Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Investigation of Similarity-based Test Case Selection for Specification-based Regression Testing

Francisco Gomes de Oliveira Neto

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dra. Patrícia Duarte Lima Machado
(Orientadora)

Campina Grande - Paraíba - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O48i Oliveira Neto, Francisco Gomes de.
Investigation of similarity-based test case selection for specification-base regression testing / Francisco Gomes de Oliveira Neto. – Campina Grande, 2014.
134 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

"Orientação: Prof.ª Dr.ª Patrícia Duarte Lima Machado".
Referências.

1. Teste de Software. 2. Teste de Regressão. 3. Engenharia de Software Empírica. I. Machado, Patrícia Duarte Lima. II. Título.

CDU 004.415.53(043)=111

"INVESTIGATION OF SIMILARITY-BASED TEST CASE SELECTION FOR SPECIFICATION-BASED REGRESSION TESTING"

FRANCISCO GOMES DE OLIVEIRA NETO

TESE APROVADA EM 30/07/2014

[illegible]

PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Orientador(a)

EMANUELA GADELHA CARTAXO, D.Sc, UFCG
Examinador(a)

EDUARDO HENRIQUE DA SILVA ARANHA, Dr., UFRN
Examinador(a)

[illegible]

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

[illegible]

ADENILSO DA SILVA SIMÃO, Dr., USP
Examinador(a)

CAMPINA GRANDE - PB

Universidade Federal
de Campina Grande

Declaro, para os devidos fins, que participei por videoconferência da apresentação da defesa da Tese de Doutorado de **Francisco Gomes de Oliveira Neto**, intitulada: "Investigation of Similarity-based Test Case Selection for Specification-based Regression Testing", em 30 de julho de 2014 e considero o trabalho aprovado.

Emanuela Gadelha Cartaxo

Emanuela Gadelha Cartaxo (UASC/UFCG)

UFCG

SUPRA OMNES LUX LUCIS

Universidade Federal
de Campina Grande

Declaro, para os devidos fins, que participei por videoconferência da apresentação da defesa da Tese de Doutorado de **Francisco Gomes de Oliveira Neto**, intitulada: “Investigation of Similarity-based Test Case Selection for Specification-based Regression Testing”, em 30 de julho de 2014 e considero o trabalho aprovado.

Eduardo Aranha

Eduardo Henrique da Silva Aranha (UFRN)

Acknowledgements

Here I express my **gratitude** to everyone that witnessed my long journey comprising this doctorate research. The joy, challenges, obstacles and sacrifices undertaken could only be accomplished through the support of important people in my life. The outcome of this endeavour provided me encouragement to pursue personal goals, and exercised my belief that there are things beyond my understanding. Ultimately, this allows me to discover not only new ideas, but, more importantly, new boundaries. A list of the people and institutions below had an important role on this accomplishment.

To **José Bener** for all his partnership, love and comforting words that have given me the drive and energy to achieve my goals. More importantly, for unlocking my mind and heart to the clarity of true happiness. For helping me to accomplish and create dreams.

To my **family** for their love, understanding and support throughout this long journey. More specifically, my parents **Luis Oliveira** and **Lucia Macedo** who have taught me from the cradle that education and knowledge are among the most valuable assets of a human being. My siblings **Emmanuelle Oliveira** and **Adam Oliveira** for their love, support and care. To **Sueli Alves** and **Benardi Souza** for their love, kindness, support and friendship.

To **Richard Torkar** for his mentoring and friendship that inspiring me to achieve higher standards, apply my best and stay true to myself. For his never-ending encouragement to push the envelope and achieve higher standards. In addition, I am thankful for allowing me a life changing experience where I could exceed my own expectations and overcome professional and personal challenges.

To **Patrícia Machado** for her trust, supervision and, more importantly, for allowing me the freedom and autonomy to conduct the research. For providing me with challenging research problems, professional inspiration and above all, valuable advices that I will carry with me and pass it along my students.

To **Emanuela Cartaxo** for nearly a decade of teaching and friendship. For providing me the foundation of an academic career and showing the importance and responsibilities of achieving and applying research results. Alongside the professional inspiration and research partnership, I am thankful for her support, trust, friendship, for listening to my worries and for teaching me to never doubt myself.

