subsystem (Saturn) was modeled and other functionalities of the VCS were left out. This subsystem was selected because robustness testing is concerned with testing the behavior of VCS in the presence of abnormal environment situations, which can only be tested when the VCS is in a conference call with other systems, which is what Saturn manages. Saturn is complex enough to demonstrate the applicability and usefulness of RUMM while still remaining manageable in the context of a case study.

Controlled experiments: Two controlled experiments (one initial experiment and its large scale replication) were conducted to evaluate the “readability” of state machines modeling crosscutting behavior using AOM, in our context AspectSM. “Readability” means the ease with which state machines can be understood, analyzed, and changed by a human to perform various tasks. Models developed with AspectSM are compared with UML state machines modeling crosscutting behavior directly. The two controlled experiments were conducted in two different geographical locations (Pakistan and China) and used two case study systems. One of the case study systems was adapted from an actual system developed by Cisco Systems, Norway.

4.5 Robustness Testing

In this section, research activities in terms of a methodology (Section 4.5.1), tool support (Section 4.5.2), and empirical evaluation (Section 4.5.3) for robustness testing are presented.

4.5.1 Methodology

Test data generation is an important component of any MBT automation. For UML models, with constraints in OCL, test data generation is a non-trivial problem. A few approaches in the literature address this issue. But most of them targets only a small subset of OCL [73, 77], are not scalable, or lack proper tool support [78]. This is a major limitation when it comes to the industrial application of MBT approaches that use OCL to specify constraints on models. To overcome these limitations, this thesis reports on a set of novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), and Alternating Variable Method (AVM), to solve OCL constraints in order to generate test data. In this context, OCL constraints on the environment are solved to emulate faulty situations and perform robustness testing.

4.5.2 Tool Support

To automated test data generation, a tool was developed in Java, whose details are provided in Section 3.2. In addition, the TRUST tool is extended for robustness testing since the existing version of the tool only supported for functional testing.

4.5.3 Empirical Evaluation

Empirical evaluation of OCL solver: In our case study, test data generation using constraint solving for MBRT of the VCS was targeted. Testing is performed at the system level and specifically focused on robustness faults, for example related to faulty situations in the network and other systems that comprise the environment of the SUT. Test cases are generated from the system state machines using TRUST [76] as discussed in Section 4.3. To execute test cases, appropriate data were needed for state variables of the system, state variables of the environment (network properties and in certain cases state variables of other VCS), and input parameters that may be used in the following UML state machine elements: (1) guard conditions on transitions, (2) change events as triggers on transitions, and (3) inputs to time events.

Empirical evaluation of TRUST for Robustness: TRUST for robustness testing was applied to the industrial case study of a Cisco VCS. Robustness test cases were generated using the extended TRUST (Section 4.5.2). OCL constraints on the environment were solved using our search-based constraint solver (Section 4.5.2) to introduce faulty situations in the environment. The test cases were executed using the setup presented in Section 3.2.

5 Summary of Results

In this section, a summary and the key results of the papers submitted as part of this thesis are presented.

5.1 Paper 1

"A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation", S. Ali, L. Briand, H. Hemmati, and R. K. Panesar-Walawege, in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010.

In this paper, we answered the following research questions via a systematic review.

RQ1: What is the research space of search-based software testing?

In the context of search-based software testing, most of empirical studies have used Genetic Algorithms, Simulating Annealing, and their various extensions to automate test case generation at the unit testing level. A few of the papers target specific application domains and the most frequently observed domains were real-time and embedded systems. Most of the papers defined test cases as test data (or test input) and did not target any specific faults. The papers that did not target any specific faults, aimed to achieve a specific coverage of a test model, thus indirectly targeting certain types of faults (models are used as automated oracles). The most frequently observed test model was the control flow graph, where a variety of control flow-based criteria were applied to generate test cases.

RQ2: How are the empirical studies in search-based software testing designed and reported?

In our context, defining good and relevant cost and effectiveness measures is a prerequisite for a useful empirical study. Almost all of the papers use appropriate (though not perfect) cost and effectiveness measures to perform empirical studies. However, there were two major problems in the majority of the papers. First, most of the papers do not account for the random variation in cost and effectiveness of SBST techniques. Even the majority of the papers that did account for the random variation did not use proper data analysis and reporting methods (descriptive statistics and statistical hypothesis testing). Thus, there is a general lack of rigor in the statistical analysis and reporting of results in most empirical studies assessing the use of search algorithms for test case generation. Second, most of the papers did not demonstrate the benefits of SBST by comparing it with simpler, techniques such as random search or hill climbing. These two factors are highly important for yielding interpretable empirical studies in the context of test case generation using SBST techniques. Furthermore, many other relevant aspects of empirical studies such as reporting of validity threats, definition of formal hypotheses, object selection strategy, and data collection methods are not reported by most of the papers. We concluded that most empirical studies in the context of test case generation using SBST techniques are still not properly conducted and reported. Improving this situation should be an important objective of the research community in future studies.

RQ3: How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

We found that the number of papers which contain well-designed and reported empirical studies in the domain of test case generation using SBST is very small. As a result, there is a limited body of credible evidence that demonstrates the usefulness of SBST techniques for test case generation. This evidence is, in addition, very partial as it mostly focuses on the use of Genetic Algorithms at the unit testing level. This evidence, however, consistently shows that the Genetic Algorithms outperform random search in terms of structural coverage. More empirical studies must be conducted to provide strong and generalizable evidence about the suitability and applicability of different search algorithms for test case generation at different testing levels and for test objectives other than structural coverage.

5.2 Paper 2

S. Ali, L. Briand, and H. Hemmati. Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems, Accepted for publication in the Journal of Software and Systems Modeling, Springer, 2011.

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

We have used AspectSM to model the crosscutting robustness behavior of an industrial videoconferencing system developed by Cisco Systems, Norway. The results of the application of the case study are summarized as follows:

Reduced modeling effort: Overall, results on the industrial case study suggested that the modeling effort can be significantly reduced when using aspect state machines for modeling crosscutting behavior using AspectSM. While modeling various crosscutting behaviors on a complete industrial case study, by analyzing the number of modeling elements involved in various models, we estimated the saved modeling effort to be around 98% on average.

Enhanced separation of concerns: Modeling crosscutting behavior in UML state machines provides enhanced separation of concerns. This means that a modeler, or several of them with possibly different expertise, can focus on each crosscutting concern separately and therefore model them separately from the core functionality and other crosscutting concerns. This is very important for our industrial partner since they have separate groups for different kinds of testing activities including functional testing, video testing, audio testing, and network testing. Using our methodology each group can model aspects which are related to their expertise and our tool can then be used to automatically weave these aspects with the behavioral base models (models developed by the functional testing group).

Systematic fault modeling: Using RUMM, we systematically identified possible classes of faults for a specific SUT based on the proposed fault taxonomy. Furthermore, we instantiated specific fault types from the identified classes, which were considered critical in the SUT environment.

5.3 Paper 3

S. Ali, T. Yue, and L. Briand Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines?, Submitted to Systems and Software Modeling Journal (SOSYM), 2011

In this paper, the "readability" of state machines when modeling crosscutting behavior using AOM, and more specifically AspectSM (Paper 2), was evaluated. Readability is indirectly measured through defect identification and fixing rates in state machines, and the scores obtained when answering a comprehension questionnaire about the system behavior. With AspectSM, crosscutting behavior is modeled using so-called "aspect state machines". Their readability is compared with that of system state machines directly modeling crosscutting and standard behavior together. An initial controlled experiment and a much larger replication were conducted with trained graduate students, in two different institutions and countries, to achieve the above objective. Two baselines of comparisons—standard UML state machines without hierarchical features (flat state machines) and standard state machines with hierarchical/concurrent features (hierarchical state machines)—were used.

Results showed that the defect identification and defect fixing rates of aspect state machines are significantly higher than the ones for the hierarchical and flat state machines. For instance, for the industrial case study in the replication, aspect state machines showed, on average, increases of 28% and 19% in defect identification rates when compared to hierarchical and flat state machines, respectively. This is most likely due to the fact that aspect state machines are less complex than hierarchical and flat state machines in terms of modeling elements such as states and transitions. But on the other hand, aspect state machines can be potentially difficult to comprehend in terms of mentally processing how an aspect is woven into its base state machine. This may explain why, based on subjects' responses to a comprehension questionnaire, results showed that the subjects that were given hierarchical state machines outperformed the ones that were assigned aspect state machines, though that difference was not statistically significant with the employed statistical tests at the chosen significance level (0.05). No significant difference in effort was observed between any types of state machines in both defect identification and comprehension. Based on the results, our practical recommendation is to model crosscutting behaviors using aspect state machines in combination with hierarchical/concurrent features of UML state machines, where applicable, in order to improve the overall readability of crosscutting behaviors.

5.4 Paper 4

S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand. Solving OCL Constraints for Test Data Generation in Industrial Systems with Search Techniques, Submitted to TOSEM, 2011.

This paper is a journal extension of the following paper:
S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand. A Search-based OCL Constraint Solver for Model-based Test Data Generation, In: Proceedings of the 11th International Conference On Quality Software (QSIC 2011), pp. 41-50, IEEE, 2011.
Test data generation is an important component of MBT automation. For UML models, with constraints in OCL, test data generation is a non-trivial problem. A few approaches in the literature exist that address this issue. But most of them, either target only a small subset of OCL [73, 77], are not scalable, or lack proper tool support [78]. This is a major limitation when it comes to the industrial application of MBT approaches that use OCL to specify constraints on models.

This paper presented a contribution by devising novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs) and (1+1) Evolutionary Algorithm (EA), to solving OCL constraints (covering the entire OCL 2.2 semantics [10]) in order to generate test data. A search-based OCL constraint solver is implemented and evaluated on the first reported, industrial case study on this topic. The following research questions were addressed in this paper:

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in the models of our industrial case study?

The results showed that (1+1) EA outperformed both RS and GA, whereas GA outperformed RS. We can observe that, with an upper limit of 2000 iterations, (1+1) EA achieved a median success rate of 80% but GA did not exceed a median of roughly 60%. All success rates for (1+1) EA were above 50% and most of them were close to 100%. Constraints with the lowest success rates were seven and eight clauses long. Even taking the lowest success rates for the most difficult constraints, which was 50%, this would entail that, with r runs of (1+1) EA, we would achieve a success rate of 1 - $(1 - 0.5)^r$. For example, with $r = 7$, we would obtain a success rate above 99%. This entails a computation time of approximately 3.8*7=27 minutes since it took on average 3.8 minutes for 2000 iterations. Given that we used a slow prototype (EyeOCL [19]) for OCL expression analysis and that we could easily parallelize the search on a cluster or a network of computers, our results suggested that our approach is effective, efficient, and therefore practical, even for difficult constraints.

RQ2: Among the considered search algorithms, which one performs best in solving OCL constraints?

The results showed that there is statistically strong evidence to claim that (1+1) EA is significantly more successful than the other analyzed algorithms, i.e., GA and RS.

In the journal version, there were the following differences:

1. Additional heuristics for various OCL operations were presented, which were absent from Paper 4.
2. The industrial case study was extended and 10 new constraints were included. In addition, more rigorous empirical evaluation was performed, where results of each individual constraint were discussed and evaluated.
3. Comparison with the most relevant existing work in literature was augmented. All industrial problems were run and compared with the corresponding tool.
4. A new empirical evaluation was presented, where individual heuristics were evaluated using additional artificial problems.
5. A new search algorithm, Alternating Variable Method (AVM), was added in the empirical evaluations.

The following research questions were answered:

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in the models developed for our industrial case study?

The results show that AVM not only outperformed all the other three algorithms, i.e., (1+1) EA, RS, and GA but in addition achieved a consistent success rate of 100%. (1+1) EA outperformed GA and RS and achieved an average success rate of 98%. Finally, GA outperformed RS, whereas GA achieved an average success rate of 65% and RS attained an average success rate of 49%. With an upper limit of 2000 iterations, (1+1) EA achieved a median success rate of 98% and GA exceeded a median of roughly 80%, whereas RS could not exceed a median of roughly 45%. All success rates for (1+1) EA were above 90% and most of them were close to 100%.

RQ2: Among the considered search algorithms (AVM, GA, (1+1) EA), which one fared best in solving OCL constraints and how do they compare to RS?

The results suggested that AVM has more chances of success than GA. Similar results were observed for (1+1) EA, which significantly outperformed GA. For AVM vs RS, for almost all of the problems and in particular the most complex ones, AVM performed significantly better than RS. Similar results were observed for (1+1) EA vs RS and GA vs RS, where both (1+1) EA and GA significantly outperformed RS.

RQ3: Does the optimized branch distance calculation improve the effectiveness of search?

In this research question, we empirically evaluated whether the fine grained fitness functions (optimized branch distance calculation we proposed) for various OCL operations really improve performance of search algorithms as compared to using simple branch distance functions, yielding *0* if an expression is true and *k* otherwise. The results showed that (1+1) EA and AVM with optimized branch distance calculations significantly improved their success rates. In worst cases, when there were no differences in success rates, (1+1) EA and AVM took significantly less iterations to solve the problems.

5.5 Paper 5

S. Ali, L. Briand, A. Arcuri, and S. Walawege. An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms, In: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), ACM/IEEE, 2011.

This paper reported our successful application of MBRT in an industrial case study. We reported on our overall experience of performing MBRT on video conferencing systems developed by Cisco Systems, Norway. We discussed how we developed and integrated various techniques and tools which were discussed in Paper 2 to Paper 5 to achieve a fully automated MBRT that is able to detect previously uncaught, critical software faults in those systems. We provided an overview of how we achieved scalable modeling of robustness behavior using aspect-oriented modeling (Paper 2 and Paper 3), test case generation using search algorithms (Paper 4 and Paper 5), and environment emulation for test case execution. Our experience and lessons learned further identified challenges and open research questions to better support the industrial application of MBRT. Our preliminary experiments revealed one critical robustness fault in a deployed version of a videoconference system.

6 Future Directions

In this section, we discuss possible future directions, based on the two main parts of thesis, i.e., robustness modeling and robustness testing.

As we discussed, our robustness modeling methodology (RUMM) was specifically developed for modeling robustness behavior to facilitate automated model-based testing. While developing the methodology, we took into consideration only those issues which are relevant for modeling the behavior of a system in the presence of faulty situations in the environment. We have not investigated whether other non-functional crosscutting concerns, such as security and dependability, can be successfully modeled using RUMM or an adapted version of it. The reason is that RUMM starts with modeling faults based on a fault taxonomy for the system environment, which may not be necessary, for instance, when modeling security concerns such as logging. In addition, since RUMM is developed for model-based testing, we only considered issues which are important to support automated testing. For instance, we focused on UML state machines, which are often used for automated testing in control and communication systems as they typically exhibit state-driven behavior. We also focused on modeling crosscutting behavior on those modeling elements of state machines that are mandatory to support test automation such as states (including state invariants, entry, exit, and do activities) and transitions (including guard, trigger, and effect). In AspectSM, we write pointcuts as OCL queries, and we have not yet empirically evaluated and compared their expressiveness when using other related languages and notations such as the one presented in [37]. We used OCL to write pointcuts as it is the only standard for writing constraints in UML models, an important advantage in industrial contexts. Last, our work for defining interactions and ordering between different aspect state machines still requires further investigation. In the future, we will investigate to which extent our profile is applicable for other types of crosscutting behaviors to be modeled as state machines. In addition, we need to investigate the effort required by developers and testers to learn and apply RUMM. A series of controlled experiments and case studies are required for this purpose, which we are planning to conduct in the future. Our work on modeling interactions and ordering between various aspects still needs further investigation and evaluation. Currently, we only evaluated the readability of AspectSM using controlled experiments. In the future, we are planning to replicate the experiment to study the readability of aspect state machines in the presence of interactions between aspects as well as compare the understandability, modeling effort, and quality of aspect state machines with flat and hierarchical state machines.

For robustness testing, we needed to model faulty situations in the network, media streams, and VCSs communicating with a VCS under test (VUT). To date, we experimented only with emulating faulty situations in the network, which is just one aspect of the environment. Although we have already modeled the faulty situations in media streams (e.g., echo in audio and miss-synchronization between audio and video) [7], we do not have an appropriate media stream emulator yet. In addition to the media stream emulator, we also need to update our test script generator to generate test scripts that will control the media streams emulator during test case execution. For emulating faulty situations in other VCSs communicating with the VUT, we have not yet modeled the VCSs from that perspective. But we do expect that the models of the VCSs should be quite similar to the models of VUT, except for the need to select test paths from the models that will trigger faulty situations. For this purpose, we do have software-based emulators for VCSs, which can be utilized to emulate faulty situations during test case execution.

Finally, we are planning to extend the TRUST tool with more sophisticated test strategies specifically tailored to discovering robustness faults in a VCS. We also plan to perform robustness testing in the presence of faulty situations in other aspects of the environment such as in media streams and VCSs.

7 Conclusion

This thesis reported a scalable, automated, and systematic Model-Based Robustness Testing (MBRT) approach, whose scalability and applicability is thoroughly evaluated in realistic settings. We discussed how we integrated different tools and techniques to achieve the ultimate goal of automated and systematic MBRT. First, we addressed how we achieved scalable modeling of robustness behavior using Aspect-oriented Modeling (AOM) and more specifically using the AspectSM profile. AspectSM is a UML profile specifically designed to model robustness behavior with minimum extensions to UML to ease practical adoption. We also provided details on the weaver for AspectSM. Second, we provided details on the use of search algorithms (e.g., Genetic Algorithms) to solve complex constraints on environmental properties, expressed with the standard Object Constraint Language (OCL), to emulate faulty situations. Third, we described the integration of the abovementioned tools with our model-based testing tool TRansformation-based tool for Uml-baSed Testing (TRUST) to achieve fully automated MBRT. Finally, we discussed the setup required to execute the test cases generated by TRUST whose execution revealed a critical robustness fault in an industrial, deployed video conferencing system (VCS) that had remained undetected by previous testing. Failures were triggered in the presence of duplicate packets in the network during a videoconference. We also reported various empirical studies that we performed to evaluate our proposed solutions, such as the evaluation of robustness modeling using two controlled experiments and one industrial case study, the evaluation of our search-based constraint solver using one industrial case study, and the evaluation of robustness testing using one industrial case study. Results showed that our modeling methodology can significantly reduce modeling effort (on average 98%), improve separation of concerns, improve readability, and ease model evolution. In addition, our OCL constraint solving heuristics turned out to be very effective and efficient, and significantly better than existing works in the literature. Furthermore, preliminary experiments with our robustness testing methodology revealed one critical robustness fault in a deployed VCS.

References for the Summary

[1] Artemis, http://artemis-ju.eu/embedded_systems, 2011

[2] "IEEE Standard Glossary of Software Engineering Terminology," IEEE, IEEE Std 610.12-19901990.

[3] "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability," *IEEE Std 982.1-2005 (Revision of IEEE Std 982.1-1988),* pp. 1-34, 2006.

[4] "Standard for Software Quality Characteristics," International Organization for Standardization, ISO-9126-32003.

[5] "Software Assurance Standard," NASA Technical Standard, NASA-STD-8739.82005.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing,* vol. 1, pp. 11-33, 2004.

[7] "IEEE Standard Classification for Software Anomalies," IEEE, IEEE Std 1044-20092009.

[8] Modeling and Analysis of Real-time and Embedded systems (MARTE), http://www.omgmarte.org/, 2010

[9] Kermeta - Breathe Life into Your Metamodels, http://www.kermeta.org/, 2010

[10] Object Constraint Language Specification, Version 2.2, http://www.omg.org/spec/OCL/2.2/, 2010

[11] T. Pender, *UML Bible*: Wiley, 2003.

[12] Meta Object Facility (MOF), http://www.omg.org/spec/MOF/2.0/, 2006

[13] F. Lagarde, H. Espinoza, F. Terrier, C. André, and S. Gérard, "Leveraging Patterns on Domain Models to Improve UML Profile Definition," in *Fundamental Approaches to Software Engineering*, 2008.

[14] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*: Tim Weilkiens, 2008.

[15] UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms, http://www.omg.org/spec/QFTP/1.1/, 2010

[16] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*: Springer, 2007.

[17] IBM OCL Parser, http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html, 2010

[18] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Cârcu, "OCLE," V2.0 ed, 2010.

[19] M. Egea, "EyeOCL Software," 2010.

[20] CertifyIt, http://www.smartesting.com/index.php/cms/en/product/certify-it, 2011

[21] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems," in *IADIS International Conference in Applied Computing*, 2005.

[22] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," in *ECOOP-Workshop on Defining Precise Semantics for UML*, 2000.

[23] M. Clavel and M. A. G. d. Dios, "Checking unsatisfiability for OCL constraints," in *In the proceedings of the 9th OCL 2009 Workshop at the UML/MoDELS Conferences*, 2009.

[24] M. Kyas, H. Fecher, F. S. d. Boer, J. Jacob, J. Hooman, M. v. d. Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electron. Notes Theor. Comput. Sci.,* vol. 115, pp. 39-47, 2005.

[25] J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. ster, "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars," *Electron. Notes Theor. Comput. Sci.,* vol. 211, pp. 159-170, 2008.

[26] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the alloy constraint analyzer," in *Proceedings of the 22nd international conference on Software engineering* Limerick, Ireland: ACM, 2000.

[27] R. Yedduladoddi, *Aspect Oriented Software Development: An Approach to Composing UML Design Models*: VDM Verlag Dr. Müller, 2009.

[28] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented Approach to Early Design Modelling," *IEEE Software,* vol. 151, 2004.

[29] R.Chitchyan, A.Rashid, P. Sawyer, J. Bakker, M. P. Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, and A. Jackson, "Survey of Aspect-Oriented Analysis and Design," Aspect-Oriented Software Engineering Special Interest Group, Lancaster University, AOSD-Europe-ULANC-9, 2005.

[30] G. Zhang, "Towards Aspect-Oriented State Machines," in *2nd Asian Workshop on Aspect-Oriented Software Development (AOASIA'06)* Tokyo, 2006.

[31] T. Cottenier, A. v. d. Berg, and T. Elrad, "The Motorola WEAVR: Model Weaving in a Large Industrial Context," in *Aspect Oriented Software Development (AOSD)*, 2007.

[32] J. Evermann, "A meta-level specification and profile for AspectJ in UML," in *Proceedings of the 10th international workshop on Aspect-oriented modeling* Vancouver, Canada: ACM, 2007.

[33] G. Zhang, M. M. Hölzl, and A. Knapp, "Enhancing UML State Machines with Aspects," in *In Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2007.

[34] J. Zhang, T. Cottenier, A. V. D. Berg, and J. Gray, "Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver," *Journal of Object Technology,* vol. 6, 2007.

[35] Z. Jingjun, "Modeling Aspect-Oriented Programming with UML Profile," 2009, pp. 242-245.

[36] J. U. Júnior, V. V. Camargo, and C. V. F. Chavez, "UML-AOF: a profile for modeling aspect-oriented frameworks," in *Proceedings of the 13th workshop on Aspect-oriented modeling* Charlottesville, Virginia, USA: ACM, 2009.

[37] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi, "An Expressive Aspect Composition Language for UML State Diagrams," in *Model Driven Engineering Languages and Systems*, 2007.

[38] J. Whittle and P. Jayaraman, "MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation," in *Models in Software Engineering*, G. Holger, Ed.: Springer-Verlag, 2008, pp. 16-27.

[39] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*: Morgan-Kaufmann, 2007.

[40] D. Xu and W. Xu, "State-based incremental testing of aspect-oriented programs," in *Proceedings of the 5th international conference on Aspect-oriented software development* Bonn, Germany: ACM, 2006.

[41] D. Xu, W. Xu, and K. Nygard, "A State-Based Approach to Testing Aspect-Oriented Programs," in *17th International Conference on Software Engineering and Knowledge Engineering* Taiwan, 2005.

[42] W. Xu and D. Xu, "A Model-Based Approach to Test Generation for Aspect-Oriented Programs," in *First Workshop on Testing Aspect-Oriented Programs*, 2005.

[43] D. Xu, W. Xu, and W. E. Wong, "Testing Aspect-Oriented Programs with UML Design Models," *International Journal of Software Engineering and Knowledge Engineering*.

[44] QTRONIC, http://www.conformiq.com/qtronic.php, 2010

[45] T. Weigert and R. Reed, "Specifying Telecommunications Systems with UML," in *UML for Real: Design of Embedded Real-time Systems*: Kluwer Academic Publishers, 2003, pp. 301-322.

[46] J. Perala, "Improving TTCN-3 Test System Robustness Using Software Fault Tolerance," in *Advances in System Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on*, 2009, pp. 48-56.

[47] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," 2011.

[48] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[49] M. Samar, C. B. Lionel, L. Yvan, and P. Massimiliano Di, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," *IEEE Transactions on Software Engineering,* vol. 99, 2010.

[50] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*: IEEE Computer Society, 2008.

[51] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," in *Proceedings of the Fifth International Conference on Quality Software*: IEEE Computer Society, 2005.

[52] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research Articles," *Softw. Test. Verif. Reliab.,* vol. 16, pp. 175-203, 2006.

[53] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths," *Advanced Design and Manufacture to Gain a Competitive Edge,* pp. 147-156, 2008.

[54] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001.

[55] L. C. Briand, M. D. Penta, and Y. Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Transactions on Software Engineering,* vol. 30, pp. 770-793, 2004.

[56] M. Harman, S. A.Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," King's College,Technical Report TR-09-032009.

[57] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering,* vol. 99, 2009.

[58] A. Andrea, "Longer is Better: On the Role of Test Sequence Length in Software Testing," International Conference on Software Testing, Verification, and Validation, 2010, pp. 469-478.

[59] C. Wohlin, P. Runeson, and M. Höst, *Experimentation in Software Engineering: An Introduction*: Springer, 1999.

[60] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - A systematic literature review," *Information and Software Technology,* vol. 51, pp. 7-15, 2009.

[61] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development," in *Proceedings of the 5th International Conference on The Unified Modeling Language*: Springer-Verlag, 2002.

[62] H. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering,* vol. 14, pp. 131-164, 2009.

[63] A. Aldini, R. Gorrieri, F. Martinelli, and J. Jürjens, "Model-Based Security Engineering with UML," in *Foundations of Security Analysis and Design III.* vol. 3655: Springer Berlin / Heidelberg, 2005, pp. 42-77.

[64] J. Péreza, N. Ali, J. A. Carsı´b, I. Ramosb, B. Álvarezc, P. Sanchezc, and J. A. Pastorc, "Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems," *Information and Software Technology,* vol. 50, pp. 969-990, 2008.

[65] T. Cottenier, A. v. d. Berg, and T. Elrad, "Stateful Aspects: The Case for Aspect-Oriented Modeling," in *Proceedings of the 10th international workshop on Aspect-oriented modeling* Vancouver, Canada: ACM, 2007.

[66] M. Shafique and Y. Labiche, " A Systematic Review of Model Based Testing Tools," Carleton University, Department of Systems and Computer Engineering, Technical Report (SCE-10-04)2010.

[67] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.,* vol. 1, pp. 11-33, 2004.

[68] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*: Addison-Wesley Professional, 2008.

[69] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.

[70] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," in *Proceedings of the 11th International Conference On Quality Software (QSIC 2011)*, 2011.

[71] "netem," 2011.

[72] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*: IEEE Computer Society, 2008.

[73] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," Citeseer, 2002.

[74] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the Cost of Model-Based Testing through Test Case Diversity," in *22nd IFIP International Conference on Testing Software and Systems (ICTSS)*, 2010.

[75] H. Hemmati, L. Briand, A. Arcuri, and S. Ali, "An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study," in *18th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*: ACM, 2010.

[76] S. Ali, L. Briand, A. Arcuri, and S. Walawege, "An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms," in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011)*, 2011.

[77] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," in *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*, 2003.

[78] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," in *International Conference on cimputational Intelligence and Security*, 2007, pp. 1048-1052.

A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation

Shaukat Ali, Lionel Briand, Hadi Hemmati, and Rajwinder K. Panesar-Walawege

Published in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010

Abstract— Metaheuristic search techniques have been extensively used to automate the process of generating test cases and thus providing solutions for a more cost-effective testing process. This approach to test automation, often coined as "Search-based Software Testing" (SBST), has been used for a wide variety of test case generation purposes. Since SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective they are at reaching their test objectives and whether they scale up to realistic development artifacts. However, approaches to empirically study SBST techniques have shown wide variation in the literature. This paper presents the results of a systematic, comprehensive review that aims at characterizing how empirical studies have been designed to investigate SBST cost-effectiveness and what empirical evidence is available in the literature regarding SBST cost-effectiveness and scalability. We also provide a framework that drives the data collection process of this systematic review and can be the starting point of guidelines on how SBST techniques can be empirically assessed. The intent is to aid future researchers doing empirical studies in SBST by providing an unbiased view of the body of empirical evidence and by guiding them in performing well designed and executed empirical studies references.

1 Introduction

Software is being incorporated into an ever increasing number of systems and hence it is becoming increasingly important to thoroughly test these systems. One challenge to testing software systems is the effort involved in creating test cases that will systematically test the system and reveal faults in an effective manner. The overall testing cost has been estimated at being almost fifty percent of the entire development cost [6], if not more. Thus, a logical response is to automate the testing process as much as possible, and test case generation is naturally a key part of this automation. A possible strategy which has drawn great interest in the automation of test case generation is the application and tailoring of metaheuristic search (MHS) algorithms [41]. The main reason for such an interest is that test case generation problems can often be re-expressed as optimization or search problems.

There has been a tremendous amount of research in applying MHS algorithms to test case generation and a large body of research exists: a search of the most relevant databases (as detailed in Section 4.2.1) found 450 articles which after reading abstracts resulted in 122 relevant articles published over the years 1996-2007 on this specific topic, often referred to as search-based software testing (SBST) [4].

Seeing the amount of research activity in this field, it is at this point in time, highly important to characterize what type of research has been performed and how it has been conducted. Among other things, it is crucial to appraise how much empirical evidence there is regarding the cost-effectiveness of SBST and to determine whether there is room for improvement in the way studies are performed and reported. The ultimate goal is to improve the quality of future research in this important, emerging field of research. In order to assess the current state of the art in SBST, we decided to conduct a comprehensive systematic review of the current literature, as this is commonly done in other scientific fields of research such as medicine [25] and social science [29]. The purpose of this systematic review is to collect, classify, and assess the empirical studies on SBST in order to assess the current body of evidence regarding the cost and effectiveness of SBST. By identifying the strengths and weaknesses of the current literature we hope to suggest improved research practices and relevant future research directions.

This paper is organized as follows: In Section 2, we provide the background relevant to the material presented in this paper. Section 3 suggests a framework used to assess the empirical studies in SBST and Section 4 presents the method used to conduct this systematic review. In Section 5, we present the results of our review whilst Section 6 outlines its validity threats. The final conclusions that we can draw from this systematic review are presented in Section 7.

2 Background

Detailed In this systematic review, we are analyzing which MHS algorithms have been used to address test case generation and what body of evidence exists regarding their cost-effectiveness. As a preliminary to the review itself, we introduce here the three main components involved in this paper: search-based software testing, systematic reviews, and empirical studies.

2.1 Search-based software testing

The main aim of software testing is to detect as many faults as possible, especially the most critical ones, in the system under test (SUT). To gain sufficient confidence that most faults are detected, testing should ideally be exhaustive. Since in practice this is not possible, testers resort to test models and coverage/adequacy criteria to define systematic and effective test strategies that are fault revealing. A test case normally consists of test data and the expected output [36]. The test data can take various forms such as values for input parameters of a function, values of input parameters for a sequence of method calls, or seeding times to trigger task executions. In the context of this review, we are not dealing with the expected outputs, but focus exclusively on the generation of test data as this has been the objective of papers making use of SBST. In order to perform test case generation, systematically and efficiently, automated test case generation strategies are employed. Bertolino [7] addresses the need for 100% automatic testing as a means to improve the quality of complex software systems that are becoming the norm of modern society. A comprehensive testing strategy must address many activities that should ideally be automated: the generation of test requirements, test case generation, test oracle generation, test case selection, or test case prioritization. In our current review, we are only dealing with test case generation. A promising strategy for tackling this challenge comes from the field of search-based software engineering [23].

Search-based software engineering attempts to solve a variety of software engineering problems by reformulating them as search problems [15]. A major research area in this domain is the application of MHS algorithms to test case generation. MHS algorithms are a set of generic algorithms that are used to find optimal or near optimal solutions to problems that have large complex search spaces [15]. There is a natural match between MHS algorithms and software test case generation. The process of generating test cases can be seen as a search or optimization process: there are possibly hundreds of thousands of test cases that could be generated for a particular SUT and from this pool we need to select, systematically and at a reasonable cost, those that comply to certain coverage criteria and are expected to be fault revealing, at least for certain types of faults. Hence, we can reformulate the generation of test cases as a search that aims at finding the required or optimal set of test cases from the space of all possible test cases. When software testing problems are reformulated into search problems, the resulting search spaces are usually very complex, especially for realistic or real-world SUTs. For example, in the case of white-box testing, this is due to the non-linear nature of software resulting from control structures such as if-statements and loops [17]. In such cases, simple search strategies may not be sufficient and global MHS algorithms1 may, as a result, become a necessity as they implement global search and are less likely to be trapped into local optima [16]. The use of MHS algorithms for test case generation is referred to as search-based software testing [4]. Mantere and Alander [35] discuss the use of MHS algorithms for software testing in general and McMinn [37] provides a survey of some of the MHS algorithms that have been used for test data generation. The most common MHS algorithms that have been employed for search-based software testing are evolutionary algorithms, simulated annealing, hill climbing, ant colony optimization, and particle swarm optimization [12]. Among these algorithms, hill climbing (HC) [12] is a simpler, local search algorithm. The SBST techniques using more complex, global MHS algorithms are often compared with test case generation based on HC and random search to determine whether their complexity is warranted to address a specific test case generation problem. The use of the more complex MHS algorithm may only be justified if it performs significantly better than HC.

2.2 Systematic reviews

Systematic reviews are a means of synthesizing existing research regarding a specific research question [29]. They are usually performed to summarize the existing evidence for a particular topic and aid in the identification of gaps in the current research and thus can form the basis of new research activity. A review protocol is created at the beginning of the review, which lays out the research questions being answered and the methodology that will be used to answer these questions. The protocol specifies a specific search strategy that is used to select as much of the relevant literature as possible and provides justification for why studies are included or excluded from the systematic review. The data to be collected to answer the research questions is also presented in the protocol. All this information is published so that readers can judge the completeness of the systematic review, and if necessary replicate it. These features distinguish the systematic review from the usual literature review or survey that is usually conducted at the beginning of a

[1] Global MHS algorithms are often contrasted with local MHS algorithms. The former are based on strategies for the search to avoid being stuck in local minima, thus being more effective in situations with complex search landscapes [12].

research activity. A systematic review synthesizes the existing work in a systematic, comprehensive, and unbiased manner.

2.3 Empirical studies for search-based software testing

Kitchenham et al. [19, 31] make the case for evidence-based software engineering that seeks to help practitioners make informed decisions related to software development and maintenance by integrating current best evidence from research with practical experience. Thus, to determine if SBST techniques can be applied in practice, we need to conduct empirical studies to assess their cost-effectiveness and scalability. The cost-effectiveness of a SBST technique is normally measured in terms of the ability of the technique to generate test cases that achieve a certain testing objective at a reasonable cost. The testing objective, as is the case with any test case generation technique, is to detect faults of a type that is explicitly defined or implicitly determined by the test model (e.g., state transition faults for a state machine model). In this review, we have focused on empirical studies of SBST techniques in order to assess whether convincing evidence exists to show their cost-effectiveness and scalability. For this purpose, it was necessary to define what we mean by an empirical study in this context and what constitutes a well designed and reported empirical study. Empirical studies are usually divided into three different types: surveys, case studies or experiments [52]. For this review, we have used a broad definition of empirical study, to include any kind of empirical evaluation that has been done in the field of SBST in order to be comprehensive in our investigation.

In order to determine what constitutes a proper empirical study in SBST, we looked at existing guidelines [27, 32, 52] for conducting empirical studies in software engineering, and those for evaluating SBST techniques in other fields. Wohlin et al. [52] and Kitchenham [32] present guidelines on how to conduct experimentation and empirical research in the specific context of software engineering whereas Johnson [27] presents a general guide for experimental analysis of algorithms. We have tailored and augmented some of these guidelines to create a specific framework for conducting and reporting empirical studies in the domain of SBST. This was necessary as SBST studies involve the analysis of automation techniques in which no human subjects are involved and presents many specific challenges. In addition, the fact that SBST techniques are based on MHS algorithms makes it important to account for the inherent random variation that exists in their results. Furthermore, there should also be some means to show that a SBST technique is really necessary for the context that it is being applied in. This can be done, for example, by showing that other simpler search techniques do not perform as well. The reason for doing this is that we want to ensure that the problems being tackled by the SBST techniques do warrant their use.

The framework was created for a dual purpose. First, it was used in this systematic review to direct the collection of data that was used to assess the current state of empirical research in SBST. Second, it can also be used as a set of guidelines for conducting and reporting future research in the field or at least as a starting point in the development of such guidelines. The next section will present the framework.

3 Framework

As presented here, this framework is not intended to provide complete operational guidelines, but rather to justify the data collection that took place to perform the systematic review presented in the next sections and to highlight some of the most important concepts and issues.

The framework is divided into four parts. First, the test problem addressed must be clearly specified. Second, the MHS algorithms adopted must be clearly defined. Third, since any SBST research should always include empirical studies aiming at assessing the cost and effectiveness of the proposed approaches, the design of such studies must be carefully described so that its validity can be assessed. Last, results must be carefully reported so as to be clearly interpretable and reproducible. Whenever relevant, we will refer to Johnson's general guidelines on the experimental analysis of algorithms [27], either to point the reader to further, more general considerations, or to show that our more specific guidelines are a specialization of these more general ones.

3.1 Test problem specification

The test problem specification includes two main parts, the purpose of testing and the test strategy that will be employed. Each of these parts directly affects the form that the search-based software testing strategy will take. Figure 1 outlines the constituent parts of a test problem specification. The general purpose of software testing is to gain sufficient confidence in the dependability of a software artifact. Explicitly, this is usually done by targeting specific types of faults at different levels (such as unit, integration, and system testing). The targeted faults can be categorized in many ways depending on the view one takes of a system. At the highest level, one differentiates functional from non-functional faults, e.g., faults related to performance, security, robustness, and safety requirements.

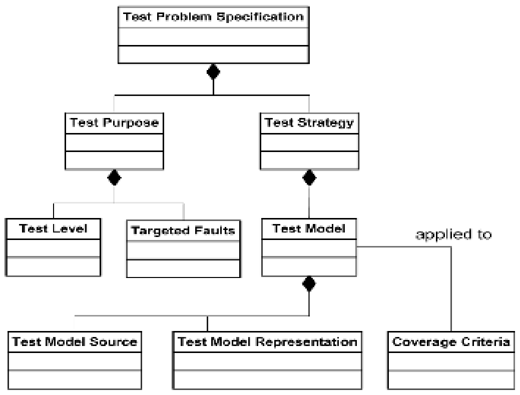


Figure 1 Concept diagram of test problem specification

A testing strategy is defined by a model of the SUT and some specific coverage criteria defined on that model. Such a model is typically referred to as a test model and the coverage criteria aim at systematically exercising the SUT based on the test model. This test model may be characterized by its source and representation (i.e., notation and semantics). Coverage criteria definitions depend on the test model representation. The source of the model implies constraints on the application of the test strategy as it depends on the availability and reliability of precise information in a specific form. As discussed in [5], possible sources for a test model can be the SUT specification, design artifacts or the source code itself. Based on the model source (specification, design or source code), different types of test models can be constructed. Typical examples of models derived from source code include control and data flow graphs, whereas test models based on SUT design include state machines or Markov usage models. To be systematic, a test strategy generates test cases to cover certain features of the test model. For instance, in the case of state machines, typical coverage criteria include the coverage of all states or all transitions, the latter being a stronger requirement, while, in the case of control flow graphs, a typical coverage criterion is branch coverage. It is important to clearly specify the coverage criteria as it is often used to measure the effectiveness of SBST techniques regarding test case generation.

3.2 Metaheuristic search algorithm specification

MHS algorithms are general strategies that need to be adapted to the problem at hand. When reporting a study, this implies describing and justifying the customizations and parameter settings for each specific algorithm. This will be required for replicating the study and also for comparisons with other SBST techniques and future studies. Each type of MHS algorithm has specific parameter settings to be reported, but the general idea is to report all settings and adjustments that may have an effect on the performance of the algorithm or are needed for replicating the study. In Figure 2 we show how a typical genetic algorithm can be used for test case generation. The important parameters to report for a genetic algorithm would be the encoding of the chromosomes, the fitness function created to guide the search, the strategy for creating the initial population, the selection strategy for selecting parents for the next generation, the various recombination operators such as crossover and mutation operators and their values, the reinsertion strategy and the stopping criteria. We discuss in [5] how these parameters affect the results of empirical studies involving the use of genetic algorithms for test case generation.

3.3 Empirical study design

This section will define the most important items that should be reported about the study definition (through its objectives and hypotheses), design, and results.

3.3.1 Objectives and experimental hypotheses

One must define what is going to be empirically assessed and compared. The objective is usually to compare various SBST techniques and alternatives in terms of code coverage, fault detection, test suite size, or test case generation time. The empirical study can be an assessment of a single SBST technique, a comparison of two or more SBST techniques, or a comparison of SBST techniques versus non-SBST techniques (i.e., not relying on meta-heuristic search algorithms). The latter includes, for example, random search, static analysis, greedy algorithms or some other specific technique for the test problem under consideration, e.g., schedulability analysis in the case of real-time systems. In any case, what is going to be compared should be precisely specified through formal test hypotheses, thus leading to appropriate statistical significance testing. One notion important here is to

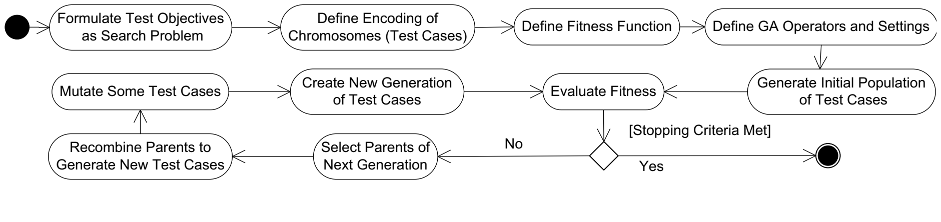

Formulate Test Objectives as Search Problem
Define Encoding of Chromosomes (Test Cases)
Define Fitness Function
Define GA Operators and Settings
Generate Initial Population of Test Cases
Evaluate Fitness
Create New Generation of Test Cases
Mutate Some Test Cases
Recombine Parents to Generate New Test Cases
Select Parents of Next Generation
No
[Stopping Criteria Met]
Yes

Figure 2 Test case generation using genetic algorithms

state the kind of hypothesis that will be used: either a one-tailed hypothesis or a two-tailed hypothesis [14]. This has an impact on how we interpret the results in terms of p-values (probability of type I errors). In the context of SBST, a one-tailed hypothesis would be used in the case when, based on the properties of the fitness function, we have a theoretical basis to assert the direction of the expected outcome. For example, when comparing a guided search algorithm such as genetic algorithm with random search, we may, based on an analysis of the fitness function, expect the genetic algorithm to be equally or more effective at hitting the search target – but not worse – and as such we would use a one-tailed hypothesis. However, as an example, when comparing two genetic algorithms with different fitness functions, where we cannot state upfront which one would fare better in terms of cost or effectiveness, we would use a two-tailed hypothesis. In other words, when the theory regarding the search algorithms under study allows us to be a priori confident regarding the possible direction of differences in cost or effectiveness, then we should use a one-tailed test as this will increase our chances to uncover a statistically significant difference.

3.3.2 Target application domain

Empirical studies should specify a target application domain in which their results are intended to be generalized. Example application domains are: real-time, concurrent, distributed, embedded, and safety-critical. Testing techniques typically target specific faults that are more relevant in certain application domains, e.g., slow response time in real-time systems. Moreover, assumptions are typically made regarding the availability of information required to build the test model. Such assumptions tend to be more or less realistic depending on the application domain. For example, if one assumes the use of the MARTE UML profile [3] to design a system and then derive a test model, this is of course more realistic in the context of embedded, real-time applications. Further, the selection of subject systems for empirical studies will then be partly determined by the target application domain.

3.3.3 Subject systems (Software Under Test or SUT) specification

After identifying the target application domain, specific SUTs fitting that domain are selected. It is important to carefully select SUTs and precisely justify why the selected SUTs are adequate matches for the target application domain as this will help the reader determine the extent to which the experimental results will generalize to this domain. This discussion should be in terms of the inherent properties of the SUT such as its size, complexity, or structure. This is particularly important when one is creating artificial SUTs specifically for the experiment, a common situation when one is trying to account for SUTs of varying size and complexity. For each SUT in the empirical study, the function of the SUT together with relevant properties affecting its representativeness of the domain should be carefully reported in order to ensure the reproducibility of the experiment and help future comparisons of cost-effectiveness results. Johnson [27] discusses the general problem of instance selection (i.e., SUTs here) in experiments (Principle 3: Use instance testbeds that can support general conclusions) and defines reproducibility (Principle 6: Ensure Reproducibility) when experimenting with algorithms as the capacity to "perform similar experiments that would lead to the same basic conclusions". The goal is to make it possible to confirm the results of an original experiment independently from the precise settings and details of the experiment. In addition to SUT properties, the hardware platform that the SUT executes on is also important to specify. Johnson [27] provides an in-depth discussion of the latter issue (Principle 7: Ensuring Comparability), which is not specific to SBST, and suggestions to address it. In its Principle 9, about well-justified conclusions, Johnson [27] also discusses the danger of drawing conclusions from small instances that are then generalized to much larger instances, as the former do not always predict well the latter, and recommends to use instances that are as large as possible.

3.3.4 Measures of cost and effectiveness for SBST techniques

Measuring effectiveness and more particularly cost in our context is inherently difficult and the validity of measures is very often context-dependent. As discussed by Johnson in [27] (Principle 6: Ensure Reproducibility), just reporting effectiveness and cost values is not very informative as it does not provide direct insights into what these values actually imply. It is nevertheless crucial, in order to draw useful conclusions from studies involving SBST techniques, to be able to use appropriate comparison baselines. In our context, one usually resorts to comparing the investigated technique to simpler, existing techniques (see Section 5 on baselines of comparisons) in order to assess the relative goodness of a search. The measures should be relevant for the particular study and comparable across the different techniques being investigated. Studies may use slight variations of an existing measure or introduce new ones, hence, it is important to explain the reasoning behind the effectiveness and cost measures and justify why they are applicable in the context they are being used. Along with the measure, the method used to collect the data related to the measures should be thoroughly explained. In the context of SBST, the effectiveness of a test case generation technique is closely related to the “quality” of the test suite generated by the technique. A good test suite can be characterized by its ability to uncover faults or to give confidence in the SUT by fulfilling a certain coverage criterion. Thus we can say that, in practice, there are two main categories of measures of effectiveness, which can be referred to as coverage-based measures and fault-based measures. In the former category, there may be many different types of measures depending on the adequacy criteria being used, for example, control-flow coverage criteria such as branch or path coverage may be used. The fault-based measures are typically fault detection scores. They can be computed based on real, known faults or are estimated through mutation analysis [48]. In the latter case, the program is seeded with faults based on mutation operators and depending on the number of faults caught, a so-called mutation score is calculated. The techniques are assessed upon how successful they are at detecting the seeded faults.

Cost measures are generally related to the speed of the technique to converge towards the test objective (in some cases it is referred to as the search technique’s “efficiency”). Some common cost measures used in the SBST domain are: (a) the number of iterations, which shows how many times a SBST technique needed to iterate in order to find its best solution, e.g., the number of generations in genetic algorithms, or cycles in ant colony optimization algorithms, (b) the cumulative number of individuals in all iterations (usually each individual represents a test case in SBST), (c) the number of fitness evaluations an algorithm needs, to find the final solution, which depends on the number of newly generated individuals (usually each new population is made up of some individuals from the previous iteration and some newly generated ones), (d) the time spent by a MHS algorithm to find test cases meeting the targeted test objective, which is sometimes referred to as “test case generation time”. This time can be either measured using clock time or CPU cycles. Clock time is the time from the “wall” clock and not easily comparable across different hardware architectures. However it is a practical measure that can be used to assess if a technique can be used in practice. CPU cycles on the other hand is a measure that can be used across techniques for comparison on other hardware architectures as well, and (e) the size of the resulting test suite, which is a surrogate measure for the cost of the time it would take to execute the resulting test suite since a larger test suite would require more resources to execute.

Among the first three cost measures, the number of iterations is a very coarse grained measure and is not as precise as the number of individuals, which in turn is not as precise as the number of fitness evaluations. The number of fitness evaluations is more precise than the number of individuals because in each iteration there are some individuals that are kept from the previous population and there is no cost for generating them. Therefore, the number of evaluations can more precisely estimate the real cost of a SBST technique. All these three measures are surrogate measures for the time used to generate the final test suite but none is perfect, because different search techniques may require a different amount of time per iteration, per creation of an individual (test case), or per fitness evaluation. For instance, it would not be a good idea to compare simulated annealing (SA) and genetic algorithms (GA) based on the number of iterations because the amount of time required for each iteration in GA and SA is likely to differ significantly.

The cost of a technique is generally measured for one of two purposes: either to compare two techniques to assess which one will cost less for the same effectiveness or to assess whether a technique can be used in practice given expected time constraints. From the measures discussed above, "test case generation time", if it has been measured under similar conditions, is the only measure that can give users an intuitive idea of whether they can apply a particular technique to their situation within the time constraints that they have. When comparing the cost of different techniques, it is also necessary to make sure that any other required resources are kept equal amongst the techniques. The fact that two techniques require the same amount of time does not mean that they have the same cost if one technique consumes much more memory than the other. Therefore all relevant types of resources must be accounted for when comparing the cost of SBST techniques.

3.3.5 Measures for scalability assessment

Scalability assessment is the process of assessing how the cost-effectiveness of a SBST technique evolves as a function of the size of the test case generation problem to be addressed. This involves one or more measures of SUT size and the analysis of their relationships with the cost or effectiveness of the SBST techniques under investigation. Some examples of measures that can be scaled up include the size of the SUT in terms of lines of code or the size of search space in terms of number and range of input data parameters. The effect of this scaling is then observed on different cost and effectiveness measures to see if the SBST technique is still cost-effective as the SUT gets larger and more complex.

3.3.6 Baselines for comparison

A SBST technique can only be assessed if it is compared with a carefully selected, meaningful baseline since the optimal solution is normally not known. Since it is difficult to assess SBST techniques in absolute terms, it is therefore important to show, as a minimum, that the problem at hand could not be addressed by some simpler means. In other words, every study should have one or more baselines of comparison when assessing SBST techniques and the minimum to be expected is a comparison with random search. The SUT investigated may, for example, be small and simple, and the fact that a SBST technique performs well may not mean much. Random search can then serve as a basic verification that the search problem cannot be addressed by a simple random search and warrants the use of a SBST technique. It is also preferable to use other simple SBST techniques, such as HC, as a comparison baseline for other more expensive SBST techniques. This further demonstrates that the use of a SBST technique is justified given the test case generation problem at hand. In addition—but this is context dependent—other SBST techniques, previously published or considered plausible alternatives, can also be used as baselines of comparisons for the proposed SBST techniques.

As discussed in [27], once baseline techniques are selected, one must ensure that reasonably efficient implementations are used for all techniques in order for cost and effectiveness to be comparable. Documentation, source code, URLs for downloadable tools, or at the very least a careful description of the implementation, should be provided.

3.3.7 Parameter settings

Most SBST techniques require parameter settings which tend to have a significant impact on their performance. In many studies, alternative parameter settings are investigated and compared. It is therefore highly important, to make any study reproducible, to specify these parameters in a precise manner. It is also interesting to justify their values based on existing studies, when possible, as this provides insights into how cost and effectiveness could be affected if they were changed or if a different SUT with different properties was used. One particularly important parameter in our context is the stopping criterion of the search (Principle 6: Ensure Reproducibility). It can be based on whether the search objective has been reached (or one is sufficiently close), execution time or a surrogate measure (due to practical constraints), or any significant progress is observed over a period of time.

3.3.8 Accounting for random variation in SBST results

Since SBST techniques use MHS algorithms; their results can vary from one execution to another. So, it is important to ensure that we run the algorithms a sufficient number of times to capture the random variation of results and be able to perform statistical comparisons with other search techniques. It is difficult to precisely specify the number of runs required in general but, as a ballpark number, it should probably be above ten, so as to allow the use of basic statistical hypothesis testing and obtain a reasonable statistical power to detect large differences [52]. Based on the expected (minimum) difference between techniques (if this can be estimated) and the statistical tests used to compare cost and effectiveness across techniques, the minimum required number of runs can be estimated using power analysis [18].

When dealing with multiple runs, in our context, we are often interested in the best run, yielding the best test suite or test case according to some fitness function (e.g., bringing the execution time of a task as close as possible to its deadline). Another frequent case is when we are interested in the frequency with which a certain target was reached across runs (e.g., test input data satisfying certain constraints). In both cases, it is important to report the execution time and other cost measures of all runs and, when relevant, information about their fitness distribution. The basic principle is that it should be possible to estimate the total cost of achieving the best solution or, depending on what is relevant, the expected cost to achieve the search target. From a more general standpoint, Johnson (Principle 6: Ensure Reproducibility) [27] warns against reporting only effectiveness and cost data for the best run.

3.3.9 Data analysis

During the design of an empirical study, it is important to decide about the data analysis methods that will be applied to cost-effectiveness and scalability results.

Data analysis methods for comparing cost-effectiveness. Performance in the case of SBST usually relates to measuring the cost-effectiveness of the various search techniques. The cost and effectiveness of a SBST technique are used together for assessing its performance. For example, a technique that has higher coverage than another technique may not be considered to have better performance, because it uses significantly more fitness evaluations (higher cost) to achieve that effectiveness, thus making it impractical for larger SUTs. Any claims of better performance should be backed by empirical evidence demonstrating lower cost or higher effectiveness when compared to the baseline and alternative techniques. In the ideal case, a study that is concentrating on measuring cost, should keep the effectiveness measures constant. For example, the study may measure the number of fitness evaluations needed to achieve 100% branch coverage. If, however, the aim is to measure effectiveness, then this can be done by keeping the cost constant, for example, by measuring how much branch coverage is achieved in some constant amount of time or number of fitness evaluations. The reported performance results should include the results of the comparison baselines. At a high level, reported results should follow the structure below:

Reporting descriptive statistics. Both cost and effectiveness distributions should be reported (e.g., as a table with descriptive statistics) and analyzed. Looking at their standard deviation may indicate the level of uncertainty in terms of cost and effectiveness associated with a SBST technique. This in turn may help determine how many runs would in practice be necessary to guarantee that we obtain a satisfactory result, i.e., achieve the objective.

Results of hypothesis testing. The purpose of statistical testing is to determine whether differences across SBST techniques in terms of central tendencies for cost and effectiveness can be attributed to chance or whether they really capture a trend. Statistical hypothesis testing is necessary as SBST techniques are always associated with a certain level of random variation in terms of cost or effectiveness. Because statistical testing is a standard practice, we will not detail it further here and interested readers may consult reference [40] for more details.

Statistical hypothesis testing should be used to accept/reject research hypotheses related to the cost-effectiveness analysis of SBST techniques and comparison baselines. The choice of a specific statistical test depends on the specific objective of SBST. In our context, hypothesis testing falls into three broad categories: (1) Comparing samples of runs in terms of effectiveness and cost. For example, comparing average or maximum branch coverage achieved across runs of alternative SBST techniques and baselines of comparison. (2) Comparing samples of runs in terms of "successful" runs. For example, comparing the proportion of runs that find a deadlock across alternative SBST techniques and baselines of comparison. (3) Comparing samples of targets (e.g., control flow branches) in terms of cost (e.g., iterations) or effectiveness (e.g., percentage of runs reaching that branch). In this last case, the samples are not independent, because observations in each sample are paired (identical targets). This leads to the application of specific statistical tests for paired samples. Moreover, though this is a standard issue, there can be two or more samples, and this will also affect the specific statistical test to be used. Moreover, as usual in other contexts, specific statistical tests have to be selected and justified based on the data distributions of the samples being compared to avoid drawing incorrect conclusions from the analysis. Statistical tests are usually classified as parametric and non-parametric [52]. When the sample follows a specific distribution (e.g., normal), certain parametric tests are applicable (e.g., t-test). Alternatively, non-parametric statistical tests are used when no appropriate assumptions can be made about the sample distributions. The issues related to selecting appropriate tests are however discussed in standard textbooks and will not be further addressed here. In Table 1, as a guideline, we provide a mapping between the analysis situations we have encountered in SBST studies and the type of statistical tests that are suitable (for the sake of simplicity, we are assuming two samples, that is, the comparison of two techniques). This mapping is illustrated with examples.

Data analysis should both address statistical and practical significance of differences among alternative search techniques. The former assesses whether differences among search techniques can be due to chance. The latter assesses whether the difference can be considered of practical significance, that is, whether they would make any difference in the day-to-day practice of test case generation given the specific test objectives being considered. For example, if statistical testing based on a large number of runs show that there is a significant difference between the cost of two search techniques in terms of time required for finding the best test suite, the actual difference may not be of practical importance if it is in the range of a few minutes. On the other hand, a lack of statistical significance despite a visible difference may be due to small samples, and therefore a lack of statistical power, which in our context means that the number of runs for each compared

Table 1 Mapping of SBST problems to statistical tests

SBST Analysis Type	**Type of Statistical Comparison**	**Example in the Context of SBST**	**Type of Statistical Test (assuming two samples)**
Comparing samples of runs in terms of effectiveness and cost	Comparing central tendencies of two or more independent samples, each corresponding to a SBST technique	Comparing maximum branch coverage achieved across all runs between two SBST techniques	Parametric *t*-tests or Non-Parametric Mann-Whitney U test
Comparing samples of runs in terms of “successful” runs	Comparing proportions in independent samples, each corresponding to a SBST technique	Comparing the proportion of runs finding deadlocks across different SBST techniques	z-score test for proportions
Comparing samples of target in terms of cost to reach them or frequency at which runs reach them	Comparing central tendencies of matched pairs across samples	Comparing the frequency, across samples of runs matching each SBST technique, according to which a branch (target) is covered. Note that the observations across samples are paired as they correspond to identical branches.	Parametric Paired *t*-tests or Non-Parametric Wilcoxon or Sign test

search technique may be too small. The larger the number of runs, the more likely one is to obtain statistical significance when observing differences.

Data analysis methods for scalability. Scalability is used to assess whether a SBST technique can be applied to either larger or more complex SUTs and still have satisfactory effectiveness and cost. If the aim of the empirical study is to show the scalability of a SBST technique then appropriate measures of size and complexity should be clearly defined. There will be at least two measures involved – one size measure that will be scaled up through successive SUTs and the other that will measure the corresponding performance (cost and effectiveness). Then the effect of scaling up a particular measure can be reported in terms of a statistical relationship (recall the unavoidable random variation). For example, we may investigate several SUTs of variable sizes in terms of lines of code and then assess whether a SBST technique can still reach a certain level of coverage at acceptable cost (e.g., measured as the number of generations) for larger SUTs and analyze how this cost evolves with the size of the SUT. A positive, exponential relationship between size and cost might then be problematic, for example, as it would undermine the applicability of the technique for large scale test models and systems. Similarly, if effectiveness (e.g., in terms of achieved coverage) is strongly decreasing as a function of SUT size, we also have a scalability problem.

As for scalability analysis, we need to characterize relationships between SUT size variables and measures of the SBST technique's cost and effectiveness. Such techniques are typically analyzed through regression analysis, though in practice, because the number of SUTs under study is likely to be small, such analysis is more likely to be qualitative, that is simply based on observing scatter plots in the cost-effectiveness and size space.