To my best friend **Arthur Yuri Jordão** for all his patience and friendship that gave me joy, happiness and has always managed to make me laugh during the most stressful moments.

To **Robert Feldt** for his valuable input, feedback and collaboration that allowed this research to venture into new territory and overcome the challenges of experimentation with model-based techniques. All the hard and intense (yet fun) work/meetings have inspired me to improve the quality of this research.

To my friends from Gothenburg: **Ali Shahrokni, Antonio Martini, Nicole Costa, Hiva Alahyari, Ana Magazinius, Emil Alégroth, Ildiko Pilan** and **Lucas Gren** for their support, friendship and kindness. They made my stay in Sweden the best experience that I lived, so far. Despite the cold temperatures, I was able to find warmth during our lunches, conversations, fyka, and game nights. I also thank **Carolina Andrade** and **Emilia Villani** for their company, support and friendship during my stay in Sweden.

To my friends **Rafaella Italiano, Matthew Shropshire, Paloma Freire, Elloá Guedes** and **Andrea Mendonça** that provided me with comfort, joy and encouragement during these 4 years.

To my friends **Alberto Burity** and **Paola Alves** who encouraged me to expand my own knowledge and explore a reality that I assumed to be familiar with. In summary, for showing me the meaning of freedom, understanding and compassion.

To my colleagues **João Felipe Ouriques** and **Ana Emília Coutinho**, and the other members of the Software Practices Laboratory (**SPLab**) for all their support, collaboration and company during this research.

To **Chalmers University of Technology** and its **Software Engineering Division**, for receiving me during 9 months and providing complete assistance, help and cooperation during my stay. During that time, I was able to interact with great Software Engineering researchers and experience life in Gothenburg.

To the funding agencies **Coordenação de Aperfeiçoamento de Pessoal do Nível Superior** (CAPES) and **Conselho Nacional de Desenvolvimento Científico e Tecnológico** (CNPq) for their financial support throughout this work.

Last but not least, I thank the staff of **Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande** (PPGCC-UFCG) for all their administrative support. More especifically, **Rebeka Lemos** for her assistance and help.

Contents

Acronyms

ALTS - Annotated Labelled Transitions System

IEEE - Institute of Electrical and Electronics Engineers

LTS - Labelled Transitions System

LTS-BT - Labelled Transitions System - Based Testing

MBT - Model-based Testing

MOF - MetaObject Facility

OMG - Object Management Group

SART - Similarity Approach for Regression Testing

SBMTE - Search-based Model Generation for Technology Evaluation

STCS - Similarity-based Test Case Selection

SUT - System Under Test

TaRGeT - Test and Requirements Generation Tool

TTCN - Testing and TEst Control Notation

UML - Unified Modelling Language

UMLAUT - Unified Modelling Language All pUrposes Transformer

WSA-RT - Weighted Similarity Approach for Regression Testing

XMI - XML Metadata Interchange

XML - eXtensible Markup Language

List of Figures

List of Tables

Abstract

During software maintenance, several modifications can be performed in a specification model in order to satisfy new requirements. Perform regression testing on modified software is known to be a costly and laborious task. Test case selection, test case prioritization, test suite minimisation, among other methods, aim to reduce these costs by selecting or prioritizing a subset of test cases so that less time, effort and thus money are involved in performing regression testing. In this doctorate research, we explore the general problem of automatically selecting test cases in a model-based testing (MBT) process where specification models were modified. Our technique, named Similarity Approach for Regression Testing (SART), selects subset of test cases traversing modified regions of a software system's specification model. That strategy relies on similarity-based test case selection where similarities between test cases from different software versions are analysed to identify modified elements in a model. In addition, we propose an evaluation approach named Search Based Model Generation for Technology Evaluation (SBMTE) that is based on stochastic model generation and search-based techniques to generate large samples of realistic models to allow experiments with model-based techniques. Based on SBMTE, researchers are able to develop model generator tools to create a space of models based on statistics from real industrial models, and eventually generate samples from that space in order to perform experiments. Here we developed a generator to create instances of Annotated Labelled Transitions Systems (ALTS), to be used as input for our MBT process and then perform an experiment with SART. In this experiment, we were able to conclude that SART's percentage of test suite size reduction is robust and able to select a subset with an average of 92% less test cases, while ensuring coverage of all model modification and revealing defects linked to model modifications. Both SART and our experiment are executable through the LTS-BT tool, enabling researchers to use our selection strategy and reproduce our experiment.

Chapter 1

Introduction

Quality is one of the key aspects of a successful product release, and that is a recurrent concern for software systems. Software engineering addresses software quality through different approaches and one of the most popular is software testing. Unfortunately, effectively testing a software is costly, thus encouraging researchers and industry practitioners to propose several cost-effective techniques in order to make software testing more affordable.

Maintaining quality is even more challenging whenever a software system changes since modifications can affect proper functioning of a system. Examples of modifications are: Bug fixing, enhancement or removal of a functionality, code refactoring, updates to the operating environment, etc. Modifications could occur both during and after software development, and therefore, (re)testing activity named *regression testing* is recommended [Agrawal et al. 1993]. The specific goal of regression testing, when comparing to other testing approaches (e.g. system, acceptance or integration testing) is to reveal defects caused by a modification, named *regression defects* or *regression faults* [Binder 1999].

Regression testing depends on the development context. For agile development with constantly changing functionalities, regression testing should be performed every time the software is saved and compiled. On the other hand, release of a new version, patch or nightly build represents the moment for regression testing during traditional development processes. In either cases, the goal is to increase the confidence that the modifications behave as expected and do not affect unchanged parts of the system [Harrold and Orso 2008; Binder 1999].

In order to verify whether modifications affected the proper functioning of the software, the test history[1] of previous versions of the software is required. Given that software undergo several modifications throughout its life cycle, the artefacts related to regression testing can grow to an extent where both the execution and management of all test cases become impracticable. Even though it is very important to assess the quality of a modified software, regression testing is very expensive [Binder 1999; Korel et al. 2002; Harrold and Orso 2008; Tamimi and Zahoor 2011].

To address this cost issue, researchers have proposed *regression test case selection* strategies to reduce the size of a test suite by selecting only a subset of test cases to be tested. This strategy is recommended whenever the costs of executing the entire test suite is higher than selecting and executing only a subset of these test cases. The drawback, on the other hand, is that some of the defects may not be revealed because some of the scenarios may not be covered during the test. Thus, an appropriate selection strategy needs to be used in order to increase chances of detecting defects.

For example, a very simple strategy is to randomly select the test cases until the subset meets the testing time and budget constraints. The random selection is easy to use in practice since it does not require any information or assumptions based on the system under test (SUT). However, without an appropriate coverage criterion to select test cases, critical requirements and components of the system may go untested increasing risks of releasing a defective product and compromising the company's reputation.

Moreover, regression test cases can be selected in two different contexts: Code-based or specification-based selection. The first considers source code modifications (e.g. refactoring) as the main guideline, for example coverage of methods, classes, code statements, among others. The specification-based selection, on the other hand, uses modified specification models, such as state machines, or UML (Unified Modelling Language) diagrams, as the main guideline for test case selection. Then, modelling elements such as transitions and states could be used as coverage criteria or test requirements.

Regression testing at the code level has been widely explored in the literature [Korel et al. 2002] and enables solution for most software's structural problems such as modified code coverage or finding data and control dependences affected by modifications. Besides,

[1]Test artefacts of previous versions. For example, test cases and reports from previous versions, analysis and correction of defects from previous versions.

the selected test cases are already written in code language enabling automatic execution of the code and test cases, instead of requiring models and documents used for specification. Consequently, the code itself is sufficient to both select test cases and test the system.

However, there are three main problems when performing code-based selection. First, the test cases are described using the source code, hindering traceability between modifications at higher levels of abstractions (e.g. requirements, subsystems, components) and the respective part of modified source code. The second problem is understanding what is being tested, since reading the test cases requires knowledge of the programming language(s) used to write the tests. The third problem is the dependence between the code and the tests, so that code refactoring or small changes in methods will require maintenance on the test suite [Chen et al. 2007; Fahad and Nadeem 2008].