3.3.10 Discussion on validity threats

Validity threats should be considered throughout any empirical study, right from the study definition and design up to the analysis and interpretation of results [52]. The following types of threats can be discussed:

Construct validity threats. Measures of cost, effectiveness, and SUT size should be appropriate and justified given the context and objectives of investigation. No measure is expected to be perfect as the above concepts are usually not readily measurable. But in practice, by using several, complementary measures of cost, effectiveness, and SUT size, one is in a position to compare the cost-effectiveness and scalability of alternative search techniques.

Internal validity threats. If a SBST technique performs better than another one, whether regarding effectiveness or cost, can it be due to something other than the SBST technique? This could possibly be due to the following: 1) poor parameter settings of one or more of the SBST techniques, 2) the biased selection of SUTs that have certain characteristics that can favor a certain SBST technique.

Conclusion validity threats

- Has random variation been properly accounted for? Since SBST techniques use MHS algorithms, randomness in results (inherent to metaheuristic approaches) should be accounted for, as discussed above. Has it been done in such a way as to enable statistical comparisons? It implies that a sufficient number of independent runs be performed to obtain a sufficient number of observations.
- Was the right statistical test employed? Statistical test procedures should be carefully selected given the hypothesis method (e.g. one-tailed vs. two tailed hypothesis) and the data collected (distributions of cost and effectiveness). Otherwise, certain required properties of a particular statistical test could be inadvertently violated leading to incorrect conclusions. For example, many statistical tests assume that data distributions be normal [52].
- Is there any practically significant difference? To answer this question, the magnitude of the differences must be reported– this is known as the effect size and determines the practical significance of the results.

External validity threats. This is a difficult issue, as whether results can be generalized depends on whether the SUTs under investigation are representative of the targeted application domain and whether the faults considered (if used to assess test effectiveness) are representative of real faults. Ideally, SBST empirical studies should also be run on many different SUTs of the target type, but every research endeavor faces limitations in terms of time and resources. At the very least, the issue should be carefully discussed and a good case should be made as to why one should be able to trust that the observed results can be generalized.

4 Research Method

In this section, we will explain our review protocol. We define the research questions that this review attempts to answer, along with how we selected papers for inclusion and the data that we extracted.

4.1 Research questions

The most important stage of any systematic review is to precisely define the research questions. Once the research questions have been specified, the systematic review can then proceed with the search strategy to identify relevant studies and extract the data required to answer the questions [13]. In this paper, we are interested in investigating empirical studies in the domain of SBST. To proceed with our investigation, we defined the following three research questions:

RQ1: What is the research space of search-based software testing?

The objective of this question is to characterize the research that has been undertaken so far. Though the research space can be identified from different angles, because our systematic review is about SBST, basic features of software testing (such as test level, targeted faults, test model, type of test cases, and application domain) and the type of MHS algorithms seem relevant characteristics to define the research space. Because of size constraints, RQ1 will not be addressed in detail in this paper and the results will be simply summarized to provide context information to the reader and facilitate the interpretation of subsequent research results. Interested readers may consult the technical report [5] corresponding to this paper for a detailed discussion of the results.

RQ2: How are the empirical studies in search-based software testing designed and reported?

A study that has been properly designed and reported (as discussed in Section 3) is easy to assess and replicate. The following sub-questions aim at characterizing some of the most important aspects of the study design and how well studies are designed and reported:

- RQ2.1: How well is the random variation inherent in search-based software testing, accounted for in the design of empirical studies?
- RQ2.2: What are the most common alternatives to which SBST techniques are compared?
- RQ2.3: What are the measures used for assessing cost and effectiveness of search-based software testing?
- RQ2.4: What are the main threats to the validity of empirical studies in the domain of search-based software testing?
- RQ2.5: What are the most frequently omitted aspects in the reporting of empirical studies in search-based software testing?

RQ3: How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

This research question attempts to synthesize the actual results reported in the studies in order to assess how much empirical evidence we currently have. To answer this question, we address the following sub-questions:

- RQ3.1: For which metaheuristic search algorithms, test levels, and fault types, is there credible evidence for the study of cost-effectiveness?
- RQ3.2: How convincing is the evidence of cost and effectiveness of search-based software testing techniques, based on empirical studies that report credible results?
- RQ3.3: Is there any evidence regarding the scalability of the metaheuristic search algorithms for test case generation?

4.2 Study selection strategy

This is the step of a systematic review that aims at ensuring the completeness of the selection of papers on which the review is based. Study selection involves two main steps: (1) selection of the source repositories and identification of the search keywords (2) inclusion or exclusion of studies based on certain inclusion and exclusion criteria.

4.2.1 Source selection and search keywords

The process of selecting papers is started by executing a search query on the source repositories, which provides a set of papers. Since this set of papers is then subsequently used for all manual inclusions and exclusions, the selection of appropriate repositories and search strings is of utmost importance as it directly affects the completeness of the systematic review. The repositories that we used are: *IEEE Xplore*, *The ACM Digital Library*, *Science Direct* (including *Elsevier Science*), *Wiley Interscience*, *Springer*, and *MIT Press*. The first two repositories covered almost all important conferences, workshops, and journal papers, which are published either by IEEE or ACM. The next four repositories were mostly used for finding papers that are published in leading software engineering journals.

We selected the following journals based on [13]: *IEEE Transactions on Software Engineering (TSE)*, *ACM Transactions on Software Engineering and Methodologies (TOSEM), IEEE Software (SW)*, *Springer: Software Testing Verification and Reliability (STVR)*, *Springer: Empirical Software Engineering, Elsevier Science: Information and Software Technology (IST)*, and *Elsevier Science: Journal of Systems and Software (JSS)*.

Since our review is about SBST, we also included journals relating to software quality assurance and evolutionary computing: *Springer: Software Quality Journal, Springer: Genetic Programming and Evolvable Machines*, *IEEE: Transactions on Evolutionary Computation*, and *MIT Press: Evolutionary Computation*. Another important source of publications that we included was the *Genetic and Evolutionary Computation Conference (GECCO)*. Based on the impact factor, GECCO is one of the top conferences in the fields of artificial intelligence, machine learning, robotics, and human-computer interaction [1] and is directly related to the field of genetic and evolutionary computation. GECCO's proceedings were published by Springer in 2003 and 2004 and afterwards by ACM.

A systematic way of formulating the search string includes (1) identifying the major search keywords based on the research questions (2) finding alternative words and synonyms for the major keywords and (3) creating a search string by joining major keywords with Boolean AND operators, and the alternative words and synonyms with Boolean OR operators.

Based on our main research focus, which is investigating empirical studies in the domain of SBST, the following major search keywords are used in this paper: *software testing* and *metaheuristic search algorithm*.

We did not use *empirical study* as a keyword because we realized that not all papers that perform an empirical study, in the broad sense that we have defined it, use this keyword.

To formulate our search query we tried a number of search strings and came to the conclusion that *'software testing'* as an expression is not a good keyword because there are many papers which don't use these two words together but are nevertheless related to software testing. These papers may use terms such as testing, test case, test data and so on. On the other hand if we used the term testing alone, we would find too many unrelated papers. So we decided to use the terms *software* and *test* linked together with a Boolean AND instead of using *'software testing'* as an expression. Using *'software'* and *'test'* will find almost all related papers to software testing, but to make sure that we do not miss any interesting papers in test case generation we used the expression of *'test case generation'* as an alternative for software testing.

Metaheuristic search algorithm is the second major term and also has many alternatives. We used general terms such as '*evolutionary algorithm*', '*meta-heuristic*', and 'search based' to explore the domain. Also, names of different MHS algorithms were used to make sure that no related papers were missed.

We also wanted to make sure that we do not miss any papers that have explicitly used the widely used term *'evolutionary testing'*, and thus included the expression of '*evolutionary testing*' as a separate search string joined with the main string by an OR Boolean operator. The above decisions lead to the following search string shown in Figure 3.

The whole string is searched in each repository in all titles, keywords, and abstracts. The expression '*evolutionary testing*' is searched in the entire contents of all papers in the repositories as well.

One problem that we realized after some manual checking of the results of the search query was the fact that some search engines, such as IEEE Xplore, differentiate between the singular and plural form of words. To deal with this, we had to add some more alternative words and expressions to the search string by adding a '*s*' to the end of all the words we already had. For example, we added '*evolutionary algorithms*', '*meta-heuristics*', '*genetic algorithms*' and so on.

After finalizing the search string, the search query was run on the search engines of different repositories.

4.2.2 Study selection based on inclusion and exclusion criteria

Metaheuristic search algorithms have been used to automate a variety of software testing activities such as test case generation, test case selection, test case prioritization, and optimum allocation of testing resources. Since the focus of this systematic review is on test case generation, it is therefore necessary to define suitable inclusion and exclusion criteria for selecting relevant papers. In this section, we will discuss and justify the inclusion and exclusion criteria that were used.

We executed our search query on all selected databases and found 450 (after removing duplicates from different repositories) research papers in total. We only included papers up to the year 2007. In order to select the relevant papers to answer our research questions, we applied a two-stage selection process. At the first stage, we excluded papers based on abstracts and titles. All the papers were divided into three sets and each set was read by a researcher. We applied the following exclusion criteria:

- Abstracts or titles that do not discuss test case generation or any of the alternate terms that we used were excluded.
- Abstracts or titles that do not discuss the application of any MHS algorithm to automate test case generation were excluded.

{(((*software* AND *test*) OR *'test case generation'*) AND (*'evolutionary algorithm'* OR *'hill climbing'* OR *'metaheuristic'* OR *'meta-heuristic'* OR *'genetic algorithm'* OR *'optimization algorithm'* OR *'search-based'* OR *'search based'* OR *'simulated annealing'* OR *'ant colony'*)) <in abstract, keywords, and title>} OR *'evolutionary testing'* <in abstract, keywords, title, and whole content>

Figure 3 The search string used for selecting the papers from repositories

If a researcher was unsure about a paper after reading its title and abstract, then the paper was included for the second phase of selection. After applying the inclusion criteria for the first phase, we were left with 122 papers.

At the second stage, we again divided the papers into three equal sets and divided them among three researchers to check the contents of each paper. We excluded papers based on the following exclusion criteria:

- Posters, extended abstracts, technical reports, PhD dissertations, and papers with less than three pages were excluded. Our goal was to account only for peer-reviewed, published papers that presented sufficient technical details.
- The papers that do not automate test case generation were excluded because this is the scope of our review.
- The papers that do not report any empirical study (see Section 2.3 for details on what we mean by empirical studies) were excluded.

In the cases where a researcher could not decide whether to keep or exclude a paper, then the paper was discussed with other researchers and a decision was made, by consensus. It is important to mention that we didn't exclude papers based on the realism of SUTs used in their case studies. The reason is that exclusion would then be subjective as no precise criterion can be defined and would probably lead to a very small number of selected papers. After applying the second phase of selection, we remained with 68 papers that contained empirical studies about test case generation using MHS algorithms. However, four of these 68 papers, presented empirical studies that had already been reported in some other paper. This occurred, for example, when the journal version of a conference paper was found. In these cases we extracted data about the study from both the conference and journal versions of the paper and reported them as one study. Thus in the rest of the review we mention only 64 papers in total, even though we did analyze 68 papers. Details on the number of papers found in each database and number of papers included after applying inclusion and exclusion criteria are listed in Table 2.

Table 2 Distribution of papers after applying inclusion and exclusion criteria

Repository	Number of Included Papers After Applying Search Query	Number of Papers After Stage 1 Exclusion Criteria	Number of Papers After Stage 2 Exclusion Criteria
IEEE Xplore	297	77	33
ACM Digital Library	117	27	22
Wiley Interscience	8	2	2
Science Direct	8	3	2
Springer	19	12	8
MIT Press	1	1	1
Total	450	122	68

4.2.3 Data extraction

We designed a data extraction form in Microsoft Excel to gather data from the research papers. We collected two sets of information from each paper. The first set included standard information [30] such as name of the paper, authors' names, a brief summary, researcher's name, and additional comments by the researcher. The second set included the information directly related to answering the research questions (see Table 3 for a summary list and [5] and Section 3 for details on each data item). To assess and improve consistency of data extraction among the researchers, a sample of papers were selected and read by all researchers and the relevant data extracted. The extracted data was then discussed by the researchers to ensure a common understanding of all data items being extracted and where necessary, the data collection procedure was refined. The final set of selected papers from each repository was then divided amongst three researchers. Each researcher read the allocated papers and extracted the data from the papers. In order to mitigate data collection errors, the data extraction forms of each researcher were read and discussed by two others. All ambiguities were clarified by discussion among the researchers.

Table 3 Research questions and type of data collected

Research Questions		Type of Data Collected
RQ 1		Type of MHS algorithms, test levels, targeted faults, test model, type of test cases, and application domain
RQ 2	RQ 2.1	Number of runs, analysis method
	RQ 2.2	Comparison baseline
	RQ 2.3	Measures of cost, measures of effectiveness
	RQ 2.4	Conclusion, external, internal, and construct validity threats
	RQ 2.5	All of the information from RQ2.1 to RQ2.4 is used, formal hypothesis, object selection strategy, data collection method
RQ 3	RQ 3.1	Test level, fault type, MHS algorithm
	RQ 3.2	Test purpose, comparison baseline, cost and effectiveness results
	RQ 3.3	Scalability results

5 Results

The following section outlines the results related to the research questions. No formal meta-analysis of the results of the empirical studies could be performed because of the variations in the way empirical studies are conducted and reported, and as such, results are compiled in structured, tabular form.

5.1 RQ1: What is the research space of search-based software testing?

As previously mentioned, we provide here only the most salient results to the research question. The reader is invited to read the technical report [5] corresponding to this paper to obtain detailed results. The results show that in the majority of the papers, SBST techniques have been applied at the unit testing level (75%). Moreover, most papers (78%) do not target any specific faults but rather focus on structural coverage of different test models. The most commonly used algorithm is the GA and its extensions (73%), followed by a more limited use of simulated annealing and its extensions (14%). There could be several reasons for this frequent use of genetic algorithms. First, there are numerous publications on the application of GA to various problems [21]. Furthermore, substantial empirical data is available for the different parameter settings required by GAs and this greatly helps the choice of appropriate parameters for a specific problem to be solved [46]. This, together with the many books [16, 26] that exist on genetic algorithms, makes it easier for researchers to learn how to adapt genetic algorithms to their context. Second, being a global search algorithm, GAs have been shown to usually perform better than local search algorithms [53], though there is no evidence showing that GA is better than other global search algorithm [21]. Last, GAs have many well known implementations in the form of commercial tools [42] and frameworks [2, 34], which greatly facilitate their practical application.

5.2 RQ2: How are the empirical studies in search-based software testing designed and reported?

The purpose of this research question is to investigate and assess the design and reporting of empirical studies in the domain of search-based software testing. To answer this question, we further divided this question into five sub-questions. By answering each sub-question individually, we will answer the main research question. Though the results are presented in tables that summarize the main findings, the reader can obtain a break-down of which papers led to these findings in the technical report [5] corresponding to this paper.

5.2.1 RQ2.1: How well is the random variation inherent in search-based software testing, accounted for in the design of empirical studies?

We discussed the necessity and importance of accounting for random variation and using appropriate data analysis methods in Section 3.3. To assess whether random variation has been accounted for, we classified the papers into two main categories: (1) papers which accounted for random variation in their design and reported this information and (2) papers which either did not account for random variation or did not report it well. To be classified in the first category, the study in the paper had to report the number of times the MHS algorithm was executed, sufficient information to determine whether the runs were independent, and report the data analysis methods used to compare alternative algorithms and baseline solutions. The independence of different runs can be determined in different ways in different MHS algorithms. For instance, in the case of the HC algorithm, if it is started from the same starting point in each run using the same strategy to select neighbors, then all the runs will not be independent and hence every time the algorithm will find the same solution. Different runs in HC are normally made independent by choosing different starting points in each run or by using a random strategy to select neighbors. Additionally, the number of runs for each MHS algorithm had to be at least ten, a ballpark figure to enable the application of statistical hypothesis testing with minimal statistical power. Papers that did not report the number of runs or were executed less than ten times were placed in the second category (Random Variation Not Accounted).

Within the first category, we further divided the papers according to the type of data analysis that had been performed. If only the average of the results or the percentage of successful runs over all runs was reported, then these papers were classified as having "poor" descriptive statistics (the definition of successful run varies across papers, but generally speaking, if the test target to be covered is found, then the run is considered successful. A test target, for example, could be a branch to cover). This is because the average does not convey any information about the dispersion of the results being examined. Papers which report the level of variation as well as the measures of central tendency are counted in the sub-category "good" descriptive statistics. The final category is the set of papers that in addition to reporting "good" descriptive statistics also reported the results of statistical hypothesis tests comparing MHS algorithms and baselines and establishing the statistical significance of differences. However, most of the papers did not have detailed information on sample distributions and the validity of statistical test assumptions. It was therefore usually not possible to determine if a paper used the correct

Table 4 Results of how random variation is accounted for in empirical studies

Random Variation Accounted			Random Variation Not Accounted	
Poor Descriptive Statistics	Good Descriptive Statistics	Statistical Data Analysis	Random variation not discussed or accounted for	Insufficient number of runs
24	8	7	20	5
38%	12%	11%	31%	8%

statistical procedure for a particular problem and data set.

The results in Table 4 show that 25 papers did not account for random variation. Most of these, 20 papers, either did not provide any information about the number of runs or just reported the result of one unknown run (the best or the only run). In five papers, the study was repeated less than ten times.

Amongst 39 papers which accounted for random variation, 24 papers reported only the average of the cost or effectiveness results across all runs, for example, the average number of killed mutants as an effectiveness result or the average number of iterations as a cost result. In some cases, the percentage of successful runs amongst all runs is reported instead of, or along with the average of the effectiveness results (e.g., average coverage or average mutation score). At least one measure of dispersion like standard deviation, variance, or the variation interval ([Min, Max]) was reported for eight papers. These papers are categorized as having "good" descriptive statistics. There were seven papers that reported statistical tests as well as good descriptive statistics. One or more of the following statistical tests were used: t-test, paired t-test, Mann-Whitney test, F-test, ANOVA, and Tukey test [40, 44]. There was one paper in this sub-category, which reported the use of statistical tests, but did not specify the specific test being used and did not provide any descriptive statistics. From the results, we can see that 39% of the papers did not account for random variation at all, and 38% of the papers only had "poor" descriptive statistics, so in total 77% of papers either did not account for random variation or reported it poorly. The remaining 23% of papers are divided between 12% providing only good descriptive statistics and just 11% performing some kind of statistical hypothesis testing to assess the statistical significance of differences that is whether they can be due to chance. To answer RQ2.1, this review suggests that SBST would greatly benefit from paying more attention to accounting for random variation in search heuristics and applying more rigor in analyzing and reporting cost and effectiveness results.

5.2.2 RQ2.2: What are the most common alternatives to which SBST techniques are compared?

In assessing the cost-effectiveness of any technique, the comparison baseline is an important factor. In order to classify the papers we defined four categories of comparison baselines: (1) ‘Global SBST’, where the baseline of comparison is a SBST technique using a global MHS algorithm, (2) ‘Local SBST’ includes the techniques that use a local MHS algorithm such as HC, (3) ‘Non-SBST’ baselines do not use a SBST technique and feature baselines such as random search, and (4) ‘Not discussed’ addresses papers that do not report any comparison baseline.

The comparison to non-SBST techniques or local SBST techniques serves a dual purpose: it helps determine if the problem at hand is simple enough to be satisfactorily solved by a simple search algorithm; otherwise it provides justification for why a more complex SBST technique is necessary. In addition, a simple baseline of comparison is necessary to assess the benefits of using complex SBST techniques.

As shown in Table 5, 16 studies did not discuss the comparison baseline at all. These studies did not include any kind of comparison; they usually introduced the use of a MHS algorithm for test case generation and performed an empirical study to show that the technique does indeed generate satisfactory test cases. These papers are missing the justification for why the SBST technique was necessary to address the test case generation problem at hand and how much better it actually is compared to other existing, simpler techniques that are available to solve the problem at hand.

There were 34 studies that reported ‘Non-SBST’ baselines within which random search is used in 24 studies, static analysis in three, greedy algorithm in three, constraint solving in one study and three studies used some other technique specific to their context. We see that random search is the most commonly used comparison baseline amongst Non-SBST techniques. There is limited use of ‘Local SBST’ baselines with only three studies using HC. There are many studies (33) that used Global SBST techniques as comparison

Table 5 Comparison baselines used in SBST in terms of number of papers

Global SBST baselines			**Local SBST baselines**	**Non-SBST baselines**					**Not Discussed**
GA and Ext.	SA and Ext.	Others	Hill Climbing	Random Search	Static Analysis	Greedy Algorithm	Constraint Solving	Others	
22	6	5	3	24	3	3	1	3	16

baselines. This is usually done when investigating the effects of different parameter settings of MHS algorithms. This is most evident within GA and SA where 22 studies used either GA or its extensions as baselines and six studies used SA and its extensions.

5.2.3 RQ2.3: What are the measures used for assessing cost and effectiveness of search-based software testing?

Assessing the cost-effectiveness of SBST techniques for test case generation is the main objective of empirical studies in our context. Therefore, measuring cost and effectiveness in a valid manner is a basic requirement for all empirical studies.

Effectiveness measures. As it is discussed in Section 3, effectiveness measures are categorized into two main classes: coverage-based and fault-based measures. Under the coverage-based category, we found three main sub-categories: (1) control flow based coverage criteria such as branch, statement, path, condition, and condition-decision coverage (2) data flow based coverage criteria such as all-DU coverage, and (3) N-wise coverage criteria, when SBST techniques are used for testing combinatorial designs [36]. In the category of fault-based measures, mutation analysis is the core strategy and mutation score and the number of mutants killed are measures that were found in this review.

We found some other measures for effectiveness, which are still related to the quality of the generated test cases, but do not fit into any of the above categories. In this review, these measures are labeled “Others”. Based on the papers included in this review, we identified two sub-classes among them and labeled the rest as miscellaneous. Papers in the first sub-category use different kinds of measures related to the execution time of test cases and we called these time-based measures. The second sub-category addresses the distribution of fitness values of individuals (solutions) as the measure of effectiveness (e.g., average, maximum fitness). Such a measure is usually used when the goal of a search algorithm is not finding a targeted solution, but the goal is to be as close as possible to the targeted solution. An example of such papers is in [8, 9], where the goal was stressing the real-time systems by scheduling input sequences to maximize delays in the execution of targeted aperiodic tasks. In this study, the cost is measured by fitness values, which shows how close the completion time of a specific task is to its deadline. Table 6 presents the number of papers in our review per the category of effectiveness measures.

Table 6 Distribution of effectiveness measures across empirical studies

Coverage-based measures			Fault - based measures	Others			No effectiveness measure
Control flow	Data flow	N-wise		Time-based measures	Fitness value of individuals	Miscellaneous	
43	2	2	11	6	5	3	3

The data we collected revealed 61 papers using one or more effectiveness measures in a total of 72 different effectiveness measurements across reported studies. There were three papers that did not discuss the effectiveness of the SBST technique at all. There were 47 instances (65%) that used some type of coverage criterion as the measure of effectiveness. The most often used criteria were control flow based criteria with 43 instances (60%). Among them, 23 instances (32%) used branch coverage, which is the most frequently used effectiveness measure. All-DU coverage, which is based on data flow analysis, was used in two instances and two instances used N-wise coverage as the coverage criterion.

There were 11 instances (15%) that used fault detection rate as the measure of effectiveness, where mutation analysis is used so as to report the mutation score or the number of killed mutants. In some cases, the fault-based measures are reported along with other effectiveness measures. Among the 14 instances (19%), which used the other measures for the quality of test cases, five papers used the fitness value of individuals and six papers used different kinds of execution-time based measures. Most of the time-based measures were related to CPU cycles spent for test case execution. They are used in studies which try to use SBST techniques to generate test cases that will find the best/worst case execution time of a program.

Looking at the results in Table 6, we can see that control flow based coverage criteria targeted at white-box testing are the most often used effectiveness measures and as we mentioned in the above discussion, branch coverage is the criterion that has received the most attention. As a result, this problem is now pretty well understood and there is a widely accepted, standard way of calculating fitness values based on approximation level and branch distance [37] on control flow graphs. Fault-based effectiveness measures received relatively little attention in the literature reporting SBST studies as compared to coverage-based measures. Similarly, the applications of SBST techniques to artifacts other than code are rare as white-box testing seems to have been by far the main focus.

Cost Measures. Based on the definition of cost measures in Section 3 and what we found in this review, we categorized cost measures into two main classes (1) ‘cost of finding the target’, which is related to the cost of automating test case generation and (2) ‘cost of executing the generated test suite’, which is related to the cost of test case execution. These are both relevant and complementary. Based on the measures found in the studies, the first category is classified into four sub-categories:

(a) the number of iterations
(b) the cumulative number of individuals in all iterations
(c) the number of fitness evaluations an algorithm needs to find the final solution
(d) test case generation time.

The only measure for the category of ‘the cost of executing generated test suite’ that we found in the papers was the size of the test suite, which is a surrogate measure for test execution time.

Table 7 shows that among 64 papers, seven papers did not perform any cost analysis and in the remaining 57 papers most empirical studies reported at least one cost measure in 70 different cost measurements reported across studies.

Based on the abovementioned classification, 62 instances (86%) used measures in the category “Cost of finding the target”. The most often used measure among them was the number of iterations, which is used in 27 instances (39%). A total of six instances (4%) used the number of individuals (test cases) and the number of fitness evaluations is used by 14 instances (20%) as the measure of cost. Finally, there were 15 instances (21%) that used the ‘test case generation time’ measure.

In the second main category, ‘cost of executing the final test suite’, the size of test suite was the only measure that we found and it was used in eight instances. Some of these instances, which report the number of test cases in the final solution, reported the cost of finding the target as well. In some of these instances, the target of the SBST technique was actually creating test suites with minimum size for covering a specific criterion such as a minimal test suite that exhibits pair-wise coverage [20].

Summarizing the results of cost measures, we can see that the most commonly used measure is the number of iterations. This measure is, however, the least precise measure based on the discussion in the framework in Section 3. Another conclusion is that most studies use cost measures only for comparison purposes with other alternative techniques. There are just 15 instances (21%) that used measures such as test case generation time, which conveys whether a particular technique is likely to be practical and scale up.

Table 7 Distribution of cost measures across empirical studies

Cost of finding the target				Cost of executing the final test suite	No cost Measure
Number of iterations	Number of individuals	Number of fitness evaluations	Test case generation time	Size of test suite	
27	6	14	15	8	7

5.2.4 *RQ2.4: What are the main threats to the validity of empirical studies in the domain of search-based software testing?*

In order to answer this question, we carefully assessed the studies using the proposed framework in Section 3. For the construct validity threats, we looked at the validity of the cost and effectiveness measures. The most frequently observed threat was using some measures of cost that have severe limitations as they are not precise. As discussed in the framework, the imprecision of cost measures such as ‘the number of iterations’ makes the comparison between different SBST techniques very coarse grained. In addition, measures such as the number of iterations, the number of individuals, and the number of fitness evaluations can only be used for comparison across SBST techniques and cannot demonstrate the practicality of SBST techniques. On the other hand, cost measures such as ‘test case generation time’, if measured as clock time, are suitable for showing the practicality of a technique under time constraints. Such measures are, however, platform dependent and therefore not easy to use for comparisons across techniques and studies.

The most frequently encountered conclusion validity threat is related to accounting for the random variation that exists in the results obtained from SBST techniques. As discussed in RQ2.1, 39% of the papers did not take the random variation of results into account and 38% did not analyze or report it properly. This leads to a frequent threat regarding the statistical significance of the results. Therefore, not accounting for randomness and not applying proper data analysis (Section 3.3 and RQ 2.1) makes it very difficult to confidently draw practical conclusions from the results reported in most studies. Moreover, among the 11% of papers that discussed statistical hypothesis tests, just one paper has discussed the practical significance of differences that is whether differences among techniques justify the use of more complex techniques.

Regarding internal validity threats, the most important concern is the instrumentation of code and the use of different tools for data collection without reporting sufficient information about them. If the data collection and code instrumentation is not done through a well-identified and available tool, then detailed information about the process of data collection should be reported. An example of this would be the use of a tool that instruments the code to collect, for instance, branch coverage information. If the tool is developed for experimentation purposes only and has not been thoroughly tested, then the coverage information generated by the tool might not be reliable and hence lead to an internal validity threat. A possible way to deal with this validity threat is to use readily available (open source, downloadable, or commercial) tools for this purpose.

The lack of clearly defining the target SUTs and having a clear object selection strategy are the most common threats to external validity. Usually the algorithms are executed on very small programs and no clear justification is provided for their choice and why they may be representative of the target domain, if specified. This can result in invalid generalization of the results.

5.2.5 RQ2.5: What are the most frequently omitted aspects in the reporting of empirical studies in search-based software testing?

In the previous sections, we have discussed the lack of properly reported descriptive statistics and statistical hypothesis testing (statistical significance) as the most commonly missing aspects in many empirical studies. Only 23% of the reviewed papers reported proper descriptive statistics or statistical significance results. In addition to this aspect, as discussed in the framework, there are other aspects that are also important and should be reported. These aspects are: discussion of validity threats, specification of formal test hypotheses, object selection strategy, parameter settings, and data collection method. For validity threats, 10% discussed conclusion validity, 6% discussed external validity, 3% discussed construct validity, and only 3% of the papers discussed internal validity threats. We found that only two papers out of 64 specified formal hypotheses, 44% of the papers discussed object selection strategies, and 39% of the papers described their data collection methods. Parameter settings (see [5]) were discussed by most, but not all of the papers (88%). However, all papers did not discuss all parameters required for their study; usually there is only a partial discussion. In some cases the authors provide justification of why they chose particular values for the parameters but this was rare.

Summarizing the above information, Table 8 depicts the most frequently omitted aspects in the reporting of empirical studies. Not reporting this information makes the full interpretation of the results very difficult. For example, poor reporting may make it difficult to determine whether differences are statistically significant, and whether differences are expected to matter in practice. It is also usually difficult to determine if results can be generalized and to what domain.

5.2.6 Conclusion

In our context, defining good and relevant cost and effectiveness measures is a prerequisite for a useful empirical study. Almost all of the papers use appropriate (though not perfect) cost and effectiveness measures to perform empirical studies. However, there were two major problems in the majority of the papers. First, most of the papers do not account for the random variation in cost and effectiveness of SBST techniques. Even the majority of the papers that did account for the random variation didn't use proper data analysis and reporting methods (descriptive statistics and statistical hypothesis testing). Thus, there is a general lack of rigor in the statistical analysis and reporting of results in most empirical studies assessing the use of MHS algorithms for test case generation. Second, most of the papers didn't demonstrate the benefits of SBST by comparing it with simpler, techniques such as random search or HC. These two factors are highly important for yielding interpretable empirical studies in the context of test case generation using SBST techniques. Furthermore, many other relevant aspects of empirical studies such as the reporting of validity threats, the definition of formal hypotheses, the object selection strategy, and data collection methods are not reported by most of the papers. We can therefore conclude that most empirical studies in the context of test case generation using SBST techniques are still not properly conducted and reported and that improving this situation should be an important objective of the research community for future studies.

Table 8 The most omitted aspects of empirical studies

The most omitted aspects in the reporting of empirical studies		Number of papers	Percentage
Good Descriptive statistics and statistical test		15	23%
Validity threats	Construct	2	3%
	Internal	2	3%
	Conclusion	7	10%
	External	4	6%
Formal Hypothesis		2	3%
Object selection strategy		28	44%
Data collection method		25	39%

5.3 How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

There is a lot of research being conducted on test case generation based on MHS algorithms. In order to draw general conclusions from the current body of work, we need to assess how convincing is the evidence regarding the cost, effectiveness, and scalability of SBST techniques. The first step is to clearly identify studies that provide complete and credible evidence from an empirical standpoint. Credible results are the consequence of a well designed and conducted empirical study. Based on the discussions in Section 3, a well designed study in the context of SBST should account for the random variation present in the results and have a meaningful comparison baseline to show that the targeted test problem benefits from a MHS approach. Therefore, in order to answer this research question, we first selected papers that at a minimum account for the random variation of results and compare their technique with the results of a simpler, non-SBST technique (such as random search, static source code analysis, or some other technique applicable to the test problem under consideration) or with HC. The first sub question, RQ3.1, will provide an overview of these papers. The second step to answer RQ3 is to select those papers that performed and reported proper data analysis. To satisfy this criterion, we expect papers to report descriptive statistics on the variation in the results (cost, effectiveness), where relevant or results of statistical hypothesis testing comparing alternative test case generation algorithms, and in particular MHS algorithms with simpler baseline alternatives. We deemed this set of papers as having credible evidence regarding the cost, effectiveness, and scalability of SBST. In sub question RQ3.2, we provide detailed information about the cost and effectiveness results presented in these papers along with a short description of the test problem that they tackled.

5.3.1 RQ3.1: For which metaheuristic search algorithms, test levels, and fault types is there credible evidence for the study of cost-effectiveness?

This sub-question provides a summary of the research papers that met the minimum criteria of accounting for random variation in results and performing comparisons with a simpler non-SBST or local SBST techniques. Out of the 64 papers that we analyzed, we found 39 that accounted for random variation of results. This number was reduced to 18, after selection of only those papers that also had either a non-SBST or a simple, local MHS comparison baseline. Thus, based on the criteria that we used, we had to exclude 46 papers as not being applicable for answering our research question. It is worth mentioning that there were 14 papers among those 46 discounted papers that had the minimum requirement of accounting for random variation, but did not have a non-SBST or local MHS comparison baseline. For example, they may have proposed an extension to a genetic algorithm that would possibly enhance its capacity for test case generation and compared their results to a genetic algorithm not having this extension. In this review, those studies are not considered as credible evidence, since they do not show, in any way, that a simple non-SBST technique such as random search or a local MHS such as HC could not, in this particular context, equal or outperform their technique. This is an important consideration, since there is no a priori reason to believe that a MHS algorithm is more cost-effective and efficient than simpler algorithms in all test case generation contexts. The size of the search space is only a weak indicator of the extent of the search challenge as the search difficulty also depends on the search space landscape and distribution of satisfactory solutions across this space. Table 9 summarizes this set of 18 papers in terms of the MHS algorithms used, the testing levels, and the fault types targeted in the empirical studies. These papers are referred to as 'Minimum Criteria papers' in Table 9.

As can be seen in Table 9, amongst the 18 papers that report credible evidence, most papers (13 out of 18) applied a SBST technique at the unit testing level. The most commonly investigated MHS algorithm is the genetic algorithm with 12 papers out of 18, followed by simulated annealing with just four papers. This trend is the same as that observed in the full set of 64 papers in Section 5.1 There are also only two papers that target specific faults, one targeting functional faults and the other non-functional faults.

5.3.2 RQ3.2: How convincing is the evidence of cost and effectiveness of search-based software testing techniques, based on empirical studies that report credible results?

Along with accounting for random variation in the results and having a non-SBST or local MHS comparison baseline, studies must also report proper descriptive statistics or statistical hypothesis testing results in order to present credible and interpretable evidence. After the application of these criteria, there were just eight papers left and the results of these papers, referred to as 'Sufficient Criteria Papers', are summarized in Table 10.

Based on the information presented in Table 10, it is apparent that there is a scarcity of convincing evidence regarding the cost-effectiveness of SBST techniques. Nevertheless, these papers are a representative sample from the different types of investigations that are performed with MHS algorithms for test case generation. MHS algorithms have been recently applied to increasingly diverse types of problems and this is seen in this sample of papers by comparing the content of the "test purpose" column across papers. This ranges from specialized purposes such as testing the performance of real time systems to more general purposes such as testing non-public methods in object-oriented programs. Despite the diversity of objectives, we can see that in most of these papers, MHS algorithms, mostly GA, were compared with random search and the results show that GA outperformed random search for the test case generation problems at hand. This suggests that this type of problems indeed requires guided search algorithms. It would also be interesting to see how the quality of the empirical studies that have been performed in this field have improved over the years. In order to investigate this, we compare three series as shown in Figure 4.

The 'All papers' series shows the number of papers per year expressed as a percentage of the total number of papers (64 papers). The 'Minimum Criteria papers' series shows the percentage per year of the papers satisfying our minimum criterion of accounting for

Table 9 Test levels, fault types, and the type of metaheuristic algorithms used by 'minimum criteria papers'

	Test Level			Fault Type		Type of Metaheuristic Search Algorithm						
Paper	**Unit**	**Integration**	**System**	**Non-Functional**	**Functional**	**GA**	**EGA**	**SA**	**ESA**	**ACO**	**GP**	**PSO**
Jones *et. al.* [28]	√	–	–	–	–	√	–	–	–	–	–	–
Puschnerand Nossal [43]	√	–	–	–	–	√	–	–	–	–	–	–
Tracey *et. al.* [47]	√	–	–	–	–	-	–	√	–	–	–	–
Bueno and Jino [10]	√	–	–	–	–	√	–	–	–	–	–	–
Michael *et. al.* [38]	√	–	–	–	–		√	–	–	–	–	–
Wegener *et. al.* [51]	√	–	–	–	–	√	–	–	–	–	–	–
Shiba *et. al.* [45]	–	–	√	–	–	√	–	–	–	√	–	–
Briand *et. al.* [8, 9]	–	–	√	√	–	√	–	–	–	–	–	–
Miller *et. al.* [39]	√	–	–	–	–	√	–	–	–	–	–	–
Watkins and. Hufnagel [50]	√	–	–	–	–	√	–		–	–	–	–
Zhan and Clark [54]	–	–	√	–	√	–	–	√	–	–	–	–
Zhan and Clark [55]	–	–	√	–	–	–	–	√	√	–	–	–
Bueno *et. al.* [11]	–	–	√	–	–	–	–	–	–	–	–	√
Harman *et. al.* [33]	√	–	–	–	–	–	√	–	–	–	–	–
Harman and McMinn [24]	√	–	–	–	–	√	–	–	–	–	–	–
Harman *et. al.* [22]	√	–	–	–	–	√	–	–	–	–	–	–
Wappler and Schieferdecker [49]	√	–	–	–	–	√	–	–	–	–	–	–
Xiao *et. al.* [53]	√	–	–	–	–	√	–	√	√	–	–	–

random variation (as reported in Table 9 and the ‘Sufficient Criteria papers’ series shows the percentage per year of papers satisfying our secondary criteria of having an appropriate baseline and proper descriptive statistics or results of statistical hypothesis testing (as reported in Table 10). From Figure 4 we can see that 40% of all papers, 55% of all minimum criteria papers and 88% of all sufficient criteria papers were published in recent years (2006 and 2007). The trends that become apparent are that firstly, the number of SBST publications has been steadily growing over the years, and secondly, that the quality of empirical studies has increased dramatically in recent years.

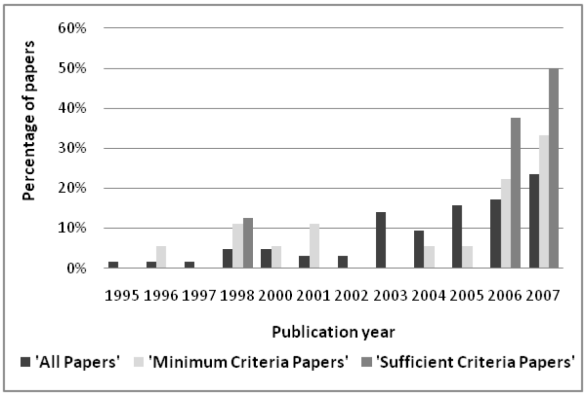


Figure 4 Quality trends of SBST empirical studies based on the publication year

5.3.3 RQ3.3: Is there any evidence regarding the scalability of metaheuristic search algorithms for test case generation?

During our systematic review, we did not find any paper specifically targeting the scalability of the MHS algorithm in the context of SBST. However, there was one paper where the authors performed a small scale scalability analysis [53]. The study was conducted on five small test objects written in C/C++. There were 36 to 87 test requirements to achieve full condition-decision coverage for all test objects and the size of the search space ranged from 26 to 232. The study was performed using different algorithms including GA, SA, Genetic Simulating Annealing (GSA), SA with Advanced Adaptive Neighbors (SA/AAN), and random search. In two of the SUTs used for the study, two different search spaces (one small and one large) were used to measure the performance (condition-decision coverage vs. the number of SUT iterations) of different MHS algorithms and random search. Based on the empirical evaluation, it was concluded

Table 10 Test purposes, comparison baselines, and result highlights for the 'sufficient criteria papers'

Paper	Test purpose	Comparison baseline	Result highlights
Puschner and Nossal, 1998	Creating an input data set with the worst-case program execution time	RS BEDG StA	In most cases, GA performed equal to or better than RS in terms of effectiveness measured as execution time of the SUT. For smaller size SUTs, GA had results as good as BEDG and StA
Briand *et. al.*, 2005 and 2006	Stressing a real-time system by creating input sequences that maximize delays in the execution of target tasks and increase chances of missing deadlines.	ScA	The technique can schedule tasks to miss the deadline(s) even though schedulability analysis identified them as schedulable. The GA is successful in bringing task completion times closer to their deadlines, thus leading to stressing the system in that respect.
Miller *et. al.*, 2006	Test case generation using genetic algorithms and program dependence graphs.	RS, GA	1) The results showed that, for simple programs there is little difference in the results (branch coverage) between RS and their proposed GA approach (TDGen). 2) The difference is seen in larger programs, where a much smaller number of generations are required to achieve 100% branch coverage. 3) It is also observed that for some SUTs, TDGen can achieve 100% branch coverage, where RS and GADGET cannot.
Watkins *et. al.*, 2005	Comparison of different fitness functions for path coverage	RS	Based on the study, it was concluded that there is no single fitness function that works well in all cases. A two-step method using two best fitness functions is therefore suggested in the paper.
Harman and McMinn, 2007	Test data generation to answer three research questions formulated based on royal road theory (see [24]) for GA	RS, HC	1) GA was able to find inputs to exercise the branches that have royal road features and HC and RT were not successful at all. 2) GA was unable to find the inputs to exercise the branches that have royal road features if crossover operators were removed. 3) HC performed better or no worse than GA for the branches that do not have royal road features.
Harman *et. al.*, 2007	Investigation of the relationship between the size of the search space (consisting of test inputs) and the performance of search algorithms measured as the number of fitness evaluations to cover a branch	RS, HC	1) There is no relationship between search space reduction and reduction in cost for random search. 2) There is significant improvement in cost reduction for both hill climbing and the genetic algorithm. 3) The reduction in cost is more for the genetic algorithm than for hill climbing. 4) There is no relationship between search space reduction and search effectiveness in terms of coverage for any of the search algorithms.
Wappler and Schieferdecker, 2007	An approach for testing non-public methods without breaking the encapsulation of the class, using an objective function specifically designed to cover non-public methods via public methods.	RS, GP	The new GP technique achieved higher overall branch coverage than RS and higher coverage of non-public methods than their existing GP based approach.
Xiao *et. al.*, 2007	Empirical evaluation of different MHS algorithms and RS for test data generation.	GA, SA, two extensions of SA (SA/AAN, GSA), RS	GA performed better than all other algorithms including random search. After GA, SA/AAN performed better in terms of both cost (number of SUT executions) and effectiveness (condition decision coverage).

HC: Hill Climbing, RS: Random Search, GA: Genetic Algorithm, SA: Simulated Annealing, GP: Genetic Programming, SA/AAN: SA with Advanced Adaptive Neighbors, GSA: Genetic SA, ScA: Schedulability Analysis, BEDG: Best Effort Data Generation, StA: Static Analysis that GA performed well for both the small and the large search space. SA/ANN was the second best. SA and GSA performed well only for the small search space. All MHS algorithms performed better than random search. As a result, we can say that scalability analyses of SBST techniques in the domain of test case generation are very rare and there is a need to focus more on scalability analysis in future studies.

5.3.4 Conclusion

Based on the discussions in the three sub-questions above, the number of papers which contain well-designed and reported empirical studies in the domain of test case generation using SBST is very small. As a result, there is a limited body of credible evidence that demonstrates the usefulness of SBST techniques for test case generation. This evidence is, in addition, very partial as it mostly focuses on the use of genetic algorithms at the unit testing level. This evidence, however, consistently shows that the genetic algorithms outperform random search in terms of structural coverage. However, this evidence is just based on eight papers and cannot be generalized to state that genetic algorithms at the unit testing level will always outperform random search regardless of the test objectives. More empirical studies must be conducted to provide strong and generalizable evidence about the suitability and applicability of different MHS algorithms for test case generation at different testing levels and for test objectives other than structural coverage.

6 Threats to the Validity of this Review

The main validity threats to our review are related to the possible incomplete selection of publications, inaccuracy of data extraction, and bias in quality assessment of studies.

6.1 Incomplete selection of publications

In Section 4.2, we have discussed and justified the systematic and unbiased selection strategy of publications. However, it is still possible to miss some relevant literature. One such instance is the existence of grey literature such as technical reports and PhD theses. In our case, this literature can be important if the authors report the complete study which is briefly reported in the corresponding published paper. In this review, we did not include such information.

Another instance that may lead to an incomplete selection of publications is the difficulty of finding an appropriate search string. In Section 4.2 we provide justification for the repositories that we selected and the search string that we used. However, there may still be some papers, which have used some other related terms other than our keywords. We refined our search string several times because we found a paper missing from our selected papers, which was in the reference list of another paper. In order to deal with this problem, we refined our search string until it included all such papers and we were sure that our set of selected papers did not miss any paper that is referred to and relevant for this review.

6.2 Inaccuracy in data extraction

Inaccurate data can be the result of subjective and unsystematic data extraction or invalid classification of data items. In our review, we tried to deal with this problem by two means. First, we defined a framework, which clearly identified the data items that should be extracted. Second, all the data extracted was reviewed by three researchers and all discrepancies were settled by discussion to make sure that the extraction was as objective as possible. Therefore, the remaining problem is the validity of the framework itself. We have defined the framework based on the current guidelines for empirical studies in software engineering and adapted them to our domain of interest based on experience. Hence, we believe that it is a good starting point, but it can be further improved by feedback and discussion from other researchers in the domain.

6.3 Unbiased quality assessment

Assessing the quality of the papers for answering RQ3 was a challenging issue. Even though the data extracted from the papers to judge their quality was detailed and based on a well thought framework, the criteria used to select the papers themselves could be thought of as subjective. Our justification for the validity of this criterion is discussed in the Section 5.3 and we re-emphasize the fact that this is the minimum requirement for having a valid empirical study in the domain of SBST.

7 Conclusion

The automation of test case generation has been a long-standing problem in software engineering. Search-based software testing, or in other words the application of metaheuristic search (MHS) algorithms for test case generation, has shown to be a very promising approach for solving this problem by re-expressing test case generation problems as search problems. As a result, a great deal of research has been conducted and published. The time was therefore ripe to perform a systematic review of the state of the art and appraise the evidence regarding the cost-effectiveness of such an approach. A systematic review is very different from more informal, traditional surveys, in the sense that it aims at being comprehensive in its coverage and repeatability by relying on well-defined paper selection and analysis procedures. This systematic review focuses, due to space constraints, on one specific but crucial aspect: the way SBST techniques have been empirically assessed. This aspect is highly important as all MHS algorithms are heuristics and therefore cannot guarantee their success in solving a test case generation problem or any other problem for that matter. Only an empirical investigation can provide the necessary confidence that a specific MHS algorithm is appropriate for a given test case generation problem.

In addition to a large-scale, systematic review, our contribution also includes guidelines, in the form of a framework, on how to conduct empirical studies in search-based software testing. Results of our review have shown that the research reported so far has mostly focused on structural coverage and unit testing. However, the research is increasingly more diversified in the types of topics being tackled. Results also show that empirical studies in this field would benefit from more standardized and rigorous ways to perform and report studies. More specifically, three important empirical issues stand out from our analysis. Studies need to, more systematically and rigorously, account for the random variation in the results generated by any MHS algorithm. Such random variation implies that alternative techniques can only be compared by statistical means, that is, statistical hypothesis testing. This, unfortunately, is not performed well in most published papers and our framework provides guidelines about which statistical test to perform in which circumstance. Last, another important issue is that it is impossible to assess how a MHS technique performs in absolute terms: to be able to conclude on its usefulness to tackle a specific test case generation problem, a proposed technique needs to be compared with simpler and existing alternatives to determine whether it brings any advantage. This is again missing in an important number of papers and needs to be carefully addressed by all studies in the future.

Despite the above limitations, credible results are available and existing results confirm that MHS algorithms are indeed promising for solving a wide variety of test case generation problems. Future research work will have to better establish their limitations and the types of problems for which they are applicable and required.

Acknowledgment

The authors wish to thank Simula School of Research and Innovation (SSRI) for funding this work.

References

[1] "Computer Science Conference Ranking," 2008, http://www.cs-conference-ranking.org/conferencerankings/topicsii.html.
[2] "Genetic Algorithms Framework," Rubicite Interactive, 2004, http://sourceforge.net/projects/ga-fwork.
[3] "UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)," Object Management Group (OMG), 2008, http://www.omg.org/cgi-bin/doc?ptc/2008-06-08.
[4] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology,* vol. 51, pp. 957-976, 2009.
[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Evolutionary Testing," in *Technical Report Simula.SE.293*: Simula Research Laboratory, 2008.
[6] B. Beizer, *Software testing techniques* Van Nostrand Reinhold Co., 1990.
[7] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *2007 Future of Software Engineering*: IEEE Computer Society, 2007.
[8] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '05)* Washington DC, USA: ACM, 2005.
[9] L. C. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines,* vol. 7, pp. 145-170, 2006.
[10] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in *Proceedings of the fifteenth IEEE international conference on Automated Software Engineering (ASE '00)* 2000, pp. 209-218.
[11] P. M. S. Bueno, W. E. Wong, and M. Jino, "Improving random test sets using the diversity oriented test data generation," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)* Atlanta, Georgia: ACM, 2007.
[12] E. K. Burke and G. Kendall, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*: Springer 2006.
[13] K. Y. Cai and D. Card, "An analysis of research topics in software engineering - 2006," *Journal of Systems and Software,* vol. 81, p. 8, 2008.
[14] J. A. Capon, *Elementary Statistics for the Social Sciences*: Wadsworth Publishing Co Inc, 1988.
[15] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Software* vol. 150, pp. 161-175, 2003.
[16] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*: World Scientific Publishing Company, 1997.

[17] R. Drechsler and N. Drechsler, *Evolutionary Algorithms for Embedded System Design*: Kluwer Academic Publishers, 2002.

[18] T. Dyba, V. B. Kampenes, and D. I. K. Sjoberg, "A systematic review of statistical power in software engineering experiments," *Information and Software Technology,* vol. 48, pp. 745-755, 2006.

[19] T. Dyba, B. A. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners " *IEEE software,* vol. 22, p. 8, January/February 2005 2005.

[20] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *The 2003 Congress on Evolutionary Computation (CEC '03)* 2003, pp. 1420-1424.

[21] M. Harman, "The Current State and Future of Search Based Software Engineering," in *2007 Future of Software Engineering*: IEEE Computer Society, 2007.

[22] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* Dubrovnik, Croatia: ACM, 2007.

[23] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.

[24] M. Harman and P. McMinn, "A theoretical empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)* London, United Kingdom: ACM, 2007.

[25] C. Hart, *Doing a Literature Review: Releasing the Social Science Research Imagination*: Sage Publications Ltd, 1999.

[26] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1997.

[27] D. Johnson, "A theoretician's guide to the experimental analysis of algorithms," in *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, 2002, pp. 215-250.

[28] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal,* vol. 11, pp. 299-306, 1996.

[29] K. S. Khan, R. Kunz, J. Kleijnen, and G. Antes, *Systematic Reviews to Support Evidence-Based Medicine: How to Review and Apply Findings of Healthcare Research*: Royal Society of Medicine Press Ltd, 2003.

[30] B. A. Kitchenham, "Guidelines for performing Systematic Literature Reviews in Software Engineering," 2007.

[31] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*: IEEE Computer Society, 2004.

[32] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering,* vol. 28, p. 14, August 2002.

[33] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '07)* London, England: ACM, 2007.

[34] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, and A. Chircop, "A Java-based Evolutionary

Computation Research System," George Mason University's ECLab Evolutionary Computation Laboratory, 2007, http://www.cs.gmu.edu/~eclab/projects/ecj/.

[35] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing,* vol. 5, pp. 315-331, 2005.

[36] A. P. Mathur, *Foundations of Software Testing*: Pearson Education, 2008.

[37] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability,* vol. 14, p. 52, 2004.

[38] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering,* vol. 27, pp. 1085-1110, 2001.

[39] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology,* vol. 48, pp. 586-605, 2006.

[40] D. S. Moore and G. P. McCabe, *Introduction to the Practice of Statistics*, Fourth ed.: W. H. Freeman, 2002.

[41] I. H. Osman and J. P. Kelly, *Metaheuristics: Theory and Applications*: Kluwer Academic Publishers, 1996.

[42] H. Pohlheim, "GEATbx - The Genetic and Evolutionary Algorithm Toolbox for Matlab," 2007.

[43] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998, pp. 134-143.

[44] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*, Third ed.: Chapman & Hall/CRC, 2003.

[45] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)* 2004, pp. 72-77.

[46] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer,* vol. 27, pp. 17-26, 1994.

[47] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)* Clearwater Beach, Florida, United States: ACM, 1998.

[48] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93)* Cambridge, Massachusetts, United States: ACM, 1993.

[49] S. Wappler and I. Schieferdecker, "Improving evolutionary class testing in the presence of non-public methods," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering (ASE '07)* Atlanta, Georgia, USA: ACM, 2007.

[50] A. Watkins and E. M. Hufnagel, "Evolutionary test data generation: a comparison of fitness functions," *Software: Practice and Experience,* vol. 36, pp. 95-116, 2006.

[51] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology,* vol. 43, pp. 841-854, 2001.

[52] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*: Kluwer Academic Publishers, 2000.

[53] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering,* vol. 12, pp. 183-239, 2007.

[54] Y. Zhan and J. A. Clark, "Search-based mutation testing for Simulink models," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '05)* Washington DC, USA: ACM, 2005.

[55] Y. Zhan and J. A. Clark, "The state problem for test generation in Simulink," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '06)* Seattle, Washington, USA: ACM, 2006.

Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems

Shaukat Ali, Lionel C. Briand, Hadi Hemmati

Published in the Systems and Software Modeling Journal (SOSYM), Springer, 2011

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

1. Introduction

Modeling software functional behavior has always been an important focus of the modeling community to support many development activities such as model-based testing (MBT) and automated code generation. Regarding model-based testing, which is the specific focus on this paper, much less attention has been given to modeling non-functional behavior such that the testing of non-functional properties (e.g., safety and robustness) can be automated. Though several UML profiles have been proposed to address the modeling of non-functional properties (including the UML profile for QoS and Fault Tolerance [5], the MARTE profile [7], and UMLSec [8]), it is not yet clear whether they can fully support test automation.

Our motivation here is to support model-based robustness testing. An IEEE Standard [10] defines robustness as *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions"*. A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, using our industrial case study as an example, modeling such robustness behavior of a videoconferencing system (VCS) is to model its behavior in the presence of hostile environment conditions (regarding the network and other communicating VCSs), such as high percentage of packet loss and high percentage of corrupt packets. The VCS should not crash, halt, or restart in the presence of, for instance, a high percentage of packet loss. Furthermore, the VCS should continue to work in a degraded mode, such as continuing the videoconference with low audio and video quality. In the worst case, the VCS should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS and must be tested systematically and automatically to be scalable.

To automate such systematic testing, one can model the system robustness behavior to such events and resort to model-based testing (MBT). However, robustness behavior is typically crosscutting many parts of the system functional model and, as a result, modeling such behavior directly within the functional models is not practical since it leads to many redundancies and hence results in large, cluttered models. To cope with this issue, we decided to adopt Aspect-Oriented Modeling (AOM) [11], which provides Separation of Concerns (SoC) during design modeling. Crosscutting concerns are modeled as aspect models and are woven into a primary model (base model), modeling non-crosscutting concerns. AOM can potentially offer several benefits such as: 1) enhanced modularization, 2) easier evolution of models, 3) increased reusability, 4) reduced modeling effort, and 5) improved readability [11, 12].

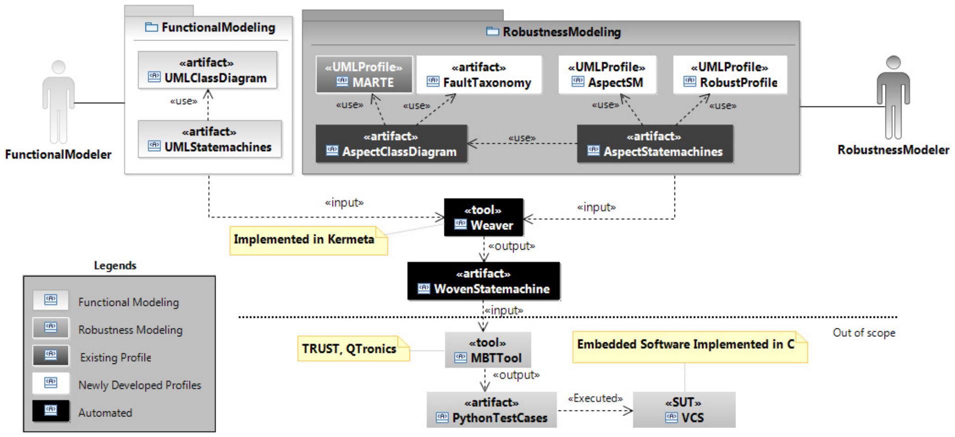


Figure 1 . An overview of RUMM

Our goal in this paper is to provide a complete solution in terms of both aspect and state machine features necessary for model-based robustness testing. Furthermore, we want to minimize the effort involved in learning a new language over standard UML and enable automated, model-based testing. To achieve this, we present a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM and assess it on an industrial case study involving a commercial videoconferencing system. Such studies are very few in the research literature and are rarely run and reported in a satisfactory manner [13]. To the knowledge of the authors, only a few industrial applications of AOM have been reported to date [14-17] and had very different objectives than RUMM. An overview of RUMM is shown in Figure 1. The core of RUMM is the definition of a UML state machine profile for AOM: AspectSM (shown as a white artifact in Figure 1 in *RobustnessModeling*). We limited our profile to UML state machines as: 1) They are the main notation currently used for model-based test case generation [18] and are particularly useful in control and communication systems, 2) Like it is often the case, our industrial case study exhibits state-based behavior so that it is natural to initially provide support for UML state machines. The profile can, however, be extended to other UML diagrams in the future, following similar principles. We rely on developing a profile instead of developing a domain specific language since, in our case study context as in many others, minimizing extensions to UML is expected to ease practical adoption. More thorough discussions on this issue are presented in Section 7. Modelers of functional aspects of the system can be different from the ones specifying its robustness behavior. The latter make use of AspectSM to model aspect state machines.

Another important part of the RUMM is another UML profile (RobustProfile) shown as a white artifact in Figure 1, based on the fault taxonomy defined by [20] and the IEEE standard classification for anomalies [21]. The profile is used by a robustness modeler to develop aspect state machines and is defined specifically to assist in defining test strategies for robustness testing. In addition, the profile helps generating test scripts based on classes of faults modeled using the profile. Once again, the profile is defined on UML state machines, as they are the main focus of this paper. We follow the widely accepted and used definitions in [20] for faults and failures. A fault is an incorrect state of a system or its environment in the presence of which the system cannot provide a correct service. Such deviation from the correct service is called a failure. A fault type is identified based on a fault taxonomy (white artifact in Figure 1) and the UML profile MARTE is used to model it in a UML class diagram (Aspect Class Diagram, dark grey artifact in Figure 1). In a subsequent step, aspect class diagrams are used to model actual faulty behavior as aspect state machines (AspectStatemachines) using both AspectSM and RobustProfile. Finally, robustness models comprising of aspect class diagrams and aspect state machines are woven into functional models once again composed of UML class diagrams and state machines. This is performed using our weaver implemented in Kermeta [22] and the woven state machines produced by the weaver can be used in turn by a model-based testing tool, for instance the TRUST tool [23] or QTronics [24], to generate executable test cases. In our case, test cases are generated in Python, which is used as a test script language by our industry partner (Cisco, Norway). Note that this paper addresses only robustness modeling and details on test case generation and execution are outside the scope of this paper.

The contributions of the paper can be summarized as follows: 1) A RobUstness Modeling Methodology (RUMM) that enables the systematic modeling of robustness behavior in a practical and scalable way, 2) A UML 2.0 profile (RobustProfile), which is based on a fault taxonomy in [20] and the IEEE standard classification for anomalies [21], to model faults, recovery mechanisms, and failure states, 3) The application of the MARTE profile in conjunction with RobustProfile to model faulty environment conditions, 4) A UML 2.0 profile (AspectSM) to support comprehensive aspect modeling for UML 2.0 state machines and enable automated robustness testing. AspectSM supports modeling crosscutting on all features of UML 2.0 state machines and supports all basic features of AOSD such as pointcuts, introduction, joinpoints, and advice; 5) An empirical evaluation and discussion of the benefits of modeling robustness behavior of an industrial system using RUMM and AspectSM; 6) Tool support, based on model transformations in Kermeta [22], to automatically weave AspectSM aspects into base state machines (modeling the core functional behavior of a system).

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

2. Case Study and Running Example

Our case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn [23]. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. In this paper, to demonstrate the applicability of RUMM, we focused on this particularly important subsystem (Saturn) and left out the other functionalities of the VCS. We selected this subsystem because robustness testing is concerned with testing the behavior of VCS in the presence of hostile environment situations, which can only be tested when the VCS is in a conference call with other systems, which is what Saturn manages. Saturn is complex enough to demonstrate the applicability and usefulness of RUMM while still remaining manageable in the context of a case study. To provide simple running examples in the next sections, we modeled a reduced version of Saturn where one can only establish calls and cannot start or stop presentations. From now onwards, we will refer to this simplified Saturn model as S-Saturn to differentiate it from the complete case study model used in Section 6 to discuss the benefits of RUMM.

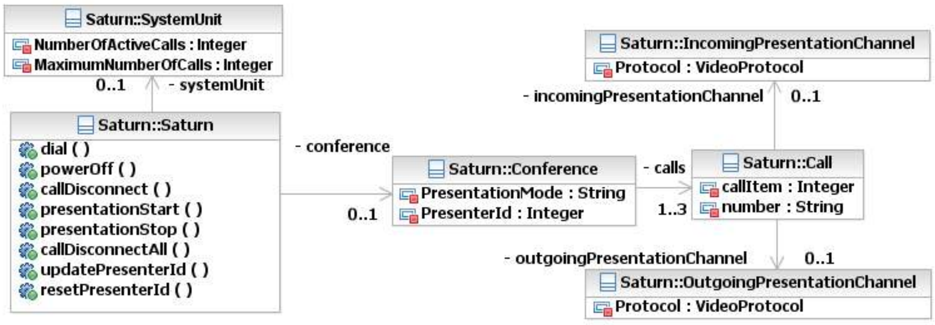
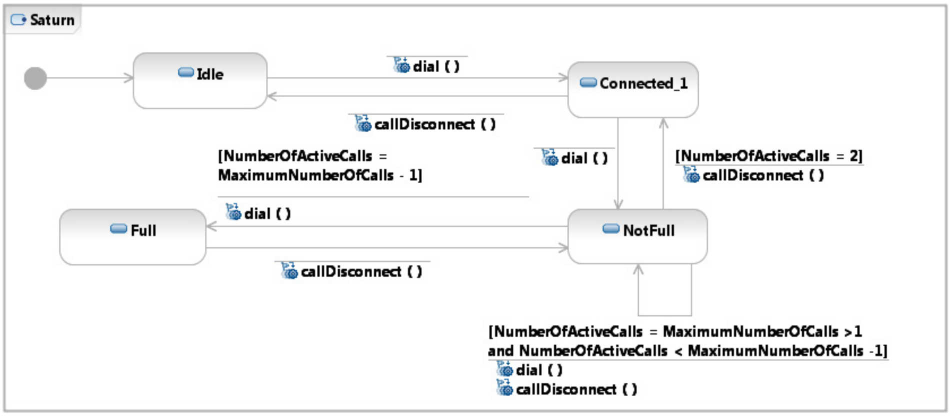


Figure 2. Conceptual model of the S-Saturn subsystem

2.1 Functional models of S-Saturn

The functional model of S-Saturn consists of a class diagram and a state machine. The class diagram of S-Saturn is shown in Figure 2 and is meant to capture information about APIs and system (state) variables, which are required to generate executable test cases and oracles in our application context. Saturn's API is modeled as a set of methods in the *Saturn* class such as *dial()* and *callDisconnect()*. In our case, the parameters of these methods are either modeled as primitive data types (e.g., *String*) or as Enumeration types (e.g., *CallProtocol*).The state variables of the system are modeled as instance variables of classes in the conceptual model. For example, two system variables in the *SystemUnit* class are *NumberOfActiveCalls* and *MaximumNumberOfCalls*. *NumberOfActiveCalls* is an *Integer* which determines the number of VCS that are currently in a Saturn videoconference, whereas *MaximumNumberOfCalls* determines the maximum number of simultaneous calls supported by Saturn.

The state machine modeling the nominal functionality of S-Saturn, referred to as a base state machine, is shown in Figure 3. It consists of four simple states. From the *Idle* state, invoking the *dial()* method of the *Saturn* class leads to the *Connected_1* state, which represents the behavior of the system when there is a conference without any presentation



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

Figure 3 . Base state machine for the Saturn subsystem

with one endpoint. As long as there exists one endpoint in the conference and no presentation is transmitting, S-Saturn stays in the *Connected_1* state and when S-Saturn dials to more endpoints, it transitions to the *NotFull* state until it connects to the maximum number of endpoints it supports and transitions to the *Full* state. Each simple state has an associated state invariant based on the system variables modeled in the conceptual model. For instance, the *Idle* state has the following state invariant:

self.systemUnit.NumberOfActiveCalls = 0 and self.conference.PresentationMode = 'off'

2.2 Robustness behavior

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

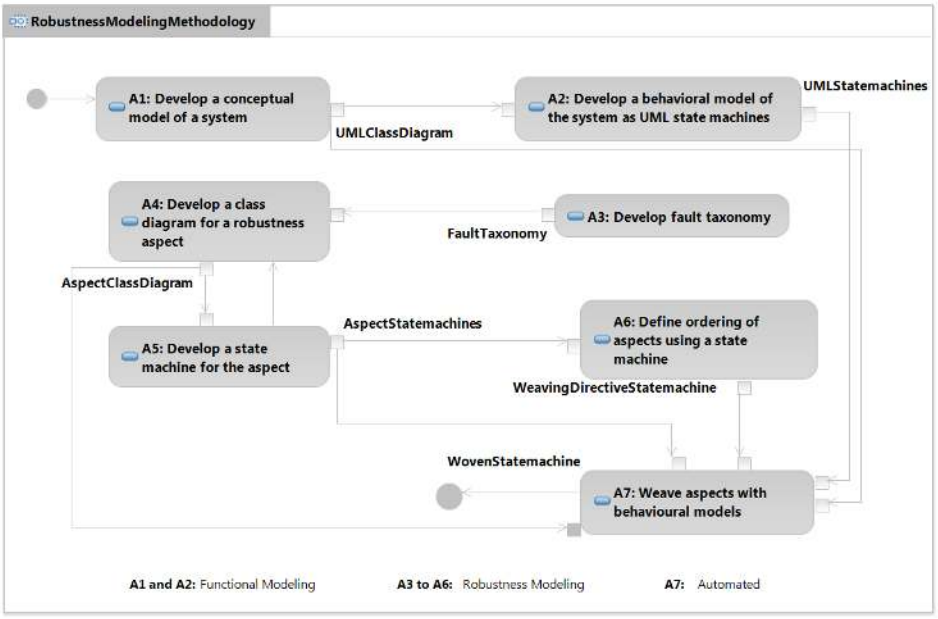

RobustnessModelingMethodology
A1: Develop a conceptual model of a system
UMLClassDiagram
A2: Develop a behavioral model of the system as UML state machines
UMLStatemachines
A4: Develop a class diagram for a robustness aspect
FaultTaxonomy
A3: Develop fault taxonomy
AspectClassDiagram
AspectStatemachines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

Figure 4. Methodology for robustness modeling (RUMM)

3. Robustness Modeling Methodology

Our goal is to devise a solution to model robustness behavior, which (1) is complete in terms of aspect and state machine features, (2) minimizes the learning curve over standard modeling skills, and (3) enable automated, model-based testing. To achieve this, we defined a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM. Recall from Section 1 that we follow the standard definition of robustness provided in the IEEE 610.12 standard [10]. Such robustness is considered very critical in many standards such as in the *IEEE Standard Dictionary of Measures of the Software Aspects of Dependability[10]*, the ISO's *Software Quality Characteristics* standard [25], and the *Software Assurance Standard* [26] by NASA. The RUMM methodology (Figure 4) is suitable for systems, which implement substantial robustness behavior to deal with faulty situations in the environment such as communication and control systems. A1 and A2 activities are related to functional modeling, whereas activities A3 to A6 are related to modeling robustness behavior. Activity A7 is automated and merges functional (base state machines) and robustness (aspects) models together into a complete model. Activities A1 to A6 are related to modeling functional and robustness behavior and are manual. In this section, we will explain very briefly each activity. Additional, detailed information will be provided in the next sections, followed by the application of RUMM in an industrial case study.

The first activity (A1) involves developing a conceptual model [27] of a SUT using a UML 2.0 class diagram based on the domain analysis of the SUT. In this activity, we model different domain concepts of the SUT as classes and relationships between them, which are determined as the result of domain analysis. In addition, we model state variables of the SUT as attributes in the class diagram. We also model public operations of the SUT (API) and external events in the SUT environment as signal receptions. The conceptual model is then used in activity A2 for developing a behavioral model of the SUT as one or more UML state machines. Attributes defined in the conceptual model are used for various purposes such as defining state invariants and defining guards on transitions. The operations and signal receptions defined in the conceptual model are used as triggers on transitions of state machines. In model-based robustness testing, one of the most important tasks is the identification and modeling of faults, in the presence of which we must test the behavior of the SUT. To systematically identify these faults, the development of fault taxonomy is required (A3) and is provided in Section 4.1. The application of the fault taxonomy to an industrial system is reported in Section 5.3. Activity A4 requires modeling different properties of the system's environment, whose violations lead to the various types of faults identified from the fault taxonomy (A3). The guidelines for this process are defined in Section 5.4. Activity A5 requires modeling robustness behavior as aspect state machines. As described in Section 4.4, this requires the use of the AspectSM profile. The profile definition is provided in Section 4.2. The control flow arrow from activity A5 to activity A4 depicts that multiple robustness aspects can be modeled one after another. Once all robustness aspects have been modeled, we may need to define the order in which the aspects should be woven into the base state machine developed in activity A2. Guidelines for modeling the ordering of aspect state machines as a weaving-directive state machine are presented in Section 4.6. Finally, activity A7 weaves aspect state machines with base state machines. For this activity, we developed a tool using Kermeta [22], a well-known model transformation environment. The details of the tool are presented in Section 4.7 and the weaving algorithm is detailed in Appendix B.

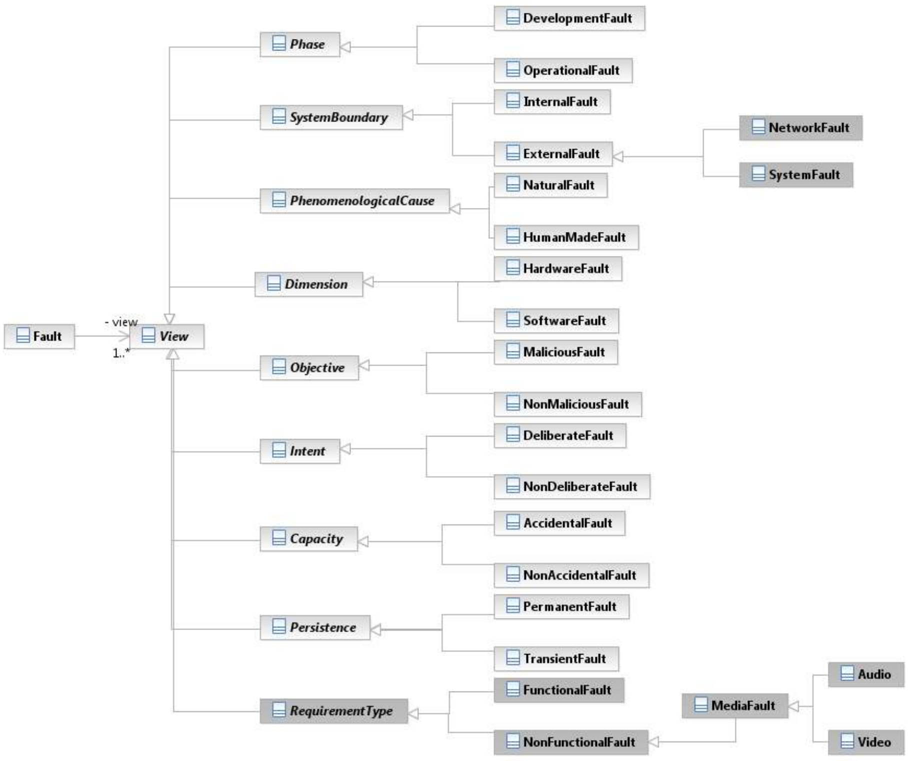


Figure 5. High level fault taxonomy

4. Concepts, Techniques, and Tools Required for RUMM

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

4.1 Definitions

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

4.1.1 *Faults and Failures in the context of UML state machines*

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

4.1.2 Fault classification based on taxonomy

Many fault taxonomies are proposed in the literature, however most of them are either specific to architectures, for instance Service-oriented Architecture (SOA) [28, 29] and Component-based Systems [30], or to application domains such as aeronautics and space [31]. We chose the widely-known and referenced fault taxonomy presented in [20] because it is very comprehensive and generic, and thus can be extended for specific needs as it was required in our case. For instance, we extended the taxonomy to accommodate for media quality faults, which are very important for a commercial VCS. The fault taxonomy for elementary fault classes provided in [20] is modeled in Figure 5 as a class diagram. Dark gray colored classes in Figure 5 show the fault classes we extended for our specific needs. The taxonomy states that a fault can be categorized based on different views/perspectives such as based on *SystemBoundary* or *Dimension*. Using *SystemBoundary* faults can be classified into either *InternalFault* or *ExternalFault* depending on where they occur. Details on classes of faults are provided in [20]. Given our goal, we extended some fault classes in the fault taxonomy to model faults which are specific to the VCS. For instance, to provide a support for modeling media-related faults, which are important for an industrial VCS, we introduced a view *RequirementType* (Figure 5) and defined two fault classes: *FunctionalFault* and *NonFunctionalFault*. We further classified

NonFunctionalFault into *MediaFault* (Figure 5), with further subclasses *Audio* and *Video*. In addition, we extended *ExternalFault*, which comprises faults in networks and external systems, into *NetworkFault* and *SystemFault* subclasses. *SystemFault* corresponds to the faults in one or more VCS communicating with the SUT. Since in robustness testing the focus is always on modeling behavior of a SUT in the presence of faults in its environment, all fault classes in the taxonomy are valid from the perspective of other VCSs communicating with the SUT. For instance, a *SoftwareFault* in a VCS communicating with the SUT can have an effect on the latter's behavior. We provide an example use of the taxonomy in Section 5.3 for our case study.

4.2 The AspectSM profile

Using the AspectSM profile, we model each aspect as a UML state machine with stereotypes (aspect state machine). The modeling of aspect state machines is systematically derived from a fault taxonomy (Figure 5) categorizing different types of faults (incorrect states [20]) in a system and its environment (such as communication medium and other systems). Such a modeling approach models each type of robustness behavior separately from the state machines modeling nominal functionality (base state machine) and hence results in enhanced separation of concerns. Furthermore, our modeling approach models crosscutting behaviors as separate aspect state machines and hence reduce modeling effort when compared to modeling robustness directly in combination with nominal behavior. The readability of models is then improved as robustness behavior that tends to be redundant when modeled directly is clearly separated out and expressed once. Following the general ideas proposed in [32] [19], to model aspects using the same notations as the base model, we used UML state machines to model both aspect and base models, which is expected to facilitate practical adoption. In industrial applications of model-based testing, it is always desirable to minimize the need to learn different notations to model different testing concerns (such as security and robustness concerns). Though profiles already exist in the literature that allow modeling aspects as UML state machines [1-3, 12, 33], we decided to define our own profile to address the three following problems:

1. Crosscutting behavior can exist on any modeling element in UML 2.0 state machines, but the existing profiles and approaches do not support all features, such as state invariants and guards [1, 12, 33, 34]. These are however crucial in the context of model-

based testing, and in particular for automated test case generation [35].

2. Existing modeling approaches using profiles require, for modeling aspect features (such as pointcut and advice), to develop new diagrams that are not part of the UML 2.0 standard [3, 4], thus making adoption in practical contexts more difficult. Indeed, such profiles require developing specific tool support for new diagrams and entails training users on how to build them. As a result, in practice, the use of non-standard modeling languages is discouraged.

3. Some of the existing approaches do not support *all* basic features of aspect orientation such as *Introduction*.

More details and discussions on related work are provided in Section 7.2

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

A UML profile enables the extension of UML for different domains and platforms, while avoiding any contradiction with UML semantics. In [36], two main approaches for profile creation are discussed. The first approach directly implements a profile by defining key concepts of a target domain, such as what was done to define SysML [37]. The second approach first creates a conceptual model outlining the key concepts of a target domain followed by creating a profile for the identified concepts. This latter approach has been used for defining profiles such as the UML profile for Schedulability, Performance, and Time specification (SPT) [38], the QoS and Fault Tolerance specifications [5], and the UML Testing Profile (UTP) [39].

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

4.2.1 Domain view of the profile

The conceptual domain model for AspectSM is shown in Figure 6 as a MOF-based [40] metamodel. The conceptual domain model defines aspect-oriented modeling concepts.

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

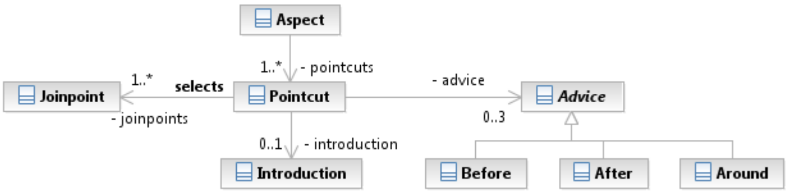


Figure 6. Conceptual domain model of the profile

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

Figure 7. Constraint on Pointcut

A *joinpoint* is a model element, which corresponds to a pointcut where an advice (additional behavior) can be applied [41]. All modeling elements in UML are possible joinpoints, where an advice can be applied [11]. For UML state machines, some examples of joinpoints include a state or a transition.

A *pointcut* selects one or more joinpoints with similar properties, where advices can be applied. A pointcut can have at most one before advice, one around advice or one after advice (Figure 6). All pointcuts are expressed with the OCL on the UML 2.0 metamodel. We decided to use the OCL to query joinpoints since it is the standard to write constraints on UML models and is also commonly used to query jointpoints (modeling elements such as states and transitions). Also, several OCL evaluators are currently available that can be used to evaluate OCL expressions such as the IBM OCL evaluator [42], OCLE 2.0 [43], and EyeOCL [44]. Furthermore, writing pointcuts as OCL expressions do not require a modeler to learn a notation that is not part of the UML standard. In the literature, several alternatives are proposed to write pointcuts [1-4, 12] but all of them either rely on languages (mostly based on wildcard characters to select joinpoints, for instance, '*' to select all joinpoints) or diagrammatic notations which are not standard, thus forcing modelers to learn and apply new notations or languages. Using the OCL, we can write precise pointcuts to select jointpoints with similar properties. We do so by selecting modeling elements (jointpoints) based on the properties of UML metaclasses. This further gives us the flexibility to specify pointcuts of varying complexities. For instance, we can specify a very complex pointcut based on all properties of a UML metaclass, e.g., a pointcut on the *Transition* metaclass, selecting a subset of transitions in a base state machine for which all properties of the *Transition* metaclass are the same. On the other hand, we can also specify a simple pointcut based on a small subset of properties of a UML metaclass. For example, a pointcut on the *Transition* metaclass selecting all those transitions from a base state machine, which have the same guards, though other properties such as triggers or effects can be different. In UML state machines, states and transitions are the most important modeling elements and all other elements are contained within them such as state invariants in states and guards and actions in transitions. Therefore, pointcuts are defined in the context of the UML metaclass *Vertex*, to query states and apply advices on states and its composing elements such as state invariants and do, entry, and exit activities. Similarly, pointcuts are also defined in the context of the UML metaclass *Transition* to query transitions and advices are applied on transitions and its containing elements such as *Guard* and *Actions*. The attributes for the *Vertex* and *Transition* metaclasses can be obtained from the UML specifications [45]. For example, a pointcut may select all transitions of a state machine which have triggers with signal events. This pointcut, defined in Figure 8, is written as an OCL expression on attributes of the UML metaclass *Transition* and selects all transitions that have triggers with signal events on them.

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

Figure 8. A pointcut in OCL selecting all transitions with

An advice is an additional behavior added at joinpoint(s) selected by a pointcut. This behavior can be added as OCL constraints or in the form of state machine modeling elements such as a guard or an effect. As most of the concepts in AOM are inspired from aspect-oriented programming (AOP) languages such as AspectJ [46], in a similar way in AOM, an advice can be of type before, after, or around. A before advice is applied before joinpoint(s), an after advice is applied after joinpoint(s), whereas an around advice replaces joinpoint(s). For example, introducing guards on all transitions of a state machine that have signal events as triggers is an example of a before advice on transitions. Table 1 summarizes the semantics of each type of advice for each UML 2.0 state machine modeling element. Examples for advice on all UML 2.0 state machine modeling elements are provided in [47].

Table 1. Definition of before, around, and after advice

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

An introduction is similar to the inter-type declaration concept in AspectJ [46] and is used in many AOM approaches [4, 48-50] to introduce new modeling elements in a base model. In a similar fashion, we use introduction in our context to introduce new modeling elements in a UML state machine, e.g., a new state or a transition. In our context, we mostly use introduction to introduce transitions in a base state machine, which correspond to faults in the environment (Section 4.1.1). We also use introduction to introduce new states in a base state machine, which are related to a robustness behavior such as the state of a system which is operating with degraded performance (Section 4.1.1).

Table 2. Extensions, generalizations, and associations of each stereotype

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

Table 3. Attributes defined for the <<*Pointcut*>> stereotype

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

Table 4. Attributes defined for the <<*Aspect*>> stereotype

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

Table 5. Attributes defined for the stereotypes related to advice

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

4.2.2 UML representation

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

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

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

Attributes associated with the <<*Aspect*>> stereotype are shown in Table 4. Attributes associated with the <<*Pointcut*>>, <<*Before*>>, <<*After*>>, and <<*Around*>> stereotypes are shown in Table 3 and Table 5. When applying these stereotypes, attributes must be supplied in accordance to the description in these tables. Examples are presented in [47].

4.2.3 Example of an application of AspectSM

We present next a small example of the application of AspectSM. On the

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

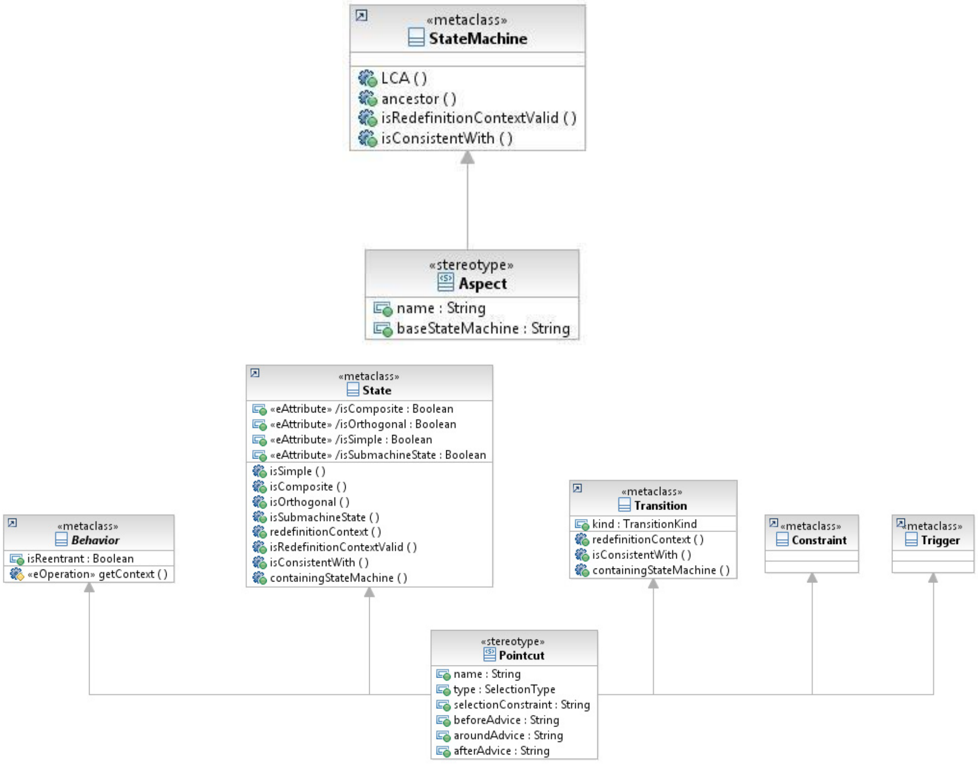


Figure 9. **<<*Aspect*>> stereotype applied to *StateMachine* metaclass (left) and <<*Pointcut*>> stereotype applied to various metaclasses (right)**

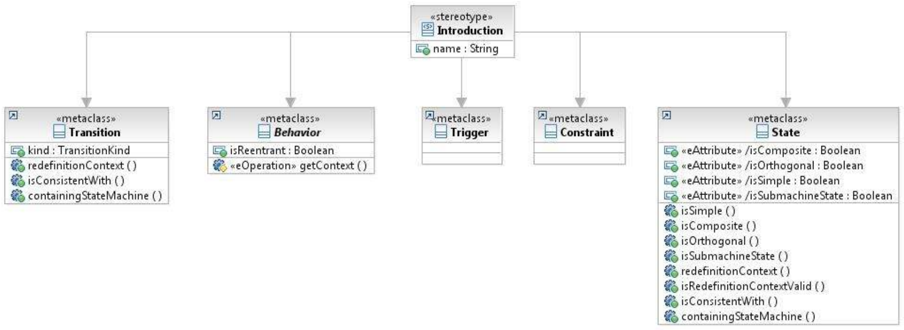


Figure 10. The <<*Introduction*>> stereotype applied to various metaclasses

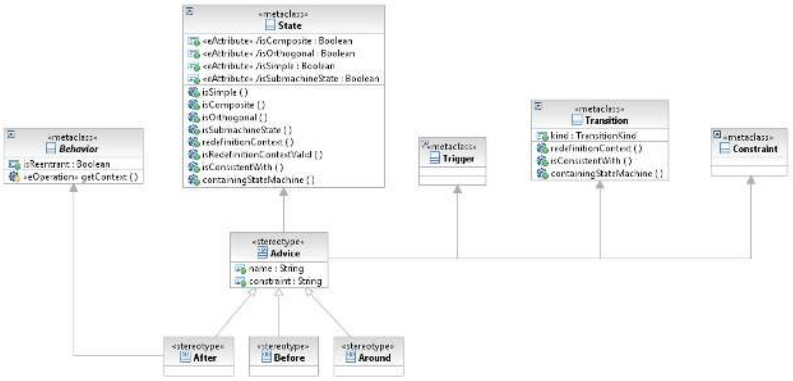


Figure 11. The <<*Advice*>> stereotype applied to various metaclasses

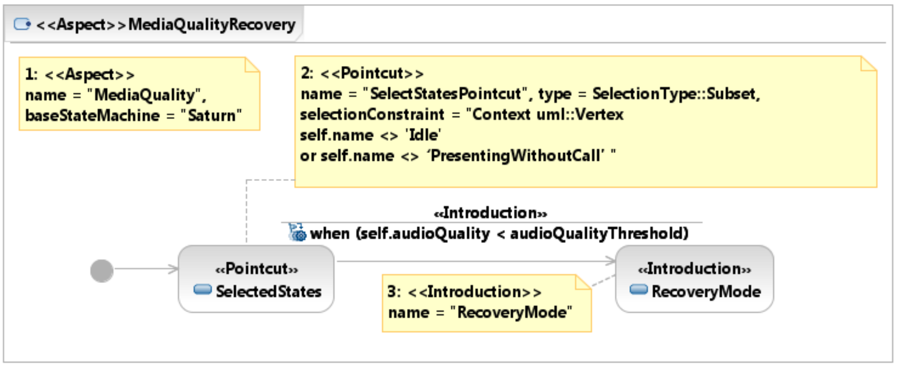

<<Aspect>>MediaQualityRecovery
1: <<Aspect>>
name = "MediaQuality",
baseStateMachine = "Saturn"
2: <<Pointcut>>
name = "SelectStatesPointcut", type = SelectionType::Subset,
selectionConstraint = "Context uml::Vertex
self.name <> 'Idle'
or self.name <> 'PresentingWithoutCall' "
«Introduction»
when (self.audioQuality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

Figure 12. An example for the application of AspectSM

4.3 RobustProfile

To help with the definition of robustness test strategies, we defined a UML profile RobustProfile to model faults and their properties. In addition, the profile supports the modeling of recovery mechanisms when a fault has occurred and the modeling of states a system can transition to when it has recovered. The profile has two sub-profiles: the first sub profile, FMProfile, deals with modeling faults and their attributes. The second sub-profile, FRProfile, deals with modeling recovery mechanisms and states of a system after recovery from a failure. Below, we provide details on the definition of these sub-profiles. We reused all the concepts presented in [20] and in addition added a few more concepts presented in Section 4.1.2. In addition, we reused all the concepts from the IEEE standard on the classification of software anomalies as defined in [21]. All these concepts from the IEEE standard were captured in a UML profile so that the standard can be used in combination with UML models. The newly introduced concepts are italicized in Table 7 and Table 6.

4.3.1 *Fault Modeling Profile (FMProfile)*

We used the same procedure to define FMProfile as that for AspectSM (Section 4.2). The domain view for FMProfile is the same as the fault taxonomy shown in Figure 5 [20]. Below, we provide a UML representation of FMProfile, which includes profile diagrams and details on stereotypes and their attributes.

Figure 13 shows a part of the profile diagram for FMProfile that is related to the abstract <<*Fault*>> stereotype class, which corresponds to the *Fault* class in Figure 5. We show different attributes of <<*Fault*>> and also show its extension relationships to UML metaclasses. Additional information about FMProfile is summarized in Table 7. The <<*Fault*>> stereotype is applied to the metaclasses *Transition*, *Trigger*, and *Event* because each fault in our case occurs when an event associated to trigger on a transition is fired (see Section 4.1). Furthermore, according to UML semantics [45], a transition can have multiple triggers, and each trigger can model different faults belonging to the same super class. For instance, a transition can model multiple external faults (*ExternalFault* in Figure 5) and one trigger on the transition can model one fault from *NetworkFault* while the other trigger can model one fault from *SystemFault*. This is the reason that the <<*Fault*>> stereotype class has an extension relationship with the *Trigger* metaclass. The attributes of <<*Fault*>> are obtained from the IEEE Standard in [21] where more details can be found on each attribute. Based on the values of these attributes, test strategies can be devised. For instance, the transitions that are stereotyped with <<*Fault*>> or any of its sub-stereotype classes with value *High* for the *severity* attribute, could be given priority over other transitions modeling faults with lower severity. In addition, complex test strategies can be defined to test the robustness of a SUT in the combined presence of faults that belong to different fault classes. For example, a test strategy can be devised that can test the behavior of a SUT in the presence of one media fault and one network fault at the same time. We also defined stereotypes for all other classes shown in the taxonomy and provide detailed information about these stereotypes in Table 7. All stereotypes inherit attributes from <<*Fault*>>.

This profile also assists in test script generation. For instance, different stereotypes can indicate for which entity (for instance, network or other systems) in the environment, test scripts are to be generated. For example, the <<*NetworkFault*>> stereotype indicates that test scripts will be generated for a network emulator and the test scripts will emulate a particular fault in the emulator. The <<*MediaFault*>> stereotype indicates that test scripts will be generated to introduce media faults in the VCS that is communicating with the SUT. It is important to distinguish between faults for different entities in the environment because different scripting languages are normally used to control these entities. In our case study, a proprietary scripting language is used for the SUT and other VCS communicating with it, whereas Python is used to control a proprietary network emulator used by our industry partner.

4.3.2 *Fault Recovery Profile (FRProfile)*

FRProfile deals with modeling recovery mechanisms associated with the occurrence of a fault. The domain view of FRProfile is shown in Figure 14. It consists of two main parts. The first part describes recovery mechanisms such as *Forward* and *Backward* [20]. The second part deals with the state of the system after a recovery mechanism is executed, which could be *Initial*, *Final*, *Failure*, or a *Degraded* state [20].

A part of the profile diagram for FRProfile is shown in Figure 15. Both recovery mechanisms and systems states refer to states in the SUT state machines and we therefore applied stereotypes <<*RecoveryMechanism*>> and <<*SystemState*>> on metaclass *Vertex*. In addition, we defined stereotypes for other classes shown in the domain view of the profile such as <<*Forward*>> and <<*Degraded*>>. These stereotypes inherit attributes from their corresponding super classes, e.g.,<<*Degraded*>> inherit attributes from <<*SystemState*>>. Details on stereotypes are shown in Table 6.

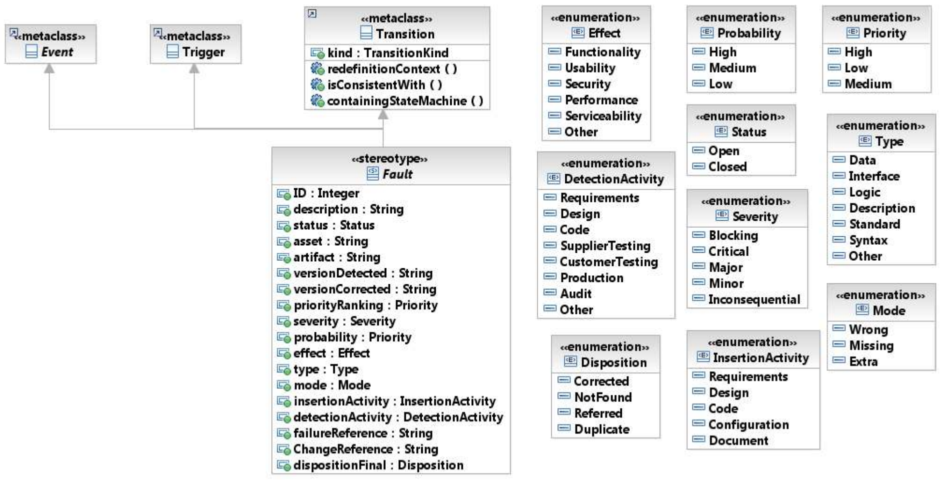


Figure 13. Profile diagram for FMProfile

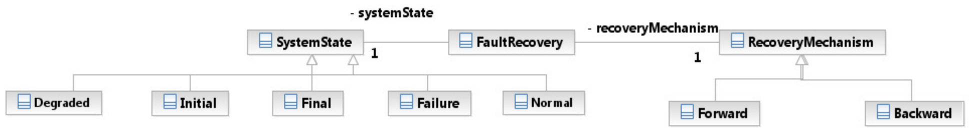


Figure 14. Domain view of FRProfile

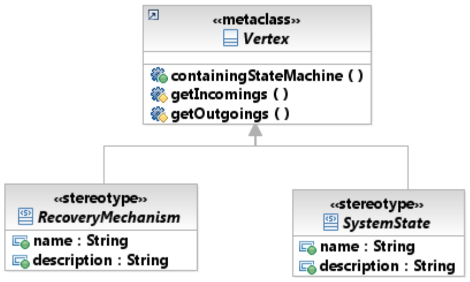


Figure 15. Profile diagram for FRProfile

4.3.3 Example of an Application of RobustProfile

This section provides a small example of the application of RobustProfile in Figure 16.

A change event *when (not self.audioQuality < audioQualityThreshold)* is fired from *SelectedStates* (stereotyped as <<*Normal*>> from RobustProfile indicating that it is a normal state) when the audio quality in a videoconference becomes lower than the allowed threshold of audio quality. This change event is stereotyped as <<*AudioFault*>> indicating that it is an audio fault (see the comment labeled C1) and its attribute values are provided in the note labeled as "1". For instance, the *effect* attribute has value *Effect::Performance* indicating that this fault affects the performance of the system. Recall that the *effect* attribute is defined based on the IEEE standard defined in [21]. The *RecoveryMode* state in Figure 16 is stereotyped as <<*Degraded*>> from RobustProfile indicating that in this state the system in functioning with degraded performance.

Table 6. Extensions and generalizations of each stereotype for FRProfile

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

Table 7.Extensions and generalizations of each stereotype for FMProfile

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

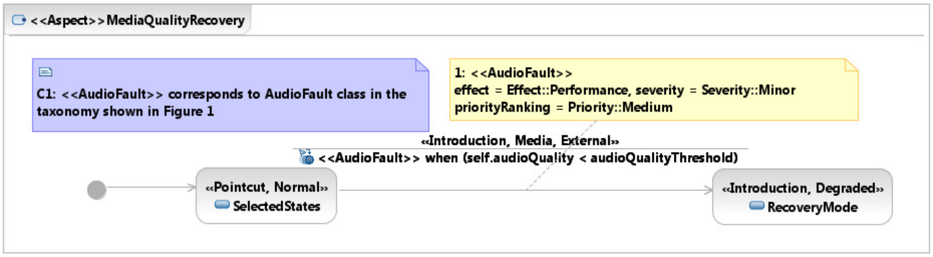

<<Aspect>>MediaQualityRecovery
C1: <<AudioFault>> corresponds to AudioFault class in the taxonomy shown in Figure 1
1: <<AudioFault>>
effect = Effect::Performance, severity = Severity::Minor
priorityRanking = Priority::Medium
«Introduction, Media, External»
<<AudioFault>> when (self.audioQuality < audioQualityThreshold)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

Figure 16. Application of RobustProfile

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

Figure 17 shows a set of guidelines to model properties of the operating environment of a SUT in a UML class diagram, violations of which lead to faults in the environment. These properties are modeled based on a fault taxonomy such as the one presented in Section 4.1.2. Faults related to the environment are mostly violations of non-functional properties (NFP) such as media properties and network properties. UML doesn't directly support modeling NFP, therefore we used part of the MARTE profile for modeling such properties [7]. The MARTE profile is an extension for UML 2.0 that allows modeling real time and embedded systems. MARTE provides a generic framework to model NFP on UML models. Moreover, MARTE provides a model library that provides NFP data types for defining various NFP properties and specific applications. MARTE also provides mechanisms to extend the model library to either extend the existing NFP data types or define entirely new NFP types.

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

Figure 17. Guidelines to model faults in aspect class diagram

Now we present an example to use the above guidelines (Figure 17) to model a class diagram, which captures the properties of the environment. Figure 18 shows a partial class diagram of the *MediaQualityRecovery* robustness behavior (Section 2.2). For this robustness behavior, we identify that the *Video* fault class from the fault taxonomy (Figure 5) is relevant. For this fault class, video frame loss in incoming video streams to a VCS is important for robustness testing of the VCS. To model video frame loss, we model a property named *videoFrameLoss* in the *MediaQualityRecovery* class shown in Figure 18. The *videoFrameLoss* property is modeled as *NFP_Percentage* defined in MARTE. The property holds the percentage of video frame loss in incoming video streams to the VCS.

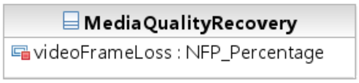


Figure 18. An Example of Modeing a Property of Environment

4.5 Aspect state machine

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

4.6 Template for Modeling Weaving-Directive state machine

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

4.7 Weaver

The aspect state machines are woven into the base state machine by a weaver, which reads the base state machine, aspect state machines, and a weaving-directive state machine and produces a woven state machine. The weaving algorithm is shown in Figure 31 in Appendix B and is based on the same weaving approach advocated in [32]. We developed a weaver for AspectSM by using Kermeta [22], which is a metamodeling language [22] that allows manipulating models by defining transformation rules at the metamodel level. We do not implement any explicit model validation, but we rely on Kermeta's model validation, which partially prevents violations of UML semantics. Kermeta conforms to OMG's metamodeling language Essential Meta Object Facility (EMOF) and Ecore [40]. Figure 21 shows the architecture of the weaver by using transformations in Kermeta to weave one or more aspect state machines into a base state machine. The AspectSM profile is defined on the UML 2.0 metamodel. An aspect state machine is defined as a UML 2.0 state machine by applying the AspectSM profile. A base state machine is a standard UML 2.0 state machine. Transformations rules in Kermeta are defined on the UML 2.0 metamodel and the AspectSM profile. Finally, the Kermeta engine uses the transformation rules that read an aspect state machine and the base state machine and weaves the aspect state machine into the base state machine. The Kermeta engine then produces a woven state machine, which is again an instance of the UML 2.0 metamodel, since the woven state machine is a standard UML 2.0 state machine. The woven state machines can then be used as input for automated model-based testing tools such as Conformiq Qtronic [24] and Smartesting Test Designer [51]. The weaver is fully automated and does not require any additional inputs from the user apart from aspect state machines and a base state machine.

The weaver is developed to support automated, model-based robustness testing, and thus aspect state machines are woven into the base state machine, which can be used for test case generation. Currently, our approach and its weaver do not support modeling and weaving interactions [12] that may occur between different aspects and may lead to conflicts between aspects during weaving. On the other hand, our weaver does support to a limited extent the handling of aspect conflicts. In [52], four classes of aspect conflicts are discussed: conflicts due to crosscutting specification, aspect-aspect conflicts, aspect-base conflicts, and concern-concern conflicts. In our application context, i.e., robustness modeling and testing, the most relevant conflicts are aspect-aspect conflicts, which are related to handling conflicts between aspects. One of the most important aspect-aspect conflicts is the ordering conflict, which is related to the order in which aspect state machines should be woven into a base state machine. Ordering conflict is most relevant in our context since, for testing purposes, we focus on modeling, weaving, and testing one or more related aspects at a time. We specify the ordering between aspect state machines in a

UML state machine containing all aspect state machines as submachine states, ordered using state machine control structure features: decision, join, and fork.

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

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

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

Figure 19. Definition of an aspect state machine

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

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

Figure 20. Definition of a weaving directive state machine

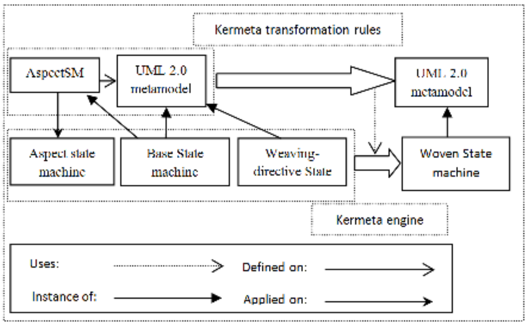


Figure 21. Aspect weaver implemented in Kermeta

5 Application of RUMM to Our Simplified Industrial Case Study

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

5.1 Activity A1: Develop a conceptual model of a system

This activity involves developing a conceptual model [27] of a system using UML 2.0 class diagram based on the domain analysis of the system. As we discussed in Section 2, the *Saturn* subsystem deals with establishing video conferencing calls, disconnecting calls, and starting/stopping presentation. In Section 2, Figure 2 shows what we refer to as a *'conceptual model'* for the system being modeled, which is here S-Saturn.

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

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

5.3 Activity A3: Identify relevant faults from fault taxonomy

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

Table 8. Media faults and their description

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

Table 9. Network faults and their description

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

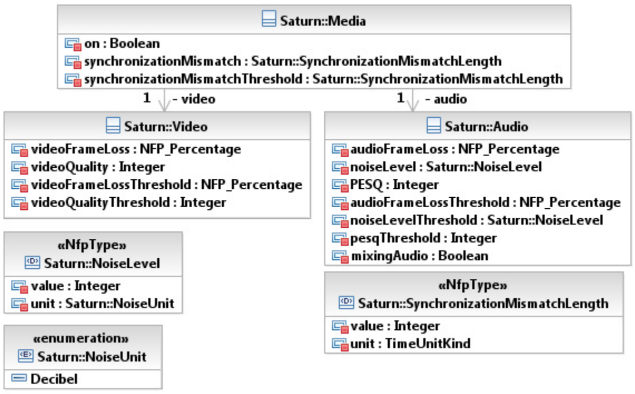
In our case study, we modeled *Media* faults in the VCSs communicating with the SUT, which are the ones that are related to quality of media such as audio, video, and their synchronization. From Figure 5, we see sub-classes of *Media* faults which are *Audio* Faults and *Video* Faults. Table 8 provides description of *Media* faults that are relevant for our case study.

In addition, network faults (*NetworkFault*, see Figure 5) are important for a VCS.

Several types of faulty situations can happen in the network that must be dealt by the VCS. We show network faults that are relevant to our case study in Table 8.

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

As advocated by the aspect-oriented paradigm, crosscutting concerns (functional or non-functional) [3] must be modeled as aspects. Activities A3 and A4 model aspects of the robustness behavior of the system using aspect state machines and aspect class diagrams. To do so, we use the AspectSM profile using the existing UML state machine notation, as presented in Section 4.2.



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

Figure 22. Class diagram for media quality attributes

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

Each aspect state machine has an associated class diagram (aspect class diagram), which is an augmentation of the conceptual model of the *Saturn* subsystem shown in Figure 2. This class diagram models the information about different kinds of faults in the fault taxonomy, such as audio and video related faults. Guidelines for such modeling based on a fault taxonomy (Section 4.1.2) are presented in Section 4.4. The *Audio* class defines audio quality attributes based on which different audio faults can be introduced, as shown in Figure 22. For instance, the *on* attribute is a *Boolean* attribute that determines if the audio is present in a videoconference. The Perceptual Evaluation of Speech Quality (PESQ) [53] is a metric for measuring audio quality. The *audioFrameLoss* is an attribute that determines the current percentage of audio frames loss during a videoconference and is defined as the MARTE type *NFP_Percentage*. The *noiseLevel* attribute is defined as the *Nfp type NoiseLevel* (modeled with <<NfpType>> from MARTE), which has two attributes: *value* that holds current noise value and *unit* contains a unit to measure audio noise such as "decibel".

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

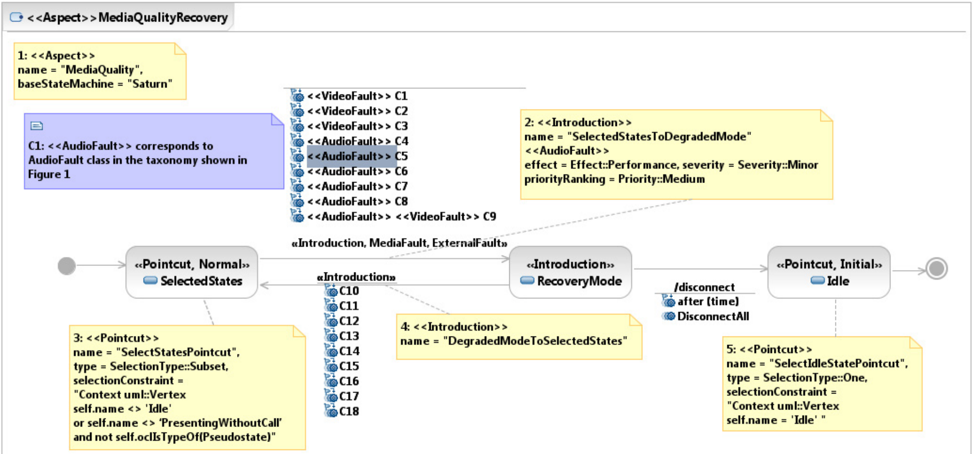
5.5 Activity A5: Develop a state machine for the robustness aspect

5.5.1 Modeling recovery from media faults

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

On the *MediaQualityRecovery* aspect state machine, the <<*Aspect*>> stereotype is described in a top-left note (labeled "1") in the upper left part of Figure 23. This aspect state machine consists of two pointcuts on states: *SelectedStates* and *Idle*, whose attribute values are described in notes explicitly linked to each <<*Pointcut*>> note. Representing pointcuts as modeling elements of UML statemachines (for instance, state in this case) enables the modeling of aspect state machines using standard UML notation, while keeping in line with UML semantics. The *SelectStatesPointcut* (see note 3 for attribute values) applied to the *SelectedStates* state selects all states of the base state machine (Figure 3) except for the *Idle* state. The *SelectIdleState* pointcut (see note 5 for attribute values) on the *Idle* state selects the *Idle* state of the base state machine (Figure 3). Whenever media quality (defined based on the quality attributes in Figure 22) falls below the acceptable level in any of the states selected by the *SelectStatesPointcut* pointcut, the system goes to the *RecoveryMode* state. This is shown as a transition—with the <<*Introduction*>>*,* <<*MediaFault*>>*,* and <<*ExternalFault*>> stereotypes (indicating this transition will be introduced in the base state machine and is modeling media faults which are external to S-Saturn) from the *SelectedStates* state to the *RecoveryMode* state with nine change events. Each change event is defined based on one media quality attribute and determines if this attribute falls below the acceptable level and is stereotyped as either <<*AudioFault*>>, <<*VideoFault*>>, or both . For example, the change event *when(not self.audio.on)* is fired from *SelectedStates* when the audio is turned off in a videoconference and is stereotyped as <<*AudioFault*>> indicating that it is an audio fault (see the comment labeled C1 and note "2" for attribute values—recall that these attributes are defined based on IEEE standard classification for anomalies [21]). If the system manages to return to acceptable media quality, it goes back to the normal state shown as a transition introduced from the *RecoveryMode* state to the *SelectedStates* state stereotyped as <<*Normal*>> (indicating that these states are normal states of S-Saturn) with again nine change events. For example, the change event *when(self.audio.on)* is fired from the *RecoveryMode* state when the audio is back in the videoconference. The state invariant of the *RecoveryMode* state ensures that

S-Saturn remains in *RecoveryMode* as long as any of the faults in the environment exists. This state invariant is simply the logical disjunction of all change events modeling the faults (Figure 24). In the other case, if the system cannot recover within time *time*, it disconnects all connected VCS and goes to the *Idle* state. This is modeled as a transition introduced between the *RecoveryMode* state and the *Idle* state with a time event and an



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

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

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

Figure 23. The *MediaQualityRecovery* aspect

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

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

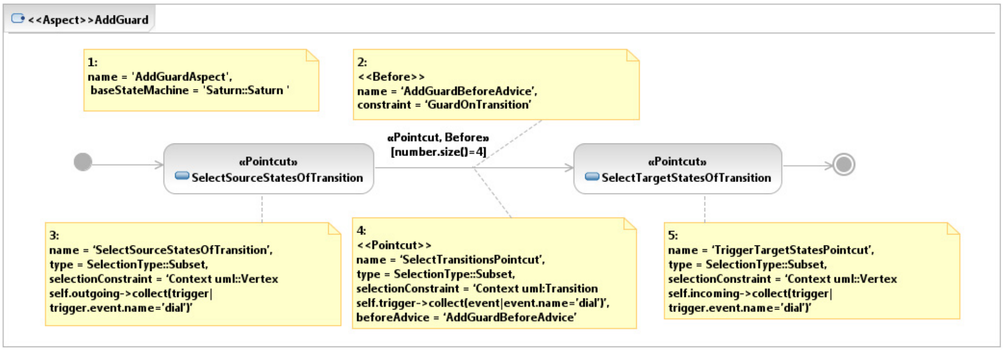
Figure 24. RecoveryMode's State invariant

5.5.2 Constraining input parameter values

The second crosscutting behavior example we present is constraining parameters of events on transitions. Since many transitions in a state machine can have the same trigger and constraints on the associated event of the trigger may be the same, redundant constraints can exist in the model and hence can be factored out as an aspect. Such constraints can be used to generate test cases exercising the system robustness with illegal inputs [54]. The aspect state machine *AddGuard* shown in Figure 25 models this crosscutting behavior. The associated class diagram for the aspect state machine is identical to Figure 2 as we do not need to model additional properties. This aspect state machine defines two pointcuts (*SelectSourceStatesOfTransition, SelectTargetStatesOfTransition*) on two states and one pointcut *SelectTransitionsPointcut* on the transition between the two states stereotyped as *<<Pointcut>>*. This aspect state machine selects all transitions which have a *dial* call event and applies a before advice *AddGuardBeforeAdvice* that adds an additional constraint *"number.size()=4"* to the existing guards on the selected transitions. This constraint ensures that the number parameter of the *dial* call event has exactly four digits.

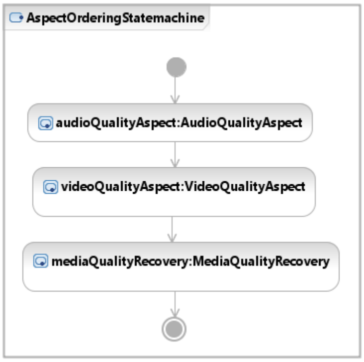
5.6 Activity A6: Define ordering of aspects using a state machine

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



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

Figure 25. State machine for the *AddGuard* aspect



AspectOrderingStatemachine
audioQualityAspect:AudioQualityAspect
videoQualityAspect:VideoQualityAspect
mediaQualityRecovery:MediaQualityRecovery

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

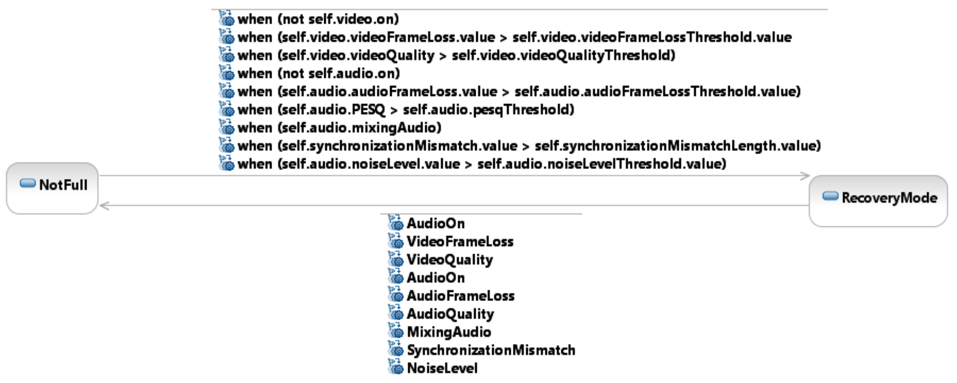
The weaving directive state machine for *MediaQualityRecovery* is shown in Figure 26. Using such state machine, we define the ordering of aspect state machines related to media quality. By weaving the aspect state machines in this order, the woven state machine will be correct for testing. The reason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

5.7 Activity A7: Weave aspects with behavioral models

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

5.7.1 Modeling recovery from media faults

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


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

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

Figure 27. Excerpt of woven state machine after applying *MediaQualityRecovery*

5.7.2 Constraining input parameter values

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

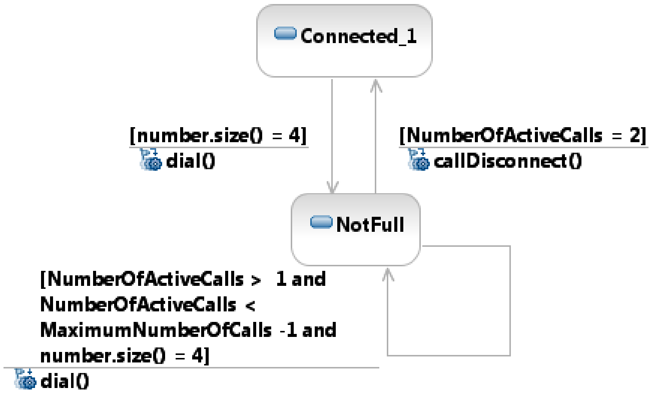

Connected_1
[number.size() = 4]
dial()
[NumberOfActiveCalls = 2]
callDisconnect()
NotFull
[NumberOfActiveCalls > 1 and NumberOfActiveCalls < MaximumNumberOfCalls -1 and number.size() = 4]
dial()

Figure 28. An excerpt of woven state machine obtained after applying *AddGuard*

Table 10. Complexity of Saturn state machines

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

6 Results from the Complete Industrial Case Study

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

6.1 Behavioral models of Saturn

Saturn consists of 20 subsystems. Each subsystem can work in parallel to the S-Saturn subsystem shown in Figure 3. For each subsystem, we modeled a class diagram to capture APIs and state variables. In addition, we modeled one or more state machines to model the behavior of each subsystem. Due to confidentiality restrictions, we do not provide names and details of the subsystems. For one subsystem (subsystem no 1), which is described in Section 2, we provided a conceptual model in Figure 2. The behavioral model of the subsystem number 1 in Table 10 consists of 15 states; four of them are modeled as submachine states to reduce model complexity. The state machines of this subsystem are presented in [47]. For other subsystems, we do not provide class diagrams and state machines, but their complexity is summarized in Table 10. It is important to note though the complexity of an individual subsystem may not look high in terms of number of states and transitions, all subsystems work in parallel to each other and therefore the overall complexity is enormous after combining them. Saturn's implementation consists of more than three million lines of C code.

6.2 Modeling robustness behavior

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

6.3 Results and discussion

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

6.3.1 Reduced modeling effort

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

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

Table 11. Modeling tasks when using and not using AspectSM*

Crosscutting behavior	Using aspects			Without aspects			Effort Saved (%)		
	S (A)	T (A)	Tr (A)	S(M/A)	T(M/A)	Tr(A)	S	T	Tr
Updating audio constraints	1	-	-	86 (M)	-	-	99%	-	-
Updating video constraints	1	-		86 (M)	-	-	99%	-	-
Media quality recovery	3	3	19	20 (A)	178	1604	-	98%	99%
Network communication	3	3	13	20 (A)	178	1082	-	98%	99%
Add Guard	2	1	-	0	22 (M)	-	-	95%	-

*S:States, T: Transitions, Tr: Triggers, A: Added, M: Modified

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

6.3.2 Enhanced separation of concerns

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

6.3.3 Improved readability

Modeling crosscutting behavior as aspect state machines keeps the base state machine less cluttered and hence easier to read. For instance, the woven state machine after applying *MediaQualityRecovery* on the *Saturn* base state machine results into a highly complex, cluttered state machine, which is difficult to read: Twenty states and 178 new transitions with 1604 triggers are added into the base state machines. Our experience is that modeling such complex state machines without aspect state machines is difficult to understand for practitioners and error-prone. Using aspect state machines, the base state machine and aspect state machines are separate and are less complex in isolation. To confirm this, we recently conducted a controlled experiment to measure the readability of aspect state machines using AspectSM [55]. Readability was measured based on the identification of defects seeded in state machines (modeled with and without AspectSM) and the score obtained when answering a comprehension questionnaire about the system behavior. The results of the experiment showed that readability with AspectSM is significantly better than that with both flat and hierarchical state machines measured in terms of inspecting models to identify seeded defect. In terms of the comprehension questionnaire, the AspectSM scores were better than flat state machines, but worse than hierarchical state machines. However, there were no significant differences between aspect and hierarchical state machines. In addition, no significant differences were observed in terms of the effort required to inspect models and detect defects.

6.3.4 Easier model evolution

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

6.3.5 Systematic fault modeling

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

6.4 Limitations

RUMM is a modeling methodology specifically developed for modeling robustness behavior to facilitate automated model-based testing. While developing the methodology, we took into consideration only those issues which are relevant for modeling the behavior of a system in the presence of faulty situations in the environment. We have not investigated whether other non-functional crosscutting concerns such as security and dependability can be successfully modeled using RUMM or an adapted version of it. The reason is that RUMM starts with modeling faults based on fault taxonomy for the system environment, which may not be necessary, for instance, when modeling security concerns such as logging. In addition, since RUMM is developed for model-based testing, we only considered issues which are important to support automated testing. For instance, we focused on UML state machines, which are often used for the automated testing in control and communication systems which typically exhibit state-driven behavior. We also focused on modeling crosscutting behavior on those modeling elements of state machines that are mandatory to support test automation such as states (including state invariants, entry, exit, and do activities) and transitions (including guard, trigger, and effect). In AspectSM, we write pointcuts as OCL queries, and we have not yet empirically evaluated and compared their expressiveness when using other related languages and notations such as the one presented in [12]. We used OCL to write pointcuts as it is the only standard for writing constraints in UML models, an important advantage in industrial contexts. Last, our work for defining interactions and ordering between different aspect state machines still requires further investigation.