At the same time, there has been a growing interest in specification-based regression testing due to the many benefits of handling high level models. In addition to help addressing those problems, specification-based approaches bring regression testing closer to a requirements/functional level as well as scaling better to large and complex systems when compared to traditional code-based approaches [Briand et al. 2002]. On the other hand, test cases generated from specification models are usually *abstract*, and thus cannot be automatically executed. Also, most specification models are usually described in high-level languages such as natural language, hence requiring reliable Model-based Testing (MBT) approaches to properly handle and relate model information to testing artefacts.

MBT benefits from a specification model to enable automatic generation, selection or execution of test cases. Therefore, the abstract specification model of software can be used as a basis for testing a concrete implementation. In this doctorate research, we address a system's behaviour and therefore, transitions and states of software models represent *steps and scenarios of use cases*. Also, modifications are represented as changes to transitions and, consequently, the states connected by those transitions, since we focus on modifications at specification models.

In addition, proposed selection strategies need to target the considered scope of testing. For example, specification-based selection is not recommended when only the source code changes (e.g. code refactoring) because source code analysis is required and providing traceability between requirements and code statements can be costly. Similarly, if we consider large and complex software systems, code-based selection may be unable to identify all parts of the code that exercise a modified requirement. In summary, both contexts complement each other and by combining both the source code and the specification elements, we are able to select test cases based on modifications performed at different levels of the system [Carver and Tai 1998; Korel et al. 2002].

1.1 Problem and Proposed Solution

This doctorate research focuses at *specification-based test case selection for regression testing*. Therefore, our goal is to provide strategies to select test cases whenever specification models of a software system are modified. In other words, our problem is *the selection of a representative subset of test cases in order to reduce the costs of regression testing at the system level.* In order to achieve cost-effective regression testing, we need to maximize the chances of defect detection, as well as minimize the number of test cases needed.

To portray our problem, imagine that during a meeting clients request a new software version in order to include and remove several functionalities. Therefore, the specification models need to be modified as well in order to reflect the new version of our software. Based on the assumption that there are not enough resources to execute all test cases for this new version, we apply test case selection to identify a representative subset to enable regression testing under the given resource's constraints.

In general, representativeness of a test set refers to a test criterion (also known as test requirement) used to identify representative test cases. Some examples of such criteria are: Code or defect coverage, critical functionalities tested, among others. Here, we consider as representative the *test cases exercising modifications* performed on a software system. Those test cases, often referred as *modification-traversing test cases*, comprise the set of test cases that, when executed, will exercise the modified parts of a software system. That is a very common criterion for selecting test cases for regression testing since empirical studies and experiments [Rothermel and Harrold 1996; Yoo and Harman 2012] show that the set of modification-traversing test cases are the closest approximation to the set of test cases able to to reveal regression defects (*fault-revealing test cases*). The question is: *How* can we identify these test cases in a test suite? Our solution is then to combine two different approaches to address the mentioned problem. They are: Model-based regression testing and similarity-based selection.

Model-based regression testing relies on MBT approaches to enable use of software models (e.g. UML diagrams or even Control Flow Graphs generated from source code) in order to automate test case selection [Farooq et al. 2010]. Moreover, those models carry details regarding software behaviour and component interactions that is often hard to see by just reading the code. In turn, similarity-based test case selection (STCS) relies on similarity functions to select the more (or less) different test cases, hence enabling removal of redundancy among test cases [Cartaxo et al. 2007a; Cartaxo et al. 2011]. The benefit with this type of selection is testing a diversity of scenarios in a SUT. Besides, similarity functions are usually mathematical functions easy to understand and incorporate in a tool.

We propose usage of a similarity function to identify modifications between different versions of specification models and allow automatic identification and selection of test cases exercising the modified parts of a specification model. Previous to this thesis we have proposed the Weighted Similarity Approach for Regression Testing (WSA-RT) technique to reduce test suites focusing on coverage of important[2] test cases exercising modified parts of the specification [de Oliveira Neto 2010]. Here we expand the technique (renaming it to Similarity Approach for Regression Testing - SART) and focus on a thorough investigation regarding the effectiveness of STCS for specification-based regression testing. Thus, the value-based approach is not discussed in this research and is instead presented as an example in Appendix A.

1.2 Research Questions and Methodology

The main goal established for this thesis is to investigate whether *similarity functions are able to identify modification-traversing test cases in a test suite*. And in order to achieve that goal, we must answer the following research questions:

RQ1: How to use similarity functions to identify modifications?