7 Related Work

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

7.1 Robustness modeling methodologies

Most of the work related to robustness modeling does not make use of AOM and focus only on modeling the behavior of a system when invalid inputs are given to the system, or on modeling exceptions in the SUT in a similar fashion to programming languages. For instance, Pintér [56] reports on the modeling of exceptions in statecharts in a similar fashion to Java mechanisms for writing exceptions (*try catch* blocks). Exceptions are modeled as events on transitions in statecharts. Such statecharts are subsequently used for model checking. Jiang [57] proposed a generic framework to model self-healing software, i.e., software which try to recover from faults during their execution. The framework supports modeling faults (such as related to invalid inputs to a system), their detection, and their resolution with the help of different patterns defined for these purposes. Self-healing is modeled as a separate model which is then combined into the functional model. Lei [58] provides a methodology to check the robustness of component-based systems in the case of invalid inputs. Test cases are then generated for invalid inputs at various states and the robustness of the system is checked. Nebut [59] provides an automatic test generation approach based on use cases extended with contracts, after transforming them into a transition system. Their approach supports both functional and robustness test generation. Robustness test cases are generated by calling use cases when their preconditions are false. Entwisle [60] proposed a framework for modeling various domain specific exception types such as network exceptions, database exceptions, and web service exceptions using use cases. This approach generates exception policy configurations from application models using model transformation and finally generates code in Java for exceptions management, such as how to catch a particular exception.

The work (RUMM) presented in this paper is different from the existing work in robustness modeling in one or more of the following ways: 1) It provides a robustness modeling methodology to model system robustness in the presence of faults in its environment; this aspect has received little attention in the literature. In contrast, most of the existing work focus only on modeling the behavior of a system when invalid inputs are given to them [56] [57] [58] [59]; 2) It is aimed at performing automated model-based robustness testing based on the robustness models for industrial systems. In contrast to the work presented in [59], our work is based on UML state machines, which are the main notation currently used for model-based test case generation [18]; 3) It relies on modeling standards, in this case UML state machines and the MARTE profile [7], to model faulty situations of the environment; 4) It uses AOM to model robustness behavior separately from the core, functional behavior, hence decreasing modeling effort by avoiding clutter in models, making them easier to read and decreasing chances of modeling errors; 5) We use standard UML extension mechanism, i.e., profile, to support robustness modeling as aspects using standard UML state machines, thus eliminating the need to adopt new notations and consequently facilitating the practical adoption of RUMM in industry; and 6) RUMM is driven by defining a fault taxonomy, thus leading to the more systematic modeling of robustness behavior. The process of defining the taxonomy helps in developing a clear and thorough understanding of the different kinds of faults that may occur in the environment against which system robustness must be tested.

7.2 AOM profiles for UML state machines

Several UML profiles for AOM have been proposed in the literature [61-64] for different UML diagrams. Since we defined a profile to define aspects on state machines, we only assess the existing AOM work focusing on state machines. We do so along three dimensions: 1) Features of UML state machines supported by a profile such as state, state invariant, do activity, entry activity, exit activity, transition, guard, trigger, and effect, 2) Features of aspect-orientation supported by a profile or a modeling approach such as pointcut, advice, and inter-type declaration (a programming construct in AspectJ [46] used to introduce new variables in a base class), 3) Representation used for the aspect-orientation features. Based on the above selection criterion, we found five related works in the literature [1-4, 9]. Table 12 and Table 13 characterize these works with respect to their coverage of important UML state machine modeling elements including state, transition and their contained elements, e.g., state invariant in state and guard in transition. For instance, in Table 12 and Table 13, the approach presented in [1] only supports modeling crosscutting behavior in states and transitions (indicated by a + sign), but not in other modeling elements (indicated by a - sign). Certain features of UML state machines which are mandatory for performing automated, model-based testing are not supported by any of the existing works. This includes state invariants and guards which, as discussed above, are essential to generating automated oracles and generating automated test data, respectively.

Table 12. Comparison of supported modeling elements related to a state

Reference	State	State Invariant	Entry Activity	Do Activity	Exit Activity
[1]	+	-	-	-	-
[2]	+	-	-	-	-
[3]	+	-	-	-	-
[4] [6]	+	-	-	-	-
[9]	+	-	-	-	-

Table 13. Comparison of supported modeling elements related to a transition

Reference	Transition	Guard	Trigger	Effect
[1]	+	-	-	-
[2]	+	-	-	-
[3]	+	+	+	-
[4] [6]	+	-	-	-
[9]	+	-	+	+

Table 14. Comparison of supported features of aspect-orientation

Reference	Before Advice	Around Advice	After Advice	Pointcut	Introduction
[1]	+	-	+	+	-
[2]	+	-	+	+	-
[3]	+	-	+	+	+
[4, 19] [6]	+	+	+	+	+
[9]	-	+	-	+	-

Table 15. Comparison of the representation of aspect-orientation features

Reference	Aspect	Advice	Pointcut	Introduction
[1]	State machine	State machine elements	Non-Standard	Not supported
[2]	State machine	Non-Standard	Non-Standard	Not supported
[3]	State machine	Non-Standard	Non-Standard	Non-Standard
[4] [6]	State machine	Non-Standard	Non-Standard	Non-Standard
[9]	Class	Activity diagram	Non-Standard	Not supported
AspectSM	State machine	State machine elements and OCL	OCL	State machine elements

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

7.3 Comparisons with Generic AOM weavers

A generic weaver, GeKo, is presented in [19], but the current implementation of the weaver is not complete (e.g., it does not support state machines) and its use requires many manual steps such as specifying mappings from pointcuts to the base model. Metamodels for pointcut and advice are defined by relaxing the UML 2.0 metamodel and are generated automatically from it using a transformation. However, there is no support for modeling pointcuts and advice based on the generated metamodels. It therefore requires developing a new diagrammatic support for these metamodels, which will not be standard, and consequently will not be supported by UML modeling tools, making the practical adoption of the weaver difficult. Another similar generic weaver, SmartAdapter, is presented in [65]. The only major difference between GeKo and SmartAdapter is that SmartAdapter requires manually writing composition rules for aspect and base models, whereas this is not required by GeKo.

An aspect composition language (SDMATA/MATA) is presented in [12, 66], which allows modeling and composing aspects on UML state machines using patterns. The selection of modeling elements of a UML state machine (concept similar to pointcuts) is performed using *state diagram patterns*. Using state diagram patterns, modeling elements are selected using regular expressions defined on diagrammatic notations that 'resemble' UML state machines (defined based on the extension of UML state machine metamodel). In AspectSM, we write pointcuts as OCL expressions to query modeling elements of a base state machine. To compare expressiveness of OCL expressions for writing pointcuts with regular expressions, a controlled experiment is required, which will be conducted in the future. The tool support for modeling patterns in SDMATA, however, is still under development. SDMATA requires defining *composition operators* (concept similar to advice) using a language based on graph transformations. As for other approaches in the literature, applying SDMATA to industrial contexts, requires learning additional, non-standard notations such as state diagram patterns.

Kermeta [22] is a model-to-model transformation language, which provides the facility to write transformation code in aspect-oriented style. Using such facility, aspects can be introduced at runtime on metaclasses (e.g., UML *Statemachine* metaclass) for introducing new attributes and operations on metaclasses or for providing definitions of existing operations in metaclasses. However, applying Kermeta for our purpose in the industrial setting requires understanding not only details of the UML metamodel, but also requires learning a new language for writing aspects. Using AspectSM, we only need simple stereotypes with a few attributes, thus reducing the learning curve and improving applicability. In other words, achieving a similar objective in Kermeta may require writing hundreds of lines of complex transformation code.

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

7.4 Testing based on Aspect-Orientation Modeling

There are also works in the literature that deal with testing aspect-oriented programs using UML-based models such as state machines [6, 67, 68]. The focus of our work is different since we do not focus on testing implementation, which is coded in an aspect-oriented programming (AOP) language such as AspectJ [46]. For instance, in our industrial system, we are targeting system level testing of an embedded software of a VCS developed by Cisco, Norway, which is implemented in a subset of C language. In addition a few approaches such as those presented in [69, 70] focus on testing components using AOM to specify their behavior as state machines. The aspects are also specified as state machines to be consistent with the notation of the core behaviors (components). The composition rules are specified in their own developed language (not following any standard), which specify how to weave aspects into the core behavior. These works focus on modeling and testing components when wrong inputs are provided to them by their users. Our purpose is also different from these approaches since we focus on modeling faulty environment (network and other VCSs) conditions of the system under test using aspect state machines and test the behavior of the VCS in the presence of these conditions.

8 Conclusion

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

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

Another very important contribution of the paper is that we performed and report on an industrial case study that suggests that using our methodology and profile may result in significantly reduced modeling effort. Such case studies are indeed very rare and, to the knowledge of the authors, none is reported on aspect state machines. Results show that modeling crosscutting behavior as a separate model (aspect state machine) leads to the modeling of significantly less states, less transitions, and also less changes to constraints such as state invariants. Modeling both standard and crosscutting behavior—in our case robustness behavior—in one state machine would lead to many redundant modeling elements and yield cluttered models that are difficult to understand. As an example, for one of the aspect state machine in our case study, we avoided the modeling of 1586 extra triggers on 178 transitions (98% reduction) by using our profile. However, this came at the cost of modeling three pointcuts for that aspect state machine, which is clearly an additional overhead, but which should be minimized by the fact that they are modeled as a UML state machine. It is however expected that the modeling effort required to model three pointcuts is significantly less than modeling 1586 triggers. In addition, the results of a recent controlled experiment [55] showed that readability of aspect state machines is significantly better than standard UML state machines, though there was no significant difference in the effort to inspect both types of state machines. Readability was measured based on the identification of defects seeded in state machines (modeled with and without AspectSM) and the score obtained when answering a comprehension questionnaire about the system behavior.

We also developed a weaver using the model transformation tool Kermeta [22] to automatically produce woven state machines. These can in turn be used for different applications, in our case model-based testing using state machines in input based on technologies such as Conformiq QTronic [24] and SmartTesting Test Designer [51]. In the future, we are planning to integrate the woven state machines produced by our weaver with our model-based testing tool TRUST [23] to automatically generate robustness test cases. TRUST [23] has already been used for generating executable functional test cases at Cisco, Norway. In the future, we will investigate to which extent our profile is applicable for other types of crosscutting behaviors to be modeled as state machines. In addition, we need to investigate the effort required by developers and testers to learn and apply RUMM. A series of controlled experiments and case studies are required for this purpose, which we are planning to conduct in the future. Our work on modeling interactions and ordering between various aspects still needs further investigation and evaluation.

ACKNOWLEDGEMENTS

We would like to thank Robert B. France and Jacques Klein for their insightful comments and feedback on this paper.

9 References

[1] Zhang, G. Towards Aspect-Oriented State Machines. In *Proceedings of the In Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development (AOASIA'06)* (Tokyo, 2006).
[2] Zhang, G. and Hölzl, M. HiLA: High-Level Aspects for UML-State Machines. In *Proceedings of the In Proceedings of the 14th Workshop on Aspect-Oriented Modeling (AOM@MoDELS'09)* (2009).
[3] Zhang, G., Hölzl, M. M. and Knapp, A. *Enhancing UML State Machines with Aspects*. 2007.
[4] Xu, D., Xu, W. and Nygard, K. A State-Based Approach to Testing Aspect-Oriented Programs. In *Proceedings of the In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering* (Taiwan, 2005).
[5] *UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms,* http://www.omg.org/spec/QFTP/1.1/, 2010
[6] Xu, D. and Xu, W. State-based incremental testing of aspect-oriented programs. In *Proceedings of the Proceedings of the 5th international conference on Aspect-oriented software development* (Bonn, Germany, 2006). ACM.
[7] *Modeling and Analysis of Real-time and Embedded systems (MARTE),* http://www.omgmarte.org/, 2010
[8] Jürjens, J. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the Proceedings of the 5th International Conference on The Unified Modeling Language* (2002). Springer-Verlag.
[9] Zhang, J., Cottenier, T., Berg, A. V. D. and Gray, J. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6, 7I (2007).
[10] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, IEEE Std 610.12-1990, 1990.
[11] Yedduladoddi, R. *Aspect Oriented Software Development: An Approach to Composing UML Design Models*. VDM Verlag Dr. Müller, 2009.
[12] Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A. and Rabbi, R. *An Expressive Aspect Composition Language for UML State Diagrams*. 2007.
[13] Runeson, H. and Höst, M. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14, 2I (2009), 131-164.
[14] Aldini, A., Gorrieri, R., Martinelli, F. and Jürjens, J. *Model-Based Security Engineering with UML*. Springer Berlin / Heidelberg, 2005.
[15] Péreza, J., Ali, N., Carsı´b, J. A., Ramosb, I., Álvarezc, B., Sanchezc, P. and Pastorc, J. A. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information and Software Technology*, 50, 9-10I (2008), 969-990.
[16] Cottenier, T., Berg, A. v. d. and Elrad, T. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *Proceedings of the Aspect Oriented Software Development (AOSD)* (2007).

[17] Cottenier, T., Berg, A. v. d. and Elrad, T. Stateful Aspects: The Case for Aspect-Oriented Modeling. In *Proceedings of the Proceedings of the 10th international workshop on Aspect-oriented modeling* (Vancouver, Canada, 2007). ACM.
[18] Shafique, M. and Labiche, Y. *A Systematic Review of Model Based Testing Tools*. Carleton University, Department of Systems and Computer Engineering, Technical Report (SCE-10-04), 2010.
[19] Kienzle, J., Abed, W. A. and Klein, J. Aspect-Oriented Multi-View Modeling. In *Proceedings of the In Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development* (Charlottesville, Virginia, USA, 2009). ACM.
[20] Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1, 1I (2004), 11-33.
[21] *IEEE Standard Classification for Software Anomalies*. IEEE, IEEE Std 1044-2009, 2009.
[22] *Kermeta - Breathe Life into Your Metamodels, IRISA and INRIA*, http://www.kermeta.org/, 2010
[23] Ali, S., Hemmati, H., Holt, N. E., Arisholm, E. and Briand, L. C. *Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies*. Simula Research Laboratory, Technical Report (2010-01), 2010.
[24] *QTRONIC, CONFORMIQ*, http://www.conformiq.com/qtronic.php, 2010
[25] *Standard for Software Quality Characteristics*. International Organization for Standardization, ISO-9126-3, 2003.
[26] *Software Assurance Standard*. NASA Technical Standard, NASA-STD-8739.8, 2005.
[27] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* Prentice Hall PTR, 2004.
[28] Bruning, S., Weissleder, S. and Malek, M. A Fault Taxonomy for Service-Oriented Architecture. In *Proceedings of the Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium* (2007). IEEE Computer Society.
[29] Chan, K. S., Bishop, J., Steyn, J., Baresi, L. and Guinea, S. *A Fault Taxonomy for Web Service Composition*. Springer-Verlag, 2009.
[30] Mariani, L. A Fault Taxonomy for Component-Based Software. *Electronic Notes in Theoretical Computer Science*, 82, 6I (2003), 55-65.
[31] Hayes, J. H. Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project. In *Proceedings of the Proceedings of the 14th International Symposium on Software Reliability Engineering* (2003). IEEE Computer Society.
[32] Ho, W.-M., Jézéquel, J.-M., Pennaneac'h, F. and Plouzeau, N. A toolkit for weaving aspect oriented UML designs. In *Proceedings of the Proceedings of the 1st international conference on Aspect-oriented software development* (Enschede, The Netherlands, 2002). ACM.
[33] Pazzi, L. Explicit Aspect Composition by Part-Whole State Charts. In *Proceedings of the In Proceedings of the Workshop on Object-Oriented Technology* (1999). Springer-Verlag.
[34] France, R., Ray, I., Georg, G. and Ghosh, S. Aspect-oriented Approach to Early Design Modelling. *IEEE Software*, 151, 4I (2004).
[35] Binder, R. V. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[36] Lagarde, F., Espinoza, H., Terrier, F., André, C. and Gérard, S. *Leveraging Patterns on Domain Models to Improve UML Profile Definition*. 2008.
[37] Weilkiens, T. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Tim Weilkiens, 2008.
[38] *UML Profile for Schedulability, Performance and Time,* http://www.omg.org/technology/documents/profile_catalog.htm, 2010
[39] Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I. and Williams, C. *Model-Driven Testing: Using the UML Testing Profile*. Springer, 2007.
[40] Steinberg, D., Budinsky, F., Paternostro, M. and Merks, E. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.
[41] Filman, R. E., Elrad, T., Clarke, S. and Aksit, M. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.
[42] *IBM OCL Parser, IBM,* http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html, 2010
[43] *OCLE,* http://lci.cs.ubbcluj.ro/ocle/, 2010
[44] *EyeOCL Software,* http://maude.sip.ucm.es/eos/, 2010
[45] Pender, T. *UML Bible*. Wiley, 2003.
[46] Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.
[47] Ali, S., Briand, L. C. and Hemmati, H. *Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems*. Simula Research Laboratory, Technical Report (2010-03), 2010.
[48] Stein, D., Hanenberg, S. and Unland, R. A UML-based aspect-oriented design notation for AspectJ. In *Proceedings of the Proceedings of the 1st international conference on Aspect-oriented software development* (Enschede, The Netherlands, 2002). ACM.
[49] Clarke, S. and Walker, R. J. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada, 2001). IEEE Computer Society.
[50] Stein, D., Hanenberg, S. and Unland, R. Designing Aspect-Oriented Crosscutting in UML. In *Proceedings of the In AOSD-UML Workshop at AOSD '02* (2002).
[51] Utting, M. and Legeard, B. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.
[52] Tessier, F., Badri, L. and Badri, M. Towards a Formal Detection of Semantic Conflicts Between Aspects: A Model-Based Approach. In *Proceedings of the The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004* (2004).
[53] *Perceptual Evaluation of Speech Quality (PESQ),* http://en.wikipedia.org/wiki/PESQ, 2010
[54] Ali, S., Iqbal, M. Z., Arcuri, A. and Briand, L. C. A Search-based OCL Constraint Solver for Model-based Test Data Generation. In *Proceedings of the Proceedings of the 11th International Conference On Quality Software (QSIC 2011)* (2011).
[55] Ali, S., Yue, T., Briand, L. C. and Malik, Z. I. *Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines?* Simula Reserach Laboratory, Technical Report(2010-11), 2010.
[56] Pintér, G. and Majzik, I. *Modeling and Analysis of Exception Handling by Using UML Statecharts*. 2005.
[57] Jiang, M., Zhang, J., Raymer, D. and Strassner, J. A Modeling Framework for Self-Healing Software Systems. In *Proceedings of the Models@run.time In conjunction with*

MoDELS/UML (2007).
[58] Lei, B., Liu, Z., Morisset, C. and Li, X. State Based Robustness Testing for Components. *Electron. Notes Theor. Comput. Sci.*, 260, 173-188.
[59] Nebut, C., Fleurey, F., Traon, Y. L. and Jezequel, J.-M. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.*, 32, 3I (2006), 140-155.
[60] Entwisle, S., Schmidt, H., Peake, I. and Kendall, E. A Model Driven Exception Management Framework for Developing Reliable Software Systems. In *Proceedings of the Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference* (2006). IEEE Computer Society.
[61] Jingjun, Z. *Modeling Aspect-Oriented Programming with UML Profile*. 2009.
[62] Júnior, J. U., Camargo, V. V. and Chavez, C. V. F. UML-AOF: a profile for modeling aspect-oriented frameworks. In *Proceedings of the Proceedings of the 13th workshop on Aspect-oriented modeling* (Charlottesville, Virginia, USA, 2009). ACM.
[63] Aldawud, O., Elrad, T. and Bader, A. UML Profile for Aspect-Oriented Software Development. In *Proceedings of the The Third International Workshop on Aspect Oriented Modeling* (2003).
[64] Evermann, J. A meta-level specification and profile for AspectJ in UML. In *Proceedings of the Proceedings of the 10th international workshop on Aspect-oriented modeling* (Vancouver, Canada, 2007). ACM.
[65] Petriu, D., Rouquette, N., Haugen, Ø., Morin, B., Klein, J., Kienzle, J. and Jézéquel, J.-M. *Flexible Model Element Introduction Policies for Aspect-Oriented Modeling*. Springer Berlin / Heidelberg.
[66] Whittle, J. and Jayaraman, P. *MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation*. Springer-Verlag, 2008.
[67] Xu, D., Xu, W. and Nygard, K. A State-Based Approach to Testing Aspect-Oriented Programs. In *Proceedings of the Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering* (2005).
[68] Xu, W. and Xu, D. A Model-Based Approach to Test Generation for Aspect-Oriented Programs. In *Proceedings of the First Workshop on Testing Aspect-Oriented Programs* (2005).
[69] Bruel, J.-m., Araújo, J., Moreira, A. and Royer, A. Using Aspects to Develop Built-In Tests for Components. In *Proceedings of the In AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language* (2003).
[70] Bruel, J. M., Moreira, A. and Araújo, J. Adding Behavior Description Support to COTS Components through the Use of Aspects. In *Proceedings of the 2nd Workshop on Models for Non-functional Aspects of Component-Based Software* (2005).

10 Appendix A: Aspects for Updating State Invariants

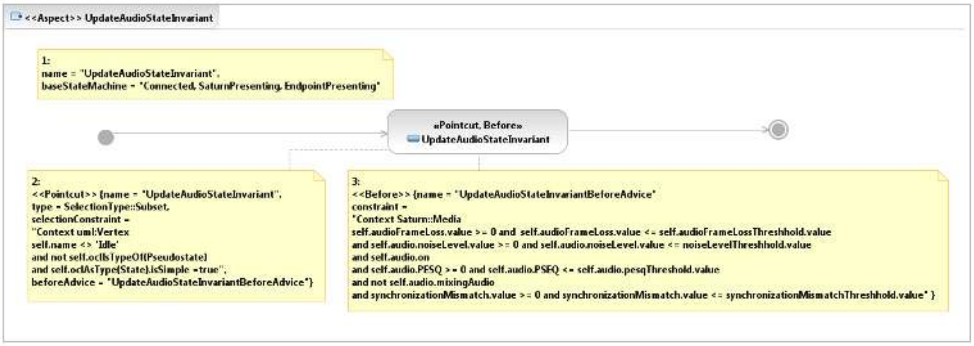
In this section, we present the details of *AudioQualityAspect* and *VideoQualityAspect*. These aspects update state invariants in the base state machine (Figure 3) with audio and video quality constraints.

10.1 Updating state invariants with audio quality attributes

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

10.2 Updating state invariants with video quality attributes

The aspect in Figure 30 updates the state invariants of states selected in the base state


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

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

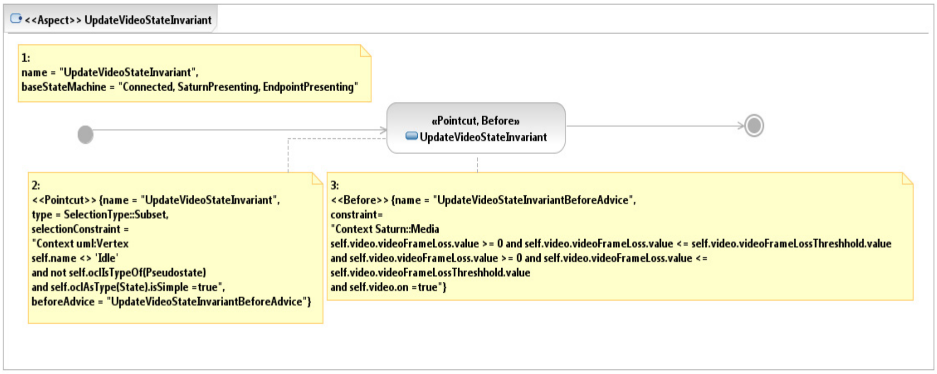
```
2:
<<Pointcut>> {name = "UpdateAudioStateInvariant",
type = SelectionType::Subset,
selectionConstraint =
"Context uml:Vertex
self.name <> 'Idle'
and not self.oclIsTypeOf(Pseudostate)
and self.oclAsType(State).isSimple =true",
beforeAdvice = "UpdateAudioStateInvariantBeforeAdvice"}
```

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

Figure 29. State machine for *AudioQualityAspect*

machine by the <<*Pointcut*>> stereotype applied on the state invariant of the state *UpdateStateInvariantsWithVideoQuality* in Figure 30 according to the before advice defined based on the video quality attributes modeled in Figure 26. The *on* attribute is a *Boolean* attribute that determines if the video is present in a videoconference. The *videoQuality* is a video quality metric for measuring video quality and is defined as an *Integer*. The *videoFrameLoss* is an *Integer* attribute that determines the current video frame loss during a videoconference.

The *<<Before>>* stereotype applied on the state invariant of the state *UpdateStateInvariantsWithVideoQuality* in Figure 30 adds an additional conjunct to state


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

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

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

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

Figure 30. State machine for the Vi*deoQualityAspect*

invariants of all selected states (see note 3 for attribute values). The woven state machines looks exactly the same as the base state machines in as only state invariants changed in this case.

11 Appendix B: Weaver Algorithm

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

Figure 31. Weaving algorithm

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

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

Figure 32. The WeavePointcut() function

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

Figure 33. The Introduction() function

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

Figure 34 (a). The WeaveNoStereotype() function

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

Figure 34 (b). The *RepeatCompostion()* function

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

Figure 34 (c). The *PointCutOnState()* function

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

Figure 35. The *PointcutOnTransition()* function

12 Appendix C: Network Communication Aspect

12.1 Description of the Aspect

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

12.2 Network Robustness (NR) Aspect (Aspect Class Diagram)

Figure 36 shows a class diagram that models the robust behavior of the system in the presence of different network faults defined based on the fault taxonomy (Figure 5) such as jitter, packet loss, low bandwidth, illegal packets for videoconferencing protocols (SIP and H323), and in the case of no network connection. Six network properties are modeled in the class diagram that models different faulty situations. Five network properties are modeled as non-functional (NF) types using the MARTE profile [7]: packet loss, jitter, bandwidth, and percentage of illegal packets for H323 and SIP protocols. The network connection is modeled as a *Boolean* attribute.

12.2.1 PacketLoss

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

12.2.2 Jitter

This property introduces delay between network packets. This delay is introduced in the unit of millisecond (ms) and checks robustness of a videoconferencing system in the presence of delayed network packets. This property has two attributes: *value* of type

Integer and *unit* of the MARTE type *TimeUnitKind*. The type *TimeUnitKind* of the MARTE profile is used to define units for time values such as millisecond and microsecond. We chose this data type so that a modeler can chose appropriate unit to measure unit. We set the default value of the *unit* attribute to millisecond (ms).

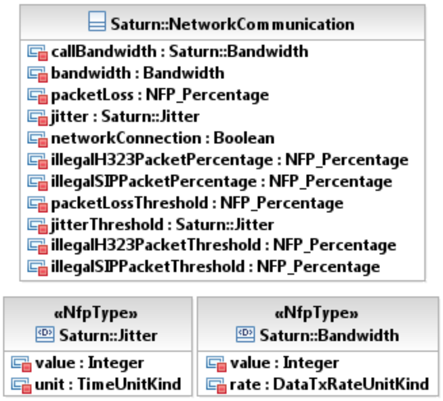

Saturn::NetworkCommunication
callBandwidth : Saturn::Bandwidth
bandwidth : Bandwidth
packetLoss : NFP_Percentage
jitter : Saturn::Jitter
networkConnection : Boolean
illegalH323PacketPercentage : NFP_Percentage
illegalSIPPacketPercentage : NFP_Percentage
packetLossThreshold : NFP_Percentage
jitterThreshold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

Figure 36. Class diagram for the NR aspect

12.2.3 Bandwidth

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

12.2.4 IllegalH323PacketPercent

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

12.2.5 IllegalSIPPacketPercent

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

12.3 Aspect State Machine for NR

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

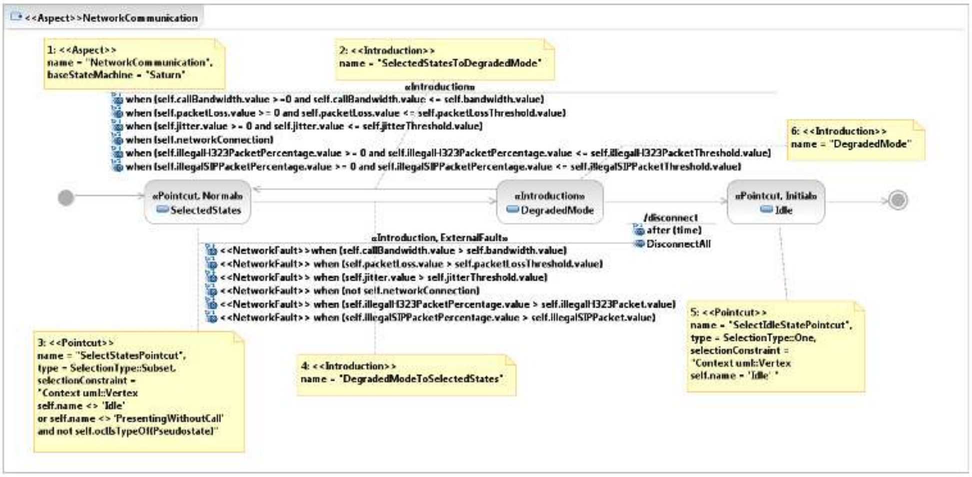

<<Aspect>>NetworkCommunication
1: <<Aspect>>
name = "NetworkCommunication",
baseStateMachine = "Saturn"
2: <<Introduction>>
name = "SelectedStatesToDegradedMode"
«Introduction»
when (self.callBandwidth.value >=0 and self.callBandwidth.value <= self.bandwidth.value)
when (self.packetLoss.value >= 0 and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

Figure 37. State machine for the 'NetworkCommunication' aspect

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

Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines?

Shaukat Ali, Tao Yue, Lionel C. Briand

Submitted to the Systems and Software Modeling (SoSyM) Journal

Abstract— Aspect-oriented Modeling (AOM) is a relatively recent and very active field of research, whose application has however been limited in practice. AOM is assumed to yield several potential benefits such as enhanced modularization, easier evolution, increased reusability, and improved readability of models, as well as reduced modeling effort. However, credible, solid empirical evidence of such benefits is lacking. We evaluate the "readability" of state machines when modeling crosscutting behavior using AOM and more specifically AspectSM, a recently published UML profile. This profile extends the UML state machine notation with mechanisms to define aspects using state machines. Readability is indirectly measured through defect identification and fixing rates in state machines, and the scores obtained when answering a comprehension questionnaire about the system behavior. With AspectSM, crosscutting behavior is modeled using so-called "aspect state machines". Their readability is compared with that of system state machines directly modeling crosscutting and standard behavior together. An initial controlled experiment and a much larger replication were conducted with trained graduate students, in two different institutions and countries, to achieve the above objective. We use two baselines of comparisons—standard UML state machines without hierarchical features (flat state machines) and standard state machines with hierarchical/concurrent features (hierarchical state machines). The results showed that defect identification and fixing rates are significantly better with AspectSM than with both flat and hierarchical state machines. However, in terms of comprehension scores and inspection effort, no significant difference was observed between any of the approaches. Results of the experiments suggest that one should use, when possible, aspect state machines along with hierarchical and/or concurrent features of UML state machines to model crosscutting behaviors.

1. Introduction

Aspect-orientation provides enhanced modularization by separating out crosscutting concerns as separate entities called aspects. Aspect-orientation is a very active field [1, 2], which has mainly focused on aspect-oriented programming (AOP), but also led to significant progress in the realms of design and modeling, denoted as aspect-oriented Modeling (AOM) [3, 4]. Crosscutting concerns, for example related to robustness or security behavior, are modeled as aspect models and are subsequently woven into a primary/base model capturing nominal functional behavior. AOM is expected to yield benefits such as improved readability, enhanced modularization, easier evolution, and increased reusability of models, as well as reduced modeling effort [4]. However, there is very little empirical evidence of such benefits. Empirical investigations, such as controlled experiments, are required to support the above claims about AOM and better understand its limitations. For example, an initial search on the IEEE, ACM, Science Direct, Wiley Interscience, and Springer digital libraries yielded 517 papers on AOM; however, none of them reported any empirical study to evaluate its benefits. This paper is a first step in that direction and reports on the first two controlled experiments assessing the benefits of AOM.

In industrial models, such as state machines, one must not only capture nominal behavior but also robustness behavior, for example describing how the system should react to abnormal environmental conditions. Such robustness is considered very critical in many standards such as in the IEEE Standard Dictionary of Measures of the Software Aspects of Dependability [5], the ISO's Software Quality Characteristics standard [6], and the Software Assurance Standard by NASA [7]. This is for example needed to support the automated robustness testing of embedded or communication systems [8] based on models. Focusing on UML state machines, as it is the most widely used notation in practice for the specification of control and communication systems [8, 9], crosscutting (e.g., robustness) behavior can result in cluttered and redundant UML state machines. As a result, modeling such crosscutting behavior directly on UML state machines can be error-prone and is expected to require significant extra modeling effort.

In a recent paper we reported on AspectSM [10], a UML profile which was defined to model crosscutting behavior on UML state machines using *extended* UML state machines, in order to facilitate the use of AOM and limit its associated learning curve. The focus of

AspectSM was on model-based test case generation for control and communication systems [8, 9], though it can potentially be applied for other purposes. Comparable approaches in the literature do not use UML extension mechanisms to provide complete AOM support: they make use of specific notations for aspect-related features that do not follow any standard. With our industrial partners, and generally in most industrial settings, AOM support should be based on the UML standard to facilitate adoption. Also, support for modeling robustness behavior as a crosscutting behavior in state invariants and guards is not supported by any existing AOM approach, though they are important features in many applications, such as the generation of automated test oracles and data generation. A detailed comparison of the AspectSM profile with other related profiles can be found in [10]. AspectSM was successfully applied to model the robustness behavior of video conferencing systems for the purpose of model-based robustness testing at Cisco Systems, Norway [10]. Results suggested that more than 95% of the modeling effort could potentially be saved. Consistent with AOM broader claims, using AspectSM to model crosscutting behavior on UML state machines as aspects, should reduce cluttering and redundancy in models.

In this paper, we report the first two controlled experiments that were conducted to evaluate the “readability” of state machines modeling crosscutting behavior using AOM, in our case AspectSM. By “readability” we denote the ease with which state machines can be understood, analyzed, and changed by a human to perform various tasks. We evaluate AspectSM models by comparing them with UML state machines modeling crosscutting behavior directly. The first controlled experiment, which was smaller in scale than the second, was conducted with 27 fully trained, graduate students taking a graduate course in ‘Advanced Software Architecture’ at the University Institute of Information Technology (UIIT) at the Pir Mehr Ali Shah Arid Agriculture University, Rawalpindi, Pakistan. The second experiment, which can be seen as a differentiated replication of the first one, was conducted at the Beijing University of Aeronautics and Astronautics (BUAA) Beijing, China, with 47 graduate students. Half of the students were taking a graduate course titled ‘Software Engineering’, while the remaining half were taking a course titled “Software Architecture”. Two case study systems were used for the controlled experiments. The first one is an Elevator control system (ECS) provided in a well-known textbook [11]— but we had to extend the case study system with two crosscutting behaviors: emergency stop and emergency call. The second case study system is a reduced version of an industrial video conferencing system developed by Cisco Systems, Norway. The readability of state machines is evaluated using three measures. The first measure is based on the ability of subjects to identify design defects seeded in state machines by checking their conformance against their specifications given as English text. The second measure is based on the ability of the subjects to fix the defects seeded in state machines. The third measure is based on subjects' scores to answer a carefully designed comprehension questionnaire. Based on these three measures, we compare the readability and also the effort resulting from using AspectSM, against both standard hierarchical and flat UML state machines. Our motivation is to assess the impact of hierarchy and/or concurrency, which is supposed to address some of the same issues as AOM in state machines (e.g., redundancy), on the relative benefits of using AspectSM.

The results of the experiments show that AspectSM helps significantly increase the identification and fixing of defects. It also leads to significantly better comprehension scores than flat state machines but hierarchical state machines look better in terms of comprehension scores, though these results were not statistically significant. In terms of the inspection effort, no significant difference was observed. For the replication, we observed similar results for defect identification and fixing but there was no significant difference observed between any of the three approaches regarding comprehension scores.

The rest of the paper is organized as follows: Section 2 describes the necessary background to understand the rest of the paper, Section 3 provides details on planning of the initial experiment and its replication, and Section 4 reports on results of the initial experiment and replication, respectively. Section 5 discusses the possible threats to validity and Section 6 compares existing, related experiments in Aspect-oriented Programming (AOP) to our experiments. Finally, we conclude our paper in Section 7.

2. Background

In this section, we provide a brief reminder of UML state machines and an overview of aspect state machines in AspectSM, the technology being evaluated in our controlled experiments.

2.1 UML State Machines

UML state machines enable modeling the dynamic behavior of a class, subsystem, or system. State machines in general are extensively used to model a variety of systems such as communication [12] and control systems [9]. Due to the ability of state machines to capture rich and detailed information, they have been used for automatic code generation [13] and the automated generation of test cases [8, 14, 15]. UML state machines provide many advanced features such as concurrency and hierarchy, which aim at supporting large-scale modeling. Concurrency enables the modeling of concurrent behavior whereas state hierarchies capture commonalities among states. A submachine state in a state machine functions like a simple state, but is referring to another state machine. A submachine can be reused in more than one state machine and may refer to other submachines [16]. They can therefore help reduce the structural complexity of state machines. State machines developed using the hierarchical features of UML will be referred to as hierarchical state machines in this paper and the ones developed without using submachine states, with only basic features of UML state machines, will be referred as flat state machines.

2.2 Aspect State Machines

AspectSM is a UML profile described in [10], which supports the modeling of system robustness behavior, which is very common type of crosscutting behavior in many types of systems such as communication and control systems [4]. An example of a robustness behavior for a communication system is related to how the system should react, in various states, in the presence of high packet loss. The system should be able to recover lost packets and continue to behave normally in a degraded mode. In the worst case, the system should go back to the most recent state and not simply crash or show inappropriate behavior. In a control system, one needs to model, for example, how the system should react, in various states, when a sensor breaks down. AspectSM allows modeling UML state machine aspects as UML state machines (aspect state machines). Such an approach, relying on a standard and using the target notation as the basis to model the aspects themselves, is expected to make the practical adoption of aspect modeling easier in industrial contexts. In our previous work [10], we thoroughly compared AspectSM with the similar existing AOM profiles. Our findings showed that only AspectSM is exclusively based on standard UML notation and OCL, thus eliminates the need of learning additional non-standard notations or languages, and therefore making it easy to reuse open source and commercial technology. This is highly important in most industrial contexts and strongly affects the adoption of modeling technologies. In addition, it is easy to train people in the industry for standard languages such as UML and the OCL.

Though AspectSM was originally defined to support scalable, model-based, robustness testing, including test case and oracle generation at Cisco Systems, Norway, a fundamental question is whether it is easier to model crosscutting concerns such as robustness with AOM in general, and AspectSM in particular, than simply relying on UML state machines to do it all. In AspectSM, the core functionality of a system is modeled as one or more standard UML state machines (called base state machines). Crosscutting behavior of the system (e.g., robustness behavior) is modeled as aspect state machines using the AspectSM profile. State machines developed using this profile will be referred as aspect state machines. A weaver [10] then automatically weaves aspect state machines into base state machine to obtain a complete model, that can for example be used for testing purposes. The AspectSM profile specifies stereotypes for all features of AOM, in which the concepts of Aspect, Joinpoint, Pointcut, Advice, and Introduction [4] are the most important ones. Below, we briefly describe these concepts along with how are they represented in the profile. Figure 1 shows the metamodel representing and relating these concepts. The complete discussion of the AspectSM profile can be found in [10]. We can see from that description that proper modeling requires the modeler to master AOM concepts and mentally determine the end result of weaving; an exercise that cannot be taken for granted and be a priori considered easier than directly modeling crosscutting concerns in a state machine. Investigating the benefits of AspectSM, and more generally AOM, is the main purpose of our experiments.

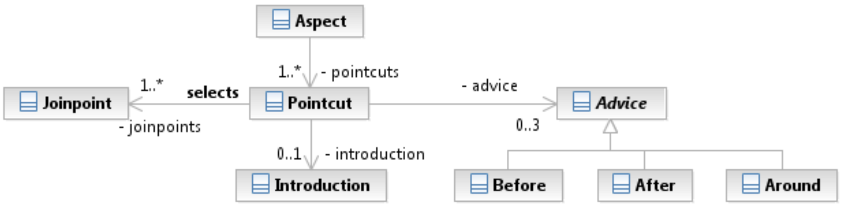


Figure 1. Conceptual domain model of the AspectSM profile

2.2.1 Main Concepts in AspectSM

Aspect. This concept describes a crosscutting concern, e.g., the robustness behavior of a system in the presence of failures in its environment (e.g., network failures in communication systems). Using the AspectSM profile, we model each aspect as a UML 2.0 state machine augmented with stereotypes and attributes.

Jointpoint. A *Joinpoint* is a model element selected by a *Pointcut* (defined next) where an *Advice* or *Introduction* (additional behavior) can be applied [4]. In the context of UML, all modeling elements in UML can be possibly joinpoints. In UML state machines, joinpoints can be, for example, *State*, *Activity*, *Constraint*, *Transition*, *Behavior*, *Trigger*, and *Event*.

Pointcut. A *Pointcut* selects one or more joinpoints, where *Advice* or *Introduction* can be applied. A *Pointcut* can have at most one *Before* advice, one *Around* advice and one *After* advice. In the AspectSM profile, all pointcuts are expressed with the Object Constraint Language (OCL) [16] on the UML 2.0 metamodel [16]. We decided to use the OCL to query joinpoints because the OCL is the standard way to write constraints and queries on UML models and can therefore be used to query jointpoints in UML state machines. Also, several OCL evaluators are currently available that can be used to evaluate OCL expressions such as the IBM OCL evaluator [17], OCLE 2.0 [18], and EyeOCL [19]. Furthermore, writing pointcuts as OCL expressions do not require the modeler to learn a notation that is not part of the UML standard. In the literature, several alternatives are proposed to write pointcuts [20-24] but all of them either rely on languages (mostly based on wildcard characters to select joinpoints, for instance, '*' to select all joinpoints) or diagrammatic notations which are not standard, thus forcing modelers to learn and apply new notations or languages. Using the OCL, we can write precise pointcuts to select jointpoints with similar properties. We do so by selecting modeling elements (jointpoints) based on the properties of UML metaclasses. This further gives us the flexibility to specify precise pointcuts as any condition defined based on some or all of the properties of a UML metaclass, e.g., a pointcut on the *Transition* metaclass, selecting a subset of transitions in a base state machine, which have triggers of type *CallEvent* and do not have any guard.

Advice. An *Advice* is one of the crosscutting behaviors of the *Aspect*. The *Advice* is attached to Joinpoint(s) selected by the *Pointcut*. In correspondence to AspectJ [25] concepts, an *Advice* can be of type *Before*, *After*, or *Around*. A *Before* advice is applied before Joinpoint(s), an *After* advice is applied after Joinpoint(s), whereas an *Around* advice replaces Joinpoint(s). For example, introducing guards on a set of transitions of a state machine is an example of a *Before* advice on transitions (*Joinpoint*).

Introduction. An *Introduction* is similar to the inter-type declaration concept in AspectJ [25]. Using *Introduction* in our context, new modeling elements (e.g., state or transition) can be introduced into a UML state machine.

2.2.2 Example of applying AspectSM

In this section, we present an example of the application of AspectSM. An aspect state machine modeling crosscutting behavior *EmergencyStop* is shown in Figure 2. This UML state machine is stereotyped as <<*Aspect*>>, which means that it is an aspect state machine. The <<*Aspect*>> stereotype has two attributes: *name* and *baseStateMachine*, whose values are shown in the note labeled as '1' in Figure 2. The *name* attribute contains the name of the aspect (*EmergencyStop* in this example), whereas the *baseStateMachine* attribute holds the name of the base state machine, on which this aspect will be woven, which is *ElevatorControl* in this example.

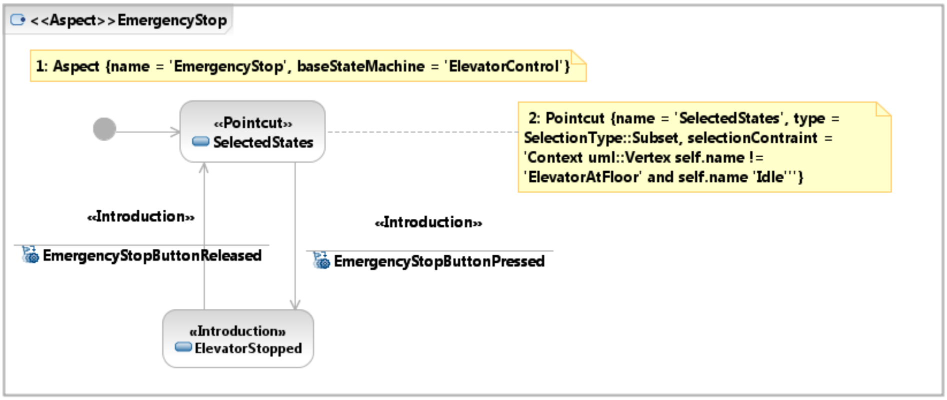

<<Aspect>>EmergencyStop
1: Aspect {name = 'EmergencyStop', baseStateMachine = 'ElevatorControl'}
«Pointcut»
SelectedStates
2: Pointcut {name = 'SelectedStates', type = SelectionType::Subset, selectionContraint = 'Context uml::Vertex self.name != 'ElevatorAtFloor' and self.name 'Idle''}
«Introduction»
EmergencyStopButtonReleased
«Introduction»
EmergencyStopButtonPressed
«Introduction»
ElevatorStopped

Figure 2. An aspect state machine for crosscutting behavior EmergencyStop

The aspect state machine consists of two states: *SelectedStates* and *ElevatorStopped*. *SelectedStates* is stereotyped as <<*Pointcut*>>, which means that this state selects a subset of states from the base state machine. There are three attributes of <<*Pointcut*>>, whose values are shown in the note labeled as '2' in Figure 2. The *name* attribute indicates the name of the pointcut and *type* denotes the type of the pointcut, which is *Subset* in this case. In AspectSM, different types of pointcuts can defined, a complete list of other types of pointcuts is presented in [10]. The third attribute *selectionConstraint* contains a query in OCL on the UML state machine metamodel, which selects all states of the base state machine except *ElevatorAtFloor* and *Idle.* All the model elements stereotyped as <<*Introduction*>> (one state, two transitions) will be newly introduced elements in the base state machine during weaving. This aspect introduces the *ElevatorStopped* state in the base state machine, and selects all states of the base state machines except *ElevatorAtFloor* and *Idle* (via *SelectedStates*) and introduces transitions from them to *ElevatorStopped* with trigger *EmergencyStopButtonPressed*. In addition this aspect introduces transitions from *ElevatorStopped* to all the states selected by *SelectedStates* with trigger *EmergencyStopButtonReleased*.

3. Experiments Planning

This section discusses the planning of the experiments according to the definition and reporting template defined by Wohlin et al. [26].

3.1 Goal, Research Questions and Hypotheses

The objective of our experiments is to assess the AspectSM profile with respect to the readability of resulting UML state machines. Readability will be looked at from three complementary points of view: model comprehensibility, the ease of detecting defects, and the ease of fixing defects for designers inspecting the models. Note that in this paper, we did not study the impact of possible interactions between aspects.

Based on the objective of our experiments, we defined the following four research questions.

- ***RQ1: Does the use of AspectSM lead to better defect identification rate when inspecting state machines as compared to hierarchical and flat state machines?***

We wish to compare the readability of AspectSM with two different types of state machines where crosscutting behavior is modeled directly: hierarchical and flat state machines. None of the expected differences between them can a priori be certain to be in a specific direction. This therefore leads to the definition of two-tailed hypotheses.

H^1_0: The defect identification rate in aspect state machines is the same as that for hierarchical state machines.

H^2_0: The defect identification rate in aspect state machines is the same as that for flat state machines.

- ***RQ2: Does the use of AspectSM lead to better defect fixing rate when inspecting state machines as compared to hierarchical and flat state machines?***

Similar to the previous question, we wish to compare the ease of defect fixing when using AspectSM with two different types of state machines directly capturing crosscutting behavior: hierarchical and flat state machines. Again, none of the expected differences between them can a priori be certain to be in a specific direction, hence leading to the definition of the following two-tailed hypotheses.

H^3_0: The defect fixing rate in aspect state machines is the same as that for hierarchical state machines.

H^4_0: the defect fixing rate in aspect state machines is the same as that for flat state machines.

- ***RQ3: Does the use of AspectSM improve the ease of comprehension when compared to hierarchical and flat state machines?***

Similar to the previous research questions, we wish to compare the comprehensibility of AspectSM with the two different types of state machines directly capturing crosscutting behavior (hierarchical and flat state machines) based on the scores to answer a comprehension questionnaire. We defined the following two-tailed null hypotheses accordingly.

H^5_0: The comprehensibility of aspect state machines is the same as that for hierarchical state machines.

H^6_0: The comprehensibility of aspect state machines is the same as that for flat state machines.

- ***RQ4: Does the use of AspectSM reduce the required inspection effort for defect identification and answering the comprehension questionnaire?***

While the two previous research questions looked at the effectiveness of using alternative models, this research question is concerned with the effort required to inspect crosscutting behavior for defect identification and answering the comprehension questionnaire. This leads again to the following two-tailed null hypotheses:

H^7_0: The effort to identify defects in aspect state machines is the same as that for hierarchical state machines.

H^8_0: The effort to identify defects in aspect state machines is the same as that for flat state machines.

H^9_0: The effort to answer the comprehension questionnaire for aspect state machines is the same as that for hierarchical state machines.

H^{10}_0: The effort to answer the comprehension questionnaire for aspect state machines is the same as that for flat state machines.

3.2 Participants

The first controlled experiment was conducted at the Pir Mehr Ali Shah Arid Agriculture University, Rawalpindi, Pakistan. The subjects in the experiment were 27 graduate students taking a graduate course in 'Advanced Software Architecture' at the University Institute of Information Technology (UIIT). The course is offered in the Master of Science program. The students in this degree already hold a Bachelor in Computer Science or Information Technology and have already been exposed to the UML notation and extensions in the form of UML profiles. On average, each student went through five development and two modeling courses. Eighteen students (out of twenty-five) have used the UML notation for their final year projects before the experiment was conducted. Twenty students gained development experience in IT companies or as teaching staff in computer science.

The replication of the above experiment was conducted at the Beijing University of Aeronautics and Astronautics (BUAA), Beijing, China. The subjects in the replication are 47 graduate students. Half of the students were taking a graduate course titled 'Software Engineering', whereas the remaining half students were taking a graduate course titled 'Software Architecture'. Both courses rely on similar teaching materials and methods and therefore we therefore can assume that all students have a similar education background regarding software engineering. These two courses are offered in the Master of Computer Software and Theory program. The students in this degree already hold a Bachelor in Computer Science and had all already been exposed to UML. On average, each student went through two software development courses and one modeling course. All of the students had at least one year of experience in development work in various industry sectors such as maritime and aerospace. In conclusion, the subjects have roughly the same background, although the subjects were in different years of their study. Their seniority was taken into the consideration while forming experimental groups as we will discuss in Section 3.4.

Our motivation in selecting these two groups of subjects was to find participants with adequate background (e.g., UML modeling) that could be trained to use our AOM approach over a short period of time. Our goal was to assess AspectSM with fully trained, competent participants in order to assess the maximum potential benefits of the approach. Most practitioners have very little knowledge of AOP and even less of AOM. Ensuring they have the required background is also difficult. This is why we relied on a group of mature and trained graduate students. The subjects were free to choose to participate or not into the experiments and were told their choice would have no effect on their course grades. All students underwent a specific, additional training for the experiments (Section 3.7). For the initial experiment, two students decided not to participate in the experiment.

3.3 Material

In this section, we provide details on the material we used for the experiments.

3.3.1 Case Study System

For the initial experiment, we only used an Elevator Control System (ECS), whereas for the replication of the experiment we used a second system as well: Video Conferencing System (VCS). Differences between the initial experiment and replication are summarized in Section 3.8. The complexity of both case study systems is summarized in Table 1 based on the number of modeling elements. For instance, in our context, the complexity is measured based on the number of states and transition in a state machine. In Appendix A, we provide partial models of VCS to illustrate various models specified using different modeling approaches.

Elevator Control System. It controls movements of an elevator in a building. For our experiments, we extended the specification of the elevator given in [11] with two additional crosscutting behaviors so that the AspectSM profile could be used to model them. These two crosscutting behaviors are: 1) Emergency call behavior (*Call*): the behavior of an elevator, when an emergency call is made, and 2) Emergency stop behavior (*Stop*): the behavior of an elevator, when the emergency stop button is pressed. Note that in Table 1 for ECS, in the replication, we improved the design of *Hierarchical* such that there are fewer states and transitions when compared to the design in the initial experiment.

Table 1. Complexity of the state machines modeling the crosscutting behaviors of the case study system

System	Experiment	Crosscutting behavior	Base state machine		Flat Approach		Hierarchical Approach		Aspect Approach		
			# S	# T	# S	# T	# S	# T	# S	# T	# P
ECS	Experiment	Call	12	15	15	27	14	18	16	18	1
		Stop	12	15	15	27	12	15	14	17	1
ECS	Replication	Call	12	15	15	27	17	21	16	18	1
		Stop	12	15	13	23	14	17	14	17	1
VCS	Replication	AQ	5	9	8	17	10	19	8	13	1
		DnD	5	9	6	15	8	20	7	13	1
		Standby	5	9	5	11	5	14	7	13	1

*S: States, T: Transitions, P: Pointcuts

Video Conferencing System: It is a core subsystem of a video conference system called Saturn developed by Cisco Systems, Norway. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels. For the replication, we used a reduced model of Saturn that is related to establishing and disconnecting videoconferences. In addition to the core functionality, we used the following three crosscutting behaviors:

1) *Audio Quality Loss (AQ):* An important robustness behavior of Saturn is to recover from audio quality loss. Whenever Saturn is in a video conference, it checks audio quality after every certain time. If the quality is within threshold it continues the normal operation, otherwise it tries to recover audio quality. If it successfully recovered the audio quality it continues its normal operation, otherwise it restarts the VCS.

2) *Do Not Disturb (DnD):* Whenever the *Do Not Disturb* feature is on, Saturn ignores all incoming calls. If Saturn is already in a call, it will remain in the call, but ignores any new incoming calls.

3) *Standby:* The *Standby* behavior of Saturn becomes active when it is idle for *m* minutes. When any activity is performed on Saturn while it is in Standby mode, it becomes active.

The crosscutting behaviors for both systems can be modeled in three different ways: 1) by applying AspectSM to derive an aspect state machine (*Aspect* Approach), 2) by directly adding states and transitions on the base state machine (*Flat* Approach), 2) by using hierarchical/orthogonal states (*Hierarchical* approach) in order to avoid redundant modeling and reduce complexity to the maximum extent. It is, however, not always possible to use the hierarchical approach successfully. For instance, separating out constraints modeling non-functional properties (e.g., video or audio quality) from state invariants is not possible using hierarchical state machines without introducing accidental complexity and redundancy as we demonstrated in [10]. Information regarding the complexity of the three resulting state machines is provided in Table 1, measured using number of states and transitions for each system. For aspect state machines, we also provide the number of Pointcuts, which also contribute to modeling complexity.

3.3.2 Design Defect Classification

Given that the correctness and completeness of defect identification through inspections are part of our evaluation criteria to compare state machines, experiment participants were asked to identify defects seeded in state machines by checking their conformance against their corresponding specifications (Section 3.4).

To help systematically inspect state machines for various types of defects, a classification of different types of design defects is required. The classification we used in the experiments is given below and was adapted from Binder's book [8]. It was provided to the participants of the experiments as part of the answer sheet (Section 3.3.5) to systematically collect their answers.

Incorrect Transition (IT): A transition that comes from or leads to a wrong state or has an incorrect guard, trigger, and/or event.

Missing Transition (MT): According to the specification, there is a transition missing from the state machine.

Extra Transition (ET): A transition is subsumed by another transition in a state machine. Such a transition is redundant in the sense that removing it still keeps the state machine in conformance to its specification.

Missing State (MS): According to its specification, a state that should be modeled in a state machine but is missing.

Incorrect State (IS): A state is incorrect if it has an incorrect state invariant, do, entry and/or exit activity.

Extra State (ES): A state is subsumed by another state. This state is considered as an extra state in the sense that removing it still keeps the state machine in conformance to its specification.

3.3.3 Seeded Defects

It is important to note that in our experiments, we were interested in studying the readability of crosscutting behaviors since AspectSM is specifically designed for that purpose. Moreover, the readability of other types of behaviors is expected to be the same with or without AspectSM. For these reasons we only seeded defects in the crosscutting behaviors. Different types of defects were selected after we carefully examined the base and aspect state machines and identified possible independent defects. Table 2 shows the distribution of these defects that were seeded in the compared state machines. Note that seeded defects in ECS are different for the initial experiment and its replication since we improved the models in the latter.

Table 2. Distribution of seeded defects in state machines

Experiment	System	Crosscutting Behavior	Aspect				Hierarchical				Flat			
			MT	IT	MS	IS	MT	IT	MS	IS	MT	IT	MS	IS
Experiment	ECS	Stop	1	-	-	-	1	-	-	-	10	-	-	-
		Call	1	1	-	1	4	2	-	1	11	9	-	-
Replication	ECS	Stop	1	-	-	-	1	-	-	-	10	-	-	-
		Call	1	1	-	1	1	1	-	1	10	10	-	1
	VCS	AQ	1	1	1	-	1	1	1	-	1	4	1	-
		DnD	-	2	1	1	-	2	1	1	-	8	1	1
		Standby	2	1		1	-	-	-	-	2	1	-	1

Because aspects model crosscutting behavior, it is expected that one defect in an aspect often corresponds to several defects in the corresponding hierarchical state machine. Similarly, because hierarchical states factor out common behavior, one defect in a hierarchical state machine often leads to several defects in its corresponding flat state machine. As a result, different numbers of defects were seeded in the three state machines in order to conceptually correspond to equivalent defects and have semantically equivalent models. Note that in Table 2, a *'-'* indicates that we didn't seed defects from a particular defect class (e.g., MT, IT).

3.3.4 Comprehension Questionnaire

As we discussed above, we also want to compare how easy it is to comprehend the various types of state machines. To this effect, a comprehension questionnaire (Appendix B) was designed to evaluate, in a repeatable and objective way, the extent to which a subject can understand the state machines. For example, some questions concern what scenario is triggered when an event happens in a certain state. The subjects were asked the same ten questions on crosscutting behaviors together for all three state machines. Participants had to answer each question by inspecting the state machine assigned to them and correctness scores were computed by accounting for partially correct answers. For example, if the answer to a question entailed to list four transitions, then pointing out each correct transition contributed 0.25 to the full mark of the question.

3.3.5 Answer Sheets

Three answer sheets were developed to collect answers for three readability measures (defect identification, defect fixing, and comprehension). The first answer sheet was developed to collect information about classes of defects that were identified by each subject, the number of defects in each class, and the location of identified defects. A table was provided to the subjects for each crosscutting behavior. The rows of the table were labeled with each defect class, whereas the columns featured two pieces of information about defects: number of defects identified in each class and location of each identified defect. The second answer sheet was developed to collect the state machine corrected by the subjects. The third answer sheet was designed to collect answers to the comprehension questionnaire.

3.3.6 Pre- and Post-Lab Questionnaire

In addition to the answer sheets, we designed pre- and post-lab questionnaires to obtain subjective opinions of the subjects on various issues. Each question in each questionnaire is a four-point Likert scale [27] question with the following meaning: one (completely agree), two (generally agree), three (generally disagree), and four (completely disagree). The pre-lab questionnaire asked questions about training, experience, and confidence of a subject about a particular modeling approach he/she was assigned. In the post-lab questionnaire, we asked questions about the applicability of the approach and level of confidence of the subjects about their solutions.

3.4 Design

In this section, we present the design of the initial experiment and its replication. In the initial experiment, we used a between-subjects design for reasons discussed in Section 3.4.1, whereas in the replication, we used both between-subjects and within-subjects designs for each of the two rounds, respectively (Section 3.4.2).

3.4.1 Design of the Initial Experiment

The design of our experiment is summarized in Table 3. Our experiment design consists of two rounds and there were three groups denoted Group 1, Group 2, and Group 3. Given the number of the subjects, this led respectively to 8, 8, and 9 subjects in each group. In each round, one group was given a different type of state machines (*Aspect*, *Hierarchical*, or *Flat*). During the training sessions (Section 3.7), each subject was equally trained to understand the three different types of state machines: *Aspect*, *Flat*, and *Hierarchical*. The subjects were also given a modeling assignment, after the training sessions, for them to practice before the actual experiment tasks. This assignment was marked by the first author of this paper and grades were used to form blocks (i.e., groups of students of equivalent skills). The experiment groups were then formed through randomization and blocking to obtain three comparable groups with similar proportions of students from each skill block.

The two rounds of the experiments were conducted in sequence on the same day.

This initial experiment used a between-subjects design, where different groups of subjects are compared when using different state machine modeling techniques. As shown in Table 3, in the first round, each group was asked to identify defects in two separate tasks corresponding to the *Call* and *Stop* crosscutting behaviors. Group 1 was given state machines modeled using the *Aspect* approach. The subjects in Group 1 were given one base state machine and one aspect state machine modeling *Call* in *Task 1*, whereas in *Task 2*, Group 1 was given the same base state machine and one aspect state machine for the *Stop* crosscutting behavior. Group 2 was given one hierarchical state machine for *Call* and one hierarchical state machine for *Stop* for *Task 1* and *Task 2*, respectively. Similarly, Group 3 was given one flat state machine for *Call* and one flat state machine for *Stop* for *Task 1* and *Task 2*, respectively. Seeded defects for each type of state machines (*Aspect*, *Hierarchical*, and *Flat*) are presented in Table 2. For each task, the subjects were allowed to take as much time as they needed, but when they finished the first task, their answer sheets for this task were collected and then they were handed the description of the second task and a new answer sheet. The starting and completion times were noted on each answer sheet by the subjects and were checked for correctness by the instructors while collecting the solutions.

For the second round, the three groups were rotated: Group 1 was asked to answer comprehension questionnaire for flat state machine, Group 2 for aspect state machines, and Group 3 for hierarchical state machines. This rotation was performed only for pedagogical reasons such that each group can be exposed to a different type of state machines than the previous round. However, since we had only two tasks due to time constraints, it was not possible for all of the groups to experience all three approaches. The starting and completion times for this task were collected following the same procedure as for Round 1.

Table 3. Design of the Initial Experiment

Round	Case study	Crosscutting behavior	Task	Group 1	Group 2	Group 3
1	ECS	Stop	DI	A	H	F
		Call		A	H	F
2		Stop and Call	AC	F	A	H

* DI: Defect Identification, AC: Answer Comprehension Questionnaire, A: Aspect, H: Hierarchical, and F: Flat

3.4.2 Design of the Replication

The design of the replication is summarized in Table 4. Our replication design consists once again of two rounds (Round 1 and Round 2) and each round was conducted on a separate day. During the training session (Section 3.7), each subject was equally trained to understand the three different types of state machines: *Aspect*, *Flat*, and *Hierarchical*. The subjects were divided to form blocks (i.e., groups of students of equivalent skills) based on their seniority in their graduate programs. The groups were then formed through randomization and blocking to obtain three comparable groups with similar proportions of students from each skill block. We divided the subjects into three groups: Group 1, Group 2, and Group 3. For Round 1, there were 17, 15, and 15 subjects, respectively. For Round 2, due to practical reasons such as time clash with courses and exams, fewer students participated than in Round 1. In Round 2, we had 14, 10, and 15 in Group 1, 2 and 3, respectively.

In Round 1, the ECS system and a between-subjects [26] design were used. We did not have a third crosscutting behavior to opt for a balanced, within-subjects design, as for the second round that is described next. Every participant was exposed to only one modeling approach. Group 1 was given state machines modeled using the *Aspect* (*A*) approach, Group 2 with the *Hierarchical* (*H*) approach and Group 3 with the *Flat* approach (*F*).

In Round 2, regarding detecting and fixing defects, we used a within-subjects design [26] since we have three crosscutting behaviors and three treatments (*Aspect*, *Hierarchical*, or *Flat*). A within-subjects design offers two main advantages. First, with this type of design, we can reduce the error variance due individual differences in human performance, which is quite common in software engineering tasks. This is due to the fact that the same group of students is exposed to all modeling approaches across the different crosscutting behaviors (e.g., *Call* and *Stop*). Second, within-subjects designs provide more statistical power as compared to a between-subjects design [26] as it leads to more observations for each treatment. Potential threats from within-subjects designs are "carryover" effects. To address this, for each of the three crosscutting behaviors, each group was given a different treatment in such a way that ordering effects were counterbalanced: each of the three modeling approaches occurred once in a different order across the three groups. For example, as shown in Table 4, for aspect *DnD*, each group was asked to detect and fix defects and Group 1, Group 2 and Group 3 were given treatment *Aspect*, *Hierarchical*, and *Flat*, respectively. For *Standby*, the three groups were rotated: Group 1 was asked to identify and fix defects for flat state machines, Group 2 used aspect state machines, and Group 3 used hierarchical state machines. Similarly, the groups were rotated again for *AQ*.

With a within-subjects design, a matched pair analysis can be applied by comparing the performance of subjects with themselves across treatments.

In both rounds, the subjects were presented with all three crosscutting behaviors together and were asked to answer questions from a comprehension questionnaire for one type of state machine. For each crosscutting behavior, the subjects were given a fixed time as shown in Table 4. Fixing the time for task execution tends to yield more differences in task effectiveness, but then results cannot be used to study time differences across treatments [26]. Note that in the replication, we ordered the crosscutting behaviors based on their complexity (Table 1) from simple to complex, in order to enable the subjects to tackle increasingly more complex models and thus smooth the learning curve.

Table 4. Design of the Replication[*]

Round	Case study	Aspect	Task	Group 1	Group 2	Group 3	Time (min)
1	ECS	Stop	DI	A	H	F	15
			DF				15
		Call	DI	A	H	F	15
			DF				15
		Stop and Call	AC	A	H	F	30
2	VCS	DnD	DI	A	H	F	15
			DF				15
		Standby	DI	F	A	N/A	15
			DF				15
		AQ	DI	H	F	A	15
			DF				15
		DnD, Standby, AQ	AC	A	H	F	30

* DI: Defect Identification, DF: Defect Fixing, AC: Answer Comprehension Questionnaire, A: Aspect, H: Hierarchical: and F: Flat

3.5 Dependent Variables

Defect Identification Rate (DIR) and Defect Fixing Rate (DFR). These variables capture whether a subject accurately identifies/fixes seeded defects. Based on the information collected in the answer sheet described in Section 3.3.5, there are several different ways to measure DIR and DFR, which we discuss below.

1) *Average DIR/DFR*

   For each type of defects, Average DIR/DFR (DIR_Average/DFR_Average) is measured as the percentage of identified/fixed defects over the total number of seeded defects:

number of identified or fixed defects / total number of seeded defects

2) *DIR and DFR on binary scale with minimum defect identification and fixing*

   As discussed in Section 3.3.1, one defect seeded in aspect state machines may correspond to more than one defect in hierarchical or flat state machines. Therefore, to allow for a meaningful combination of observations across tasks and state machines, we use a binary measure indicating whether at least one defect was found (*DIR_Binary*) or fixed (*DFR_Binary*). As long as at least one defect is identified/ fixed in a given task by a subject in hierarchical and flat state machines, value 1 is assigned to *DIR_Binary/DFR_Binary*. For example, as shown in Table 2, the flat state machine modeling the *Call* crosscutting behavior contains 10 MT defects, 10 IT defects, and one IS defect. If at least any one of these defects is identified by a subject, then *DIR_Binary* = 1; otherwise *DIR_Binary* = 0. It is important to note that we developed this measure such that comparisons across the three approaches are made possible. This is due to the reason that different numbers of defects are introduced in three types of state machines corresponding to a single defect in a crosscutting behavior.

3) *DIR and DFR on binary scale with maximum defect identification and fixing*

   This measure (*DIR_Binary_Max/DFR_Binary_Max)* is a variation of *DIR_Binary/DFR_Binary*−which is also comparable across state machines−and is assigned value 1 when all defects seeded in a crosscutting behavior are identified/fixed by a subject in a task. For instance, in Table 2, the hierarchical state machine modeling the *Call* crosscutting behavior has 10 MT defects. *DIR_Binary_Max* = 1, if all these defects are identified by a subject, otherwise it is assigned 0. In comparison with the measure *DIR_Binary*, this measure is stricter in the sense that it requires all the seeded defects in each of the three types of state machines to be identified to obtain a value 1. None of these measures are perfect but such binary measures are necessary to combine all observations in one data set. We will interpret differences in results of binary measures if they arise. The purpose of defining this measure is the same as for the previous measure: render possible comparisons across the three approaches, but in a different way.

Score of the responses to the comprehension questionnaire (SCQ). Correctness of the responses to the comprehensive questionnaire is calculated as follows:

Sum of scores of all questions / 10

In the formula above, the score for each question is calculated based on the marking procedure discussed in Section 3.3.4 and 10 is the total number of questions in the questionnaire.

Required effort (Effort). Required effort is measured in minutes taken by a subject to identify defects in each crosscutting behavior. Similarly, we also measure effort in minutes taken by a student to answer comprehension questionnaire. It is simply measured as Completion time – Starting time.

Table 5 summarizes which dependent variables are used to answer research questions presented in Section 3.1.

3.6 Data Collection

For the initial experiment, the solutions were collected from the subjects and were marked by the first author of this paper. In the replication, the solutions were marked by the second author of this paper. The data was encoded into a JMP [28] data file to perform the statistical analysis.

Table 5. Dependent variables corresponding to each research question

Research Question	Dependent Variables
RQ1	DIR_Average, DIR_Binary, DIR_Binary_Max
RQ2	DFR_Average, DFR_Binary, DFR_Binary_Max
RQ3	SCQ
RQ4	Effort

For the experiment, data integrity was checked using the following rule: for the same subjects and for each step, the starting time should precede the completion time, and the completion time of the current task must precede the starting time of the next task. For the replication, since the time for each task was fixed (Section 3.4.2), the answer sheets for a task were collected before handing over the next task to the subjects to ensure that the each subject used exactly the same time. In addition, to avoid mistakes in marking the solutions, the first two authors double-checked the solutions marked by the other. Moreover, for a sample of randomly selected solutions, the first two authors also checked the consistency of the entries in the JMP file with the marks on the answer sheets and no inconsistencies were detected.

3.7 Training

In the initial experiment, the subjects were trained by the first author of this paper. Two three-hour sessions were given on the following topics: 1) Recap of UML state machines since the subjects were already familiar with this topic preceding the training (Section 3.2), 2) Introduction to the Object Constraint Language (OCL), 3) Introduction to aspect-oriented software development (AOSD), and 4) Aspect-oriented modeling (AOM) using the AspectSM profile. Each topic was accompanied with several examples and interactive class assignments. As previously discussed, the subjects were given a home assignment after the training sessions to practice the three state machine modeling approaches and groups were later formed based on the grades of this assignment.

For the replication, the subjects were trained by the second author of this paper. One three-hour session was given on the same topics as the ones used in the initial experiment. However, in this case, there were no class assignments given to the students due to practical constraints.

3.8 Replication

There are several potential reasons why replications of experiments are necessary in software engineering [29]. Our replication was motivated by the following reasons: 1) to reduce the validity threats that were observed in the intitial experiment, 2) to increase the sample sizes and improve the statistical power of results, 3) to address the problems identified in the design and material. The differences between the initial experiment and its replication are summarized below:

3.8.1 Reduced External Validity Threats

In the replication, we reduced external validity threats by doing the following. 1) We added an additional case study, which is a reduced version of an industrial videoconferencing system developed by Cisco, Norway. In addition, we included three real crosscutting behaviors of the videoconferencing system. 2) We replicated the experiment in a different geographical area with graduate students from a different education system.

3.8.2 Improved hierarchical modeling of ECS

The ECS system was used in both the initial experiment and its replication. For the replication, we improved the design for *Hierarchical*. The *Stop* crosscutting behavior of ECS in the replication is modeled with a reduced number of modeling elements as compared to its design in the initial experiment.

3.8.3 Improved Assignments of Subjects to Treatments

In the initial experiment (Section 3.4.1), we rotated the groups for two tasks (defect identification and answering comprehension questionnaire) such that each group can inspect the state machines modeled with a different approach. Though this rotation was done for pedagogical reasons, since we had only two tasks for ECS (Section 3.3.1), not all of the groups could experience state machines modeled with all three approaches. In the replication, in contrast, we used a within-subjects design for the VCS system, where each group was exposed to all treatments exactly once. As discussed above, this also led to higher statistical power and a reduction in variance associated with individual differences by enabling the use of matched pair analysis.

3.8.4 Coverage of UML Features

In the initial experiment, we covered most of the advanced features of UML state machines such as composite states and submachine states, but we didn't cover orthogonal states. In the replication, we included a crosscutting behavior with orthogonal states (*Call* in ECS).

3.8.5 *Other Differences*

In the initial experiment, we measured readability from two perspectives: defect identification and answering a comprehension questionnaire. In the replication, we added another perspective: defect fixing. In the initial experiment, we gave subjects as much time as they wanted to perform each task. The results did not, however, reveal any significant differences between various approaches in terms of time (Section 4.4). In the replication, we fixed the time for each task and this expectedly led to most subjects using most of the allocated time. As expected, the differences across treatments, if any, are in such a context only visible in terms of effectiveness (e.g., defect identification/fixing rates) [26].

3.9 Overview of Statistical Tests

In this section, we provide justifications for the statistical tests run for our data analysis.

3.9.1 Statistical Tests

Using statistical testing, we check whether the differences between modeling approaches are statistically significant to determine if we can reject the null hypotheses stated in Section 3.1. For all statistical tests reported in this section, we used a significance level of α=0.05, though exact p-values are also reported. To check if, overall, there exist significant differences among the three approaches under investigation, we performed the one-way ANOVA test [30] on each dependent variable defined on an interval scale, i.e., *DIR_Average, DFR_Average, SCQ, and Effort*. Our samples for all dependent variables meet all assumptions of the ANOVA test, which are as follows: 1) the samples should be approximately normal, 2) the samples must be independent, and 3) variances of populations must be equal. To check for normality, we performed the Shapiro–Wilk W test [30] for each dependent variable. The results showed that their distributions do not strongly depart from normality. The second assumption also holds since our samples are collected on different groups of the subjects, working independently. To check the equivalence of variances, we performed the Bartlett's test [30] which showed that the variances across samples are equal for all dependent variables. In addition to the one-way ANOVA test, we also performed the Kruskal–Wallis one-way analysis of variance test [30], which is a non-parametric equivalent of the one-way ANOVA test. The results of both tests turned out to be consistent.

For those dependent variables for which one-way ANOVA results were significant, we performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30], which is the ANOVA post-hoc test. As an adaptation of *t*-test, the Tukey_Kramer HSD test is designed to handle the increase in Type-I error resulting from multiple comparisons. It assumes normally distributed samples and requires samples of equal or comparable size, or otherwise yield conservative results [30]. We have (nearly) equal sample sizes (see Section 3.4) and our dependent variable distributions do not strongly depart from normality as the results of the Shapiro–Wilk W test [30] showed. We also report the mean differences between pairs of approaches indicating the direction in which the result is significant. We also performed the Wilcoxon Signed-Rank test [30], which is a non-parametric equivalent of Tukey_Kramer HSD. The results of both tests were consistent.

For Round 2 in the replication, since our design is a within-subjects design, we performed the matched pairs *t*-test, in addition to the one-way ANOVA and pair-wise comparisons with Tukey-Kramer HSD since matched pairs analysis improves statistical power over independent sample testing, as discussed in Section 3.4. In our context, a pair is the same student performing the same type of task (e.g., defect identification) on different crosscutting behaviors (e.g., *DnD* and *Standby*) on state machines designed with different approaches (e.g., *Aspect* and *Hierarchical*). We double checked the results of the matched pairs *t*-test with a Wilcoxon matched pairs test, which is an equivalent, non-parametric test. The results of the tests turned out to be consistent. Since the results of both parametric and non-parametric tests are consistent, we only report the results of the parametric tests in this paper.

For *DIR_Binary, DFR_Binary, DFR_Binary_Max,* and *DIR_Binary_Max*, we performed the Fisher's exact test [30] to compare the defect identification/fixing proportions for the various state machines. These four measures are binary and observations can be therefore classified into two categories (either 0 or 1 showing 'not found' or 'found', respectively), which is exactly what the Fisher's exact test is designed for. For these binary variables, for Round 2 in the replication, since our design is a within-subjects design, we also performed the McNemar's Test [31] for matched pairs analysis. This test is specifically designed for matched pairs analysis of binary data.

We performed all the tests mentioned in this section using JMP [28] except for the McNemar's Test [31], for which we used the web-based application [31].

3.9.2 *Power Analysis*

Power analysis can be used during the design stage of an experiment to determine how many subjects are likely to be needed, or after the fact to help interpret non-significant results. The latter may be due to small samples sizes and effect sizes that are smaller than expected. Power analysis is particularly important for controlled experiments in software engineering that involve human subjects, as they normally suffer from small sample sizes because of the limited availability of trained subjects and the high cost of conducting experiments. In our context, like in most software engineering experiments, the number of subjects is imposed by external constraints and a retrospective power analysis, as suggested in [32], helps interpret non-significant results in such conditions. For each statistical test considered, such an analysis estimates the minimum effect size at which we can observe an acceptable level of power (typically 80%). This means that above that minimum, we can probably interpret a non-significant result as an absence of effect. Below this threshold the effect might be present but remain undetected.

In our experiments, we are interested in comparing the *Aspect* approach to *Hierarchical* and *Flat* approaches. We perform power analysis for the dependent variables that did not yield significant results and followed the method of calculating power as reported in [32], which requires a fixed sample size, a set significance level (0.05) and power level (80%), and uses the observed variance to calculate the corresponding, minimum effect size. We didn't use standardized effect sizes as suggested by Cohen [33] since those cannot be easily interpreted in a software engineering context.

4. Results and Discussion

We analyze and present our experiments results in this section. We present the results for the four research questions in Section 4.1, Section 4.2, Section 4.3, and Section 4.4, respectively. Within each section, we provide results for both the initial experiment and its replication, and a plausible explanation of the results. In Section 4.5, we provide concluding remarks on the results and discussions.

4.1 Results and Analysis for Defect Identification (RQ1)

In this section, we report results for RQ1 presented in Section 3.1. As shown in Table 5, we will answer this research question based on the *DIR_Average*, *DIR_Binary*, and *DIR_Binary_Max* dependent variables, for both the initial experiment (Section 4.1.1) and its replication (Section 4.1.2). We provide individual discussions of the results for each experiment in Section 4.1.3 followed by an overall discussion (Section 4.1.4).

4.1.1 Results for the Initial Experiment

Regarding *DIR_Average*, from Table 6 we can observe higher values for *Aspect* than for *Hierarchical* and *Flat*. More specifically, the *Aspect* group performed 56% and 62% better than the *Hierarchical* and *Flat* groups. These results show that it is easier to correctly detect the defects seeded in aspect state machines than in the flat and hierarchical state machines. The most plausible explanation is that the number of model elements (Section 3.3.1) for aspect state machines is lower than in the other two types of state machines (Table 1) and complexity of pointcuts written as OCL queries does not override this effect. In addition, we checked whether the differences observed for *DIR_Average* are statistically significant to determine if we can reject the null hypotheses stated in Section 3.1. As shown in Table 7, we observed significant differences for *DIR_Average*. Since the results were statistical significant, we further performed Tukey_Kramer HSD for a pair-wise comparison of modeling approaches. The results showed that *Aspect* significantly outperformed both *Flat* and *Hierarchical* in terms of *DIR_Average* as p-values are lower than α (Table 8).

Table 6. Descriptive statistics for various DIR measures

Experiment	System	Measure	Crosscutting Behavior	Approach		
				Aspect	Hierarchical	Flat
Experiment	ECS	DIR_Average	Call and Stop	0.81	0.25	0.19
		DIR_Binary		0.94	0.56	0.67
		DIR_Binary_Max		0.69	0.13	0.28
Replication	ECS	DIR_Average	Stop	0.29	0.46	0.40
		DIR_Binary		0.29	0.46	0.73
		DIR_Binary_Max		0.29	0.46	0.06
		DIR_Average	Call	0.27	0.53	0.16
		DIR_Binary		0.52	0.93	0.73
		DIR_Binary_Max		0.05	0.06	0
		DIR_Average	Call and Stop	0.28	0.5	0.28
		DIR_Binary		0.41	0.7	0.73
		DIR_Binary_Max		0.17	0.26	0.03
	VCS	DIR_Average	DnD	0.30	0.1	0.18
		DIR_Binary		0.71	0.4	0.6
		DIR_Binary_Max		0	0	0
		DIR_Average	Standby	0.6	-	0.33
		DIR_Binary		0.6	-	0.64
		DIR_Binary_Max		0.6	-	0
		DIR_Average	AQ	0.25	0.19	0.26
		DIR_Binary		0.66	0.57	1
		DIR_Binary_Max		0	0	0
		DIR_Average	DnD, Standby, and AQ	0.36	0.15	0.25
		DIR_Binary		0.66	0.5	0.71
		DIR_Binary_Max		0.15	0	0

For *DIR_Binary* as shown in Table 6, for *Aspect*, 93.7% of the subjects managed to catch at least one defect from any of the defect types seeded in both tasks. This is 37.5% and 27% higher than for *Hierarchical* and *Flat*, respectively. For *DIR_Binary_Max*, we observed a pattern similar to *DIR_Binary* for both tasks, as shown in Table 6. *DIR_Binary_Max* is higher for *Aspect* than that of *Hierarchical* and *Flat*, i.e., for the *Aspect* group, 68.7% of the subjects managed to find all the defects seeded in both tasks, which is 56.2% more than for the *Hierarchical* group and 40.9% more than for *Flat* (see Table 6). As we discussed in Section 3.9, we performed the Fisher's exact test to check statistical significance of difference in binary variables and the results are provided in Table 9. For *DIR_Binary*, *Aspect* significantly outperformed *Hierarchical*, but there were no significant differences observed for *Aspect* vs *Flat*. In the case of *DIR_Binary_Max*, *Aspect* significantly outperformed both *Hierarchical* and *Flat*.

Table 7. Results for one-way ANOVA for DIR_Average

Experiment	**System**	**Crosscutting behavior**	**p-value**
Experiment	ECS	Call and Stop	**0.0001**
Replication	ECS	Stop	0.55
		Call	**0.003**
		Stop and Call	**0.04**
	VCS	DnD	0.10
		Standby	0.12
		AQ	0.64
		DnD, Standby, and AQ	**0.02**

Table 8. Comparisons of all pairs for DIR_Average using Tukey_Kramer HSD

Experiment	**System**	**Crosscutting Behavior**	**Aspect vs Hierarchical**		**Aspect vs Flat**	
			Mean Difference (Aspect - Hierarchical)	**p-value**	**Mean Difference (Aspect - Flat)**	**p-value**
Experiment	ECS	Stop and Call	0.36	**0.02**	0.44	**0.005**
Replication	ECS	Call	-0.25	**0.04**	0.11	0.48
		Stop and Call	-0.21	**0.03**	0.003	0.99
	VCS	DnD, Standby, and AQ	0.20	**0.01**	0.10	0.27

Table 9. Two tailed Fisher's exact test for DIR binary measures at α=0.05

Experiment	**System**	**Measure**	**Aspect vs. Hierarchical**		**Aspect vs. Flat**	
			Difference in proportion (Aspect - Hierarchical)	**p-value**	**Difference in proportion (Aspect - Flat)**	**p-value**
Experiment	ECS	DIR_Binary	0.375	**0.03**	0.27	0.09
		DIR_Binary_Max	0.56	**0.003**	0.40	**0.03**
Replication	ECS	DIR_Binary (Stop)	-0.17	0.46	-0.43	**0.03**
		DIR_Binary_Max (Stop)	-0.17	0.46	0.22	0.17
		DIR_Binary (Call)	-0.40	**0.01**	-0.20	0.29
		DIR_Binary_Max (Call)	-0.0007	1	0.05	1
		DIR_Binary	-0.28	**0.02**	-0.32	**0.01**
		DIR_Binary_Max	-0.09	0.54	0.14	0.10
	VCS	DIR_Binary (DnD)	0.31	0.21	0.11	0.69
		DIR_Binary_Max (DnD)	0	-	0	-
		DIR_Binary (Standby)	-	-	-0.04	1
		DIR_Binary_Max (StandBy)	-	-	0.6	**0.001**
		DIR_Binary (AQ)	0.09	0.71	-0.33	0.06
		DIR_Binary_Max (AQ)	0	-	0	-
		DIR_Binary	0.16	0.28	-0.05	0.80
		DIR_Binary_Max	0.15	0.07	0.15	**0.02**

4.1.2 Results for the Replication

In this section, we provide results for the replication for defect identification. First, we provide the results for the ECS followed by the results for VCS.

Results for the ECS system. Table 6 shows descriptive statistics for various measures of *ECS*. For *Stop*, *DIR_Average* for *Hierarchical* (0.46) and *Flat* (0.40) is better than *Aspect* (0.29). For *Call*, again *Hierarchical* has higher *DIR_Average* (0.53) than *Aspect* (0.27). However, in this case *Aspect* has higher *DIR_Average* than *Flat* (0.16). For *Stop* and *Call* together *Hierarchical* has higher *DIR_Average* (0.53) than *Aspect* and *Flat*, and *DIR_Average* is tied between *Aspect* and *Flat*. For *Stop*, *DIR_Binary* is higher (0.73) for *Flat* than *Hierarchical* and *Aspect*, which are 0.46 and 0.29 respectively. *DIR_Binary* of *Call* for *Hierarchical* (0.93) is higher than *Aspect* (0.52) and *Flat* (0.73), respectively. For *Call* and *Stop* together, *Flat* (0.73) has higher *DIR_Binary* than *Hierarchical* (0.7) and *Flat* (0.41). For *DIR_Binary_Max* in *Stop*, *Hierarchical* (0.46) outperformed *Aspect* (0.15) and *Flat* (0.40), but for *Call*, *Hierarchical* and *Aspect* show values for *DIR_Binary_Max* of 0.06 and 0.5, respectively, whereas *Flat* has *DIR_Binary_Max* of 0. For *Stop* and *Call* together, *Hierarchical* is better than both *Aspect* and *Flat*.

In addition, we checked the statistical significance of DIR_Average using one-way ANOVA, as discussed in Section 3.9. Table 7 shows the ANOVA results for ECS, where the p-values are made bold when below than our chosen significance level (0.05). For ECS, we observed significant differences in *DIR_Average* for Call and Stop and Call together. We then performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30]. The results are presented in Table 8. For *DIR_Binary* and *DIR_Binary_Max*, we performed the two-tailed Fisher's Exact test, whose results are also summarized in Table 9.

Results for the VCS System. Table 6 shows the descriptive statistics for various measures of VCS. For *DnD*, *Aspect* outperformed both *Hierarchical* and *Flat* for *DIR_Average* and *DIR_Binary*; however, *DIR_Binary_Max* is 0 for all groups. Note that we could not model the *Standby* crosscutting behavior with *Hierarchical*. Again, for *Standby*, *Aspect* outperformed *Flat* for *DIR_Average* and *DIR_Binary_Max*, whereas we observed the reverse for *DIR_Binary* (Table 6). In case of *DIR_Average* and *DIR_Binary* for *AQ*, *Flat* outperformed *Aspect*, which in turn outperformed *Hierarchical*. For all three crosscutting behaviors together, *Aspect* outperformed *Hierarchical* and *Flat* in terms of *DIR_Average* and *DIR_Binary_Max*, whereas for *DIR_Binary*, *Flat* (1.0) outperformed *Aspect* (0.66), which in turn outperformed *Hierarchical* (0.57).

In addition, the ANOVA results (Table 7) showed significant differences in *DIR_Average* with Call and Stop and Call together. We therefore performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30]. The results of the test are reported in Table 8. For VCS, in addition we performed the matched pairs *t*-test (Section 3.9) as reported in Table 10. We observed that *Aspect* significantly outperformed *Hierarchical* and *Flat* with p-values of 0.002 and 0.02 (Table 10), respectively. Hence, it shows that *Aspect* has a high likelihood of having higher *DIR_Average* than both *Flat* and *Hierarchical*. For *DIR_Binary* and *DIR_Binary_Max*, we performed the two-tailed Fisher's Exact test, whose results are also summarized in Table 9. The results of the McNemar's test for matched pairs analysis of these binary measures are reported in Table 10. For DIR_Binary_Max, *Aspect* significantly performed better than both *Hierarchical* and *Flat*. For DIR_Binary, *Aspect* significantly outperformed *Hierarchical*, but *Flat* significantly performed better than *Aspect*.

Table 10. Results of the matched pairs for VCS at α=0.05 for various DIR measures

Measure	Test	Pair of approaches	Mean Difference	p-value
DIR_Average	*t*-test	Aspect-Hierarchical	0.27	**0.002**
DIR_Average		Aspect-Flat	0.19	**0.02**
DIR_Binary	McNemar's test	Aspect-Hierarchical	0.16	**0.03**
DIR_Binary		Aspect-Flat	-0.05	**0.02**
DIR_Binary_Max		Aspect-Hierarchical	0.15	**0.001**
DIR_Binary_Max		Aspect-Flat	0.15	**5.42e-07**

4.1.3 Discussion

In this section, we discuss the results reported in Section 4.1.1 and Section 4.1.2. First, we provide discussion of the results for each experiment individually followed by an overall discussion.

Analysis of Results for the Initial Experiment. Based on the experiment results reported in Section 4.1.1, we conclude that overall, *Aspect* state machines are better than *Flat* and *Hierarchical* ones in terms of the overall defect identification rate, even though the difference between *Aspect* and *Flat* for one of the binary measures (*DIR_Binary*) is not statistically significant given our selected α (0.05) and sample size. One reasonable explanation is that, when compared with flat and hierarchical state machines, aspect state machines are much less complex in terms of number of states and transitions (Table 1); therefore it is expected to be much easier to identify defects in aspect state machines. It is also interesting to note that the additional complexity introduced by pointcuts in *Aspect* does not have any visible negative effect on defect identification.

We further analyzed non-significant results using the power analysis reported in Table 11. The table shows the estimated effects size thresholds corresponding to 80% power for *DIR_Binary* (*Aspect* vs *Flat*) that yielded non-significant results in the previous section (*Minimum effect size*). This means that for effect sizes less than these thresholds, power is less than 80% thus entailing a significant risk of error (type II) in not rejecting the null hypotheses. In other words, for effect sizes below these thresholds, we cannot draw conclusions with confidence from the statistical test results in Table 11. The *Average* column in Table 11 shows the average values for the dependent variables, when combining all the observations being compared. The last column shows the percentage of *Average* that corresponds to the minimum effect size. The result of power analysis for *DIR_Binary* regarding *Aspect* vs *Flat* (Table 11) shows an estimated effect size of 0.20 (24% of average) to achieve 80% power. The observed effect size is 0.14, which is lower than this estimated effect size thus explaining the lack of significance. This suggests that we need to collect more observations, if we want to draw conclusions with confidence for effect sizes below 24% of the average, regarding which approach (*Aspect* or *Flat*) is better in terms of *DIR_Binary*.

Table 11. Estimation of the effect size corresponding to 80% power for ECS[*]

Experiment	Measure	p-value	Observed Effect Size	Minimum Effect Size	Average	Minimum Effect Size/Average
Experiment	DIR_Binary (A vs F)	0.09	0.14	0.20	0.80	0.24
Replication	DIR_Average (A vs F)	0.99	0.001	0.13	0.29	0.46
	DIR_Binary_Max (A vs H)	0.54	0.04	0.15	0.22	0.69

* A: Aspect, H: Hierarchical, F: Flat

Analysis of Results for the Replication. In this section, we provide a discussion on DIRs for each crosscutting behavior individually and all crosscutting behaviors together for the replication. Recall that DIRs are measured with three dependent variables: *DIR_Average*, *DIR_Binary*, and *DIR_Binary_Max*. Results for all those variables for which the results were statistically significant are summarized in Table 12. The first column lists the dependent variables which are used to answer RQ1 (Table 5). The second column represents a pair of approaches being compared and each dependent variable has two rows in this column: *A>X* and *X>A* denoting whether *Aspect* (A) is significantly better than *Hierarchical* or *Flat* (X), and *Hierarchical* or *Flat* (X) are significantly better than *Aspect* (A), respectively. The third column (labeled "Crosscutting Behavior (X)") presents two pieces of information: 1) name(s) of the crosscutting behavior(s) for which the results were significant for ECS, 2) name of the approach in brackets against which the results were significant, i.e., the approach is either significantly better than *Aspect* if it is in the row X>A or vice-versa if it is in the row A>X. For example, in case of *DIR_Average* in the row labeled *"X>A"*, *Call (H)* means that *Hierarchical* is significantly better than *Aspect* for the *Call* crosscutting behavior. If results were significant for all crosscutting behaviors together, for instance in the case of ECS, when the observations are combined for *Call* and *Stop* for a dependent variable (e.g., *DIR_Average*), we denote it as *All* in the table. The fourth column is similar to the third column except that it presents the results of VCS. The sixth column is similar to the fourth column, but the only difference is that the sixth column represents the results of the matched pairs tests, whereas the fourth column shows the results of Tukey-Kramer HSD for VCS. The fifth column represents the type of the matched pairs tests applied to each dependent variable. For instance, the McNemar's test is applied to the two binary dependent variables. Non-significant results are indicated by "-" in Table 12.

Table 12. Summary of statistically significant results for DIR measures*

Dependent Variable	Approach pair	Round 1	Round 2		
		ECS	VCS (Tukey-Kramer HSD)	VCS (Matched Pairs)	
		Crosscutting Behavior (X)	Crosscutting Behavior (X)	Test	Crosscutting Behavior (X)
DIR_Average	A>X	-	All (H)	*t*-test	All (H), All (F)
	X>A	Call (H), All (H)	-		-
DIR_Binary	A>X	-	-	McNemar's test	All (H)
	X>A	Stop (F), Call (H), All (F), All (H)	-		All (F)
DIR_Binary_Max	A>X	-	Standby (F), All (F)		All (H), All (F)
	X>A	-	-		-

* X: Either H (Hierarchical) or F (Flat), A: Aspect, H: Hierarchical, F: Flat, '-' indicates non-significant results.

From Table 12, we can see that in the case of the ECS system, we observed a significance difference across the three approaches for *Call* and for *Stop* and *Call* together in terms of *DIR_Average*, where *Hierarchical* fared significantly better than *Aspect*. This could be due to the reason that in this first round, students were more familiar with standard UML state machines as compared to aspect state machines. For VCS, in case of *DIR_Average*, *Aspect* has significantly higher *DIR_Average* than *Hierarchical* for all crosscutting behaviors together (column 4, row 1, in Table 12). The results of the matched pairs *t*-test on VCS shows consistent results with Tukey-Kramer HSD, since in both cases *Aspect* significantly outperformed *Hierarchical* and *Flat*.

In case of *DIR_Binary*, for ECS, again *Hierarchical* and *Flat* significantly performed better than *Aspect*, whereas for VCS we didn't observe significant differences between approaches using the Tukey-Kramer HSD test. However, based on the results of matched pairs analysis with the McNemar's test for *DIR_Binary*, we observed that *Aspect* significantly outperformed *Hierarchical*, whereas *Flat* significantly outperformed *Aspect*. Regarding the latter, it could be due to an inherent bias of *DIR_Binary* towards *Flat* as finding just one defect out of all seeded defects will give *Flat* a maximum score (Section 3.5). For *DIR_Binary_Max* in ECS we didn't observe any significant differences. With VCS, *Aspect* significantly outperformed *Flat* for *Standby* and *Aspect* significantly performed better than *Flat* for all crosscutting behaviors together. Similar results were observed for the McNemar's test for *DIR_Binary_Max*, where *Aspect* outperformed both *Flat* and *Hierarchical*. In conclusion, a plausible explanation for the results presented above is that, when compared with flat and hierarchical state machines, aspect state machines are much less complex in terms of number of states and transitions (Table 1); therefore it is expected to be easier to identify defects in aspect state machines. The fact that the reader has to mentally weave the aspects with the base state machines to get the full picture does not seem to be a severe hindrance for these defect identification tasks.

By looking at the above results, it is also interesting to note that the results of Round 2 are different than those of Round 1 since all the results in Round 2, as opposed to Round 1, are in favor of AspectSM, except the McNemar's test results for *DIR_Binary* between *Aspect* and *Flat*. This could be due to the following reasons: 1) AspectSM entails a steep learning curve as the experience gained by the subjects of the *Aspect* group in Round 1 helped them in performing significantly better than the subjects in other groups in Round 2, 2) AspectSM may be more beneficial when modeling more complex crosscutting behaviors—recall that VCS used in Round 2 is more complex than ECS used in Round 1 (Table 1).

To discuss non-significant results, we performed power analysis, which results are summarized in Table 11 and Table 13. Note that we did so only for those cases where the results were not significant when observations were combined for all crosscutting behaviors. In Table 11, in case of *DIR_Average* (*Stop* and *Call*) regarding three approaches shows an estimated minimum effect size of 0.13 (46% of average) to achieve 80% power in Table 11. The observed effect size is 0.001, which is much lower than 0.13. Since this is a quite large effect size threshold, to draw useful conclusions with confidence regarding which approach (*Aspect* or *Flat* or *Hierarchical*) is better in terms of *DIR_Average* for *Stop* and *Call*, we probably need more observations. Similar results are obtained for other dependent variables for which the results were not significant, as shown in Table 11 and Table 13.

Table 13. Estimation of the effect size corresponding to 80% power for VCS[*]

Dependent Variable	p-value	Observed Effect Size	Minimum Effect Size	Average	Minimum Effect Size/Average
DIR_ Average (A vs F)	0.27	0.05	0.1	0.31	0.32
DIR_Binary (A vs H)	0.28	0.1	0.11	0.28	0.39
DIR_Binary (A vs F)	0.80	0.05	0.11	0.31	0.35
DIR_Binary_Max (A vs H)	0.0	0.08	0.18	0.6	0.30

* A: Aspect, H: Hierarchical, F: Flat

4.1.4 Overall Discussion

In this section, we discuss the RQ1 results of the initial experiment and the replication together. Table 14 summarizes the statistically significant results of the initial experiment and its replication. The first column represents dependent variables for defect identification, i.e., *DIR_Average*, *DIR_Binary*, and *DIR_Binary_Max*. The second column denotes the pair of approaches being compared, e.g., *A>X* reports on whether the *Aspect* (A) approach is significantly better than *Hierarchical* and/or *Flat* (X). In our particular case, we have three approaches *Aspect*, *Hierarchical*, and *Flat* denoted as *A*, *H*, and *F* respectively in the table. In addition, each dependent variable has two corresponding rows: *A>X* and *X>A* reporting on whether *Aspect* is significantly better than *Hierarchical* or *Flat*, and *Hierarchical* or *Flat* are significantly better than *Aspect*, respectively. The third column tells the name(s) of the approaches(s) for which the results were significant for ECS in the experiment. For example, for RQ1, in the case of *DIR_Average* in the row labeled *"A>X"*, *H* means that *Aspect* is significantly better than *Hierarchical*. The fourth and fifth columns are similar to the third column, but the only difference is that these columns represent the results for ECS and VCS for the replication using Tukey-Kramer HSD. The seventh column presents the results of matched pairs for VCS, whereas the sixth column lists tests being applied for matched pairs analysis for all the dependent variables.

In the table, *"-"* indicates non-significant results.

For *DIR_Average*, in the experiment, for ECS, *Aspect* performed significantly better than both *Flat* and *Hierarchical*. In contrast, in the replication, we observed that *Hierarchical* outperformed *Aspect* for *DIR_Average*. This can be explained from the fact that the subjects in the initial experiment had more training and previous experience in modeling as compared to the subjects in the replication (Section 3.7). This can be further seen from the results of the VCS system in the replication, where *Aspect* significantly performed better than *Hierarchical* and *Flat* using matched pairs analysis for *DIR_Average*, consistent with those for the ECS system in the initial experiment.

Table 14. Summary of statistically significant results for both experiments*

Dependent Variable	Pair of approaches	Experiment	Replication			
		ECS	ECS	VCS (Tukey-Kramer HSD)	VCS (Matched Pairs)	
DIR_Average	A>X	H, F	-	H	*t*-test	H, F
	X>A	-	H	-		-
DIR_Binary	A>X	H	-	-	McNemar's test	H
	X>A	-	H, F	-		F
DIR_Binary_Max	A>X	H, F	-	F		H, F
	X>A	-	-	-		-

* X: Either H (Hierarchical) or F (Flat), A: Aspect, H: Hierarchical, F: Flat, and '-' indicates non-significant results.

We observed similar results for *DIR_Binary*. In the initial experiment, *Aspect* significantly outperformed *Hierarchical* for ECS but for the replication, we observed that *Flat* and *Hierarchical* performed significantly better than *Aspect*. Again, this is probably due to the differences in training that the subjects received in the initial experiment and replication. For *DIR_Binary_Max*, we observed consistent results for the initial experiment and the replication, in which *Aspect* outperformed *Flat* and *Hierarchical*.

4.2 Results and Analysis for Defect Fixing (RQ2)

In this section, we present results for defect fixing (RQ2) based on the *DFR_Average*, *DFR_Binary*, and *DFR_Binary_Max* dependent variables (Section 3.5). Recall from Section 3.8 that defect fixing was only conducted in the replication.

4.2.1 Results for the ECS system

For ECS, in case of the *Stop* crosscutting behavior (Table 15), *Hierarchical* scored 0.66 for *DFR_Average* outperforming *Aspect* (0.64) and *Flat* (0.49). For *DFR_Binary*, *Flat* (0.93) outperformed *Hierarchical* (0.66) and *Aspect* (0.64). For *DFR_Binary_Max* (Table 15), *Hierarchical* (0.66) outperformed *Aspect* (0.64) and *Flat* (0). In *Call*, *Aspect* (0.64) outperformed both *Hierarchical* (0.63) and *Flat* (0.31) and similar results were observed for *DFR_Binary* and *DFR_Binary_Max* for *Stop*. For *Call* and *Stop* taken together, *Hierarchical* scored 0.65 for *DFR_Average*, outperforming *Aspect* (0.64) and *Flat* (0.40). For *DFR_Binary*, *Flat* (0.93) outperformed *Aspect* (0.79) and *Hierarchical* (0.76) respectively, whereas *Aspect* (0.55) outperformed *Hierarchical* (0.53) and *Flat* (0.06) for *DFR_Binary_Max* (Table 15).

Table 15. Descriptive statistics for various DFR measures

System	Measure	Crosscutting Behavior	Approach		
			Aspect	Hierarchical	Flat
ECS	DFR_Average	Stop	0.64	0.66	0.49
	DFR_Binary		0.64	0.66	0.93
	DFR_Binary_Max		0.64	0.66	0
	DFR_Average	Call	0.64	0.63	0.31
	DFR_Binary		0.94	0.86	0.93
	DFR_Binary_Max		0.47	0.4	0.13
	DFR_Average	Call and Stop	0.64	0.65	0.40
	DFR_Binary		0.79	0.76	0.93
	DFR_Binary_Max		0.55	0.53	0.06
VCS	DFR_Average	DnD	0.5	0.125	0.16
	DFR_Binary		0.71	0.4	0.26
	DFR_Binary_Max		0.28	0	0
	DFR_Average	Standby	0.4	-	0.59
	DIR_Binary		0.4	-	0.85
	DFR_Binary_Max		0.4	-	0.07
	DFR_Average	AQ	0.62	0.16	0.43
	DFR_Binary		1	0.5	0.7
	DFR_Binary_Max		0.33	0	0
	DFR_Average	DnD, Standby, and AQ	0.52	0.14	0.38
	DFR_Binary		0.74	0.45	0.58
	DFR_Binary_Max		0.33	0	0.02

Table 16. Results of One-way ANOVA for DFR_Average at α=0.05

System	Crosscutting behavior	p-value
ECS	Stop	0.48
	Call	**0.03**
	Stop and Call	**0.02**
VCS	DnD	**0.008**
	Standby	0.23
	AQ	**0.0003**
	All	**0.0003**

The one-way ANOVA results presented in Table 16 show that there are significant differences for *DFR_Average* (*Call*) and *DFR_Average* (*Stop* and *Call*). For these variables, we performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30], reported in Table 17. The results of the two-tailed Fisher exact test for binary variables (*DFR_Binary* and *DFR_Binary_Max*) are shown in Table 18, where p-values are bold when below our selected level of significance.

Table 17. Comparisons of all pairs using Tukey_Kramer HSD for DFR_Average

System	Measure	Aspect vs Hierarchical		Aspect vs Flat	
		Mean Difference (Aspect - Hierarchical)	p-value	Mean Difference (Aspect - Flat)	p-value
ECS	DFR_Average (Call)	0.01	0.99	0.33	**0.04**
	DFR_Average	-0.0002	0.99	0.24	**0.04**
VCS	DFR_Average (DnD)	0.37	**0.02**	0.34	**0.02**
	DFR_Average (AQ)	0.46	**0.0002**	-0.26	0.06
	DFR_Average	0.37	**0.0002**	0.13	0.18

Table 18. Two tailed Fisher's exact test for DFR binary measures

System	Measure	Aspect vs. Hierarchical		Aspect vs. Flat	
		Difference in proportion (Aspect- Hierarchical)	p-value	Difference in proportion (Aspect- Flat)	p-value
ECS	DFR_Binary (Stop)	-0.01	1	-0.28	0.08
	DFR_Binary_Max (Stop)	-0.01	1	0.64	**0.0001**
	DFR_Binary (Call)	0.07	0.58	0.0007	1
	DFR_Binary_Max (Call)	0.07	0.73	0.33	0.06
	DFR_Binary	0.02	1	-0.13	0.15
	DFR_Binary_Max	0.02	1	0.49	**0.0001**
VCS	DFR_Binary (DnD)	0.31	0.21	0.44	**0.02**
	DFR_Binary_Max (DnD)	0.28	0.11	0.28	**0.04**
	DFR_Binary (Standby)	-	-	-0.45	**0.03**
	DFR_Binary_Max (Standby)	-	-	0.32	0.12
	DFR_Binary (AQ)	0.50	**0.002**	0.3	0.05
	DFR_Binary_Max (AQ)	0.33	**0.04**	0.33	0.06
	DFR_Binary	0.28	**0.03**	0.15	0.22
	DFR_Binary_Max	0.33	**0.001**	0.30	**0.0005**

4.2.2 Results for the VCS system

For VCS, in case of *DnD*, *Aspect* outperformed both *Hierarchical* and *Flat* for all three defect fixing measures as it can be seen from the means reported in Table 15. For the *Standby* crosscutting behavior, for *DFR_Average* and *DFR_Binary*, *Flat* outperformed *Aspect*, whereas *Aspect* outperformed *Flat* for *DFR_Binary_Max*. Recall that for *Standby*, we didn't have a solution using the *Hierarchical* approach. For *AQ*, *Aspect* outperformed *Hierarchical* and *Flat* for all three defect fixing measures. For all three crosscutting behaviors together, *Aspect* outperformed *Hierarchical* and *Flat* for all three defect fixing dependent variables.

The results of one-way ANOVA presented in Table 16 show that there are significant differences for *DFR_Average* (*DnD*), *DFR_Average* (*AQ*), and *DFR_Average* (*DnD*, *Standby*, and *AQ*). For these variables, since one-way ANOVA results were significant, we performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30] and the results are given in Table 17. For binary variables *DFR_Binary* and *DFR_Binary_Max*, we report the results of the Fisher exact test in Table 18. In all these tables, bold p-values highlight statistically significant results and the mean differences between pairs of approaches indicate the direction of the effect.

The results for the matched pairs *t*-test for *DFR_Average* are shown in Table 19. For all three crosscutting behaviors together, *Aspect* significantly outperformed *Hierarchical*; however there is no significant difference between *Aspect* and *Flat*. For matched pairs analysis of the binary dependent variables, the results of the McNemar's test are shown in Table 19, where *Aspect* significantly outperformed *Hierarchical* and *Flat* regarding *DFR_Binary_Max*. For *DFR_Binary*, a significant difference is once again observed between *Aspect* and *Flat* but not between *Aspect* and *Hierarchical*.

Table 19. Results of matched pairs for VCS for various DFR measures at α=0.05

Dependent Variable	Pair of approaches	Mean Difference	Test	p-value
DFR_Average	Aspect-Hierarchical	0.31	*t*-test	**0.004**
DFR_Average	Aspect-Flat	0.14		0.11
DFR_Binary	Aspect-Hierarchical	0.33	McNemar's test	0.832
DFR_Binary	Aspect-Flat	0.15		**0.03**
DFR_Binary_Max	Aspect-Hierarchical	0.33		**2.98e-08**
DFR_Binary_Max	Aspect-Flat	0.31		**4.17e-07**

4.2.3 Discussion

In this section, we provide a discussion on DFRs for each crosscutting behavior individually and all crosscutting behaviors together. DFRs are measured based on three dependent variables: *DFR_Average*, *DFR_Binary*, and *DFR_Binary_Max*. Statistically significant results are summarized in Table 20. The first column lists the dependent variables which are used to answer RQ2 (Table 5). The second column denotes pairs of approaches being compared and each dependent variable has two rows in this column: *A>X* and *X>A* denoting whether *Aspect* (A) is significantly better than *Hierarchical* or *Flat*

(X), and *Hierarchical* or *Flat* (X) are significantly better than *Aspect* (A), respectively. The third column (labeled "Crosscutting Behavior (X)") presents two pieces of information: 1) name(s) of the crosscutting behavior(s) for which the results were significant for ECS, 2) name of the approach in brackets against which the results were significant, i.e., the approach is either significantly better than *Aspect* if located in row X>A or vice versa if located in row A>X. For example, in case of *DIR_Average* in the row labeled *"X>A"*, *Call (H)* means that *Hierarchical* is significantly better than *Aspect* for the *Call* crosscutting behavior. If results were significant for all crosscutting behaviors together, for instance in the case of ECS, when the observations are combined for *Call* and *Stop* for a dependent variable (e.g., *DIR_Average*), we denote it as *All* in the table. The fourth column is similar to the third column except that it presents the results of VCS. The fifth column is similar to the fourth column, but the only difference is that it reports the results of the matched pairs *t*-test, whereas the fourth column shows the results of Tukey-Kramer HSD for VCS. In Table 20, a *"-"* indicates non-significant results.

From Table 20, we can see that overall *Aspect* significantly performed better than *Flat* in terms of *DFR_Average* and *DFR_Binary_Max*, but there are no significant differences between *Aspect* and *Hierarchical*. When compared to the results of DIRs from Round 1, the results are in favor of AspectSM because the students gained experience with AspectSM while identifying defects. In addition, due to the lower complexity of aspect state machines (Table 1), it was easier for the subjects to fix the defects. For Round 2, in the case of VCS, *Aspect* is overall significantly better than *Hierarchical* and *Flat* as it can be seen from the results of Tukey-Kramer HSD for all three DFR variables in Table 20. The results of the matched pairs *t*-test for *DFR_Average* and the McNemar's test for the two binary dependent variables yielded consistent results. Similar to defect identification, plausible explanation is that, when compared with flat and hierarchical state machines, aspect state machines are much less complex in terms of number of states and transitions (Table 1); therefore it is expected to be much easier to fix defects in aspect state machines.

To further investigate non-significant results, we performed power analysis, which results are summarized in Table 21. Note that we did so only for those cases where the results are not even significant when observations are combined for all crosscutting behaviors. In case of *DFR_Average* (*Call* and *Stop*) regarding three approaches, the results of the power analysis shows an estimated minimum effect size of 0.20 (60% of average) to achieve 80% power in Table 21. The observed effect size is 0.07, which is much lower than the estimated effect size (0.20) thus explaining lack of significance. Given that 60% is a large threshold, this suggests that we need to collect more observations to draw conclusions with confidence regarding which approach (*Aspect* or *Flat* or *Hierarchical*) is better in terms of *DFR_Average* for *Call* and *Stop*. Similar results are obtained for other dependent variables for which the results were not significant in Table 21.

Table 20. Summary of statistical significant results[*]

Dependent Variable	Approach pair	Round 1	Round 2	
		ECS	VCS (Tukey-Kramer HSD)	VCS (Matched Pairs *t*-test)
		Crosscutting Behavior (X)	Crosscutting Behavior (X)	Crosscutting Behavior (X)
DFR_Average	A>X	Call (F), All (F)	DnD (H), DnD (F), AQ (H), All (H)	All (H)
	X>A	-	-	-
DFR_Binary	A>X	-	DnD (F), Standby (F), AQ (H), All (H)	All (F)
	X>A	-	-	-
DFR_Binary_Max	A>X	Stop (F), All (F)	DnD (F), AQ (H), All (H), All (F)	All (H), All (F)
	X>A	-	-	-

* X: Either H (Hierarchical) or F (Flat), A: Aspect, H: Hierarchical, F: Flat, and '-' indicates non-significant results.

Table 21. Estimation of the effect size corresponding to 80% power[*]

System	Measure (Approaches)	p-value	Observed Effect Size	Minimum Effect Size	Average	Minimum Effect Size/Average
ECS	DFR_ Average (A vs H vs F)	0.48	0.07	0.20	0.60	0.34
	DFR_ Average (A vs H)	0.99	0.001	0.15	0.65	0.23
	DFR_Binary(A vs H)	1	0.01	0.15	0.78	0.19
	DFR_Binary (A vs F)	0.15	0.06	0.12	0.85	0.14
	DFR_Binary_Max (A vs H)	1	0.01	0.18	0.54	0.33
VCS	DFR_ Average (A vs H vs F)	0.18	0.06	0.12	0.45	0.27
	DFR_Binary (A vs H vs F)	0.22	0.07	0.15	0.66	0.22

* A: Aspect, H: Hierarchical, F: Flat

4.3 Results and Analysis for Comprehensibility (RQ3)

In this section, we present results for answering comprehension questionnaire (RQ3) based on the *SCQ* dependent variable (Section 3.5).

4.3.1 Results for the Initial Experiment

The descriptive statistics for *SCQ* are presented in Table 22. We observed that *Hierarchical* yields higher correctness than *Aspect* and *Flat*. More specifically, *Hierarchical* performed 21.8% and 40% better than *Aspect* and *Flat*, respectively. We checked the significance of the results by applying one-way ANOVA to SCQ (Table 23), which shows significant differences between the approaches as the p-value is 0.002. Since one-way ANOVA results are significant, we performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD. The results showed that differences are not significant between *Aspect* vs *Hierarchical* and *Aspect* vs *Flat*. However, the results are significant between *Hierarchical* and *Flat*, but we do not report them here since this is not the focus of our study.

4.3.2 Results for the Replication

For the replication with ECS, we observed that *Hierarchical* yields higher comprehensibility (*SCQ*) than *Aspect* and *Flat* as it can be seen from the results reported in Table 22. For VCS, we observed that *Aspect* scored on average 6.92, which is higher than *Hierarchical* (6.6) and *Flat* (6.4) in Table 22. A one-way ANOVA with *SCQ* for ECS is reported in Table 23 and shows a significant difference. However, the results of a pair-wise comparison using Tukey-Kramer HSD shows, once again, significant differences only between *Hierarchical* and *Flat*. For VCS, the result of one-way ANOVA on *SCQ* showed no significant differences (Table 23).

Table 22. Descriptive statistics for SCQ

Experiment	**System**	**Crosscutting Behavior**	**Approach**		
			Aspect	**Hierarchical**	**Flat**
Experiment	ECS	Call and Stop	6.38	8.56	4.50
Replication	ECS	Call and Stop	5.52	7.06	5.33
	VCS	DnD, Standby, and AQ	6.92	6.6	6.4

Table 23. Results of One-way ANOVA for SCQ at α=0.05

Experiment	**System**	**Crosscutting Behavior**	**p-value**
Experiment	ECS	Stop and Call	**0.002**
Replication	ECS	Stop and Call	**0.02**
	VCS	DnD, Standby, and AQ	0.79

4.3.3 Discussion

In overall, the differences between *Aspect* vs *Hierarchical* and *Aspect* vs *Flat* are not significant. One plausible explanation is that for *Aspect* the subjects needed to carefully read and understand *Pointcut* specifications in the *Aspect* state machines. With more training and practice on AspectSM, subjects would be expected to gain better comprehension of aspect state machines as compared with flat and hierarchical state machines, for which they had more prior experience.

The power analysis results for *SCQ* for the initial experiment, when comparing *Aspect vs Hierarchical* and *Aspect vs Flat,* revealed that we need minimum effect sizes of 1.27 (17% of average) and 1.65 (30% of average), respectively, to achieve 80% power (Table 24). These effect sizes are larger than the observed effect sizes, i.e., 1.08 and 0.94, thus explaining lack of significance. For VCS, power analysis revealed similar results, where we need minimum effect size of 1.10 (17% of the average) to achieve 80% of power as reported in Table 24. The minimum effect size, i.e., 1.10 (score out of 10) is much larger than the observed effect size (0.23). Thus, overall, if we want to investigate effects below the minimum thresholds mentioned above, the results of power analysis suggest that we need to collect more observations either by increasing the number of subjects and/or adding more case studies with crosscutting behaviors.

Table 24. Estimation of the effect size corresponding to 80% power for SCQ

Experiment	System	Pair of Approaches	p-value	Observed Effect Size	Minimum Effect Size	Average (score out of 10)	Minimum Effect Size/Average
Experiment	ECS	Aspect vs Hierarchical	0.09	1.08	1.27	7.41	0.17
Replication	ECS	Aspect vs Flat	0.15	0.94	1.65	5.5	0.3
	VCS	Aspect vs Hierarchical vs Flat	0.79	0.23	1.10	6.64	0.17

4.4 Results and Analysis for Effort (RQ4)

In this section, we present results for effort (RQ4) based on the *Effort* dependent variable (Section 3.5). Recall from Section 3.8 that the effort was measured only for the initial experiment and thus in this section we only present results and analysis for the initial experiment.

4.4.1 Results

From Table 25, we can observe that in *Task 1*, the subjects took approximately 34 minutes on average for *Hierarchical* to identify defects. However, both *Aspect* and *Flat* took the same average time to complete the task: 32 minutes. *Task 2* took three and six minutes less for *Aspect* than for *Hierarchical* and *Flat* to identify defects. For answering the comprehension questionnaire (*Task 3*), the subjects took nine and five minutes more for *Hierarchical* than *Aspect* and *Flat,* respectively. In summary, there is no practically significant time difference across the three state machines.

As discussed in Section 3.9.1, we applied the one-way ANOVA test to assess the statistical significance of differences for *Effort* (for each task) distributions across the three approaches. Table 26 shows the results of the test, where significant differences were observed for the *Effort* of Task 3. Since one-way ANOVA results were significant, we performed a pair-wise comparison of the distributions obtained for the three state machines using Tukey_Kramer HSD [30]. The results are presented in Table 26, where *Hierarchical* took significantly more time than *Aspect* (p-value=0.01), whereas *Aspect* took less time than *Flat*, though the latter is not significant (p-value=0.39).