RQ2: How to select test cases based on the identified modifications?

[2]The technique used a value-based approach where weights were manually assigned to test cases in order to determine their importance.

RQ3: How to address redundancy issues occurring in test suites from both regression test suites and MBT approaches?

RQ4: Is our selection strategy beneficial for regression testing when compared to direct application of similarity functions on a test suite?

Alongside the difficulties in exploring our selection strategy, its evaluation was one of our main concerns since an experiment would be necessary to obtain significant conclusions and assess our hypothesis. Thus, we asked how can we evaluate our proposed model-based technique? Model-based (MB) techniques are frequently evaluated through case studies with specific models, leading to conclusions that are hardly general and often limited due to the context of the study. Systematic, empirical evaluation, such as an experimental study, on the other hand, allows the investigation into the strengths and weaknesses of a technique, backed by extensive empirical data and rigorous statistical analysis. But at the same time, it requires large samples of realistic models (i.e. large number of models with characteristics such as size and type that are typical of models used in organizations developing software), often unavailable to researchers.

To address this issue we decided to use stochastic model generation with descriptive statistics of realistic models [de Oliveira Neto et al. 2013]. Combining information from actual models used in practice with stochastic model generators would allow us to generate large number of models that share characteristics with industrial models. The automatically generated models will create a *space of models* that can be used as input for our technique. We named this approach Search-Based Model Generation for Technology Evaluation (SBMTE).

For being a contribution of our study, we also investigated some aspects towards usage of SBMTE for evaluation of MB techniques in general, more specifically, experimental evaluation of MB techniques. Would it possible to obtain large samples of models to enable statistical analysis of a technique's performance? If so, would our generator tool be able to generate realistic models to achieve conclusions of practical significance? And ultimately, would that practical significance help technology transfer of MB techniques? Definition and usage of a model generator tool has provided valuable insight towards evaluation of MB techniques, such as existing challenges that hinder experimental studies and how we are able to overcome them.

An overview of our methodology, including schedule, activities and artefacts is presented in Figure 1.1. Based on the research questions, we began a literature review to find directions towards an appropriate combination of similarity-based strategies and specification-based regression test, leading, then to the proposal of WSA-RT. Next, we started to use WSA-RT on case studies with both toy and real[3] specification models to gather data and to improve our strategy towards a more precise selection of test cases. The result was the current version of our technique named Similarity Approach for Regression Testing, or simply SART.

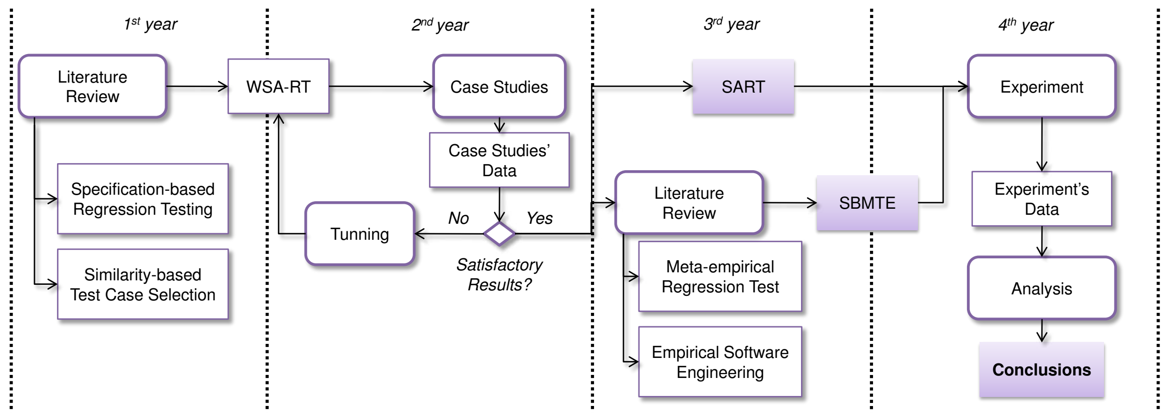


Figure 1.1: The activities, schedule and artifacts of our research.