Table 25. Results for one-way ANOVA test for Effort using Tukey_Kramer HSD

Measure	Effort (Task1)	Effort (Task2)	Effort (Task3)
p-value	0.86	0.09	**0.02**

Table 26. Comparisons of all pairs for Effort (Task 3) using Tukey_Kramer HSD

Aspect vs Hierarchical		Aspect vs Flat	
Mean Difference (Aspect-Hierarchical)	p-value	Mean Difference (Aspect-Flat)	p-value
-9.31	**0.01**	-4.3	0.39

4.4.2 Discussion

There were no significant differences in effort between any pair of approaches for defect identification (*Task1* and *Task2*). This means that the effort spent for identifying defects across the three state machines is roughly the same. Regarding *Task 3* (i.e., answering the comprehension questionnaire), we only observed significant differences for *Effort* between the *Aspect* and *Hierarchical* groups, where the hierarchical group took significantly more time than the *Aspect* group (Table 26). Between *Aspect* and *Flat*, for Task 3, we didn't observe significant differences in terms of *Effort*.

The power analysis in Table 27 shows that the minimum effect size corresponding to 80% power is 3.62 minutes (i.e., 16% of the average effort for the combined groups). The observed effect size is 2.14 minutes, thus explaining lack of significance. Drawing reliable conclusions for effect sizes below 3.62 minutes would require larger sample sizes. However, note that the difference between the two averages, i.e., 2.14 minutes and 3.62 minutes, is small and therefore practically negligible.

Table 27. Estimation of the effect size corresponding to 80% power

Measure	p-value	Observed Effect Size	Minimum Effect Size	Average (Minutes)	Minimum Effect Size/Average
Effort (Task 3 for Aspect vs Flat)	0.39	2.13	3.62	22.43	16%

4.5 Concluding Remarks

Based on the above results and discussions, we suggest that aspect state machines should be used to model crosscutting behavior, but one should always use, when applicable, hierarchical state machines features within aspect state machines to further improve their comprehensibility. There are cases in which hierarchical state machines (submachines) are not applicable and aspect state machines are then the only option. For example, separating out constraints modeling non-functional properties (e.g., video or audio quality) from state invariants is not possible using hierarchical state machines without introducing accidental complexity and redundancy as we demonstrated in [10]. Easier identification/fixing of defects in aspect state machines also implies that it is easier to ensure their conformance to specifications.

5. Threat to Validity

Below, we discuss the threats to validity of our controlled experiments based on the guidelines presented in [26].

5.1 Conclusion Validity Threats

Conclusion validity threats are concerned with factors that can influence the conclusion that can be drawn from the results of the experiments. As with most controlled experiments in software engineering, our main conclusion validity threat is related to the sample size on which we base our analysis. For the initial experiment, we performed a two-round experiment to maximize the number of observations within time constraints. However, the lack of significance of certain differences (e.g., the difference in *SCQ* for *Aspect* vs *Hierarchical* and *Aspect* vs *Flat*, effort for answering the comprehension questionnaire (*Aspect* vs *Flat*), and *DIR_Binary* for *Aspect* vs *Flat*) may be due to low statistical power if actual effect sizes are below a certain threshold (Section 4.5). Studying the presence of smaller effect sizes requires replicating the experiment and collecting additional data points. Due to this reason, we replicated the experiment with an additional industrial case study including three crosscutting behaviors (Section 3.3.1) and with more subjects

(Section 3.2) to increase the sample sizes and thereby the power of statistical tests. Statistical conclusions were drawn by applying appropriate statistical tests based on a careful analysis of their assumptions (Section 3.9).

5.2 Internal Validity Threats

Internal validity threats exist when the outcome of results are influenced by external factors and are not necessarily due to the application of the treatment being studied. Through our experiment design (between-subjects design) for the initial experiment and Round 1 of the replication, we have tried to minimize the chances of other factors being confounded with our primary independent variable: the use of aspect state machines. We avoided any biased assignment of subjects to groups by using blocking based on assignment marks.

In Round 2 of the replication, regarding identification and fixing defects, we used a within-subjects design and matched pairs *analysis*. The strength of this design is that the variation due to differences in subjects is eliminated as each subject acts as its own control. A within-subjects design may however be subject to learning effects, for example due to using the same material for various tasks (e.g., defect identification and defect correction) that could result into improved performance from one task to the next. To counterbalance such effects, we rotated our groups to each crosscutting behavior for each task (e.g., defect identification) as we discussed in Section 3.4.2. In the initial experiment, we gave the subjects as much time as they wished to use for each task, though there was a time limit by which they had to finish all the tasks. No time differences were observed across modeling approaches. In the replication, we gave the subjects a fixed amount of time for each activity. Such an approach only enables, however, an investigation of the effect of the modeling approaches in terms of effectiveness.

5.3 Construct Validity Threats

There are two possible construct validity threats in our experiments. Regarding readability based on defect identification rates, due to time and resource constraints, we couldn't seed all types of defects in the defect classification (Section 3.3.2). It is also not practically feasible to devise case studies containing all types of defects from the defect classification. Anyhow, we tried to maximize the defect classification coverage based on the available case studies and seeded defects of types MT, IT, MS, and IS to compute defect identification rates. The second threat of construct validity is that we were not able to investigate all features of aspect-orientation (such as all types of basic advice) due to the nature of the crosscutting behaviors in our case studies.

5.4 External Validity Threats

This is typically the most common threat in controlled experiments. Due to time constraints, case studies and tasks are usually small, and this often tends to minimize the differences among treatments. As we see in Table 1 for ECS, for crosscutting behavior *Call*, the flat state machine has 15 states and 27 transitions. Similarly, for the *Stop* crosscutting behavior, we have 13 states and 25 transitions. Such numbers are at least representatives of the state machines of classes and small components. In addition, we also replicated the experiment on an industrial case study with three crosscutting behaviors and more students to further reduce external validity threats. However, because crosscutting concerns are expected to have an even higher impact on large models, we expect the use of AspectSM to be even more beneficial in such cases. It is worth noting that we replicated the experiment in a different geographical area and education system to reduce external validity threats. One may also question the use of students as subjects for the experiment. Note that many practitioners have very little knowledge of AOP or AOM in general, and hence require significant training and cost to teach them AOM. Due to this reason, we chose a group of experienced graduate students with a suitable educational background (Section 3.2). In addition, some studies in [34-36] reported on the performance of trained software engineering students for various tasks when compared with professional developers. These differences turned out not to be statistically significant when compared to junior and intermediate developers, thus suggesting that there is no evidence that students trained for the tasks at hand may not be used as subjects in place of professionals.

6. Related Work

Most experimentation in Aspect-Oriented Software Development (AOSD) has been conducted to evaluate aspect-oriented programming when compared to object-oriented programming in terms of development time, errors in development, and performing maintenance tasks. An initial search on the IEEE, ACM, Science Direct, Wiley Interscience, and Springer digital libraries yielded 517 papers; however, none of them reported any controlled experiment to evaluate AOM approaches. A controlled experiment [37] was performed in industry settings to measure effort and errors using aspect-oriented programming for applying different maintenance tasks related to the tracing crosscutting concern, i.e., the use of logging to record execution of a program. The results showed that aspect-orientation resulted in reducing both development effort and number of errors.

Another experiment is reported in [38], which compares aspect-orientation (AspectJ) with a more traditional approach (Java) in terms of development time for crosscutting concerns. A similar experiment is reported in [39] focusing on development time to perform debugging and change activities on object-oriented programs using AspectJ. Both of these experiments revealed mixed results, i.e., aspect-orientation has positive impact on development time only for certain tasks. For instance, Aspect-oriented Programming (AOP) seems to be more beneficial when the crosscutting concern is more separable from the core behavior.

An exploratory study is reported in [40] to assess if AOP has any impact on software maintenance tasks. Eleven software professionals were asked to perform different maintenance tasks using Java and AspectJ. The results of the experiment revealed that AOP performed slightly better than Object-oriented Programming (OOP), but there were no statistically significant results observed. Another exploratory study is reported in [41] to measure fault-proneness with AOP. Three evolving AOP programs were used and data about different faults made during their development were collected. The experiment revealed two major findings: 1) Most of the faults were due to lack of compatibility between aspect and base code, 2) The presence of faults in AOP features such as Pointcuts, Advice, and inter-type declarations was as likely as for normal programming features. The results turned out to be statistically significant.

An experiment is reported in [42], where two software development processes based on a same aspect modeling approach (i.e., the Theme approach [43]) are compared to determine their impacts on maintenance tasks such as adding new functionality or improving existing functionality. The first process (aspectual process) involves generating AO code in AspectJ from Theme AO models, whereas the second process (hybrid process) involves generating object-oriented code in Java from Theme models. Maintenance tasks are measured based on metrics such as size, coupling, cohesion, and separation of concerns. The results showed that on average the aspectual process took lesser time than the hybrid process.

An exploratory study is reported in [44], which aims to assess if aspects can help reducing effort on resolving conflicts that can occur during model compositions. To do so, they compared AOM with non-AOM in terms of effort to resolve conflicts and number of conflicts resolved on six releases of a software product line. The results of the study showed that aspects improved modularization and hence helped better localize conflicts, which in turn resulted in reducing the effort involved in resolving conflicts.

Our controlled experiments are different from the above experiments from several perspectives. First, our controlled experiments focused on the design phase of the software development life cycle and Aspect-Oriented Modeling. Most of the experiments in the literature have focused on comparing AOP with OOP. We evaluated the "readability" (i.e., defect identification, defect fixing, and answering comprehension) of crosscutting behaviors modeled as aspect state machines as compared to directly modeling them in UML state machines. We further compared the effort for defect identification and answering comprehension for the experiment. Apart from these differences, we observed results in our experiments to be consistent to some of the results observed in the literature. For instance, similar to the results on development time using AspectJ reported in [41], we didn't observe any reduced effort in inspecting state machines developed using our AspectSM approach. Also, similar to results reported in [38] and [39], where they observed inconsistent results for different measures corresponding to different program development and maintenance activities, our results also differed for defect identification/fixing rates and responses to the comprehension questionnaire.

7. Conclusion and Future Work

Aspect-oriented Modeling (AOM) is a very active field of research and can potentially yield several benefits while modeling systems, including enhanced separation of concerns, improved readability, easier model evolution, increased reusability, and reduced modeling effort. However, to authors' knowledge, there is no reported empirical evidence regarding such benefits.

This paper reports the results of the first two controlled experiments in the literature to report on the evaluation of AOM, and more precisely whether AOM can help improve the "readability" of UML state machines in terms of design defect identification, defect fixing, comprehension, and inspection effort. The specific AOM approach under evaluation is a recently published UML profile (AspectSM), which was specifically designed to model crosscutting behavior (e.g., robustness behavior) using standard UML 2 state machines with a lightweight extension for aspect-oriented features. The AspectSM profile has been previously applied to an industrial case study for automated, state-based robustness testing.

The readability of state machines modeling crosscutting behavior using AspectSM (aspect state machines) is compared with standard UML 2 state machines using advanced features such as hierarchy and concurrency (hierarchical state machines) and without hierarchical features (flat state machines).

Results show that the defect identification and defect fixing rates of aspect state machines are significantly higher than the ones for the hierarchical and flat state machines. For instance, for the industrial case study in the replication, aspect state machines show, on average, increases of 28% and 19% in defect identification rates when compared to hierarchical and flat state machines, respectively. This is most likely due to the fact that aspect state machines are less complex than hierarchical and flat state machines in terms of modeling elements such as states and transitions. But on the other hand, aspect state machines can be potentially difficult to comprehend in terms of mentally processing how an aspect is woven into its base state machine. This may explain why, based on subjects' responses to a comprehension questionnaire, results show that the subjects that were given hierarchical state machines outperformed the ones that were assigned aspect state machines, though that difference was not statistically significant. No significant difference in effort was observed between any types of state machines in both defect identification and comprehension. Based on the results above, our practical recommendation is to model crosscutting behaviors using aspect state machines in combination with hierarchical/concurrent features of UML state machines, where applicable, in order to improve the overall readability of crosscutting behaviors.

In the future, we are planning to replicate the experiment to study the readability of aspect state machines in the presence of interactions between aspects as well as compare the understandability, modeling effort, and quality of aspect state machines with flat and hierarchical state machines.

8. References

[1] M.S. Ali, M.A. Babar, L. Chen, K.-J. Stol, A Systematic Review of Comparative Evidence of Aspect-oriented Programming, Information and Software Technology, 52(9), 871-887.
[2] R.Chitchyan, A.Rashid, P. Sawyer, J. Bakker, M.P. Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, A. Jackson, Survey of Aspect-Oriented Analysis and Design, in, 2005.
[3] R.E. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley Professional, 2004.
[4] R. Yedduladoddi, Aspect Oriented Software Development: An Approach to Composing UML Design Models, VDM Verlag Dr. Müller, 2009.
[5] IEEE Standard Dictionary of Measures of the Software Aspects of Dependability, IEEE Std 982.1-2005 (Revision of IEEE Std 982.1-1988), (2006), 1-34.
[6] Standard for Software Quality Characteristics, in, International Organization for Standardization, ISO-9126-3, 2003.
[7] Software Assurance Standard, in, NASA Technical Standard, NASA-STD-8739.8, 2005.
[8] R.V. Binder, Testing object-oriented systems: models, patterns, and tools, Addison-Wesley Longman Publishing Co., Inc., 1999.
[9] D. Drusinsky, Modeling and Verification using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, 1st ed., Newnes, 2006.
[10] S. Ali, L.C. Briand, H. Hemmati, Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems, in, Accepted for publication in the Systems and Software Modeling (SOSYM) Journal, 2011.
[11] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley Professional, 2000.
[12] T. Weigert, R. Reed, Specifying Telecommunications Systems with UML, in: UML for Real: Design of Embedded Real-time Systems, Kluwer Academic Publishers, 2003, pp. 301-322.
[13] SmartState, http://www.smartstatestudio.com/, 2010
[14] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann, 2007.
[15] R. Cavarra, C. Crichton, J. Davies, A. Hartman, L. Mounier, Using UML for automatic test generation in: International Symposium on Software Testing and Analysis (ISSTA '02), 2002.
[16] T. Pender, UML Bible, Wiley, 2003.
[17] IBM OCL Parser, http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html, 2010
[18] OCLE, http://lci.cs.ubbcluj.ro/ocle/, 2010
[19] EyeOCL Software, http://maude.sip.ucm.es/eos/, 2010
[20] G. Zhang, Towards Aspect-Oriented State Machines, in: 2nd Asian Workshop on Aspect-Oriented Software Development (AOASIA'06), Tokyo, 2006.
[21] G. Zhang, M. Hölzl, HiLA: High-Level Aspects for UML-State Machines, in: In Proceedings of the 14th Workshop on Aspect-Oriented Modeling (AOM@MoDELS'09), 2009.
[22] G. Zhang, M.M. Hölzl, A. Knapp, Enhancing UML State Machines with Aspects, in:

In Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), 2007.
[23] D. Xu, W. Xu, K. Nygard, A State-Based Approach to Testing Aspect-Oriented Programs, in: 17th International Conference on Software Engineering and Knowledge Engineering, Taiwan, 2005.
[24] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, R. Rabbi, An Expressive Aspect Composition Language for UML State Diagrams, in: Model Driven Engineering Languages and Systems, 2007.
[25] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, 2003.
[26] C. Wohlin, P. Runeson, M. Höst, Experimentation in Software Engineering: An Introduction, Springer, 1999.
[27] A.N. Oppenheim, Questionnaire design, interviewing, and attitude measurement, Pinter Pub Ltd, 1992.
[28] JMP, http://www.jmp.com/, 2010
[29] F. Shull, J. Singer, D. I. K. Sjøberg, Guide to Advanced Empirical Software Engineering, Springer, 2008.
[30] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, Chapman and Hall/CRC, 2007.
[31] McNemar's Test, http://www.fon.hum.uva.nl/Service/Statistics/McNemars_test.html, 2011
[32] L. Thomas, Retrospective Power Analysis, Conservation Biology, 11(1), (1997), pp. 276-280.
[33] T. Dyba°, V.B. Kampenes, J.E. Hannay, D.I.K. Sjøberg, Systematic review: A systematic review of effect size in software engineering experiments, Information and Software Technology, 49(11-12), (2007), pp. 1073-1086.
[34] M. Höst, B. Regnell, C. Wohlin, Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, Empirical Software Engineering, 5(3), (2000), pp. 201-214.
[35] E. Arisholm, D.I.K. Sjoberg, Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software, IEEE Transactions on Software Engineering, 30(8), (2004), pp. 521-534.
[36] R.W. Holt, D.A. Boehm-Davis, A.C. Shultz, Mental Representations of Programs for Student and Professional Programmers, in: M.O. Gary, S. Sylvia, S. Elliot (Eds.) Empirical Studies of Programmers: Second Workshop, Ablex Publishing Corp., 1987, pp. 33-46.
[37] P. Durr, L. Bergmans, M. Aksit, A Controlled Experiment for the Assessment of Aspects - Tracing in an Industrial Context, in, University of Twente, CTIT, Enschede, 2008.
[38] S. Hanenberg, S. Kleinschmager, M. Josupeit-Walter, Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, 2009.
[39] R.J. Walker, E.L.A. Baniassad, G.C. Murphy, An initial assessment of aspect-oriented programming, in: 21st international conference on Software engineering, ACM, Los Angeles, California, United States, 1999.
[40] M. Bartsch, R. Harrison, An exploratory study of the effect of aspect-oriented programming on maintainability, Software Quality Control, 16(1), (2008), pp. 23-44.
[41] F. Ferrari, R. Burrows, v. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, J. Maldonado, An exploratory study of fault-proneness in evolving aspect-oriented programs, in:

Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, Cape Town, South Africa, 2010.
[42] K. Farias, A. Garcia, J. Whittle, Assessing the impact of aspects on model composition effort, in: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, ACM, Rennes and Saint-Malo, France.
[43] A. Carton, C. Driver, A. Jackson, S. Clarke, Model-Driven Theme/UML, in: K. Shmuel, O. Harold, F. Robert, J. Jean-Marc, z, quel (Eds.) Transactions on Aspect-Oriented Software Development VI, Springer-Verlag, 2009, pp. 238-266.
[44] A. Hovsepyan, R. Scandariato, S.V. Baelen, Y. Berbers, W. Joosen, From aspect-oriented models to aspect-oriented code?: the maintenance perspective, in: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, ACM, Rennes and Saint-Malo, France.

Appendix A: Models for Video Conferencing System (VCS)

In this Appendix, we provide the description and models of one of the case studies that we used in the replication. Note that we provide this information only for one crosscutting behavior AudioQuality (AQ) just to provide an idea of how models developed using different modeling approaches looks like. The VCS case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels. The class diagram of Saturn in shown in Figure 3, whereas base state machine of Saturn is shown in Figure 4. From the Idle state, calling the *dialSaturn()* method of the Saturn class, it goes to *Connected*, which represents the behavior of the system when there is a conference with one endpoint. As long as there exists one endpoint in the conference, Saturn stays in the *Connected* state and when Saturn dials to more endpoints, it goes to the *NotFull* state. When it connects to the maximum number of endpoints it supports, it goes to the *Full* state. The maximum number of connections is six in this VCS.

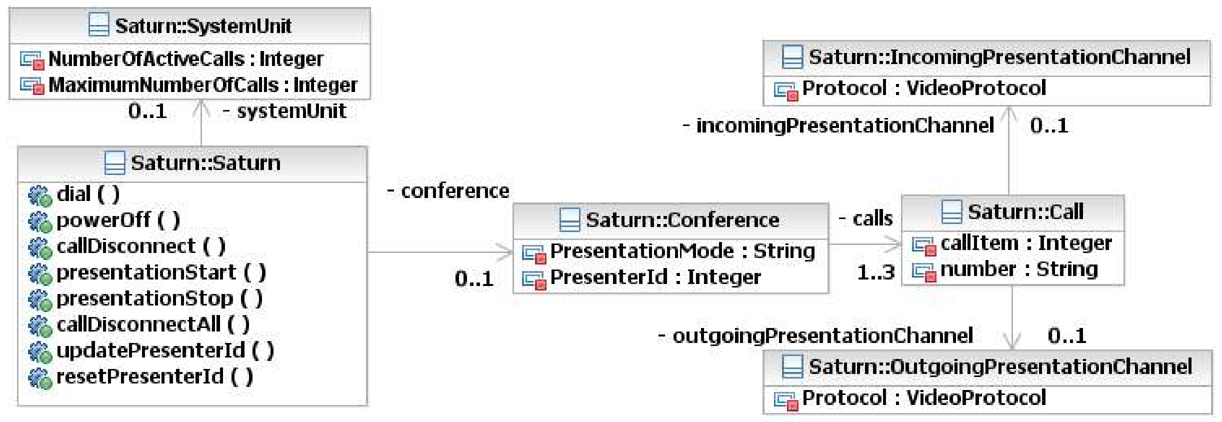


Figure 3. Conceptual model of the S-Saturn subsystem

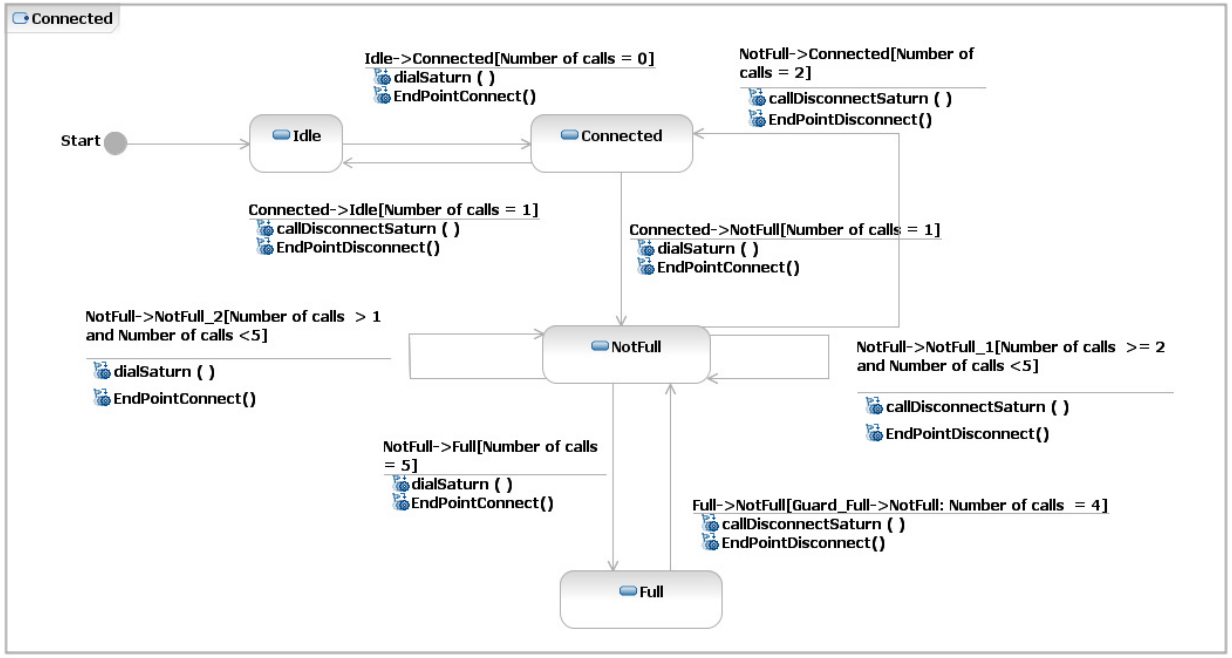
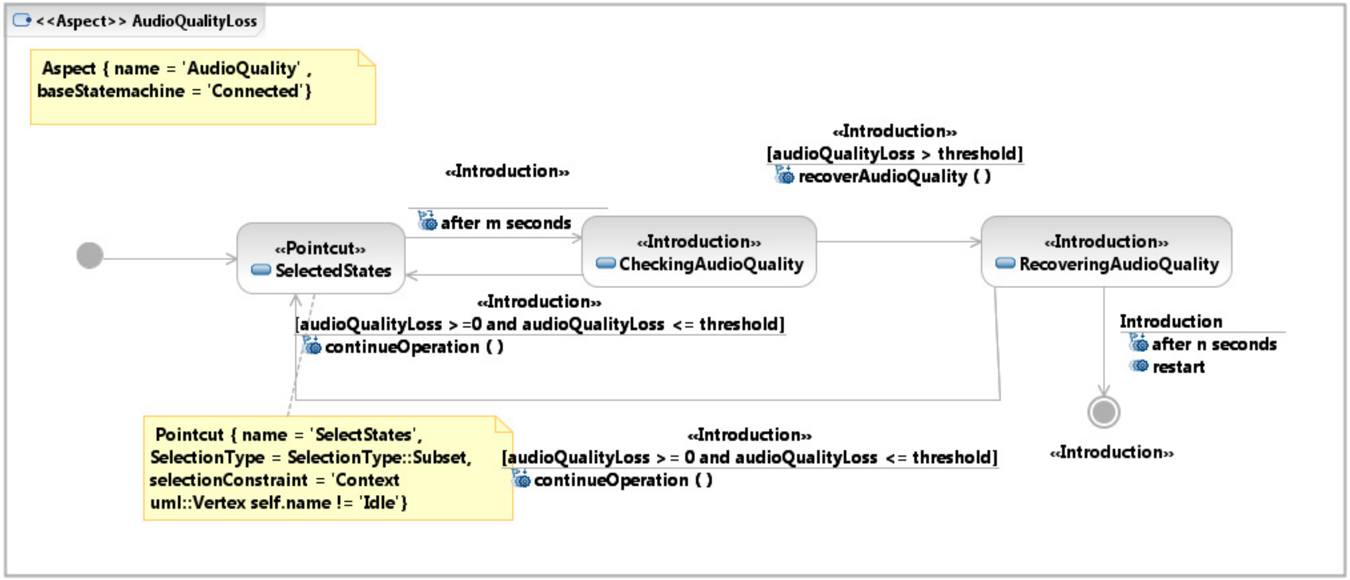

Connected
Start
Idle
Connected
NotFull
Full
Idle->Connected[Number of calls = 0]
dialSaturn ( )
EndPointConnect()
NotFull->Connected[Number of calls = 2]
callDisconnectSaturn ( )
EndPointDisconnect()
Connected->Idle[Number of calls = 1]
callDisconnectSaturn ( )
EndPointDisconnect()
Connected->NotFull[Number of calls = 1]
dialSaturn ( )
EndPointConnect()
NotFull->NotFull_2[Number of calls > 1 and Number of calls <5]
dialSaturn ( )
EndPointConnect()
NotFull->NotFull_1[Number of calls >= 2 and Number of calls <5]
callDisconnectSaturn ( )
EndPointDisconnect()
NotFull->Full[Number of calls = 5]
dialSaturn ( )
EndPointConnect()
Full->NotFull[Guard_Full->NotFull: Number of calls = 4]
callDisconnectSaturn ( )
EndPointDisconnect()

Figure 4. Base state machine for VCS

Models for AQ

An important robustness behavior of Saturn is to recover from audio quality loss. Whenever Saturn is in a videoconference, it checks audio quality after every *m* seconds. If the quality is within *threshold* it continues the normal operation, otherwise it tries to recover audio quality. If it is successful in less than *n* seconds it continues its normal operation, otherwise after *n* seconds it restarts the VCS. The aspect state machine for AQ is shown in Figure 2. AQ modeled with *Hierarchical* is shown in Figure 6 and Figure 7, whereas with the *Flat* approach is shown in Figure 8.


<<Aspect>> AudioQualityLoss
Aspect { name = 'AudioQuality' , baseStatemachine = 'Connected'}
«Pointcut» SelectedStates
«Introduction» after m seconds
«Introduction» CheckingAudioQuality
«Introduction» [audioQualityLoss > threshold] recoverAudioQuality ( )
«Introduction» RecoveringAudioQuality
«Introduction» [audioQualityLoss > =0 and audioQualityLoss <= threshold] continueOperation ( )
Introduction after n seconds restart
Pointcut { name = 'SelectStates', SelectionType = SelectionType::Subset, selectionConstraint = 'Context uml::Vertex self.name != 'Idle'}
«Introduction» [audioQualityLoss > = 0 and audioQualityLoss <= threshold] continueOperation ( )
«Introduction»
Figure 5. Aspect state machine for AQ

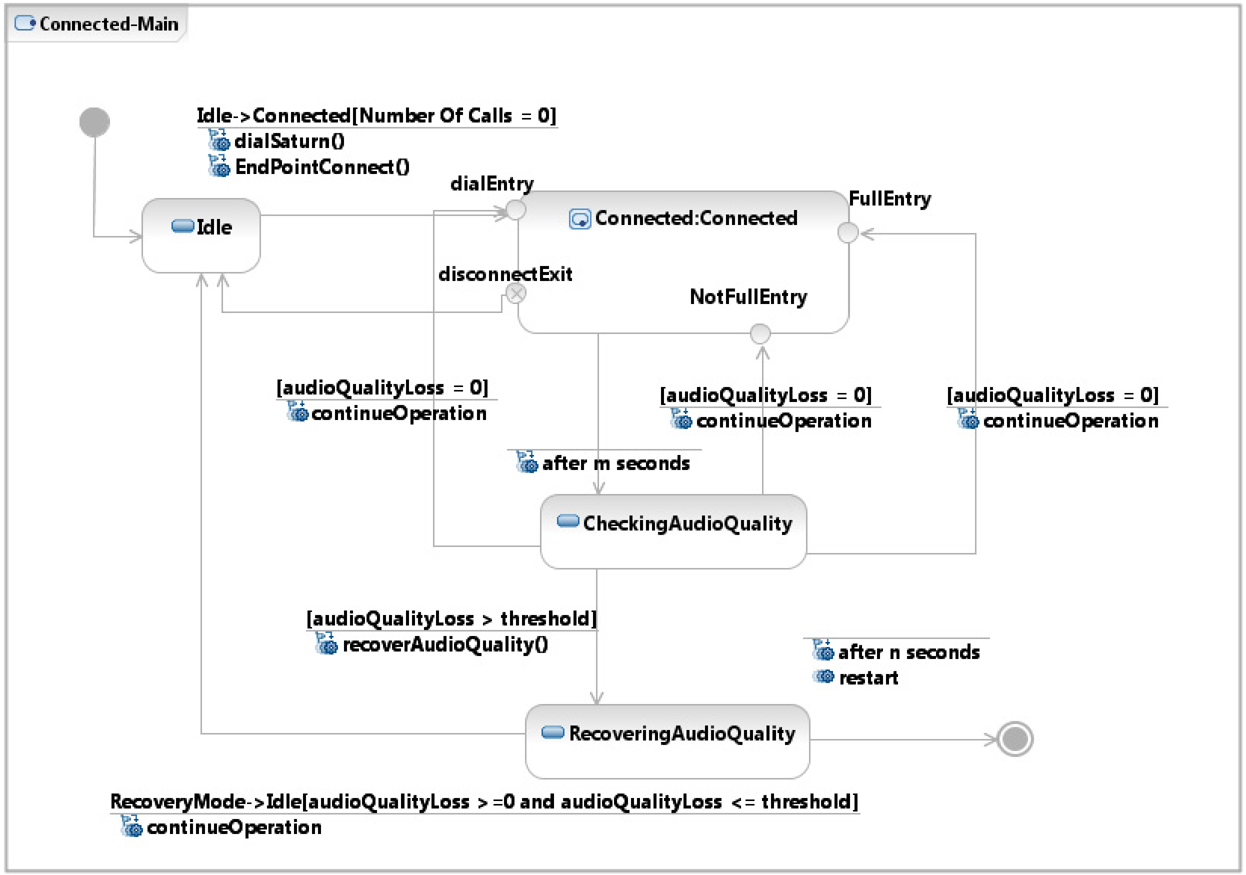


Figure 6. AQ modeled with the Hierarchical approach

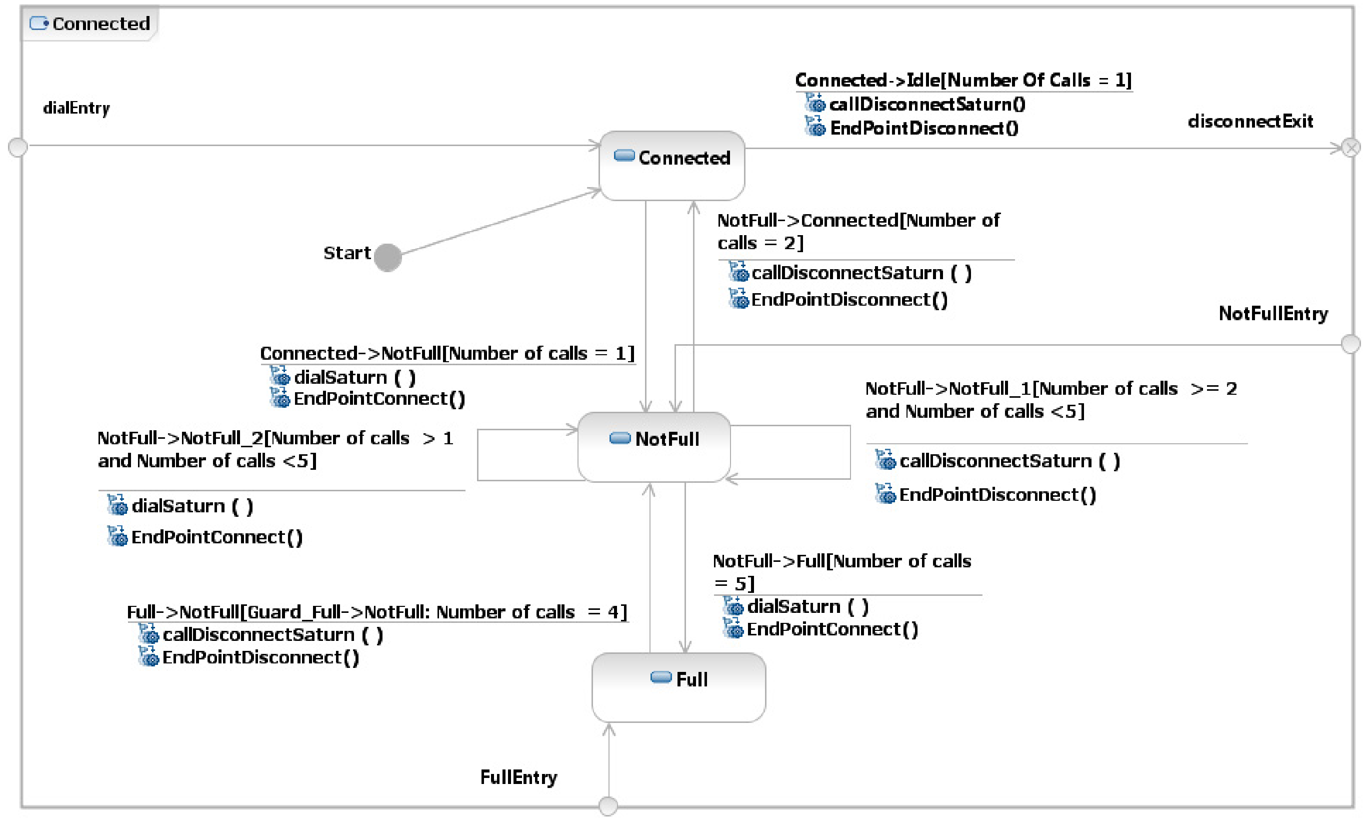


Figure 7. AQ modeled with the Hierarchical approach

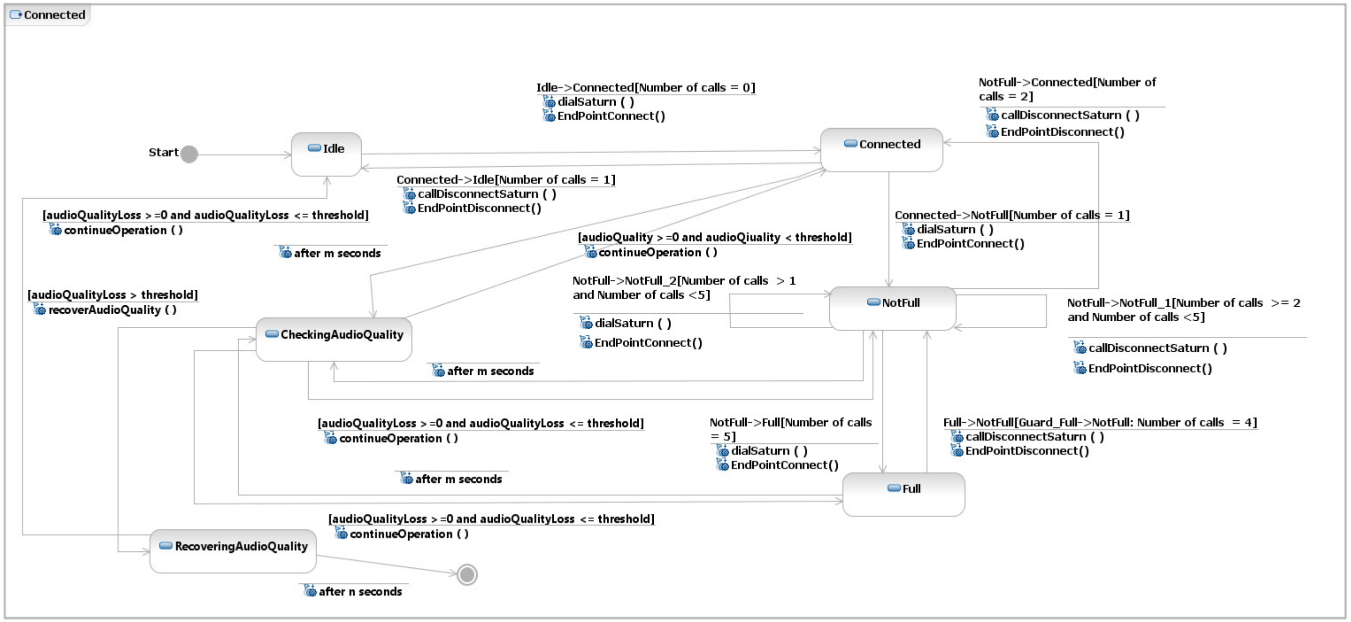

Connected
Idle->Connected[Number of calls = 0]
dialSaturn ( )
EndPointConnect()
NotFull->Connected[Number of calls = 2]
callDisconnectSaturn ( )
EndPointDisconnect()
Start
Idle
Connected
Connected->Idle[Number of calls = 1]
callDisconnectSaturn ( )
EndPointDisconnect()
[audioQualityLoss > =0 and audioQualityLoss <= threshold]
continueOperation ( )
Connected->NotFull[Number of calls = 1]
dialSaturn ( )
EndPointConnect()
after m seconds
[audioQuality > =0 and audioQiuality < threshold]
continueOperation ( )
NotFull->NotFull_2[Number of calls > 1 and Number of calls <5]
[audioQualityLoss > threshold]
recoverAudioQuality ( )
NotFull
NotFull->NotFull_1[Number of calls >= 2 and Number of calls <5]
CheckingAudioQuality
dialSaturn ( )
EndPointConnect()
callDisconnectSaturn ( )
EndPointDisconnect()
after m seconds
[audioQualityLoss > =0 and audioQualityLoss <= threshold]
continueOperation ( )
NotFull->Full[Number of calls = 5]
dialSaturn ( )
EndPointConnect()
Full->NotFull[Guard_Full->NotFull: Number of calls = 4]
callDisconnectSaturn ( )
EndPointDisconnect()
after m seconds
Full
[audioQualityLoss > =0 and audioQualityLoss <= threshold]
continueOperation ( )
RecoveringAudioQuality
after n seconds

Figure 8. AQ modeled with the Flat approach

Appendix B: Comprehension Questionnaire for Replication

1. Explain the possible subsequent scenario when *Saturn* is in a videoconference with three endpoints and audio quality is within the allowed threshold value?

2. Explain the possible subsequent scenario when *Saturn* is *Idle* and audio quality is greater than the allowed threshold?

3. Explain the possible subsequent scenario when *Saturn* couldn't recover audio quality and is in a videoconference with one endpoint?

4. Explain the possible subsequent scenario when *Saturn* is connected to one endpoint and *Do Not Disturb mode* is turned *On*?

5. Explain the possible subsequent scenario when *Saturn* is in *Do Not Disturb mode* and is connected to two endpoints and two more endpoints are dialing to *Saturn* at the same time?

6. Explain possible subsequent scenario when *Saturn* is in *Connected* state for *m* minutes?

7. Explain the possible subsequent scenario when *Saturn* is in *StandbyOn* and an endpoint dials to it?

8. Explain the possible subsequent scenario when *Saturn* is *Idle* and an endpoint dials to it with H254 videoconference protocol?

9. Explain the possible subsequent scenario when *Saturn* is in a videoconference with two endpoints with SIP protocol and a third endpoint dials to Saturn with H323 videoconference protocol?

10. Explain the scenario when *Saturn* is in a videoconference with six endpoints with SIP protocol and another endpoint dials to Saturn with H323 videoconference protocol?

Solving OCL Constraints for Test Data Generation in Industrial Systems with Search Techniques

Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

Submitted in the ACM Transactions on Software Engineering and Methodology. Conference version appeared in : In: Proceedings of the 11th International Conference on Quality Software (QSIC 2011), pp. 41-50, IEEE, 2011

Abstract—Model-based testing (MBT) aims at automated, scalable, and systematic testing solutions for complex industrial software systems. To increase chances of adoption in industrial contexts, software systems should be modeled using well-established standards such as the Unified Modeling Language (UML) and Object Constraint Language (OCL). Given that test data generation is one of the major challenges to automate MBT, this is the topic of this paper with a specific focus on solving OCL constraints, which is a necessary step to generate appropriate test data. Though search-based software testing has been applied to test data generation for white-box testing (e.g., branch coverage), its application to the MBT of industrial software systems has been limited. In this paper, we propose a set of search heuristics targeted to OCL constraints to guide test data generation and automate MBT in industrial applications. These heuristics are used to develop an efficient OCL solver exclusively based on search. In this paper, we evaluate these heuristics for search algorithms such as Genetic Algorithms, (1+1) Evolutionary Algorithm and Alternating Variable Method. We empirically evaluate our heuristics using complex artificial problems followed by empirical analyses to evaluate the feasibility of our approach on one industrial system. Though the focus is on OCL constraints, many of the principles introduced here could be adapted to other high level constraint languages based on first-order logic and set theory.

1. Introduction

Model-based testing (MBT) has recently received increasing attention in both industry and academia [1]. MBT leads to systematic, automated, and thorough system testing, which would often not be possible without models. However, the full automation of MBT, which is a requirement for scaling up to real-world systems, requires supporting many tasks, including preparing models for testing (e.g., flattening state machines), defining appropriate test strategies and coverage criteria, and generating test data to execute test cases. Furthermore, in order to increase chances of adoption, using MBT for industrial applications requires using well-established standards, such as the Unified Modeling Language (UML) and its associated language to write constraints: the Object Constraint Language (OCL) [2].

OCL is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and set theory. It is at a higher expressive level than Boolean predicates written in programming languages such as C and Java. The language allows modelers to write constraints at various levels of abstraction and for various types of models. For example, it can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post conditions of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) [3] (which is a standard defined by Object Management Group (OMG) for defining meta-models). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles, which is a standard way of extending UML for various domains using pre-defined extension mechanisms.

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger and/or state variables) to achieve full transition coverage, then it is essential to provide input values to the event that triggers the transition such that the values satisfy the guard. Another example can be to generate valid parameter values based on the pre-condition of an operation.

Test data generation is an important component of MBT automation. For UML models, with constraints in OCL, test data generation is a non-trivial problem. A few approaches in the literature exist that address this issue. But most of them, as we will explain in more details later in the paper, either target only a small subset of OCL [4, 5], are not scalable, or lack proper tool support [6]. This is a major limitation when it comes to the industrial application of MBT approaches that use OCL to specify constraints on models.

This paper provides and assesses novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), and Alternating Variable Method (AVM), to the solving of OCL constraints (covering the entire OCL 2.2 semantics [2]) in order to generate test data. A search-based OCL constraint solver is implemented and evaluated on the first reported, industrial case study on this topic. Note that many of the principles introduced here could be easily adapted to other high-level constraint languages based on first-order logic and set theory, in other modeling languages, and this makes our contribution of general value for test data generation based on models with constraints.

The rest of the paper is organized as follows: Section 2 discusses the background and Section 3 discusses related work. In Section 4, we present the definition of distance function for various OCL constructs. Section 5 discusses the case study and analysis of results of the application of the approach on an industrial case study, whereas Section 6 provides an empirical evaluation of heuristics on a set of artificial problems, and Section 7 provides an overall discussion of the both empirical evaluations. Section 8 discusses the tool support, Section 9 addresses the threats to validity of our empirical study, and finally Section 10 concludes the paper.

2. Background

Several software engineering problems can be reformulated as a search problem, such as test data generation [7]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good’’ solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [8-10].

To use a search algorithm, a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search algorithms toward fitter solutions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms (GAs) are the most well-known [8], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. Alternating Variable Method (AVM) is a local search algorithm, which is similar to the Hill Climbing algorithm, with the main difference that it can have larger modifications. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (RS) as comparison baseline [8].

3. Related Work

There are a number of approaches that deal with the evaluation of OCL constraints. The basic aim of most of these approaches is to verify whether the constraints can be satisfied. Though most of the approaches do not generate test data, they are still related to our work since they require the generation of values for validating the constraints. These approaches can therefore be adapted for generating test data. In Section 3.1, we discuss the relationship between OCL solvers and OCL evaluators. In Section 3.2, we discuss the OCL-based constraint solving approaches in the literature, whereas Section 3.3 discusses the approaches that use search-based heuristics for testing.

3.1 Comparison with OCL Constraints Evaluation

An OCL evaluator tells whether a constraint on a class diagram satisfies an instantiation of the class diagram provided to it. Several OCL evaluators are currently available that can be used to evaluate OCL constraints such as the IBM OCL evaluator [22], OCLE 2.0 [23], EyeOCL [24], and the OCL evaluation in CertifyIt by Smartesting [25]. Our work is different from these works since we automatically generate instances of a class diagram with the aim of finding a particular instantiation that solves a provided constraint. Note that for our purpose, i.e., solving OCL constraints to generate test data, an OCL evaluator is a necessary component because of two reasons: 1) an evaluator tells if a constraint is solved, 2) an evaluator helps in calculating the fitness (e.g., using a branch distance [26]) of an OCL expression that guides a search algorithm to find a solution. Note that any OCL evaluator can be integrated with our tool.

Table 1: Summary of OCL Constraint Solving Approaches

Technique	Translation to Formalism	Intermediate Representation	Complete OCL	OCL Parts Missing or Additional Requirements
Alloy Analyzer [11]	Yes	Alloy	No	Real, String, Enumerations, Limited operations on collections, attributes
Aertryck & Jensen [4]	Yes	FSA	No	Collections, Real, String, Enumerations
Diestefano et al. [12]	Yes	BOTL	No	String, real, enumerations
Clavel et al. [13]	Yes	FOL	No	String, Real, collections other than Set, Enumeration
Bao-Lin et al. [6]	No	DNF	No	Not discussed in the paper
Benattou et al. [5]	No	DNF	No	Class Inheritance, Generalization, Association
Aichernig [14]	Yes	CSP	No	Handles a small subset, collections iterators, Bag, Sequence,
UMLtoCSP [15]	Yes	CSP	No	Enumerations
Queralt et al [16]	Yes	FOL	No	Operations that cannot be converted to select() or size() operations, e.g., collect.
Winkelman [17]	Yes	Graph constraints	No	Collection operations except size(), isEmpty(). Enumerations
Kyas et al [18]	Yes	PVS	No	Not discussed in the paper
Kreiger [19]	Yes	SAT in CNF	No	Adds a non standard extension, String, Real, Enumerations
Weißler [20]	No	Test Tree	No	Collections, Enumerations
Gogolla [21]	Yes	Formal Logic	No	Desired properties of snapshot to be specified in a language ASSL

3.2 OCL-based Constraint Solvers

A number of approaches use constraint solvers for analyzing OCL constraints for various purposes. These approaches usually translate constraints and models into a formalism (e.g., Alloy [11], temporal logic BOTL [12], FOL [13], Prototype Verification System (PVS) [18], graph constraints [17]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [27], model checker [12], Satisfiability Modulo Theories (SMT) Solver [13], theorem prover [13], [18]). Satisfiability Problem (SAT) solvers have also been used for evaluating OCL specifications ,e.g., for OCL operation contracts (e.g., [28], [19]).

Table 2: Summary of OCL Constraint Solving Approaches

Technique	**Tool Support**	**Application on Case Study**	**Approach Type**	**Test Data Generation**
Alloy Analyzer [11]	Yes	Simple example	SAT Solver	No
Aertryck & Jensen [4]	Yes	Simple example	SAT Solver	Yes
Diestefano et al. [12]	Yes	Simple example	Model Checking	No
Clavel et al. [13]	Yes	Simple example	SMT Solver	No
Bao-Lin et al. [6]	No	Simple example	Partition Analysis	Yes
Benattou et al. [5]	No	Simple example	Partition Analysis	Yes
Aichernig [14]	Yes	Simple example	CSP Solving	No
UMLtoCSP [15]	Yes	Simple example	CSP Solving, Instance Generation	No
Queralt et al [16]	No	No	Reasoning	No
Winkelman [17]	No	No	Instance Generation	No
Kyas et al [18]	Yes	Simple example	Theorem Proving, Interactive	No
Kreiger [19]	Yes	Simple example	SAT Solver	No
Weißler [20]	Yes	Simple example	Partition Testing	Yes
Gogolla [21]	Yes	Simple example	Interactive	No

Some approaches are reported in the literature to solve OCL constraints and generate data that evaluates the constraints to true. The data generated can then be used as test data. Most of these approaches only handle a small subset of OCL and UML, and are based on formal constraint solving techniques, such as SAT solving (e.g., [4]), constraint satisfaction problem (CSP) (e.g., [14], [15]), higher order logic (HOL) [29], and partition analysis (e.g., [6], [5]). The work presented in [15] is one of the most sophisticated approaches reported so far. However, its focus is on the verification of correctness properties, though it generates an instantiation of the model as part of its process. The major limitation of the approach is that the search space is bounded and, as the bounds are raised, CSP faces a quickly increasing combinatorial explosion (as discussed in [15]). The task of determining the optimal bounds for verification is left to the user, which is not simple and requires repeated interactions with the user. Models of industrial applications can have hundreds of attributes and manually finding bounds for individual attributes is often impractical. We present the results of an experiment that we conducted to compare our novel approach with this approach in Section 6. Existing approaches for OCL constraint solving do not fully fit the needs we identified with our industrial partners. Almost all of the existing works only support a small, insufficient subset of OCL (Table 1 and Table 2). Most of the approaches, as shown in Table 1, are only limited to simple numerical expressions and do not handle collections, which are used widely to specify expressions that navigate over associations. These limitations are due to the high expressiveness of OCL that makes the definitions of constraints easier, but their analysis more difficult. The conversion of OCL to a SAT formula or a CSP instance can easily result in a combinatorial explosion as the complexity of the model and constraints increases (as discussed in [15]) . For instance, one factor that could easily lead to a combinatorial explosion, when converting an OCL constraint into an instance of SAT formula, is when the number of variables and their ranges increase in a constraint. Conversion to a SAT formula requires that a constraint must be encoded into Boolean formulae at the bit-level and as the number of variables increases in the constraint, chances of a combinatorial explosion increase. For industrial scale systems, as in our case, this is a major limitation, since the models and constraints are generally quite complex. Most of the discussed approaches either do not support the OCL constructs present in the constraints that we have in our industrial case study or are not efficient to solve them (see Section 6). Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems.

Earlier we discussed approaches that convert OCL expressions to other constraint languages. There are a number of other constraint solvers (such as in [30] [31]) that have their own constraint solving languages. To the best of our knowledge, mappings from OCL constraints to these constraint languages have not been reported. If such mappings were provided, they might entail the same limitations as the existing approaches based on mappings and translation, due to the gap in abstraction and level of expressiveness between OCL and the target languages. For instance, in Gecode [30], only the *Set* data type is supported, whereas the OCL supports many other collection data types, e.g., *Bag*, which cannot be directly translated into a Set (bags allow duplicated elements, whereas sets do not). As a consequence, it does not seem trivial (if possible at all) to translate the constraints using bags. COMET [30] is another constraint solver that requires constraints to be programmed in its own constraint programming language, which is a superset of Java/C++. Similar to Gecode, full translation of OCL into this language is either complex or not possible at all, e.g., COMET also only supports *Set*. Moreover, OCL supports OCL-specific data types such as *OCLAny*, *OCLVoid*, and *OCLInvalid* and UML-specific data types such as *OCLState* and *OCLMessage*, which may not be directly translated. In addition, there are several OCL-specific operations such as *oclIsValid()* and *oclIsUndefined()* and UML-specific operations such as *oclIsInState()* and *isSignalSent()*, which are dependent on UML semantics.

Even when a translation of OCL to other constraint languages is feasible, such translation would incur a significant computational overhead, particularly in cases where there are significant differences in abstractions and no straightforward mappings. Depending on the time that a constraint solver takes to solve a constraint, such extra overhead might not be negligible when comparisons are made with solvers that work directly on OCL. To complicate things even further, even if we wanted to use a constraint solver that does not handle OCL directly, we would not only need to translate OCL constraints, but we would also need to translate metamodels/models (e.g., state machines) into the respective language of those constraint solvers. On the other hand, our constraint solver fully supports UML and the UML profiling mechanism, thus enabling the solving of constraints even on profiled models. This is one of the requirements in many of the case studies of our industrial partners, where we have to solve constraints on profiled UML models.

Most of the above approaches are different from our work, since we want to generate test data based on OCL constraints provided by modelers on UML state and class diagrams. These diagrams may be developed for environment models (for example, as in [32]) or system models (for example [33]) and the modeler should be allowed to use the entire standard (OCL 2.2). We want to provide inputs for which the constraints are satisfied, and not just verify if inputs comply with them. We also want a tool that can be easily integrated with different state-based testing approaches and is completely automated.

3.3 Search-based Heuristics for Model Based Testing

The application of search-based heuristics for MBT has received significant attention recently (e.g., [34], [35]). The idea of these techniques is to apply heuristics to guide the search for test data that should satisfy different types of coverage criteria on state machines, such as state coverage. Achieving such coverage criteria is far from trivial since guards on transitions can be arbitrarily complex. Finding the right inputs to trigger these transitions is not simple. Heuristics have been defined based on common practices in white-box, search-based testing, such as the use of branch distance and approach level [26]. Our goal is to tailor these heuristics to OCL constraint solving for test data generation. Instead of using search algorithms, another possible approach to cope with the combinatorial explosion faced in solving OCL constraints could be to use hybrid approaches that combine formal techniques (e.g., constraint solvers) with random testing (e.g. [36]). However, we are aware of no work on this topic for OCL and, even for common white-box testing strategies, performance comparisons of hybrid techniques with search algorithms are rare [37].

4. Definition of the Fitness Function for OCL

To guide the search for test data that satisfy OCL constraints, it is necessary to define a set of heuristics. A heuristic tells *'how far'* input data are from satisfying the constraint. For example, let us say we want to satisfy the constraint *x=0*, and suppose we have two data inputs: *x1:=5* and *x2:=1000*. Both inputs *x1* and *x2* do not satisfy *x=0*, but *x1* is heuristically closer to satisfy *x=0* than *x2*. A search algorithm would use such a heuristic as a fitness function, to reward input data that are closer to satisfy the target constraint.

In this paper, to generate test data to solve OCL constraints, we use a fitness function that is adapted from work done for code coverage (e.g., for branch coverage in C code [26]). In particular, we use the so called branch distance (a function *d()*), as defined in [26]. The function *d()* returns 0 if the constraint is solved, otherwise a positive value that heuristically estimates how far the constraint was from being evaluated to true. As for any heuristic, there is no guarantee that an optimal solution (e.g., in our case, input data satisfying the constraints) will be found in reasonable time, but nevertheless many successful results based on such heuristics are reported in the literature for various software engineering problems [7]. In cases where we want a constraint to evaluate to false, we can simply negate the constraint and find data for which the negated constraint evaluates to true. For example, if we want to prevent firing a guarded transition in a state machine, we can simply negate the guard and find data for the negated guard.

In this section, we give examples of how to calculate the branch distance for various kinds of OCL expressions including primitive data types (such as *Real* and *Integer*) and collection-related types (such as *Set* and *Bag*). In OCL, all data types are subtypes of *OCLAny*, which is categorized into two subtypes: primitive types and collection types. Primitive types are *Real*, *Integer*, *String*, and *Boolean*, whereas collection types include *Collection* as super type with subtypes *Set*, *OrderedSet*, *Bag*, and *Sequence*. A constraint can be seen as an expression involving one or more *Boolean* clauses connected with logical operators such as and and or. A constraint can be defined on variables of different types, such as equalities of integers and comparisons of strings. As an example, consider the UML class diagram in Figure 1 consisting of two classes: *University* and *Student*. Constraints on the class *University* are shown in Figure 2.

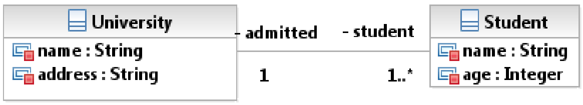


Figure 1. Example class diagram

```
context Student inv ageConstraint:
        self.age>15

context University inv numberOfStudents:
        self.student->size() > 0
```

Figure 2. Example constraints

The first constraint states that the age of every Student should be greater than 15. Based on the type of attribute age of the class Student, which is Integer, the comparison in the clause is determined to involve integers. The second constraint states that the number of students in the university should be greater than 0. In this case, the *size()* operation, which is defined on collections in OCL and returns an Integer denoting the number of elements in a collection, is called on collection student (containing elements of the class Student). Even though an operation is called on a collection, the comparison is between two integers (return value from operation *size()* and *0*).

Following we will discuss branch distance functions for different types of clauses in OCL.

4.1 Primitive types

A Boolean variable b is either true (when the branch distance is 0, i.e., *d(b)=0*), or false (when *d(b)=k*, where *k* is an arbitrary positive constant, for example *k=1*). If a *Boolean* variable is obtained from an operation call, then in general the branch distance would take one of only two possible values (*0* or *k*). For example, when the operation *isEmpty()* is called on a collection, the branch distance would either take *0* or *k*, unless a more fine grained specialized distance calculation is specified (e.g., returning the number of elements in the collection). For some types of OCL operations (e.g., *forAll()*) we can provide more fine grained heuristics. We will provide more details on these operations and their corresponding branch distance calculations later in Section 4.2.

Table 3. Branch distance calculations for OCL's operations for Boolean

Boolean operations	Distance function
A	if A is true then 0 otherwise k
not A	if A is false then 0 otherwise k
A and B	d(A)+d(B)
A or B	min (d(A),d(B))
A implies B	d(not A or B)

if A then B
else C
	d((A and B) or (not A and C))

A xor B	d((A and not B) or (not A and B))

* A and B are Boolean expressions or variables.

Table 4. Branch distance calculations of OCL's relational operations for numeric data

Relational operations	Distance function
x=y	if abs(x-y) = 0 then 0 otherwise abs(x-y)+k
x<>y	if abs(x-y) <> 0 then 0 otherwise k
x<y	if x-y < 0 then 0 otherwise (x-y)+k
x<=y	if x-y <= 0 then 0 otherwise (x-y)+k
x>y	if (y-x) < 0 then 0 otherwise (y-x)+k
x>=y	if (y-x) <= 0 then 0 otherwise (y-x)+k

The operations defined in OCL to combine *Boolean* clauses are *or*, *xor*, and, *not*, *if then else*, and *implies*. For these operations, branch distances are adopted from [26] since they work in a similar way as in programming languages and are shown in Table 3. Operations *implies*, and *xor* are syntax sugars that usually do not appear in programming languages such as C and Java, and can be re-expressed using combinations of and and or operators. The evaluation of *d()* on a predicate composed by two clauses is specified in Table 3 and can simply be computed for more than two clauses recursively.

```
if not (C1.oclIsKindOf(C2))
    d(C1=C2) := 1
otherwise if C1→size() <> C2→size()
    d(C1=C2) := 0.5 + 0.5*nor(d (C1→size()=C2 → size()))
otherwise
    $d(C1 = C2) := 0.5 * \sum_{i=1}^{C1\to size()} nor(d(pair_i)) \Big/ C1 \to size()$
where, d(pair_i) is the distance between elements in the ith
position in the two sorted collections, e.g.,
d(C1.at(i)=C2.at(i)) and nor is a normalizing function [38]
defined as nor(x)=x/(x+1). Suppose C1 and C2 are two OCL
collections.
```

Figure 3. Branch distance equality of collections

When a predicate or one of its parts is negated, then the predicate is transformed by moving the negation inward to the basic clauses, e.g., *not (A and B)* would be transformed into *not A or not B*.

For the numeric data types, i.e., *Integer* and *Real*, the relational operations that return *Booleans* (and so can be used as clauses) are <, >, <=, >=, and <>. For these operations, we adopted the branch distance calculation from [26] as shown in Table 4. In OCL, several other operations are defined on Real and Integer such as +, -, *, /, *abs()*, *div()*, *mod()*, *max()*, and *min()*. Since these operations are not used to compare two numerical values in clauses, there is no need to define a branch distance for them. For example, considering *a and b* of type *Integer* and a constraint *a+b*3<4*, then the operations + and * are used only to define the constraint. The overall result of the expression *a+b*3* will be an *Integer* and the clause will be considered as a comparison of two values of *Integer* type. For the *String* type, OCL defines several operations such as =, +, *size()*, *concat()*, *substring()*, and *toInteger()*. There are only three operations that return a Boolean: equality operator =, inequality <> and *equalsIgnoreCase()*. In these cases, instead of using k if the comparisons are false, we can return the value from any string matching distance function to evaluate how close any two strings are. In our approach, we implemented the edit distance [9] function, but any other string matching distance function can easily be incorporated.

4.2 Collection-Related Types

Collection types defined in OCL are *Set*, *OrderedSet*, *Bag*, and *Sequence*. Details of these types can be found in the standard OCL specification [2]. OCL defines several operations on collections. An important point to note is that, if the return type of an operation on a collection is *Real* or *Integer* and that value is used in an expression, then the distance is calculated in the same way as for primitive types as defined in Section 4.1. An example is the *size()* operation, which returns an Integer. In this section, we discuss branch distances for operations in OCL that are specific to collections, and that usually are not common in programming languages for expressing constraints/predicates and hence are not discussed in the literature.

4.2.1 Equality of collections (=)

In OCL constraints, we may need to compare the equality of two collections. We defined a branch distance for comparing collections as shown in Figure 3. The main goal is to improve the search process by providing a more fine grained heuristic than using a simple heuristic which simply calculates 0 if the result of an evaluation is true and *k* otherwise. In Figure 3, a branch distance for equality (=) of collections is calculated in one of the following three ways.

First, if collections *C1* and *C2* are not of the same kind, i.e., *not (C1.oclIsKindOf(C2))* evaluates to *true*, then the distance is simply *1*. Note that any other constant could have been used to represent the maximum distance. Whenever, the distance is *1*, it means that the collections are of different types, and the search algorithms must be guided to make the two collections of the same types.

Once the first condition is satisfied, the search algorithms must be guided such that the collections have equal number of elements. The second condition in the formula checks if the collections, which are of the same type, have different sizes. In that case, the search is guided to generate collections of equal size, i.e., *C1 → size()=C2 → size()*. We compute *d(C1 → size()=C2 → size())* and since *size()* returns an integer, this distance calculation is simply performed using the equality operation on numerical data as shown in Table 4. The maximum distance value that can be taken by *d(C1=C2)* in this case can be derived as follows:

d(C1=C2) = 0.5 + 0.5*nor(d (C1 → size()=C2 → size()))
using the formula of equality for numerical data from Table 4, we can derive:
d(C1=C2) = 0.5 + 0.5*nor(abs(C1 → size()-C2 → size())+k)
using the definition of nor(X), and suppose Y= abs(C1 → size()-C2 → size())+k, we can derive
d(C1=C2) = 0.5 + 0.5*(Y/ (Y+1))

In the above equation, *Y/(Y+1)* always computes a value less than *1*. Equation *0.5 + 0.5*(Y/ (Y+1))* therefore always takes a value between *0.5* to and *1*. Whenever, *d(C1,C2)* is greater than *0.5* and less than *1*, this means that collections do not have the same number of elements.

Table 5. Minimum and Maximum Distance Values For Distance Calculation for Equality Of Collections

Condition	**Minimum**	**Maximum**
not (C1.oclIsKindOf(C2))	1	1
C1 → size() <> C2 → size()	>=0.5	<1
not (C1.oclIsKindOf(C2)) and C1 → size() = C2 → size()	0	<0.5

For the third condition, i.e., if collections are of the same type and have equal numbers of elements, the distance is calculated based on comparing elements in both collections. First, we sort both collections based on their elements, regardless of type of the collection (i.e., whether they are sets, bags or sequences). For sorting, a natural order among the elements must be defined. For instance, if collections consist of integers, then we simply sort based on the integer values. However, for other types of elements (e.g., enumerations), there is no pre-defined natural order and, in these cases, we sort using the name of the identifiers of elements (e.g., a sequence of enumerations *{B, A, D, C}* would be sorted into *{A, B, C, D}*). If a collection consists of collections, we flatten the structure until we reach the primitive types and sort them based on all primitive types. Notice that how the sorting is done is not important. The important property that needs to be satisfied is that, if two collections are equal (regardless of the type of collection), then the sorting algorithm should produce the same paired alignment. For example, the set *{B, A, C}* is equal to *{C, B, A}* (the order in the sets has no importance), and their alignment using the name of enumeration elements produces the same sorted sequence *{A, B, C}*.

Once the element of both collections are sorted, we sum the distances between each pair of elements in the same position in the collections (i.e., distance between the i_{th} element of *C1* with the i_{th} element of *C2*) and finally take the average by dividing the sum with the number of elements in *C1*. When all elements of C1 are equal to *C2*, then *d(pair)* yields *0* and as a result *d(C1=C2) = 0*. The maximum value *d(C1=C2)* can take in this case can be derived as follows:

$$\mathrm{d(C1 = C2)} \coloneqq 0.5 * \sum_{\mathrm{i=1}}^{\mathrm{C1 \to size()}} \mathrm{nor(d(pair_i))} \Big/ \mathrm{C1 \to size()}$$

Using definition of nor, the above equation can be rewritten as:

$$\mathrm{d(C1 = C2)} := 0.5 * (\sum_{\mathrm{i=1}}^{\mathrm{C1 \to size()}} \mathrm{d(pair_i)/(d(pair_i) + 1)} \Big/ \mathrm{C1 \to size()})$$

d(pair$_i$)/(d(pair$_i$)+1) will always compute a value less than *1*. Considering a simple example, in which collections consists of Boolean values, using the formula from Table 3, *d(pair$_i$)* can take k as the maximum value. So the formula will be reduced to:

$$\mathrm{d(C1 = C2)} := 0.5 * (\sum_{\mathrm{i=1}}^{\mathrm{C1 \to size()}} \mathrm{k/(k + 1)} \Big/ \mathrm{C1 \to size()})$$

$$\mathrm{d(C1=C2)} := 0.5 * (\mathrm{k}/(\mathrm{k} + 1))$$

Since *(k/(k+1))* computes a value below one, the above formula will always compute a value below 0.5. To further explain the computation of branch distance, when condition *not (C1.oclIsKindOf(C2)) and C1 → size() = C2 → size()* is *true*, we provide an example below:

Example 1: Suppose *C1= {2,1,3}*, *C2 = {5,4,9}*, then the distance will be calculated as follows:

$$\mathrm{d(C1 = C2)} := 0.5 * \sum_{\mathrm{i=1}}^{\mathrm{C1 \to size()}} \mathrm{nor(d(pair_i))} \Big/ \mathrm{C1 \to size()}$$

$$\mathrm{d(C1 = C2)} := 0.5 * \sum_{\mathrm{i=1}}^{3} \mathrm{nor(d(pair_i))} \Big/ 3$$

After sorting, C1 and C2 will be {1,2,3} and {4,5,9} respectively.

$d(C1 = C2) := 0.5 * (nor(d(1 = 4)) + nor(d(2 = 5)) + nor(d(3 = 9)))/3$

Using k=1 and formula of equality of two numeric values from Table 4

$d(C1=C2) := 0.5*(nor(4)+nor(4)+nor(7))/3$

$d(C1=C2) := 0.28$

Table 6. Branch distance calculation for operations checking objects in collections

Operation	**Distance function**
includes (object:T): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } self\rightarrow size()} d(object = self.at(i))$
excludes (object:T): Boolean, where T is any OCL type	$\sum_{i=1}^{self\rightarrow size()} d(object <> self.at(i))$
includesAll (c:Collection(T)): Boolean, where T is any OCL type	$\sum_{i=1}^{self\rightarrow size()} \min_{j=1 \text{ to } c\rightarrow size()} d(c.at(i) = self.atj)$
excludesAll(c:Collection(T)): Boolean, where T is any OCL type	$\sum_{i=1}^{self\rightarrow size()} \sum_{j=1}^{self\rightarrow size()} d(c.at(i) <> self.atj)$
isEmpty(): Boolean	$d(self \rightarrow size() = 0)$
notEmpty(): Boolean	$d(self \rightarrow size() <> 0)$

As illustrated in Table 5 the three conditions in Figure 3 match three distinct value ranges, thus ensuring that the distance is always superior in the first case and the lowest in the third case, thus properly guiding the search.

4.2.2 Operations checking existence of one or more objects in a collection

OCL defines several operations to check the existence of one or more elements in a collection such as *includes()* and *excludes()*, which check whether an object does or does not exists in a collection, respectively. Whether a collection is empty is checked with *isEmpty()* and *notEmpty()*. Such operations can be further processed for a more refined calculation of branch distance than simply calculating a distance *0* when an expression is *true* and *k* otherwise. The refined calculations of branch distances for these operations are described in Table 6.

For includes *(object:T)*, a branch distance is the minimum distance from all distances (calculated using the heuristic for equality as listed in Table 4) between object and each element of the collection (*self*) on which includes is invoked. When any element of self is equal to object, the distance will be *0*, and the overall distance will therefore be *0*. When none of the collection elements is equal to object, then we select the element in the collection with minimum distance. The example below illustrates how branch distance is calculated:

Example 2: Suppose *C= {1,2,3}* and we have an expression *C → includes(4)*, then the branch distance will be calculated as:

$$\mathrm{d}\ (\mathrm{C} \rightarrow \mathrm{includes}(4)) := \ \mathrm{min}_{\mathrm{i=1\ to\ self\rightarrow size()}}\ \mathrm{d}(\mathrm{object} = \mathrm{self.at(i)})$$

$$\mathrm{d}\ (\mathrm{C} \rightarrow \mathrm{includes}(4)) := \min_{\mathrm{i=1\ to\ 3}} \mathrm{d}(\mathrm{object} = \mathrm{C.at(i)})$$

$$\mathrm{d}\ (\mathrm{C} \rightarrow \mathrm{includes}(4)) := \mathrm{min}\ (\mathrm{d}(1 = 4), \mathrm{d}(2 = 4), \mathrm{d}(3 = 4)))$$

Using k=1, and formula of the equality of two integers from Table 4

$$\mathrm{d}\ (\mathrm{C} \rightarrow \mathrm{includes}(4)) := \mathrm{min}\ (4,3,2)$$

$$\mathrm{d}\ (\mathrm{C} \rightarrow \mathrm{includes}(4)) := 2$$

For excludes *(object:T)*, a branch distance is calculated in a similar way as includes, except that we use the distance heuristic for inequality (<>) and sum up the distances of all elements in the collection, which are equal to object. The example below illustrates how a branch distance is computed using the formula.

Table 7. Branch distance for forAll and exists

Operation	**Distance function**
forAll(v1,v2, …vm|exp)	if (self→size()) = 0 then 0 otherwise $\frac{\sum_{i_1=1}^{\mathrm{self\rightarrow size()}} \sum_{i_2=1}^{\mathrm{self\rightarrow size()}} \dots \sum_{i_m=1}^{\mathrm{self\rightarrow size()}} \mathrm{d(expr\ (self.at(i_1),...self.at(i_m))}}{(\mathrm{self\rightarrow size()})^m}$
exists(v1,v2, …vm|exp)	$\mathrm{min}_{i_1,i_2 \dots i_m \in \mathrm{1to\ self\rightarrow size()}}\ \mathrm{d(expr(self.at(i_1), \dots\ self.at(i_m))}$
isUnique(v1|exp)	$\frac{\sum_{i=1}^{(\mathrm{self\rightarrow size()}-1)} \sum_{j=i+1}^{(\mathrm{self\rightarrow size()})} \mathrm{d(expr(self.at(i))} <> \mathrm{expr(self.at(j)))}}{((\mathrm{self} \rightarrow \mathrm{size}()) * (\mathrm{self} \rightarrow \mathrm{size}() - 1))/2}$
one(v1|exp)	$\mathrm{d(self} \rightarrow \mathrm{select(exp)} \rightarrow \mathrm{size()}\ = \ 1)$

Example 3: Suppose *C= {1,2,2}* and we have an expression *C → excludes(2)*, then

$$\mathrm{d(C} \rightarrow \mathrm{excludes}(2)) := \sum_{\mathrm{i=1}}^{\mathrm{self\rightarrow size()}} \mathrm{d(object} <> \mathit{self.at}(\mathrm{i}))$$

$$\mathrm{d(C} \rightarrow \mathrm{excludes}(2)) := \sum_{\mathrm{i=1}}^{3} \mathrm{d(object} <> \mathit{C.at}(\mathrm{i}))$$

$\mathrm{d}(\mathrm{C} \rightarrow \mathrm{excludes}(2)) := \mathrm{d}(1 <> 2) + \mathrm{d}(2 <> 2) + \mathrm{d}(2 <> 2)$

Using k=1, and formula from Table 4

$\mathrm{d}(\mathrm{C} \rightarrow \mathrm{excludes}(2)) := 0 + 1 + 1$

$\mathrm{d}(\mathrm{C} \rightarrow \mathrm{excludes}(2)) := 2$

In a similar fashion, we calculate branch distance of *includesAll* and *excludesAll* (Table 6), where we check if all elements of one collection are present/absent in another collection. For *includesAll*, we sum, over all elements of a collection, their minimum distance among all the elements of another collection as shown in the formula for *includesAll* in Table 6. For *excludesAll*, we sum all distances between all possible pairs of elements across the two collections, as shown in the formula for *excludesAll* in Table 6. Branch distance calculations for *isEmpty* and *notEmpty* are also defined in Table 6.

4.2.3 Branch distance for iterators

OCL defines several operations to iterate over collections. Below, we will discuss branch distances for these iterators.

The *forAll()* iterator operation is applied to an OCL collection and takes as input a *boolean-expression*, then it determines whether the expression holds for all elements in the collection. To obtain a fine grained branch distance, we calculate the distance of the *boolean-expression* by computing the distance on all elements in the collection and summing the results. The function for *forAll* presented in Table 7 is generic for any number of iterators. For the sake of clarity in the paper, we assume that function $expr(v_1,v_2, \ldots v_m)$ in Table 7 evaluates an expression expr on a set of elements $v_1,v_2, \ldots v_m$. To explain expr, suppose we have a collection *C={1,2,3}* and an expression *C → forAll(x,y | x*y >0)*, then *expr(C.at(1),C.at(2))* entails calculating *"d(x*y>0)"*, where *x=C.at(1)*, i.e., *1* and *y= C.at(2)*, i.e., *2*. The keyword self in the table refers to the collection on which an operation is applied, *at(i)* is a standard OCL operation that returns the i_{th} element of a collection, and *size()* is another OCL operation that returns the number of elements in a collection. The denominator $(self \rightarrow size())^m$ is used to compute the average distance over all element combinations of size m since we have $(self \rightarrow size())^m$ distance computations. Notice that calculating the average distance is important to avoid bias towards decreasing the size of the collection. For example, since it is a minimization problem (i.e., we want to minimize the branch distance), there would be a bias against larger collections as they would tend to have a higher branch distance (there is a number of branch distance additions that is polynomial in the number of iterators and collection size). A search operator that removes one element from the collection would always produce a better fitness function, so it would have a clear gradient toward the empty collection. An empty collection would make the constraint true, but it can have at least two kinds of side effects: first, if a clause is conjuncted with other clauses that depend on the size (e.g., *C → forAll(x|x>5) and C → size()=10)*, then there would likely be plateaus in the search landscape (e.g., gradient to increase the size towards 10 would be masked by the gradient towards the empty collection); second, because in our context we solve constraints to generate test data, we want to have useful test data to find faults, and not always empty collections. In general, to avoid side effects such as unnecessary fitness plateaus, our branch distance functions are designed in a way that, if there is no need to change the size of a collection to solve a constraint on it, then the branch distances should not have bias toward changing its size in one direction or another.

Below, we further illustrate the branch distance for *forAll* with the help of examples:

Example 4: Suppose we have a collection *C= {1,2,3}* and the expression is *C → forAll(x|x=0)*. In this example, we have just one iterator *x*, and therefore *m=1*. In this case, the formula will be:

$$d(\mathrm{C} \rightarrow \mathrm{forAll}(\mathrm{x}|\mathrm{x} = 0)) := \sum_{i_1=1}^{\mathrm{C}\rightarrow\mathrm{size}()} d(\mathrm{expr}(\mathrm{C.at}(i_1)) / \mathrm{C} \rightarrow \mathrm{size}()$$

The branch distance in this case will be calculated as:

d(C→forAll(x | x=0)) := (d(expr(C.at(1))+ d(expr(C.at(2))+ d(expr(C.at(3)))/3

d(C→forAll(x | x=0)) := (d(expr(1)+ d(expr(2)+ d(expr(3))/3

Considering k=1 and using the definition of expr and formulae from Table 3 and Table 4

d(C→forAll(x | x=0)) := (2+3+4)/3

d(C→forAll(x | x=0)) :=9/3 = 3

Example 5: Suppose we have a collection *C= {1,2}* and the expression is *C → forAll(x,y|x*y >0)*. In this case, we have two iterators *x* and *y* and thus the formula will become:

$$d(C \rightarrow \text{forAll}(x,y|x*y>0)) := \frac{\sum_{i_1=1}^{C\rightarrow size()} \sum_{i2=1}^{C\rightarrow size()} d(expr(C.at(i_1), C.at(i_2))}{(C \rightarrow size())^2}$$

The branch distance in this case will be calculated as:

d(C→forAll(x,y | x*y>0)) ≔ (d(expr(C.at(1), C.at(1)))+ d(expr(C.at(1), C.at(2)))+ d(expr(C.at(2), C.at(1)))+ d(expr(C.at(2), C.at(2))))/4

d(C→forAll(x,y | x*y>0)) ≔ (d(expr(1, 1))+ d(expr(1, 2))+ d(expr(2, 1))+ d(expr(2, 2)))/4

d(C→forAll(x,y | x*y>0)) ≔ (d(1*1>0)+ d(1*2>0)+ d(2*1>0)+ d(2*2>0))/4

Considering k=1 and using formulae from Table 3 and Table 4
d(C→forAll(x,y | x*y>0)) ≔ (0+0+0+0)/4
d(C→forAll(x,y | x*y>0)) ≔ 0

In a similar fashion, the formula can be used for any number of iterators (*m*).

The *exists()* iterator operation determines whether a *boolean-expression* holds for at least one element of the collection on which this operation is applied. The generic distance form for *exists()* is shown in Table 7. The definition of *exists()* is very similar to *forAll()* except for two differences. First, instead of summing distances across all element combinations of size m, we compute the minimum of these distances, since any element satisfying exp makes *exists() true*. Second, we do not have a denominator since no average needs to be computed. The expr() function works in the same way as for *forAll()*. Below we further illustrate branch distance calculation using two examples.

Example 6: Suppose we have a collection *C= {1,2,3}* and the expression is *C → exists(x|x=0).* In this example, we have just one iterator, i.e., *x*. The formula will be:

$$d(C \rightarrow \text{exists}(x|x=0)) := \min_{i_1 \in 1 \text{to } 3} d(expr(C.at(i_1))$$

The branch distance in this case will be calculated as:

d(C→exists(x | x=0)) ≔ min (d(expr(C.at(1)), d(expr(C.at(2)), d(expr(C.at(3)))

d(C→exists(x | x=0)) ≔ min (d(expr(1), d(expr(2), d(expr(3))

Considering k=1, the definition of expr, and using formulae from Table 3 and Table 4
d(C→exists(x | x=0)) ≔ min (2,3,4)

d(C→exists(x | x=0)) ≔ 2

Example 7: Suppose we have a collection *C= {1,2}* and the expression is *C →*

*exists(x,y|x*y>1)*. In this case, we have two iterators *x* and *y* and thus the formula will become:

$$d(C \rightarrow \text{exists}(x, y|x * y > 0)) := \min_{i_1,i_2 \in 1 \text{to } C\rightarrow \text{size}()} d(\text{expr}(C.\text{at}(i_1), C.\text{at}(i_2))$$

The branch distance in this case will be calculated as:

d(C→exists(x,y | x*y>0)) ≔ min (d(expr(C.at(1), C.at(1))), d(expr(C.at(1), C.at(2))), d(expr(C.at(2), C.at(1))), d(expr(C.at(2), C.at(2))))

d(C→exists(x,y | x*y>0)) ≔ min (d(expr(1, 1)), d(expr(1, 2)), d(expr(2, 1)), d(expr(2, 2)))

d(C→exists(x,y | x*y>0)) ≔ min (d(1*1>1), d(1*2>1), d(2*1>1), d(2*2>1))

Considering k=1, the definition of expr, and using formulae from Table 3 and Table 4
d(C→exists(x,y | x*y>0)) ≔ min (1,0,0,0)

d(C→exists(x,y | x*y>0)) ≔ 0

In a similar fashion, as explained with Example 6 and Example 7, the formula can be used for any number of iterators (m).

Table 8. Special Rules for Select() Followed By Size() when exp is false

Operation	Distance function
>, >=	d(exp)= if C → size() <= z() then (z()-C → size()) + k else nor((z()- C → select(P) → size())+k + nor(d(P)))
<,<=	d(exp) = if C → size() >= z() then (C → size()-z())+k else nor((C → select(P)→size()-z())+k + nor(d(not P)))

<>	d(exp) = if C → select(P) → size() = 0 then d(P)
	if C → select(P) → size() = C → size() then d(not P)
	if 0 < C → select(P) → size() < C → size() then min(d(P), d(not P))

=	d(exp) = if C → select(P) → size() > z() then (C → select(P) → size()-z())+k + nor(d(not P))
	if C → select(P) → size() < z() then (z()- C → size())+k + nor(d(P))

* In the above table, *k=1*, *nor(x)=x/(x+1)* and *d(P)* is simply the sum of *d()* over the elements in *A*

In addition, we also provide branch distance for *one()* and *isUnique()* operations in Table 7. The one operation returns true only if *exp* evaluates to true for exactly one element of the collection. The *isUnique()* operation returns true if exp on each element of the source collection evaluates to a different value. In this case, the distance is calculated by computing and summing the distances between each element of the collection and every other element in the collection. Since in this formula, we are computing *(((self → size())*(self → size()-1)))/(2)* distances, we compute the average distance by using this formula in the denominator. Again, calculating the average distance is important to avoid bias in the search towards decreasing the size of the collection as we discussed for *forAll*. Below we provide an example of how we calculate branch distance for *isUnique()*.

Example 8: Suppose we have a collection *C= {1,1,3}* and the expression is *C → isUnique(x|x)*. In this example, we have just one iterator, i.e., *x*. Using the formula of branch distance for *isUnique()*,

$$d(C \rightarrow \text{isUnique}(x|x)) := \frac{\sum_{i=1}^{(C\rightarrow \text{size}()-1)} \sum_{j=i+1}^{(C\rightarrow \text{size}())} d(\text{expr}(C.\text{at}(i)) <> \mathit{expr}(C.\mathit{at}(j)))}{((C \rightarrow \text{size}()) * (C \rightarrow \text{size}() - 1))/2}$$

$d(C\rightarrow\text{isUnique}(x|x)) := (d(\text{expr}(C.\text{at}(1)) <> \text{expr}(C.\text{at}(2))) + d(\text{expr}(C.\text{at}(1) <> \text{expr}(C.\text{at}(3))) + d(\text{expr}(C.\text{at}(2)) <> \text{expr}(C.\text{at}(3))))/((3*2)/2)$

$d(C\rightarrow\text{isUnique}(x|x)) := (d(1 <> 1) + d(1 <> 3) + d(1 <> 3))/3$

$d(C\rightarrow\text{isUnique}(x|x)) := (1+0+0)/3$

$d(C\rightarrow\text{isUnique}(x|x)) := 1/3$

Select, reject, collect operations select a subset of elements in a collection. The *select()* operation selects all elements of a collection for which a *Boolean* expression is true, whereas *reject()* selects all elements of a collection for which a *Boolean* expression is false. In contrast, the *collect()* iterator may return a subset of elements that does not belong to the collection on which it is applied. Since all these iterators (like the generic iterator operation) return a collection and not a *Boolean* value, we do not need to define branch distance for them, as discussed in Section.4.1. However, an iterator operation (such as *select()*) followed by another OCL operation, for instance *size()*, can be combined to make a Boolean expression of the following form:

$$\text{exp} = C \rightarrow \text{selectionOp}\ (P) \rightarrow \text{size}()\ \ \text{RelOp}\ \ z()$$

Where *C* is a collection, *selectionOp* is either select, reject, or collect, *P* is a *boolean-expression*, *RelOp* is a relational operation from set {<,<=,=,<>,>,>=}, and *z()* is a function that returns a constant. A simple way of calculating branch distance for the above example, when *RelOp* is =, and *selectionOp* is *select* would be as follows:

```
exp = C → select (P) → size()  =  z()
If  exp = true then
  d(exp)=0
else
```

d(exp)= | C → select(P) → size()-z()|+k

An obvious problem of calculating branch distance in this way is that it does not give any gradient at all to help search algorithms solve *P*, which can be arbitrarily complex. To optimize branch distance calculation in this particular case, we need special rules that are defined specifically for each *RelOp*.

For > and >=, when *exp* is *false*, this means that the size of resultant collection of the expression *C → select(P)* is less than the size which will make the branch distance *0*. In this case, first we need a collection with size greater that *z()*, and then we need to obtain those elements of *A* that increase the value of *size()* returned by *C → select(P) → size()*. This can be achieved by the rule shown in the first row of Table 8. The normalization function *nor()* is necessary because the branch distance should first reward any increase in *C → size()* until it is greater than *z()* regardless of the evaluation of *P* on its elements. Then, once the collection *C* has enough elements, we need to account for the number of elements for which *P* is true by using *((z() – C → select(P) → size())+k)*. The function *d(P)* returns the sum of branch distance evaluations of a predicate *P* over all the elements in *C* and provides additional gradient by quantifying how close are collection elements from satisfying *P*. Below, we further illustrate this case with an example:

Example 9: Suppose we have a collection *C= {1,1,3}* and the expression is *C → select(x|x>1) → size()>=3*. Using the formula of branch distance for the case when *RelOp* is >, >=.

In this case, C→size() is 3, which is equal to z(), i.e., 3, so the formula for branch distance calculation is:

d(C → select(x|x>1) →size()>=3) :=nor((z()- C → select(P) → size())+k + nor(d(P)))

Assuming k=1,

d(C → select(x|x>1) → size()>=3) :=nor((3-1)+1 + nor(d(x>1)))

d(C → select(x|x>1) → size()>=3) :=nor(3 + nor(d(x>1)))

d(C → select(x|x>1) → size()>=3) :=nor(3 + nor(d(1>1)+d(1>1)+d(3>1)))

Using k=1, and formula from Table 4

d(C → select(x|x>1) → size()>=3) :=nor(3 + nor(1+1+0))

d(C → select(x|x>1) → size()>=3) :=nor(3 + 0.667)

$d(C \rightarrow select(x|x>1) \rightarrow size()>=3) :=0.78$

For < and <=, when exp is false, this means that *C → select(P) → size()* is greater than the size which will make the branch distance *0*. Similar to the previous case, the distance computation account for those elements of *C* that decrease the value of *size()* returned by *C → select(P) → size()*, and uses *nor(d(not P))* to provide additional gradient to the search, as shown in second row of Table 8.

For the cases when the value of *RelOp* is inequality (<>), the rule is shown in the third row of Table 8. Recall that our expression is in the following format: *exp = C → selectionOp (P) → size() RelOp z()*. For this rule, there are three cases based on the value of *C → select(P) → size()*. Recall that *d(P)* is simply the sum of all *d()* on all elements of *C*. The first case is when *C → select(P) → size() = 0*, where *P* does not hold for any element in *C*. To guide the search towards increasing the size of the collection, *d(exp)* will be *d(P)* so as to minimize the sum of distances of all elements with *P*. The second case is when *P* is *true* for all elements of *C*, which means that *C → select(P) → size() = C → size()*. To guide the search in decreasing the size of the collection, for reasons that are similar to the first case, we define *d(exp)* as *d(not P)*. When *0 < C → select(P) → size() < C → size()*, we can guide the search to either increase or decrease the size of the collection and thus define *d(exp)* as *min(d(P), d(not P))*.

For the cases when the value of *RelOp* is equality (=), the rule is shown in the fourth row of Table 8. There are two important cases, which work in a similar way as the first and second cases as reported in Table 8. The first case is when *C → select(P) → size() > z()*, where we need to decrease *C → select(P) → size()*, which can be achieved by minimizing *(C → select(P) → size()-z())+k + nor(d(not P))*. The second case is when *C → select(P) → size() < z()*. For this case, we need to increase the number of elements in *C* for which *P* holds and must minimize *(z()-C → select(P) → size())+k + nor(d(P))*.

Note that we only presented formulae in Table 8 for the cases when the iterator operation considered *selectionOp* is *select*, however, the formulae can simply be extended for other iterator operations. The *collect* operation works in the same way as *select*, and hence the formulae in Table 8 can simply be adapted by replacing *select* with *collect* in the formulae. For instance, for the case when *RelOp* is > or >=, formula for collect would be:

$$d(exp)= (z()\text{- } C \rightarrow \text{collect}(P) \rightarrow \text{size}())+k + nor(d(P))$$

The *reject* operation works in a different way than *select* since it rejects all those elements for which a *Boolean* expression is *true*. But *reject(P)* can be simply transformed into *select(not P)*.

In addition to the rules for an iterator followed by *size()*, we defined two new rules when a *select()* is followed by *forAll()* or *exists()* that are shown in Table 9. For example, *C → select(P1) → forAll(P2)* (first row in Table 9) implies that for all elements of *C* for which *P1* holds, *P2* should also hold. In other words, *P1 implies P2*. Therefore, *C → select(P1) → forAll(P2)* can simply be transformed into *C → forAll(P1 implies P2*). Similarly, a *select(P1)* followed by an *exists(P2)* can simply be transformed into *exists(P1 and P2)*. This means that there should be at least one element in *C* for which *P1* and *P2* holds. Notice that a sequence of selects can be simply combined, e.g., *C → select(P1) → select (P2)* is equivalent to *C → select(P1and P2)*.

The effectiveness of all these rules for calculating branch distance is empirically evaluated in Section 6.

4.3 Tuples in OCL

In OCL several different values can be grouped together using tuples. A tuple consists of different parts separated by a comma and each part specifies a value. Each value has an associated name and type. For example, consider the following example of a tuple in OCL:

Tuple{firstName = "John", age= 29}

This tuple defines a *String firstName* of value *"John"* and an *Integer age* of value *29*. Each value is accessed via its name. For example, *Tuple{firstName = "John",age= 29}.age* returns *29*. There are no operations allowed on tuples in OCL because they are not subtypes of *OCLAny*. However, when a value in a tuple is accessed and compared, a branch distance is calculated based on the type of the value and the comparison operation used. For example, consider the following constraint:

Tuple{String: firstName = "John", Integer: age= 29}.age > 20

In this case, since age is an Integer and comparison operation is >, we use the branch distance calculation of numerical data for the case of > as defined in Table 4.

4.4 Special Cases

In this section, we will discuss branch distance calculations for some special cases including enumerations and other special operations provided by OCL, such as for example *oclInState*.

Table 9. Special Rules for Select() Followed by ForALL and Exists

Operation	**Distance function**
C → select(P1) → forAll(P2)	d(C → forAll(P1 implies P2))
C → select(P1) → exists(P2)	d(C → exists (P1 and P2))

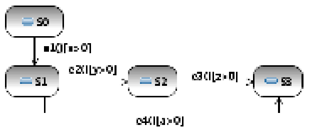


Figure 4. A dummy example to explain *oclInState()*

4.4.1 Enumerations

Enumerations are datatypes in OCL that have a name and a set of enumeration literals. An enumeration can take any one of the enumeration literals as its value. Enumerations in OCL are treated in the same way as enumerations in programming languages such as Java. Because enumerations are objects with no specific order relation, equality comparisons are treated as basic Boolean expressions, whose branch distance is either *0* or *k*.

4.4.2 oclInState

The *oclInState(s:OclState)* operation returns *true* if an object is in a state represented by *s*, otherwise it returns *false*. This operation is valid in the context of UML state machines to determine if an object is in a particular state of the state machine. *OclState* is a datatype similar to enumeration. This datatype is only valid in the context of *oclInState* and is used to hold the names of all possible states of an object as enumeration literals. In this particular case, the states of an object are not precisely defined,i.e., each state of the object is uniquely identified based on the names of the states. For example, a class *Light* having two states: *On* and *Off*, is modeled as an enumeration with two lietrals *On* and *Off*. In this example, *s:OclState* takes either *On* or *Off* value and the branch calculation is same as for enumerations. However, if the states are defined as state invariants, which is a common way of defining states in a UML state machine as an OCL constraint [2], then the branch distance is calculated based on two special cases depending on whether we can directly set the state of an object by manipulating the state variables or not. Below, we will discuss each case separately.

The first case is when the state of an object can be manipulated by directly setting its state defining attributes (or properties) to satisfy a state invariant. In this case, state invariants— which are OCL constraints—can be satisfied by solving the constraints based on heuristics defined in the previous sections. Note that each state in a state machine is uniquely identified by a state invariant and there is no overlapping between state invariants of any two states (strong state invariants [39]). For instance, in our industrial case study, we needed to emulate faulty situations in the environment for the purpose of robustness testing, which were modeled as OCL constraints defined on the properties of the environment. In this case, it was possible to directly manipulate the properties of the envrionment emulator based on which the state of the environment is defined and each state was uniquely identified based on its state invariant. A simple example of such state invariant for the environment is given below:

```
self.packetLoss.value > 5 and self.packetLoss.value <=10
```

The above state invariant defines a faulty situation in the environment, when the value of packet loss in the environment is greater than 5% and less or equal to 10%. This constraint can easily be solved using the heuristics defined in the previous sections and the value of packetLoss generated by our constraint solver can be directly set for the environment.

In the second case, when it is not possible to directly set the state of an object, the approach level heuristic [26] can be used in conjuction with branch distance to make the object reach the desired state. We will explain this case using a dummy example of a UML state machine shown in Figure 4. The approach level calculates the minimum number of transitions in the state machine to reach the desired state from the closest executed state. For instance, in Figure 4, if the desired state is *S3* and currently we are in *S1*, then the approch level is *1*. By calculating the approach level for the states that the object has reached, we can obtain a state that is closest to the desired state (i.e., it has the minimum approach level). In our example, the closest state based on the approach level is *S1*. Now, the goal is to transition in the direction of the desired state in order to reduce the approach level to *0*. This goal is achieved with the help of branch distance. The branch distance is used to heuristically score the evaluation of the OCL constraints on the path from the current state to the desired state (e.g., guards on transitions leading to the desired state). The distance is calculated based on the heuristics defined in this paper. The branch distance is used to guide the search to find test data that satisfy these OCL constraints. An event corresponding to a transition can occur several times but the transition is only triggered when the guard is true. The branch distance is calculated every time the guard is evaluated to capture how close the values used are from solving the guard. In the example, we need to solve guard *'a>0'* so that whenever *e4()* is triggered we can reach *S3*. Since the guards are written in OCL, they can be solved using the heuristics defined in the previous sections. In the case of MBT, it is not always possible to calculate the branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance its highest possible value. More details on this case can be found for example in [40].

4.4.3 oclIsTypeOf(),oclIsKindOf(), and oclIsNew()

These three operations are special operations defined for all objects in the OCL. The *oclIsTypeOf (t:Classifier)* returns true if *t* and the object on which this operation is called have the same type. The *oclIsKindOf(t:Classifier)* operation returns *true* if *t* is either the direct type or one of the supertypes of the object on which the operation is called. The operation *oclIsNew()* returns true if the object on which the operation is called is just created. These three operations are defined to check the properties of objects and hence are not used for test data generation, therefore we do not explicitly define branch distance calculation for these operations. However, whenever these operations are used in constraints, the branch distance is calculated as follows: if the invocation of an operation evaluates to *true*, then the branch distances is *0*, else the branch distance is *k*, as for any *boolean* function for which more fine grained heuristic is not provided.

4.4.4 User-defined Operations

Apart from the operations defined in the standard OCL library, OCL also provides a facility for the users to define new operations. Body of these operations is written using

OCL expressions and may call the standard OCL library operations. As we discussed in Section 4, we only provide specialized branch distance calculations for the operations defined in the standard OCL library. For user-defined operations, we calculate a branch distance according to the return types of these operations. If a user-defined operation returns a Boolean,to provide more fine grained fitness functions, it is possible to use testability transformations on those operations, as for example in search-based software testing of Java software [41]. In our tool, we have not implemented and evaluated this type of testability transformations, and further research would be needed to study their applications in OCL. For any other return type but Boolean, we define a branch distance using the rules defined in Section 4.4. For instance, consider a user-defined OCL operation named *operation1()*, which is defined on a collection and returns a collection, and the following constraint defined on it:

$$\mathrm{c1} \rightarrow \mathrm{operation1()} \rightarrow \mathrm{isEmpty()}$$

In this case, the branch distance is calculated based on the heuristic for *isEmpty()* as defined in Section 4.2.

Table 10. Statistics of Complexity of Constraints

# of Clauses	Frequency
8	1
7	8
6	23
5	10
2	6
1	9

Table 11. OCL Data Types Used in Constraints

OCL Data Types Used	Frequency
Integer	13
Boolean	2
Integer and Enumeration	31
Integer, Enumeration, and Boolean	11

```
context Saturn inv synchronozationConstraint:
      self.media.synchronizationMismatch.value  > self.media.synchronizationMismatchThreshold.value)
```

Figure 5. A constraint checking synchronization of audio and video in a videoconference

5. Case study: Robustness Testing of Video Conference System

This case study is part of a project aiming to support automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn [42], developed by Cisco Systems, Inc, Norway. Saturn is modeled as a UML class diagram meant to capture information about APIs and system (state) variables, which are required to generate executable test cases in our application context. The standard behavior of the system is modeled as a UML 2.0 state machine. In addition, we used Aspect-oriented Modeling (AOM) and more specifically the AspectSM profile [33] to model robustness behavior separately as aspect state machines. The robustness behavior is modeled based on different functional and non-functional properties, whose violations lead to erroneous states. Such properties can be related to the SUT or its environment such as the network and other systems interacting with the SUT. A weaver later on weaves robustness behavior into the standard behavior and generates a standard UML 2.0 state machine. More details and models of the case study, including a partial woven state machine, are provided in [33]. The woven state machine produced by the weaver is used for test case generation. In the current, simplified case study, the woven state machine has 12 states and 103 transitions. Out of these 103 transitions, only 83 transitions model robustness behavior as change events and 57 transitions out of these 83 have identical change conditions, including 42 constraints using *select()* and *size()* operations. A change event is defined with a 'when' condition and it is triggered when this condition is met during the execution of a system. An example of such a change event is shown in Figure 5. This change event is fired during a videoconference when the synchronization between audio and video passes the allowed threshold. *synchronizationMismatch* is a non-functional property defined using the MARTE profile [43], which measures the synchronization between audio and video in time. In order to traverse these transitions appropriate test data is required that satisfies the constraints specified as guards and when conditions (in case of change events). The complexity of these constraints, which are all in a conjunctive normal form, is reported in Table 10 in terms of number of clauses. Most constraints contain between 6 and 8 clauses.

The different OCL data types used in these constraints are shown in Table 11 and we can see that all primitive types are being used in our case study.

In our case study, we target test data generation for model-based robustness testing of the VCS. Testing is performed at the system level and we specifically target robustness faults, for example related to faulty situations in the network and other systems that comprise the environment of the SUT. Test cases are generated from the system state machines using our tool TRUST [42]. To execute test cases, we need appropriate data for the state variables of the system, state variables of the environment (network properties and in certain cases state variables of other VCS), and input parameters that may be used in the following UML state machine elements: (1) guard conditions on transitions, (2) change events as triggers on transitions, and (3) inputs to time events. We have successfully used the TRUST tool to generate test cases using different coverage criteria on UML state machines, such as all transitions, all round trip, modified round trip strategy [44].

5.1 Empirical Evaluation

This section discusses the experiment design, execution, and analysis of the evaluation of the proposed OCL test data generator on the VCS case.

5.1.1 Experiment Design

We designed our experiment using the guidelines proposed in [8, 45]. The objective of our experiment is to assess the efficiency of search algorithms such as GAs to generate test data by solving OCL constraints. In our experiments, we compared four search techniques: AVM, GA, (1+1) EA, and RS (Section 4). AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in search-based software engineering [8]. (1+1) EA is simpler than GAs, but in previous software testing work we found that it can be more effective in some cases (e.g., see [38]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [8].

From this experiment, we want to answer the following research questions.

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in industrial system models?

RQ2: Among the considered search algorithms (AVM, GA, (1+1) EA), which one fares best in solving OCL constraints and how do they compare to RS?

5.1.2 Experiment Execution

We ran experiments for 57 OCL predicates from the VCS industrial case study that we discussed earlier. The number of clauses in each predicate varies from one to eight and the median value is six. The complexity of the problems is summarized in Table 10, where we provide details on the distribution of numbers of clauses. In Table 11, we summarized the data types and OCL specific operations used in the problems.

Fitness evaluations are computationally expensive, as they require the instantiation of models on which the constraints are evaluated on. Each algorithm was run 100 times to account for the random variation inherent to randomized algorithms [46], which for our case study was enough to gain enough statistical confidence on the validity of our results. We ran each algorithm up to 2000 fitness evaluations on each problem and collected data on whether an algorithm found a solution or not. On our machine (Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Microsoft Windows 7 operating system), running 2000 fitness evaluations takes on average 3.8 minutes for all algorithms. The number of fitness evaluations should not be too high to enable enough experimentations on different constraints within feasible time, but should still represent a reasonable "budget" in an industrial setting (i.e., the time the software testers are willing to wait when solving constraints to generate system level test cases).

Instead of putting a limit to the number of fitness evaluations, in practice we can put a limit on time depending on practical constraints. This mean we can run a search algorithm with as many iterations as possible and stop once a predefined time threshold is reached (e.g., 10 minutes) if the constraint has not been solved yet. The choice of this threshold could be driven by the testing budget. However, though useful in practice, using a time threshold would make it significantly more difficult and less reliable to compare different search algorithms (e.g., accurately monitoring the passing of time, side effects of other processes running at same time, inefficiencies in implementation details).

A solution is represented as an array of variables, the same variables that appear in the OCL constraint we want to solve. For the used GA, we set the population size to 100 and the crossover rate to 0.75, with a 1.5 bias for rank selection. We use a standard one-point crossover, and mutation of a variable is done with the standard probability *1/n*, where *n* is the number of variables. Different settings would lead to different performance of a search algorithm, but standard settings usually perform well [46]. As we will show, our constraint solver is already very effective in solving OCL constraints, so we did not feel the need for tuning to improve the performance even further.

To compare the algorithms, we calculated their success rates. The success rate of an algorithm is defined in general as the number of times it was successful in finding a solution out of the total number of runs. In our context, it is the success rate in solving constraints.

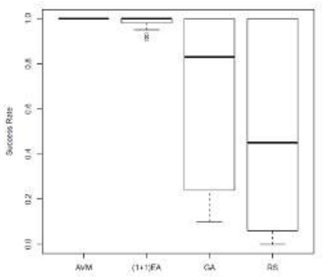


Figure 6. Success rates for various algorithms

5.1.3 Results and Analysis

Figure 6 shows a box plot representing the success rates of the 57 problems for AVM, (1+1) EA, GA, and RS. For each search technique, the box-plot is based on 57 success rates, one for each constraint. The results show that AVM not only outperformed all the other three algorithms, i.e., (1+1) EA, RS, and GA but in addition achieved a consistent success rate of 100%. (1+1) EA outperformed GA and RS and achieved an average success rate of 98%. Finally, GA outperformed RS, where GA achieved an average success rate of 65% and RS attained an average success rate of 49%. We can observe that, with an upper limit of 2000 iterations, (1+1) EA achieves a median success rate of 98% and GA exceeds a median of roughly 80%, whereas RS could not exceed a median of roughly 45%. We can also see that all success rates for (1+1) EA are above 90% and most of them are close to 100%.

Table 12 . Success rates For Individual Problems

Problem Id	Complexity	AVM	(1+1)EA	GA	RS
0	8	1	0,98	0,21	0,02
1	5	1	1	0,95	0,83
2	7	1	0,91	0,17	0,01
3	7	1	0,95	0,15	0,01
4	7	1	0,92	0,1	0,01
5	7	1	0,96	0,11	0
6	6	1	1	0,87	0,68
7	6	1	0,99	0,88	0,59
8	5	1	0,98	0,84	0,53
9	5	1	1	0,83	0,45
10	5	1	1	0,81	0,33
11	5	1	0,98	0,78	0,39
12	7	1	1	0,29	0,07
13	6	1	1	0,54	0,3
14	6	1	0,95	0,3	0,06
15	6	1	0,95	0,25	0,1
16	6	1	1	0,19	0,02
17	6	1	0,98	0,24	0,04
18	7	1	0,96	0,34	0,11
19	6	1	1	0,6	0,12
20	6	1	0,98	0,25	0,04
21	6	1	0,97	0,23	0,04
22	6	1	0,99	0,18	0,04
23	6	1	1	0,17	0,05
24	6	1	1	0,91	0,67
25	5	1	1	1	0,93
26	5	1	0,99	0,88	0,42
27	5	1	1	0,75	0,51
28	5	1	1	0,77	0,4
29	6	1	0,99	0,16	0,08
30	7	1	0,96	0,37	0,13
31	6	1	1	0,55	0,15
32	6	1	0,96	0,19	0,02
33	6	1	0,93	0,21	0,07
34	6	1	0,96	0,21	0,02
35	6	1	0,98	0,23	0,04
36	6	1	1	0,95	0,93
37	5	1	1	0,99	1
38	5	1	0,99	0,89	0,76
39	5	1	1	0,86	0,7
40	6	1	1	0,9	0,59
41	5	1	1	0,84	0,65
42	1	1	1	1	1
43	1	1	1	1	1
44	1	1	1	1	1
45	1	1	1	1	1
46	1	1	1	1	1
47	1	1	1	1	1
48	2	1	1	1	1
49	1	1	1	1	1
50	1	1	1	1	1
51	2	1	1	1	1
52	2	1	1	1	1
53	1	1	1	1	1
54	2	1	1	1	1
55	1	1	1	1	1
56	2	1	1	1	1

Table 13. Results for The Fisher's Exact Test At Significance Level of 0.05

ID	AVM vs (1+1) EA		AVM vs GA		AVM vs RS		(1+1 EA) vs GA		(1+1) EA vs RS		GA vs RS	
	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR
29	1	3	3,84E-40	1029	7,78E-48	2187	2,82E-38	339	6,70E-46	721	0,12	2
30	0,12	9	8,62E-26	340	8,48E-43	1302	1,89E-20	36,	1,39E-36	138	0,0001	3
31	1	1	8,93E-17	164	5,29E-41	1108	8,93E-17	164	5,29E-41	1108	3,55E-09	6
32	0,12	9	1,08E-37	840	1,14E-55	7919	1,09E-31	89	4,47E-49	844	0,0001	9
33	0,01	16	3,75E-36	743	5,76E-49	2505	5,21E-27	46	5,69E-39	155	0,007	3
34	0,12	9	3,75E-36	743	1,14E-55	7919	3,22E-30	79	4,47E-49	844	2,50E-05	10
35	0,49	5	1,11E-34	662	1,02E-52	4310	2,27E-31	129	4,47E-49	84	0,0001	6,
36	1	1	0,059	11	0,01	16	0,05	11	0,01	16	0,76	1
37	1	1	1	3	1	1	1	3	1	1	1	0,3
38	1	3	0,0007	25	2,48E-08	64	0,004	8	3,64E-07	21	0,02	2
39	1	1	7,49E-05	33	1,43E-10	86	7,49E-05	33	1,43E-10	86	0,009	3
40	1	1	0,001	23	5,03E-15	140	0,001	23	5,03E-15	140	6,14E-07	6
41	1	1	1,59E-05	39	1,56E-12	108	1,59E-05	39	1,56E-12	108	0,003	3
42	1	1	1	1	1	1	1	1	1	1	1	1
43	1	1	1	1	1	1	1	1	1	1	1	1
44	1	1	1	1	1	1	1	1	1	1	1	1
45	1	1	1	1	1	1	1	1	1	1	1	1
46	1	1	1	1	1	1	1	1	1	1	1	1
47	1	1	1	1	1	1	1	1	1	1	1	1
48	1	1	1	1	1	1	1	1	1	1	1	1
49	1	1	1	1	1	1	1	1	1	1	1	1
50	1	1	1	1	1	1	1	1	1	1	1	1
51	1	1	1	1	1	1	1	1	1	1	1	1
52	1	1	1	1	1	1	1	1	1	1	1	1
53	1	1	1	1	1	1	1	1	1	1	1	1
54	1	1	1	1	1	1	1	1	1	1	1	1
55	1	1	1	1	1	1	1	1	1	1	1	1
56	1	1	1	1	1	1	1	1	1	1	1	1

Table 14. Results for The Fisher's Exact Test at Significance Level of 0.05

	AVM vs (1+1) EA		AVM vs GA		AVM vs RS		(1+1 EA) vs GA		(1+1) EA vs RS		GA vs RS	
ID	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR	p-Value	OR
0	0,49	5	3,75E-36	743	1,14E-55	7919	8,33E-33	146	5,41E-52	1552	2,50E-05	1
1	1	1	0,059	12	7,26E-06	42	0,05	12	7,26E-06	42	0,01	3
2	0,003	21	2,64E-39	959	2,23E-57	13333	5,39E-28	46	7,96E-45	639	7,48E-05	13
3	0,059	12	5,29E-41	1108	2,23E-57	13333	1,15E-33	96	1,95E-49	1151	0,0003	12
4	0,006	18	1,04E-45	1732	2,23E-57	13333	7,47E-35	94	6,70E-46	722	0,009	8
5	0,12	9	1,05E-44	1564	2,21E-59	40401	2,01E-38	167	1,02E-52	4310	0,0007	26
6	1	1	0,0001	31	2,41E-11	95	0,0001	31	2,41E-11	95	0,002	3
7	1	3	0,0003	28	5,03E-15	140	0,002	9	1,35E-13	46	4,85E-06	5
8	0,49	5	1,59E-05	39	1,11E-17	178	0,0007	8	5,69E-15	35	3,59E-06	5
9	1	1	7,26E-06	42	1,59E-21	245	7,26E-06	42	1,59E-21	245	2,91E-08	6
10	1	1	1,48E-06	48	4,05E-28	405	1,48E-06	48	4,05E-28	405	6,86E-12	8
11	0,49	5	1,30E-07	57	1,12E-24	312	1,21E-05	11	1,14E-21	61	3,17E-08	5
12	1	1	1,33E-30	487	5,76E-49	2505	1,33E-30	487	5,76E-49	2506	7,42E-05	5
13	1	1	3,17E-17	171	5,77E-30	465	3,17E-17	171	5,77E-30	465	0,0009	2
14	0,059	12	5,77E-30	465	3,77E-50	2922	2,68E-23	40	2,01E-42	252	1,26E-05	6
15	0,059	12	2,87E-33	595	1,04E-45	1732	2,26E-26	51	3,71E-38	150	0,008	3
16	1	1	1,08E-37	840	1,14E-55	7919	1,08E-37	840	1,14E-55	7919	0,0001	9
17	0,49	5	5,75E-34	627	1,02E-52	4310	1,13E-30	123	4,47E-49	844	5,87E-05	7
18	0,12	9	1,60E-27	387	1,05E-44	1564	4,57E-22	41	2,01E-38	167	0,0001	4
19	1	1	1,34E-14	134	9,76E-44	1423	1,34E-14	134	9,76E-44	1423	9,47E-13	11
20	0,49	5	2,87E-33	595	1,02E-52	4310	5,40E-30	116	4,47E-49	845	2,99E-05	7
21	0,24	7	1,11E-34	663	1,02E-52	4310	4,93E-30	92	1,42E-47	597	0,0001	7
22	1	3	1,73E-38	896	1,02E-52	4310	1,22E-36	296	9,48E-51	1422	0,002	5
23	1	1	2,64E-39	959	2,13E-51	3490	2,64E-39	959	2,13E-51	3490	0,01	4
24	1	1	0,003	20	9,76E-12	99	0,003	21	9,76E-12	100	4,55E-05	5
25	1	1	1	1	0,01	16	1	1	0,01	16	0,01	16
26	1	3	0,0003	28	4,54E-23	276	0,002	9	1,90E-21	91	6,44E-12	10
27	1	1	1,07E-08	68	1,32E-18	193	1,07E-08	68	1,32E-18	193	0,0007	3
28	1	1	5,71E-08	61	3,90E-24	300	5,71E-08	61	3,90E-24	300	1,67E-07	5

Table 12 shows success rates for individual problems to further analyze the results. We observe that problems 42 to 56 were solved by all the algorithms. The reason is that these problems are the simplest problems comprising of either one or two clauses, as it can be seen from the complexity column in Table 12 and Table 15. The problems with higher complexity (higher number of clauses) are the most difficult to solve for GA and RS, as shown in Table 15. As the complexity is increasing, the success rates of GA and RS are decreasing. However, in the case of AVM and (1+1) EA, we do not see a similar pattern. AVM managed to maintain the average success rate of 100% even for the most complex problems. In the case of (1+1) EA, the minimum average success rates are for the problems with complexity of seven clauses, which is 95%. Based on these results, we can see that our approach is effective and efficient, and therefore practical, even for difficult constraints (RQ1).

Table 15 . Average Success Rates For Problems of Varying Complexity

Complexity	**AVM**	**(1+1) EA**	**GA**	**RS**
1	1	1	1	1
2	1	1	1	1
5	1	0,995	0,86	0,60
6	1	0,98	0,43	0,20
7	1	0,95	0,21	0,04
8	1	0,98	0,21	0,02

Table 16. Results for The paired Mann-Whitney U-test At Significance Level of 0.05

Pair of approaches	**p-Value**
AVM vs (1+1) EA	2.653988e-05
AVM vs GA	2.507670e-08
AVM vs RS	2.485853e-08
(1+1) EA vs GA	2.506828e-08
(1+1) EA vs RS	2.480008e-08
GA vs RS	1.822280e-08

To check the statistical significance of the results, we carried out a paired Mann-Whitney U-test (paired per constraint) at the significance level of 0.05 on the distributions of the success rates for the four algorithms. In all the four distribution comparisons, p-values were very close to 0, as shown in Table 16. This shows a strong statistical difference among the four algorithms when applied on all 57 constraints of our case study.

In addition, we performed a Fisher's exact test at the significance level of 0.05 between each pair of algorithms based on their success rates for the 57 constraints. The results for the Fisher's exact test are shown in Table 13 and Table 14. In addition to statistical significance, we also assessed the magnitude of the improvement by calculating the effect size in a standardized way. We used odds ratio [45] for this purpose, as the results of our experiments are dichotomous. Table 13 and Table 14 also show the odds ratio for various pairs of approaches for all 57 problems. For AVM vs (1+1) EA, we did not observe significant differences for most of the problems, except for Problem 2 and Problem 33, where AVM significantly performed better than (1+1) EA. In addition, odds ratios between AVM and (1+1) EA for 23 problems are greater than 1, implying that AVM has more chances of success than (1+1) EA. For 35 problems out of 57, the odds ratio is 1 suggesting that there is no difference between these two algorithms. For AVM vs GA, for 38 problems AVM significantly performed better than GA as p-values are below 0.05 (our chosen significance level). The odds ratios for most of the problems, except for the problems with 1 or 2 clauses, are greater than one, thus suggesting that AVM has more chances of success than GA. Similar results were observed for (1+1) EA, where for 38 problems it significantly outperformed GA. For AVM vs RS, for almost all of the problems except the ones with one or two clauses, AVM performed significantly better than RS. Similar results were observed for (1+1) EA vs RS and GA vs RS.

To check the complexity of the problems, we repeated the experiment on the negation of each of the 57 problems. All algorithms managed to find solutions for all these problems very quickly. Most of the time and for most of the problems, each algorithm managed to find solutions in a single iteration. This result confirmed that the actual problems we targeted with search were difficult to solve.

```
context Saturn inv synchronizationConstraint:
    self.systemUnit.NumberOfActiveCalls  > 1 and
    self.systemUnit.NumberOfActiveCalls  <= self.systemUnit.MaximumNumberOfActiveCalls and
    self.media.synchronizationMismatch.unit = TimeUnitKind::s and (self.media.synchronizationMismatch.value  >= 0 and
    self.media.synchronizationMismatch.value  <= self.media.synchronizationMismatchThreshold.value) and
    self.conference.PresentationMode = Mode::Off and
    self.conference.call→select(call | call.incomingPresentationChannel.Protocol <> VideoProtocol::Off)→size()=2 and
    self.conference.call→select(call |  call.outgoingPresentationChannel.Protocol <> VideoProtocol::Off)→size()=2
```

Figure 7. Condition for a change event

Table 17. Average Time Took By Algorithms to Solve the Problems

Algorithm	Average Time to Solve Constraints (Seconds)
AVM	2.96
(1+1) EA	99
GA	182
RS	423

Table 18. Statistics of Complexity of Constraints

Problem #	# of Clauses	OCL Data Type Used (Number of variable is 1)	Search-based Solver with (1+1)EA (Seconds)	Search-based Solver with AVM (Seconds)	UML2CSP (Seconds)
I43	1	Boolean	0.26	0.07	0.01
I44	1	Boolean	0.10	0.07	0.01
I45	1	Integer	0.07	0.03	0.01
I46	1	Integer	0.07	0.03	0.01
I47	1	Integer	0.07	0.03	0.01
I48	1	Integer	0.13	0.04	0.01
I49	2	Integer	1.41	0.26	0.01
I50	2	Integer	1.56	0.4	0.01
I51	1	Integer	0.12	0.04	0.01
I52	2	Integer	1.76	0.25	0.01
I53	2	Integer	1.72	0.26	0.01
I54	1	Integer	0.09	0.04	0.01
I55	2	Integer	1.25	0.24	0.01
I56	1	Integer	0.08	0.04	0.01
I57	2	Integer	1.48	0.23	0.01

Based on the above results, we recommend using AVM and (1+1) EA for as many iterations as possible (RQ2). We can see from the results that, even when we set the number of iterations to 2000, AVM managed to achieve a 100% success rate with 26 iterations on average. On the other hand, (1+1) EA managed to achieve a 98% success rate with an average of 743 iterations. Note that in case studies with more complex problems, a larger number of iterations may be required to eventually solve the problems.