In order to enable SART's evaluation through an experiment, we began searching for approaches to overcome the lack of availability of models to use as input in our MBT process. As a result, we proposed SBMTE in collaboration with Professor Robert Feldt from Chalmers University of Technology (Gothenburg, Sweden) and created our generator tool to obtain large samples of realistic specification models. The next step consisted in evaluation of our research by defining, planning and executing an experiment based on our usage of a stochastic model generator. Finally, the data collected during the experiment was analysed and provided answers to our research questions.

[3]The specification models obtained were not for industrial tools, instead, we used specification for open-source and academic tools.

1.3 Contributions

Several contributions are presented throughout this work, but for now we highlight two main contributions. The first one is a similarity-based test case selection technique (SART). Most model-based regression test case selection relies on model comparison to identify modifications, that can lead to dependencies with a specific type of model, or high costs and complexity for comparison of large specification models or software systems. On the other hand, our technique relies on similarity functions also known as distance functions that are easy to adapt and independent of a model type.

The second contribution is a tool that uses stochastic model generation as an alternative for empirical evaluation of model-based techniques. As a result our experiment can be easily adapted and executed by other researchers. By providing means to overcome the general lack of samples of specification models, we intend to encourage researchers to perform experimental studies with their own techniques. This evaluation approach allows early validation of an MB technique enabling the technique's improvement before presenting it to industry practitioners.

1.4 Chapter Concluding Remarks

This introduction chapter presented an overview of our entire doctorate research. Further details regarding the remainder of our research is discussed in the upcoming chapters according to the following structure.

Chapter 2 provides the background required to better understand the targeted solution of our research, such as model-based regression testing and test case selection. Next, the technique used to execute our strategy is detailed in Chapter 3. Other proposed work and their relation to our proposed technique are discussed in Chapter 4. We discuss stochastic model generation and present our model generator tool in Chapter 5, in order to allow the reader to understand the evaluation methodology and the experimental design discussed in Chapter 6. Results and analysis are then presented in Chapter 7 followed by Chapter 8 where conclusions are drawn and future work is discussed.

Chapter 2

Theoretical Background

Software testing is a part of the software development process where tests are designed and then executed in order to investigate quality attributes and find defects [Jorgensen 2002]. One of the goals for testing software is revealing defects or demonstrating to developers or customers that the software meets its requirement.

Whenever discussing software testing it is important to clarify the differences between the terms “error”,“fault”, “defect”, and “failure”. According to IEEE definition, a mistake or *error* is a human action that caused a *defect* (also known as *fault*) to appear in the product that the person is working on (e.g. requirements specification or software components). The defect has no impact on the operation of the software if it is not encountered when the software is used. However, if the defect is encountered, the product *fails* to meet the user’s needs. Therefore, defects can compromise business reputation or economic viability, and in some cases, even the environment or public safety [IEEE 2013].

In order to test a software system, a set of test cases (named *test suite*) must be chosen. These *test cases* are composed of elements able to describe software behaviour, such as the system’s pre-conditions, inputs, expected outputs, states of the system, among others. If the output produced by the System Under Test (SUT) matches the expected output specified in the test case, then we assign a *pass* verdict to the executed test case. A *fail* verdict is assigned otherwise, and thus begins the investigation and correction of the defect(s) that caused this failure.

This format for test case can be seen throughout all of the system’s levels [Beizer 1990; Jorgensen 2002], from method calls (e.g. unit level) to components and, ultimately, the system. The latter is the focus of our research, thus our test cases are usually described in natural language and need to be executed manually (named *abstract test cases*). Albeit the difficulty in tracing abstract test cases to the respective executable code parts, it is easier to identify which functionalities or use case scenarios are being tested. Thus, they are recommended for black-box testing approaches, for example acceptance testing.

Despite its benefits, it is known that software testing is costly [Beizer 1990; Harrold and Orso 2008]. Therefore, most of the times software testing is not executed properly or skipped, hence compromising the quality of software being produced. Studies for reducing costs of software testing have been proposed and are still being researched by the software engineering community. Three approaches are well known for addressing this issue: Test case prioritization, test suite minimization (or reduction), test case selection.