5.2 Comparison with UMLtoCSP

UMLtoCSP [15] is the most widely used and referenced OCL constraint solver in the literature. To assess the performance of UMLtoCSP to solve complex constraints such as the ones in our current industrial case study, we conducted an experiment. We repeated the experiment for 57 constraints from our industrial application, whose complexity is summarized in Table 10. An example of such constraint, modeling a change event on a transition of Saturn's state machine, is shown in Figure 7. This change event is fired when Saturn is successful in recovering the synchronization between audio and video. Since UMLtoCSP does not support enumerations, we converted each enumeration into an Integer and limited its bound to the number of literals in the enumeration. We also used the MARTE profile to model different non-functional properties, and since UMLtoCSP does not support UML profiles, we explicitly modeled the used subset of MARTE as part of our models. In addition, UMLtoCSP does not allow writing constraints on inherited attributes of a class, so we modified our models and modeled inherited attributes directly in the classes. We set the range of Integer attributes from 1 to 100. Since the UML2CSP tool did not support UML 2.x diagrams, we also needed to recreate our models in a UML 1.x modeling tool.

We ran the experiment on the same machine as we used in the experiments reported in the previous section. Though we let UMLtoCSP attempt to solve each of the selected constraints for one hour each, it was not successful in finding any valid solution for the 42 problems comprising of 5-8 clauses. A plausible explanation is that UMLtoCSP is hampered by a combinatorial explosion problem because of the complexity of the constraints in the model. However, such constraints must be expected in real-world industrial applications as our Cisco example is in no way particularly complex by industrial standards. In contrast, our constraint solver managed to solve each constraint within at most 2.96 seconds using AVM and 99 Seconds using (1+1) EA, as shown in Table 17. For the remaining 15 problems, which are simple constraints comprising of either one or two clauses, UMLtoCSP managed to find solutions. Each of these constraints has one variable of either Integer or Boolean type. The results of the comparison of UMLtoCSP with our tool for these simple clauses (problems 42-56) are shown in Table 18. We provide the time taken by UML2CSP to solve each problem in seconds, which is reported by the tool itself and 0.01 second (maximum precision) for all fifteen constraints. For these same 15 problems, we ran our tool 100 times for each of them. In Table 18, we report the average time taken by our tool to solve each problem over 100 runs. Since we used the same machine to run experiments for both tools, it is clear from the results that for all fifteen simple problems, UMLtoCSP took less time than our tool (which is on average less than one second and in the worst case less than two seconds). But considering that UMLtoCSP fails to solve the more complex problems and its issues regarding limited support of OCL constructs (as already discussed), we conclude it is not practical to apply UMLtoCSP in large systems having complex constraints.

6. Empirical Evaluation of Optimization Defined as Fitness Functions

In this section, we empirically evaluate the fine grained fitness functions that we defined in Section 4 for various OCL operations to see if they really improve performance of search algorithms as compared to using simple branch distance functions, yielding *0* if an expression is *true* and *k* otherwise.

6.1 Experiment Design

To empirically evaluate whether the functions defined in Section 4 really improve the branch distance, we carefully defined artificial problems to evaluate each heuristic since not all of the OCL constructs were used in the industrial case study. The model we used for the experiment consists of a very simple class diagram with one class *X*. *X* has one attribute *y* of type *Integer*. The range of *y* was set to -100 to 100. We populated 10 objects of class *X*. The use of a single class with 10 objects was sufficient to create complex constraints. For each heuristic, we created an artificial problem, which was sufficiently complex to remain unsolved by random search. We checked this by running all the artificial problems (100 times per problem) using random search for 20,000 iterations per problem, and random search could not manage to solve most of the problems most of the times, except for problems *A9* and *A10*. **Table 19** lists the artificial problems and the corresponding heuristics that we used in the experiments. We prefixed each problem with A to show that it is an artificial problem. For the evaluation, we used the best algorithms among search algorithms used in the industrial case study (Section 5.1 and in other works [38]: (1+1) EA and AVM. In this experiment, we address the following research question:

RQ3: Does optimized branch distance calculations improve the effectiveness of search over simple branch distance calculations?

Table 19. Artifical Problems for Heuristics

Problem #	**Heuristic**	**Example**
A1	forAll()	X.allInstances() → forAll(b|b.y=47)
A2	exists()	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → exists(b|b.y=92)
A3	isUnique()	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → isUnique(b|b.y)
A4	one()	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → one(b|b.y=95)
A5	select()size()	X.allInstances() → select(b|b.y=0) → size()>6
A6	select()size()	X.allInstances() → select(b|b.y=0) → size()<=1
A7	select()size()	X.allInstances() → select(b|b.y > 90) → size() > 4 and X.allInstances() → select(b|b.y > 90) → select(b|b.y=92) → size() <> 0
A8	select()size()	X.allInstances() → select(b|b.y=0) → size() = 5
A9	includes()	X.allInstances() → collect(b|b.y) → includes(17)
A10	excludes()	X.allInstances() → collect(b|b.y) → excludes(0)
A11	includesAll()	let c = Set{-1,87,19,88} in X.allInstances() → collect(b|b.y) → includesAll(c)
A12	excludesAll()	let c = Set{0,1,2,3} in X.allInstances() → select(b|b.y>0 and b.y<5) → size()>=5 and X.allInstances() → select(b|b.y>0 and b.y<5) → collect(b|b.y) → excludesAll(c)
A13	select()forAll()	X.allInstances() → select(b|b.y<>47) → forAll(b|b.y*b.y=-100)

To answer this research question, we compared branch distance calculations based on heuristics defined in Section 4 and without heuristics (i.e., branch distance calculations either return *0* when a constraint is solved or *k* otherwise). We will refer to them here as Optimized (*Op*) and Non-Optimized (*NOp*) branch distance calculations, respectively.

6.2 Experiment Execution

We ran experiments 100 times for (1+1) EA and AVM, with *Op* and *NOp*, and for each problem listed in Table 19 . We let (1+1) EA and AVM run up to 2000 fitness evaluations on each problem and collected data on whether the algorithms found solutions for *Op* and *NOp*. We used a PC with Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Microsoft Windows 7 operating system for the execution of experiment. To compare the algorithms for *Op* and *NOp*, we calculated the success rate, which is defined as the number of times a solution was found out of the total number of runs (100 in this case).

6.3 Results and Analysis

Table 20 shows the results of success rates for *Op* and *NOp* for each problem and both algorithms ((1+1) EA and AVM). To compare if the differences of success rates among *Op* and *NOp* are statistically significant, we performed the Fisher‘s exact test [47] at the significance level of 0.05. We chose this test since for each run of algorithms the result is binary, i.e., either the result is ‘found’ or ‘not found’ and this is exactly the situation for which the Fisher’s exact test is defined. We performed the test only for the problems having success rates greater than *0* and not equal to each other for both *Op* and *NOp* (i.e., for problems *A2*, *A3*, and *A4* in the case of 1+1 (EA) and problem *A9* for AVM)). For 1+1 (EA), the p-values for all the three problems (*A2*, *A3*, and *A4*) are 0.0001, thus indicating that the success rate of Op is significantly higher than NOp. For problems *A1*, *A5*, *A6*, *A7*, *A8*, *A12*, and *A13*, the results are even more extreme as *Op* shows a 100% success rate, whereas *NOp* has 0% success rate. For the problems *A9* and *A10*, the success rates are 100% for both *Op* and *NOp* and hence conclusions cannot simply be drawn based on these rates. For these problems, we further compared the number of iterations taken by (1+1) EA for *Op* and *NOp* to solve the problems. We used Mann-Whitney U-test [[47], at a significance level of 0.05, to determine if significant differences exist between *Op* and *NOp*. We chose this test based on the guidelines for performing statistical tests for randomized algorithms [45]. Table 21 shows the results of the test. The p-values are bold-faced when the results are significant. In Table 21, we also show the mean differences for the number of iterations and execution time between *Op* and *NOp* to show the direction in which the results are significant. In addition, we report effect size measurements using Vargha and Delaney’s Â12 statistics, which is a non-parametric effect size measure. We chose this effect size measure using again the guidelines reported in [45]. In our context, the value of Â12 tells the probability for Op to find a solution in more iterations than NOp.

This means that the higher the value of Â12, the higher the chances that Op will take more iterations to find a solution than NOp. If Op and NOp are equal then the value of Â12 is 0.5. With 1+1 (EA), for A9 and A10, Op took significantly less iterations to solve the problems (Table 21) as both p-values are below 0.05. In addition, for A9 and A10, values of Â12 are 0.19 and 0.46, respectively, thus showing that the only 19% and 46% of the time Op took more iterations to solve the problem than NOp.

Table 20. Results of Fisher Exact Test for Success Rate of Optimized and Non-Optimized at alpha=0.05

Problem #	**Success Rate (1+1)EA (*NOp*) in %**	**Success Rate for (1+1)EA (*Op*) in %**	**Fisher Exact Test for (1+1)EA (p-value)**	**Success Rate for AVM (*NOp*) in %**	**Success Rate for AVM (*Op*) in %**	**Fisher Exact Test for AVM (p-value)**
A1	0	100	-	0	100	-
A2	2	100	**0,0001**	0	59	-
A3	1	95	**0,0001**	0	99	-
A4	3	100	**0,0001**	0	100	-
A5	0	100	-	0	100	-
A6	0	100	-	0	100	-
A7	0	100	-	0	100	-
A8	0	100	-	0	100	-
A9	100	100	-	16	100	**0,0001**
A10	100	100	-	100	100	-
A11	0	94	-	0	99	-
A12	0	100	-	0	34	-
A13	0	100	-	0	100	-

For AVM, the results of success rates for *A10* were tied between *Op* and *NOp* (Table 20). Therefore, we further compared *Op* and *NOp* for these problems based on the number of iterations AVM took to solve these problems. As discussed before, we applied Mann-Whitney U-test [47] at significance level of 0.05 to determine if significant differences exist between *Op* and *NOp*. Table 22 shows mean differences, p-values, and *Â12* values. We observed that for the problem *Op* took less iterations to solve the problems and significant differences were observed for *A10* as the p-value is 0.04, which is less than our significance level of 0.05.

Based on the above results, we can answer our research question presented earlier: does the optimized branch distance calculation improve the effectiveness of search? We can clearly see from the results that (1+1) EA and AVM with optimized branch distance calculations significantly improve the success rates. In worst cases, when there is no differences in success rates between *Op* and *NOp*, (1+1) EA and AVM took significantly less iterations to solve the problems.

Table 21. Results of t-test at alpha=0.05 ((1+1)EA)

Problem #	Mean Difference (OP-NOP)	Â12	p-value
A9	-654,38	0,19	**0,0001**
A10	-1,01	0,46	**0,004**

Table 22. Results of t-test at alpha=0.05 (AVM)

Problem #	Mean Difference (OP-NOP)	Â12	p-value
A10	-0,23	0,52	**0,04**

7. Overall Discussion

In this section, we provide an overall discussion based on the results of the experiments on the industrial case study and the artificial problems. Based on the results from the industrial case study, we observe that AVM and (1+1) EA perform better as compared to GA and RS since the algorithms achieve 100% and 98% success rates for all 57 constraints on average, respectively (Section 5.1). For the experiments based on artificial problems (Section 6.3), we observe that AVM and (1+1) EA with optimized branch distance calculations outperform non-optimized branch distance calculations. However, we notice that for certain artificial problems, the performance of (1+1) EA is significantly better than AVM. For instance, in Table 20 for *A2*, (1+1) EA manages to find solutions for all 100 runs, whereas AVM could only manage to find solutions for 59 runs. We further perform a Fisher's exact test to determine if the differences are statistically significant at the significance level of 0.05 between these two algorithms. We obtain a p-value of 0.001 suggesting that (1+1) EA is significantly better than AVM for *A2*. Since AVM is a local search algorithm and A2 is a complex problem, AVM can be expected to be less efficient than (1+1) EA. Similar results are obtained for *A12*. Conversely, for other problems, i.e., for *A3* and A11, AVM seems more successful than (1+1) EA. For *A3*, AVM manages to find solutions 99 times, whereas (1+1) EA manages to find solutions 95 times (Table 20). In this case, we obtain a p-value of 0.21 when we applied the Fisher's exact test, hence suggesting that the differences are not statistically significant between the two algorithms. Similarly, for *A11*, AVM found solutions 99 times, whereas (1+1) EA found solutions for 94 times (Table 20). In this case, we obtain again a p-value of 0.11, which is lower than our chosen significance level (0.05); hence suggesting that the differences are not significant between these two algorithms.

Based on the results of our empirical analysis, we provide the following recommendations about using AVM and (1+1) EA: If the constraints need to be solved quickly, we recommend using AVM, since it is quicker in finding solutions as we discussed in Section 6, even though its performance was worse than (1+1) EA for two artificial problems. If we are flexible with time budget (e.g., the constraints need to be solved only once, and the cost of doing that is negligible compared to other costs in the testing phase), we rather recommend running (1+1) EA for as many iterations as possible as we notice that the success rate for (1+1) EA was 98% on average for the industrial case study, whereas for the artificial problems, it either fares better or equal to AVM.

The difference in performance between AVM and (1+1) EA has a clear explanation. AVM works like a sort of greedy local search. If the fitness function provides a clear gradient towards the global optima, then AVM will quickly converge to one of them. On the other hand, (1+1) EA puts more focus on the exploration of the search landscape. When there is a clear gradient toward global optima, (1+1) EA is still able to reach those optima in reasonable time, but will spend some time in exploring other areas of the search space. This latter property becomes essential in difficult landscapes where there are many local optima. In these cases, AVM gets stuck and has to re-start from other points in the search landscape. On the other hand, (1+1) EA, thanks to its mutation operator, has always a non-zero probability of escaping from local optima.

8. Tool Support

To efficiently solve OCL constraints, we developed a search-based OCL constraint solver, since current OCL solvers were not able to handle the complexity of the constraints in our models for the industrial case study within reasonable time (Section 6). Figure 8 shows the architecture diagram for our Search-based Constraint solver. We developed a tool in Java that interacts with an existing library, an OCL evaluator, called the EyeOCL Software (EOS) [24]. EOS is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our tool only requires interacting with EOS for the evaluation of constraints. We selected to use EOS as it is one the most efficient evaluators currently available. Any other OCL evaluator can also be easily integrated with our tool. Our tool implements the calculation of branch distance (*DistanceCalculator*) for various expressions in OCL as discussed in Section 4, which aims at calculating how far are the test data values from satisfying constraints. The search algorithms employed are implemented in Java as well and include Genetic Algorithms, (1+1) Evolutionary Algorithm, and Alternating Variable Method (AVM).

9. Threats to Validity

To reduce construct validity threats, we chose as an effectiveness measure the search success rate, which is comparable across all four search algorithms (AVM, (1+1) EA, GA and RS). Furthermore, we used the same stopping criterion for all algorithms, i.e., number of fitness evaluations. This criterion is a comparable measure of efficiency across all the algorithms because each iteration requires updating the object diagram in EyeOCL and evaluating a query on it.

The most probable conclusion validity threat in experiments involving randomized algorithms is due to random variations. To address it, we repeated experiments 100 times to reduce the possibility that the results were obtained by chance. Furthermore, we performed Fisher exact tests to compare proportions and determine the statistical significance of the results. We chose Fisher exact test because it is appropriate for dichotomous data where proportions must be compared [45], thus matching our situation. To determine the practical significance of the results obtained, we measured the effect size using the odds ratio of success rates across search techniques.

A possible threat to internal validity is that we have experimented with only one configuration setting for the GA parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience on testing problems. Parameter tuning can improve the performance of GAs, although default parameters often provide reasonable results [46].

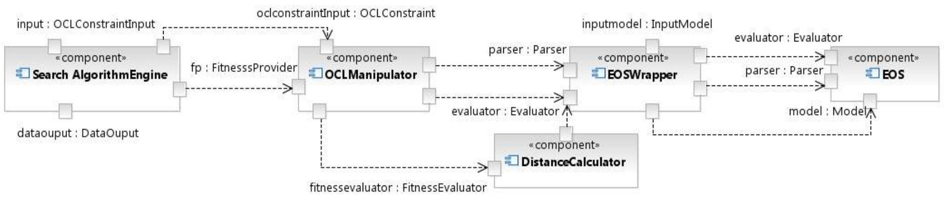


Figure 8. Architecture diagram for search-based constraint solver

We ran our experiments on an industrial case study to generate test data for 57 different OCL constraints, ranging from simpler constraints having just one clause to complex constraints having eight clauses. Although the empirical analysis is based on a real industrial system our results might not generalize to other case studies. However, such threat to external validity is common to all empirical studies. In addition to the industrial case study, we also conducted an empirical evaluation of each proposed branch distance calculation using small yet complex artificial problems to demonstrate that the effectiveness of our heuristics holds even for more complex problems. In addition, empirically evaluating all proposed branch distance calculations on artificial problems was necessary since it was not possible to evaluate them for all features of OCL in the industrial case study due to its inherent properties.

In the empirical comparisons with UMLtoCSP, we might also have wrongly configured the tool. To reduce the probability of such an event, we contacted the authors of UMLtoCSP who were very helpful in ensuring its proper use. From our analysis of UMLtoCSP, we cannot generalize our results to traditional constraint solvers in general when applied to solve OCL constraints. However, empirical comparisons with other constraints solvers were not possible because, to the best of our knowledge, UMLtoCSP is not only the most referenced OCL solver but also the only one that is publically available. However, because the problems encountered with UMLtoCSP are due to the translation to a lower-level constraint language, we expect similar issues with the other constraint solvers.

10. Conclusion

In this paper, we presented a search-based constraint solver for constraints written in the Object Constraint Language (OCL). The goal is to achieve a practical, scalable solution to support test data generation for Model-based Testing (MBT). Existing OCL constraint solvers have one or more of the following problems that make them difficult to use in industrial applications: (1) they support only a subset of OCL; (2) they translate OCL into formalisms such as first order logic, temporal logic, or Alloy, and thus result into combinatorial explosion problems. These problems limit their practical adoption in industrial settings.

To overcome the abovementioned problems, we defined a set of heuristics based on OCL constraints to guide search-based algorithms (Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), Alternating Variable Method (AVM)) and implemented them in our search-based OCL constraint solver. More specifically, we defined branch distance functions for various types of expressions in OCL to guide search algorithms. We demonstrated the effectiveness and efficiency of our search-based constraint solver to generate test data in the context of the model-based, robustness testing of an industrial case study of a video conferencing system. Even for the most difficult constraints, with research prototypes and no parallel computations, we obtain test data within 2.96 seconds on average.

As a comparison, we ran 57 constraints from the industrial case study on one well-known, downloadable OCL solver (UMLtoCSP) and the results showed that, even after running it for one hour, no solutions could be found for most of the constraints. Similar to all existing OCL solvers, because it could not handle all OCL constructs and UML features, we had to transform our constraints to satisfy UMLtoCSP requirements.

We also conducted an empirical evaluation in which we compared four search algorithms using two statistical tests: Fisher's exact test between each pair of algorithms to test their differences in success rates for each constraints and a paired Mann-Whitney U-test on the distributions of the success rates (paired per constraint). Results showed that

AVM was significantly better than the other three search algorithms, followed by (1+1) EA, GA and RS respectively. We also empirically evaluated each proposed branch distance calculation using small yet complex artificial problems. The results showed that the proposed branch distance calculations significantly improve the performance of solving OCL constraints for the purpose of test data generation when compared to simple branch distance calculations. Based on the results of our empirical analyses, we recommend using AVM if the constraints need to be solved quickly since it is quicker in finding solutions, even though its performance was worse than (1+1) EA for two complex artificial problems with difficult search landscapes. In other cases, if we are flexible with time budget (e.g., the constraints need to be solved only once), we rather recommend using (1+1) EA for as many iterations as possible since (1+1) EA has 98% success rate on average for the industrial case study, whereas for the artificial problems, it either fares better or equal to AVM.

Though focused on OCL in this paper, the general search-based solution and heuristics we propose here to make the search more efficient could be adapted to other high level constraint languages based on first-order logic and set theory. In the future, we are also planning to extend our solver to automatically instantiate models by solving constraints defined on their metamodels for the purpose of model-transformation testing, which is an increasingly important challenge.

ACKNOWLEDGEMENT

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE. We thank Marius Christian Liaaen (Cisco Systems, Inc Norway) for providing us the case study. We are also grateful to Jordi Cabot, an author of UML2CSP, for helping us to run UML2CSP on our industrial case study.

REFERENCES

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*: Morgan-Kaufmann, 2007.

[2] OCL, "Object Constraint Language Specification, Version 2.2," Object

Management Group (OMG), 2011.

[3] MOF, "Meta Object Facility (MOF)," 2006.

[4] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," in *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*, 2003.

[5] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," Citeseer, 2002.

[6] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," in *International Conference on cimputational Intelligence and Security*, 2007, pp. 1048-1052.

[7] M. Harman, S. A.Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," King's College,Technical Report TR-09-032009.

[8] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering,* vol. 99, 2009.

[9] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research Articles," *Softw. Test. Verif. Reliab.,* vol. 16, pp. 175-203, 2006.

[10] A. Andrea, "Longer is Better: On the Role of Test Sequence Length in Software Testing," International Conference on Software Testing, Verification, and Validation, 2010, pp. 469-478.

[11] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems," in *IADIS International Conference in Applied Computing*, 2005.

[12] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," in *ECOOP-Workshop on Defining Precise Semantics for UML*, 2000.

[13] M. Clavel and M. A. G. d. Dios, "Checking unsatisfiability for OCL constraints," in *In the proceedings of the 9th OCL 2009 Workshop at the UML/MoDELS Conferences*, 2009.

[14] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," in *Proceedings of the Fifth International Conference on Quality Software*: IEEE Computer Society, 2005.

[15] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*: IEEE Computer Society, 2008.

[16] D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.,* vol. 168, pp. 70-118, 2005.

[17] J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. ster, "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars," *Electron. Notes Theor. Comput. Sci.,* vol. 211, pp. 159-170, 2008.

[18] M. Kyas, H. Fecher, F. S. d. Boer, J. Jacob, J. Hooman, M. v. d. Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electron. Notes Theor. Comput. Sci.,* vol. 115, pp. 39-47, 2005.

[19] M. P. Krieger, A. Knapp, and B. Wolff, "Automatic and Efficient Simulation of

Operation Contracts," in *9th International Conference on Generative Programming and Component Engineering*, 2010.

[20] S. Weißleder and B.-H. Schlingloff, "Deriving Input Partitions from UML Models for Automatic Test Generation," in *Models in Software Engineering*: Springer-Verlag, 2008, pp. 151-163.

[21] M. Gogolla, F. Bttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.,* vol. 69, pp. 27-34, 2007.

[22] IBM, "IBM OCL Parser," IBM, 2011.

[23] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Cârcu, "OCLE," V2.0 ed, 2010.

[24] M. Egea, "EyeOCL Software," 2010.

[25] CertifyIt, "CertifyIt," Smarttesting, 2011.

[26] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Softw. Test. Verif. Reliab.,* vol. 14, pp. 105-156, 2004.

[27] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the alloy constraint analyzer," in *Proceedings of the 22nd international conference on Software engineering* Limerick, Ireland: ACM, 2000.

[28] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," in *8th International Workshop on OCL Concepts and Tools.* vol. 15: ECEASST, 2008.

[29] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A Specification-Based Test Case Generation Method for UML/OCL," in *Worksshop on OCL and Textual Modelling, MoDELS*: Lecture Notes in Computer Science, Springer, 2010.

[30] Gecode, "Gecode," 2011.

[31] COMET, "COMET," 2011.

[32] M. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2010.

[33] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.

[34] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths," *Advanced Design and Manufacture to Gain a Competitive Edge,* pp. 147-156, 2008.

[35] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*: IEEE Computer Society, 2008.

[36] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes,* vol. 30, pp. 263-272, 2005.

[37] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *Journal of Systems and Software,* vol. 83, pp. 2379-2391.

[38] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability,* 2011.

[39] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*:

Addison-Wesley Longman Publishing Co., Inc., 1999.

[40] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing," in *IFIP International Conference on Testing Software and Systems (ICTSS)*, 2010.

[41] H. Li and Gordon, "Bytecode Testability Transformation " in *Symposium on Search based Software Engineering Co-located with ESEC/FSE*: ACM SIGSOFT, 2011.

[42] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.

[43] MARTE, "Modeling and Analysis of Real-time and Embedded systems (MARTE)," 2010.

[44] S. Ali, L. Briand, A. Arcuri, and S. Walawege, "An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms," in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011)*, 2011.

[45] A. Arcuri and L. Briand., "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *International Conference on Software Engineering (ICSE)*, 2011.

[46] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*: Springer's Lecture Notes in Computer Science (LNCS) 2011.

[47] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*: Chapman and Hall/CRC, 2007.

An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms

Shaukat Ali, Lionel C. Briand, Andrea Arcuri, Suneth Walawege

In: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), ACM/IEEE, 2011.

Abstract— Systematic and rigorous robustness testing is very critical for embedded systems, as for example communication and control systems. Robustness testing aims at testing the behavior of a system in the presence of faulty situations in its operating environment (e.g., sensors and actuators). In such situations, the system should gracefully degrade its performance instead of abruptly stopping execution. To systematically perform robustness testing, one option is to resort to model-based robustness testing (MBRT), based for example on UML/MARTE models. However, to successfully apply MBRT in industrial contexts, new technology needs to be developed to scale to the complexity of real industrial systems. In this paper, we report on our experience of performing MBRT on video conferencing systems developed by Cisco Systems, Norway. We discuss how we developed and integrated various techniques and tools to achieve a fully automated MBRT that is able to detect previously uncaught software faults in those systems. We provide an overview of how we achieved scalable modeling of robustness behavior using aspect-oriented modeling, test case generation using search algorithms, and environment emulation for test case execution. Our experience and lessons learned identify challenges and open research questions for the industrial application of MBRT.

1 Introduction

Model-based robustness testing (MBRT) is concerned with testing the behavior of a system in the presence of faulty situations in its operating environment. An IEEE Standard [1] defines robustness as *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions"*. A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, in our industrial application of MBRT for Video Conferencing Systems (VCS) developed by Cisco Systems, Norway, we model the robustness behavior of a VCS in the presence of hostile environment conditions (regarding the network and other communicating VCSs), such as a high percentage of packet loss and corrupt packets. The VCS should not crash, halt, or restart in the presence of such problems. Furthermore, the VCS should continue to work in a degraded mode, such as continuing the videoconference with low audio and video quality. In the worst case, the VCS should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS, and so it must be accurately tested.

MBRT is considered very critical for embedded systems, for example communication and control systems as is the case of our industrial case study. Such robustness is also considered very critical in many standards such as in the IEEE Standard Dictionary of Measures of the Software Aspects of Dependability [2], the ISO's Software Quality Characteristics standard [3], and the Software Assurance Standard [4] by NASA. Systematic and rigorous robustness testing however requires integration of many tools and techniques in an efficient way.

In this paper, we report on our experience of applying MBRT for VCSs developed by Cisco. Note that such industrial applications of MBRT and even more generally of model-based testing (MBT) are very rare in the literature [5]. These applications are very much needed to evaluate the applicability of MBT in realistic settings. The main contribution of this paper is the integration of the following techniques and tools to achieve the ultimate goal of systematic and rigorous MBRT: 1) Use UML and the MARTE profile to model properties of the environment, whose violations lead to faulty situations the VCS must be robust to; 2) Use aspect-oriented modeling (AOM) to achieve scalable robustness modeling that improves readability of models, reduces modeling complexity, supports enhanced separation of concerns (SOC), and helps in model evolution; 3) Use search algorithms to solve complex OCL constraints on properties of the environment to introduce faulty situations; 4) Integration of the tool support for all of the above with our extensible model-based testing tool (TRUST) [6]. Robustness test case execution requires a special setup to emulate the operating environment. We discuss how we emulate the environment for the MBRT of Cisco's VCS. A preliminary experiment of MBRT in Cisco revealed a critical robustness fault in an already tested VCS. Finally, we discuss our experiences and lessons learned while performing MBRT in Cisco.

The rest of the paper is organized as follows: Section 2 provides a brief description of our case study, Section 3 provides an overview on scalable robustness modeling using AOM and UML/MARTE, and Section 4 discusses test case generation using the TRUST tool. In Section 5, we discuss about robustness test case execution and results from our preliminary experiment with MBRT. Section 6 provides lessons learned and our experiences regarding MBRT in Cisco. Section 7 compares our work with the existing works in the literature. Finally, Section 8 concludes the paper.

2 Case Study

Our case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn. The core functionality to be modeled manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. In this paper, we focused on this particularly important subsystem (Saturn) and left out the other functionalities of Saturn. We selected this subsystem because robustness testing is concerned with testing the behavior of Saturn in the presence of faulty environment situations, which can only be tested when Saturn is in a conference call with other systems. Saturn is complex enough to investigate the applicability and usefulness of MBRT in realistic conditions, while still remaining manageable in the context of a case study.

To test the robustness of Saturn, we modeled its behavior in the presence of faulty situations in the network. The behavior of the network can be very unpredictable due to busy routers, high bandwidth demanding traffic (audio and video streaming) and low speed connections. Hence, Saturn is supposed to work even under the presence of faulty situations in a degraded mode. By degraded mode, we mean that the system should continue to behave as in the non-faulty situation, except that the quality (such as audio and video) or the performance is degraded by running applications at a lower speed. The system must try to recover from the degraded mode and go back to a normal mode of operation. In the worst case, the system must return to the most recent safe state. An example of a safe state of a VCS is the idle state, in which the VCS is not in a videoconference with any VCS.

3 Scalable Robustness Modeling

In this section, we discuss our scalable robustness modeling approach. In Section 3.1, we provide and briefly present partial models for the functional behavior of Saturn. In Section 3.2 we discuss how we model robustness behavior with aspect state machines using our proposed AspectSM profile.

3.1 Functional Behavior of Saturn

The functional behavior of Saturn consists of a set of class diagrams and a set of UML state machines. An excerpt of class diagram for the Saturn subsystem described in Section 2 is shown in Figure 1.

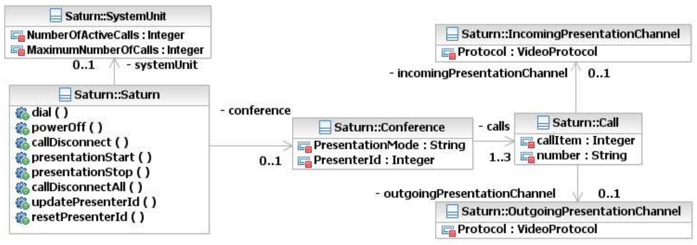


Figure 1. Class diagram for Saturn

The UML class diagram is meant to capture information about APIs and system (state) variables, which are required to generate executable test cases in our application context. Saturn’s API is modeled as a set of methods in the *Saturn* class such as *dial()* and *callDisconnect()*. The state variables of the system are modeled as instance variables of classes. For example, two system variables in the *SystemUnit* class are

NumberOfActiveCalls and *MaximumNumberOfCalls*. *NumberOfActiveCalls* is an *Integer*, which determines the number of VCS that are currently in a Saturn videoconference, whereas *MaximumNumberOfCalls* determines the maximum number of simultaneous calls supported by Saturn.

The functional behavior of Saturn is modeled as four submachine states. The first submachine state contains three simple states, whereas the second contains two additional submachine states, each having three simple states. This gives in total eleven simple states and 41 transitions in three levels. The flattened state machine consists of 70 transitions and 11 states. The complete models are provided in [7].

3.2 Robustness Modeling using RUMM

Previously, we defined a RobUstness Modeling Methodology (RUMM) to model robustness behavior using AOM [7]. Our goal was to devise a solution to model robustness behavior, which (1) is complete in terms of aspect and state machine features, (2) minimizes the learning curve over standard modeling skills, and (3) enable automated, model-based testing. RUMM consists of a series of systematic activities to model robustness behavior. We do not present here details of these activities, however, interested readers may find them in [7]. In this paper, we on modeling robustness behavior using the AspectSM profile. Using the AspectSM profile, we model each aspect as a UML state machine with stereotypes (aspect state machine). The modeling of aspect state machines is systematically derived from a fault taxonomy [7] categorizing different types of faults (faults in the environment such as communication medium and media streams that lead to faulty situations in the environment). Each aspect state machine has a corresponding aspect class diagram modeling different properties of the environment using the MARTE profile, whose violations lead to faulty situations in the environment.

3.2.1 Modeling aspect class diagram

For the robustness behavior presented in Section 2, we were interested in modeling the behavior of Saturn in the presence of faulty situations in the network. For this purpose, we decided to model the following network properties: packet loss, packet delay, duplicate packet, corrupt packet, and reorder packet. These properties are modeled in a class diagram as shown in Figure 2. All of these properties are modeled using the MARTE profile [8]. For instance, the packet loss property introduces packet loss during communication and is measured in terms of percentage. This property is defined to be of the MARTE type *NFP_Percentage*, which is defined in the MARTE profile for this purpose. Another property we defined is packet delay. This property is defined as a new, non-functional property (NFP) data type stereotyped as <<*NfpType*>> defined in MARTE (Figure 2). The new NFP type includes other properties such as *unit* of type *TimeUnitKind*. *TimeUnitKind* in MARTE defines units for time values such as millisecond and microsecond. We chose this data type so that a modeler can choose an appropriate time unit.

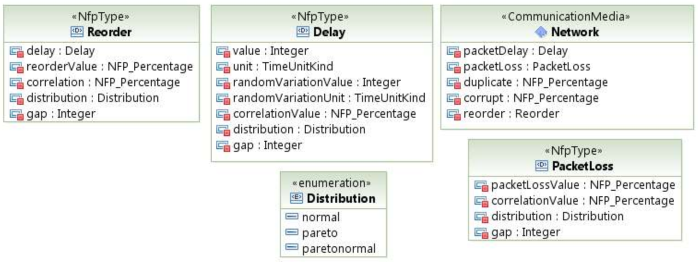

«NfpType»
Reorder
delay : Delay
reorderValue : NFP_Percentage
correlation : NFP_Percentage
distribution : Distribution
gap : Integer
«NfpType»
Delay
value : Integer
unit : TimeUnitKind
randomVariationValue : Integer
randomVariationUnit : TimeUnitKind
correlationValue : NFP_Percentage
distribution : Distribution
gap : Integer
«CommunicationMedia»
Network
packetDelay : Delay
packetLoss : PacketLoss
duplicate : NFP_Percentage
corrupt : NFP_Percentage
reorder : Reorder
«NfpType»
PacketLoss
packetLossValue : NFP_Percentage
correlationValue : NFP_Percentage
distribution : Distribution
gap : Integer
«enumeration»
Distribution
normal
pareto
paretonormal

Figure 2 Aspect class diagram for network communication

3.2.2 Modeling aspect state machine.

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

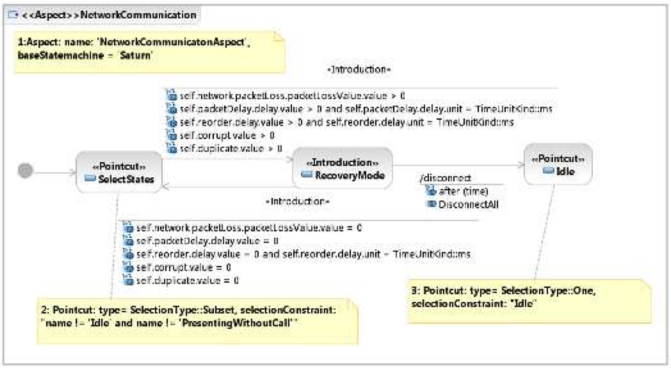

<<Aspect>>NetworkCommunication
1:Aspect: name: 'NetworkCommunicatonAspect', baseStatemachine = 'Saturn'
«Introduction»
self.network.packetLoss.packetLossValue.value > 0
self.packetDelay.delay.value > 0 and self.packetDelay.delay.unit = TimeUnitKind::ms
self.reorder.delay.value > 0 and self.reorder.delay.unit = TimeUnitKind::ms
self.corrupt.value > 0
self.duplicate.value > 0
«Pointcut» SelectStates
«Introduction» RecoveryMode
«Pointcut» Idle
/disconnect
after (time)
DisconnectAll
«Introduction»
self.network.packetLoss.packetLossValue.value = 0
self.packetDelay.delay.value = 0
self.reorder.delay.value = 0 and self.reorder.delay.unit = TimeUnitKind::ms
self.corrupt.value = 0
self.duplicate.value = 0
2: Pointcut: type= SelectionType::Subset, selectionConstraint: "name != 'Idle' and name != 'PresentingWithoutCall'"
3: Pointcut: type= SelectionType::One, selectionConstraint: "Idle"

Figure 3 Aspect state machine for network communication

A pointcut named *'SelectStatesPointcut'* on the state *'SelectedStates'* is shown in Figure 3 (see note 2), which selects all states of the base state machine except for the *Idle* and *PresentingWithoutCall* states. New transitions modeling robustness behavior of the system from all states selected by the *'SelectStatesPointcut'* pointcut to a new state *'RecoveryMode'* stereotyped with the <<*Introduction*>> stereotype are introduced. These transitions are modeled as UML change events. For instance, when *self.corrupt.value>0* in any of the states selected by the pointcut, the system goes to *'RecoveryMode'*, which is stereotyped as <<*Introduction*>> indicating that this state will be introduced in the base state machine. In this state, the system tries to recover the corrupt packets. If the system is successful, the transition with the change event *'self.corrupt.value =0'* takes the system back to the original state, which is one of the states selected by the *SelectedStates* state. If the system cannot recover within time *t*, then the system disconnects all the systems and goes to the *'Idle'* state, stereotyped as <<*Pointcut*>> (see Figure 3). This is modeled as a new transition from the *'RecoveryMode'* state to the *'Idle'* state, with a time event *after(t)*, and a new effect *'DisconnectAll'* with opaque behavior *disconnect*, which disconnects all the connected systems to the system.

4 Test Case Generation

In this section, we discuss how we extended our MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST) [6] for robustness testing.

4.1 An Overview of TRUST

In our previous work [6], we developed TRUST, whose software architecture and implementation strategy facilitate its customization to different contexts by supporting extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages. For example, the tool is extensible with respect to coverage criteria and it lets the user implement and integrate new coverage criteria with minimum changes to the tool [9]. The tool takes as input a UML class diagram and one or more UML state machines and outputs test scripts.

4.2 Integration of the AspectSM Weaver with TRUST

A weaver is a tool that takes as input a base model and one or more aspects and produces a woven model [10]. We developed a weaver for AspectSM using a set of transformation rules in Kermeta [11]. Figure 4 shows the architecture diagram for the weaver. The aspect weaver works in two steps. First it weaves aspect class diagram into the UML class diagram (e.g., Figure 1) corresponding to the base state machine using the transformation rules written in Kermeta [11]. These rules take as input an aspect class diagram (e.g., Figure 2) corresponding to an aspect state machine to be woven, a class diagram (e.g., Figure 1) corresponding to the base state machine, and output a class diagram which is the class diagram corresponding to the base state machine augmented with the aspect class diagram. In the second step, one or more aspect state machines (e.g., Figure 3) are woven into the base state machine. Since our queries (Pointcuts [7]) are in OCL, which need to be evaluated during the weaving process, we need to convert OCL expressions into Kermeta expressions. This is achieved through the *OCLToKermeta* component. Finally, *AspectStateMachineWeaver* produces a woven state machine which is a standard UML state machine. This state machine is then provided to the TRUST tool for test case generation.

4.3 Integration of Search-based Constraint Solver with TRUST

Emulating faulty situations in the operating environment of a VCS requires solving complex OCL constraints on the properties of the environment. These constraints must be solved during test case generation to emulate the faulty situations (i.e., to set the environment properties in a way for which such faulty situations occur). To efficiently solve these constraints, we developed a search-based OCL constraint solver [12], since current OCL solvers were not able to handle the complexity of our model's constraints within reasonable time. Figure 5 shows the architecture diagram for our Search-based Constraint solver. We developed a tool in Java that interacts with an existing library, an OCL evaluator called the EyeOCL Software (EOS) [13]. EOS is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our tool implements the calculation of branch distance (*DistanceCalculator*) [12] for various expressions in OCL, which aims at calculating how far are environment properties from satisfying constraints. The search algorithms employed are implemented in Java as well and includes Genetic Algorithms and (1+1) Evolutionary Algorithm [12].

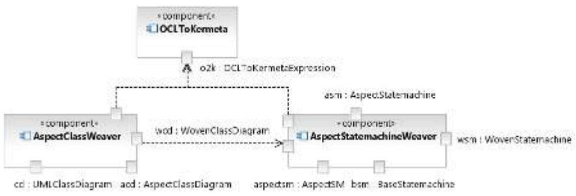


Figure 4 Architecture diagram for the weaver

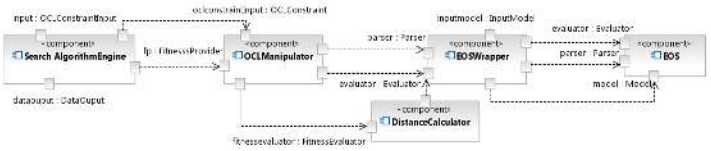


Figure 5 Architecture diagram for search-based constraint solver

5 Test Case Execution

In this section, we provide details on robustness test case execution. Section 5.1 describes our setup required for test case execution and Section 5.2 provides results of test case execution corresponding to the case study provided in Section 2.

5.1 Setup for Test Case Execution

Figure 6 shows our test execution setup for executing robustness test cases generated by TRUST. The current setup involves *Saturn*, which is the system under test (SUT) and three video conferences systems (*VCSs*). Since the execution of test cases requires emulating faulty situations in the environment, we needed a network emulator. For this purpose, we relied on software-based emulation facility (*netem* [14]). The setup of network emulator requires setting up a *PC* with three network interface cards (*NICs*). All communication to/from *Saturn* (*SUT* in Figure 4) passes through *NetworkEmulator*. *Saturn* is connected to *NIC3* of *NetworkEmulator* and all incoming and outgoing traffic from *Network* comes through *NIC1*. *NIC1* is bridged to *NIC3* and hence all the traffic goes to *Saturn* via *NIC3*. Our test case execution system is directly connected to *NIC2* of network emulator and through this *NIC* all faulty situations in the network are introduced by test scripts. All other communication from the test execution system to *SUT* and *VCSs* takes place through *NIC2* of *NetworkEmulator*. We separated them because if the faulty situations are introduced via the same NIC as other communication flows, we might end up affecting the commands that introduce faulty situations. Thus, we may end up not introducing faulty situations at all.

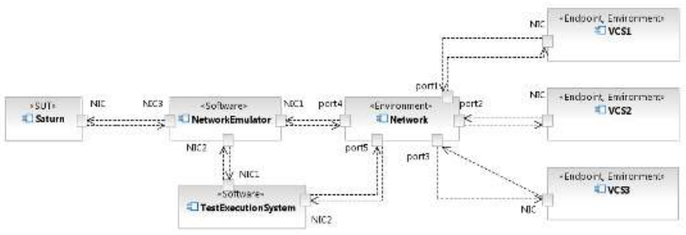


Figure 6 Setup for robustness test case execution

5.2 Preliminary Test Case Execution Results

For our current case study (Section 2), we used our weaver (Section 4.2) to produce a woven state machine. The woven state machine was given as input to TRUST (Section 4.1), which was configured to generate test cases using *All Transition Coverage* implemented by depth first search. In total 72 test cases were generated by TRUST. OCL constraints (change events in Figure 3) were solved using our search-based constraint solver (Section 4.3) to generate test data and introduce faulty situations in the environment. We executed test cases using the setup presented in Section 5.1. The execution of test cases found one robustness fault (halt and restart) in *Saturn*, when more than 10% duplicate packets were introduced in network communication. Our approach had more chances to catch this fault compared to existing practices in Cisco. MBT is more systematic and is in our case specifically tailored to catch robustness faults. Our approach indeed focuses on automatically testing the robustness of Saturn over various functional scenarios in the presence of several faulty situations in the network. In contrast, current robustness testing at Cisco is based on scripts written manually by testers to test a few network properties over a few of functional scenarios.

6 Experience and Lessons Learned

This section reports our experience of performing model-based robustness testing (MBRT) at Cisco. As often with many control and communication systems in industry, robustness testing is very critical for Cisco's Video Conferencing Systems (VCS). Currently, robustness testing at Cisco is driven by manually written test scripts, which is a common scenario in many industries. Due to time and resource constraints (e.g., system-level test cases are run with hardware-in-the-loop), only a limited number of test scripts can be written and only a limited number of faulty situations can be emulated. In these constrained cases, it is hence essential to carry out robustness testing in an automated and systematic way.

In order to support scalable modeling, aspect-oriented modeling (AOM) is adapted to support robustness modeling in the context of embedded systems and UML state machines (Section 3). Test cases are then generated based on system models including robustness behavior, using coverage criteria such as all round trip paths and all transitions criteria [9] (Section 4). Such an approach guarantees to cover important test scenarios that could be missed by manual testing, and thus leading to more systematic and comprehensive testing. Furthermore, the models can be used to generate effective, automated oracles (e.g., state invariants). Test cases are then executed using environment emulators (Section 5).

In the section below, we report on our experience of performing MBRT in Cisco. Since such reports are very rare in the literature (see Section 7), we believe that such section would provide useful insights in terms of the challenges we faced and the effectiveness of the solutions adopted in practice.

6.1 Robustness Modeling

In this section, we describe our experience and provide lessons learned obtained from modeling robustness behavior of Saturn, a VCS developed by Cisco. Details on our experience with functional modeling can be found in [6].

6.1.1 Experiences with AOM

Modeling the robustness behavior was performed by the authors with the help of testers in Cisco, who are currently involved in robustness testing. The modeling was done as part of a research project regarding the application of model-based testing technology in industry.

Before modeling, it was important to have meetings with software engineers at Cisco to understand the specifications of the robustness behavior implemented in Saturn. When the specifications were sufficiently understood, the modeling process started. The testers themselves were involved in the modeling of the robustness and functional behavior. The models were discussed and revised several times during the modeling, to ensure that the behavior is modeled completely and correctly. The robustness modeling took around seven hours. Understanding the specification took approximately four hours, whereas the actual modeling took approximately three hours. All the modeling was done with IBM Rational Software Architect (RSA) 7.5 as our UML profile (AspectSM) is also implemented in RSA. Note that this time accounts only for modeling the robustness behavior of Saturn in the presence of faulty situations in the network.

As we discussed in Section 3, robustness behavior crosscuts functional behavior. When robustness behavior is modeled directly with the functional model, the complexity of the resulting model increases enormously due to redundant modeling elements, which are scattered across the model (e.g., repeated in each state of the functional model). Modeling such redundant behavior requires substantial modeling effort if not modeled using an AOM methodology as the same behavior has to be modeled in several places in the model. As we discussed in Section 3, we employed AOM, and more specifically the AspectSM profile to reduce this accidental modeling complexity. Based on our experience with the Saturn VCS, we saved more than 95% of the modeling effort when measured by the number of modeled elements involved in the VCS robustness behaviors [7]. Of course, this effort is saved at the expense of learning and applying various stereotypes defined in AspectSM. We will further investigate the effort required to learn and apply AspectSM with more industrial case studies and controlled experiments in the future. However, the percentage of saving is so large that we consider these results to be very promising. In addition, modeling robustness behavior using AspectSM significantly improves the readability of the models as suggested from the results of a controlled experiment reported in [15].

Modeling crosscutting behavior in UML state machines provides enhanced separation of concerns. This means that a modeler/tester, or several of them with possibly different expertise, can focus on each crosscutting concern separately. They can model these crosscutting concerns separately from the core functionality and other crosscutting concerns (aspects). Our tool [7] can then be used to automatically weave these aspects with the behavioral models.

6.1.2 Experiences with MARTE

As we discussed in Section 3.2, we used a small subset of the MARTE profile to model properties of environment, whose violations lead to the faulty situations in the environment. The MARTE profile has a package dedicated to modeling non-functional properties (NFP). It provides different data types such as *NFP_Percentage* and *NFP_DataTxRate*, which are helpful to model properties of the environment, for instance jitter and packet loss in networks. When the built-in data types of MARTE are not sufficient, the open modeling framework of MARTE can be used to define new NFP types by either extending the existing NFPs or by defining completely new NFPs. For instance, we extended MARTE's NFPs and define several properties of the environment when modeling *echo* in audio streams and modeling *miss-synchronization* between audio and video streams coming to a VCS [7]. From our experience in using MARTE, we can conclude that the MARTE profile and its open modeling framework were sufficient to model relevant properties of the Saturn operating environment. In addition, the fact that

MARTE is a standard UML profile by OMG and hence is supported by many modeling tools [8] facilitates the adoption of modeling in industrial contexts since models are assets to be reused and modified over many years.

6.2 Test Case Generation

In this section, we discuss our experiences regarding the generation of robustness test cases.

6.2.1 Experiences with the TRUST tool

We have previously reported [6] the successful application of the TRUST tool in two companies to support functional test case generation. In our current application, we extended TRUST for robustness testing. For this purpose, we only needed to change the transformation rules in MOFScript [14] that generate the concrete test scripts. The modified transformation rules generate appropriate commands in the test scripts that emulate faulty situations in the environment. Generally, the transformation rules written in, e.g., Kermeta [11], MOFScript [14], or Query/View/Transformation (QVT) [16] are relatively compact and easy to read, write, and change as opposed to manipulating models using programming languages such as Java and C++. For the current implementation, we used MOFScript as Model-to-Text (M2T) transformation language, because it was the only M2T transformation language with good enough tool support (at the time of writing this paper).

6.2.2 Experiences with environment fault emulation

The most challenging part for test case generation was emulating faulty situations in the environment to test a system's robustness against them. A faulty situation in the operating environment is emulated when the properties of the environment are violated (Section 3.2). These violations are specified as change events (OCL constraints) on aspect state machines that lead to faulty states. To obtain a test suite that covers all the states in such UML models, it is hence important to find environment configurations for which these OCL constraints are evaluated to be true. Unfortunately, some of these constraints are complex, comprising of up to eight conjuncted clauses and hence are very difficult to solve using existing OCL solvers. For instance, we experimented with one well-known, downloadable OCL solver (UMLtoCSP) [17]. The results showed that, even after running that tool for 10 hours, no solutions could be found for most of the constraints. The reason is that the existing OCL solvers require the conversion of OCL to lower-level languages such as a Satisfiability (SAT) formula [18] or a Constraint Solving Problem (CSP) [17] instance and hence can easily result in combinatorial explosion as the complexity of the model and constraints increase (as discussed in [17]). For industrial scale systems, as in our case, this is a major limitation, since the models and constraints are generally quite complex. Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems. To solve this issue, we developed a new OCL solver based on search algorithms and managed to solve the same constraints in 3.8 minutes on average [12] on a regular PC. This gives empirical evidence that it is possible to quickly and directly solve complex industrial constraints written in a high-level language such as OCL, and hence efficiently emulates faulty situations in the operating environment for robustness testing purposes.

As we discuss in Section 6.2, we developed an OCL constraint solver in Java that interacts with an existing library, an OCL evaluator called EyeOCL Software (EOS) [17]. Our tool implements a set of heuristics as discussed in [12] for various expressions in OCL using EOS's API, which are then used by search algorithms to guide the search for input data that satisfy such constraints. We used EOS for both parsing and evaluating OCL expressions. We experienced that EOS is one of the most efficient OCL evaluators and provides a very simple API to evaluate and parse OCL expressions. In our experience, the only major downside of EOS is that, to evaluate/parse OCL expressions, EOS requires class and/or object diagrams to be loaded into its memory in a specific format. To facilitate this, we wrote a MOFScript transformation that takes the UML class diagram (modeling state variables, method calls, and signal receptions of the SUT) as input and generates a Java wrapper class that includes a set of EOS method calls for making class and object diagrams. During test case generation, we solve the constraints on the environment properties to emulate faulty situations in the environment using EOS and search algorithms. Another issue when solving an OCL constraint using a search algorithm is that it requires evaluating the OCL expression many times, and hence the speed of constraint solving is dependent on the efficiency of the selected OCL evaluator. Recall from Section 4.3 that we developed our TRUST testing tool with an open architecture such that any other OCL evaluator and parser (more efficient) can be easily replaced with EOS if required.

6.3 Test Case Execution

This section discusses our experience with test case execution at Cisco.

6.3.1 Experiences with setting up environment emulators

Executing robustness test cases is expensive because it requires setting up special equipment (hardware and/or software-based emulators) to emulate faulty situations in the environment. The emulators required in our current industrial case study are targeting networks, media streams and VCS. In our case, we only experimented with the network emulator because all communications between VCSs takes place via the network. It is hence important to test a VCS's behavior in the presence of faulty situations in the network. In our current application, we setup network emulator (*netem* [14]) once and then used it for testing without any additional settings for executing each test case.

6.3.2 Experiences with test case execution

Applying standard MBT criteria on UML state machines modeling the VCS results in test suites that are often to expensive or time-consuming to fit available test resources. For instance, in our current experiment, using a very simple coverage criterion on our (partial) case study (Section 5.2) resulted in 72 test cases, which would a take a long time to run in the test lab at Cisco Norway. This is expected to be a problem on most industrial systems, especially when modeling robustness along with the functional behavior. Executing large test suites is not practically feasible in many industrial contexts due to limited time and resources. For instance, running one robustness test case requires booking a specialized testing lab and takes on average 15 minutes on a Cisco's VCS. To cope with this practical problem, and in general to apply MBT in industry, there is the need of smart techniques to automatically select smaller subsets of test cases that can be run within testing budgets [19].

6.4 Current Limitations

As we discussed in Section 3, we need to model the faulty situations in the network, media streams, and VCSs communicating with a VCS under test (VUT). To date, we experimented only with emulating faulty situations in the network, which is just one aspect of the environment. Although we have already modeled the faulty situations in media streams (e.g., echo in audio and miss-synchronization between audio and video) [7], we do not have an appropriate media stream emulator yet. In addition to the media streams emulator, we also need to update our test script generator to generate test scripts that will control the media streams emulator during test case execution. For emulating faulty situations in other VCSs communicating with the VUT, we have not yet modeled the VCSs from that perspective. But we do expect that the models of the VCSs should be quite similar to the models of VUT, except for the need to select test paths from the models that will trigger faulty situations. For this purpose, we do have software-based emulators for VCSs, which can be utilized to emulate faulty situations during test case execution.

7 Related Work

Most of the work related to MBRT focuses on modeling and testing the behavior of a system when invalid inputs are given to the system, or in cases when exceptions (similar to exceptions in a programming language) are thrown in the SUT. For instance, Pintér and Majzik [20] report on the modeling of exceptions in statecharts in a similar fashion to Java mechanisms for writing exceptions (*try/catch* blocks). Exceptions are modeled as events on transitions in statecharts. Such statecharts are subsequently used for model checking. Lei et al. [21] provide a methodology to check the robustness of component-based systems in the case of invalid inputs. Test cases are then generated for invalid inputs at various states and the robustness of the system is checked. Nebut et al. [22] provide an automatic test generation approach based on use cases extended with contracts, after transforming them into a transition system. Their approach supports both functional and robustness test generation. Robustness test cases are generated by calling use cases when their preconditions are false.

The work presented in this paper is different from the existing work in MBRT in one or more of the following ways: 1) It focuses on modeling and testing system robustness in the presence of faults in its environment; this aspect has received little attention in the literature. In contrast, most of the existing work focus only on the behavior of a system when receiving invalid inputs [20] [21]. In contrast to the work presented in [22], our work is based on UML state machines, which is the main notation currently used for model-based test case generation [5]; 2) It uses AOM to model robustness behavior separately from the core, functional behavior, hence decreasing modeling effort by avoiding clutter in models, making them easier to read and decreasing chances of modeling errors; 3) It relies on modeling standards, in this case UML state machines and the MARTE profile [8], to model faulty situations of the environment. Using standards eliminate the need to adopt new notations and consequently facilitates the technology transfer to industry, as there are commercial modeling tools supporting UML and its extensions.

Other related works are the ones which employ search algorithms for non-functional testing. A recent systematic review [23] on the application of search algorithms for non-functional testing reveals that existing works focused on performance, quality of service, security, usability, and safety testing. None targeted robustness testing using search algorithms, as in our work.

8 Conclusion and Future Work

Model-based robustness testing (MBRT) is a solution for systematic and rigorous robustness testing for industrial embedded systems, as for example communication and control systems. MBRT involves testing the behavior of a system in the presence of faulty situations in its operating environment.

In this paper, we reported our experience of applying MBRT to video conferencing systems (VCSs) developed by Cisco Systems, Norway. Such industrial applications of MBRT and even more generally of model-based testing (MBT) are very rare in the literature. They are however very important to evaluate the scalability and applicability of MBT in realistic settings. We discussed how we integrated different tools and techniques to achieve the ultimate goal of automated and systematic MBRT. First, we discussed how we achieved scalable modeling of robustness behavior using Aspect-oriented Modeling (AOM) and more specifically using the AspectSM profile. AspectSM is a UML profile specifically designed to model robustness behavior with minimum extensions to UML to ease practical adoption. We also provided details on the weaver for AspectSM. Second, we provided details on the use of search algorithms (e.g., Genetic Algorithms) to solve complex constraints on environmental properties to emulate faulty situations. Third, we described the integration of the abovementioned tools with our model-based testing tool TRUST to achieve fully automated MBRT. Finally, we discussed the setup required to execute the test cases generated by TRUST and preliminary results when running the case studies on the VCS under test. The execution of test cases revealed a robustness fault in the VCS that had remained undetected by previous testing, in the presence of duplicate packets in the network during a videoconference. We then summarized our experiences and lessons learned while applying MBRT at Cisco.

This paper reports on a successful application of modeling to support testing in a real industrial setting. The results reported in this paper provide useful insights into the challenges and benefits of applying MBRT in a typical embedded system environment. One key success factor is to be able to address serious scalability issues (e.g., in constraint solving), which usually are not faced when dealing with small/artificial problem instances. However, there are still many research questions that need to be addressed. In the future, we are planning to extend the TRUST tool with more sophisticated test strategies specifically tailored to discovering robustness faults in a VCS. We also plan to perform robustness testing in the presence of faulty situations in other aspects of the environment such as in media streams and VCSs.

9 References

[1] "IEEE Standard Glossary of Software Engineering Terminology," IEEE, IEEE Std 610.12-19901990.

[2] "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability," *IEEE Std 982.1-2005 (Revision of IEEE Std 982.1-1988),* pp. 1-34, 2006.

[3] "Standard for Software Quality Characteristics," International Organization for Standardization, ISO-9126-32003.

[4] "Software Assurance Standard," NASA Technical Standard, NASA-STD-8739.82005.

[5] M. Shafique and Y. Labiche, " A Systematic Review of Model Based Testing Tools," Carleton University, Department of Systems and Computer Engineering, Technical Report (SCE-10-04)2010.

[6] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.

[7] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.

[8] "Modeling and Analysis of Real-time and Embedded systems (MARTE)," 2010.

[9] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] R. Yedduladoddi, *Aspect Oriented Software Development: An Approach to Composing UML Design Models*: VDM Verlag Dr. Müller, 2009.

[11] "Kermeta - Breathe Life into Your Metamodels," Rennes and Britanny IRISA and INRIA, 2010.

[12] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," in *Proceedings of the 11th International Conference On Quality Software (QSIC 2011)*, 2011.

[13] M. Egea, "EyeOCL Software," 2010.

[14] "netem," 2011.

[15] S. Ali, T. Yue, L. C. Briand, and Z. I. Malik, "Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines?," Simula Reserach Laboratory, Technical Report(2010-11), 2010.

[16] "Query/View/Transformation (QVT)," 2011.

[17] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*: IEEE Computer Society, 2008.

[18] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," in *8th International Workshop on OCL Concepts and Tools.* vol. 15: ECEASST, 2008.

[19] H. Hemmati, L. Briand, A. Arcuri, and S. Ali, "An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study," in *18th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*: ACM, 2010.

[20] G. Pintér and I. Majzik, "Modeling and Analysis of Exception Handling by Using UML Statecharts," in *Scientific Engineering of Distributed Java Applications*, 2005, pp. 58-67.

[21] B. Lei, Z. Liu, C. Morisset, and X. Li, "State Based Robustness Testing for Components," *Electronic Notes of Theoratical Computer Science,* vol. 260, pp. 173-188.

[22] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jezequel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions of Software Engineering,* vol. 32, pp. 140-155, 2006.

[23] W. Afzal, R. Torkar, and R. Feldt, "A Systematic Review of Search-based Testing for Non-functional System Properties," *Information and Software Technology,* vol. 51, pp. 957-976, 2009.