In test case prioritization, a *priority is assigned* to test cases and those with highest priorities are scheduled to execute first in an attempt to reveal defects at early stages of testing, enabling early correction of these defects. In turn, the goal for test suite minimization (also known as test suite reduction) is to choose a subset of test cases with *equivalent coverage* in relation to the original test suite, concerning a specific criterion, or test requirement. The size reduction is mainly provided by removing redundant test cases, i.e. test cases that provide the same coverage concerning the test requirement analysed (e.g. transitions or statements covered) [Ma et al. 2005]. Test case selection is used to *select a subset* of test cases that meets resource constraints of the test. Despite being similar to minimization techniques, the selected subset may not provide the same coverage as the original test suite.

Among proposed work in literature, this research focuses on test case selection. Concepts related to our selection strategy, such as regression testing and model-based testing will be addressed in the next sections of this chapter. Also, in order to explain our evaluation methodology, definitions of meta-empirical studies and experiments are presented.

2.1 Model-Based Testing (MBT)

Model-based testing is a *black-box* approach for automatically generating tests from models representing software [El-Far 2001]. The generated test cases are executed in order to evaluate correspondence between software implementation and its specification model, thus a formal model specifying software behaviour is required [Dalal et al. 1999]. The main activities and artefacts of an MBT approach are presented on Figure 2.1. Given that the MBT process starts with software requirements, testing can begin once the requirements are specified.

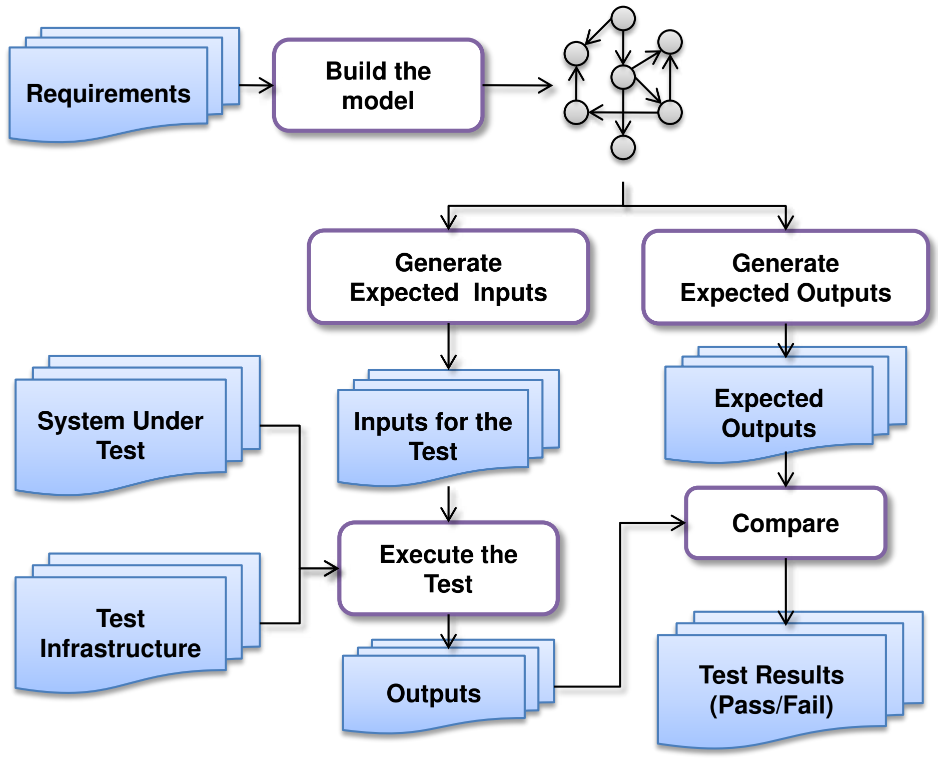


Figure 2.1: Activities and artifacts of an MBT Approach.

MBT approaches provide two main benefits. The first one is that models can help in communication between developers and testers; the second one is that these activities can be performed automatically, hence reducing costs and effort related to software testing. However, MBT approaches are seen by many to be too reliant on the specification model.

The model needs to represent information accurately since good results with MBT techniques also depend on good models provided as input [Beizer 1990]. For example, bad models with inconsistencies or ambiguous information will affect negatively the performance of an MBT technique. There are several aspects that affect the quality of a model, such as completeness and correctness of information being represented and the expertise of people responsible for creating and maintaining the model. By building a *good model* we are able to design test cases using information from models (e.g. expected inputs and outputs).

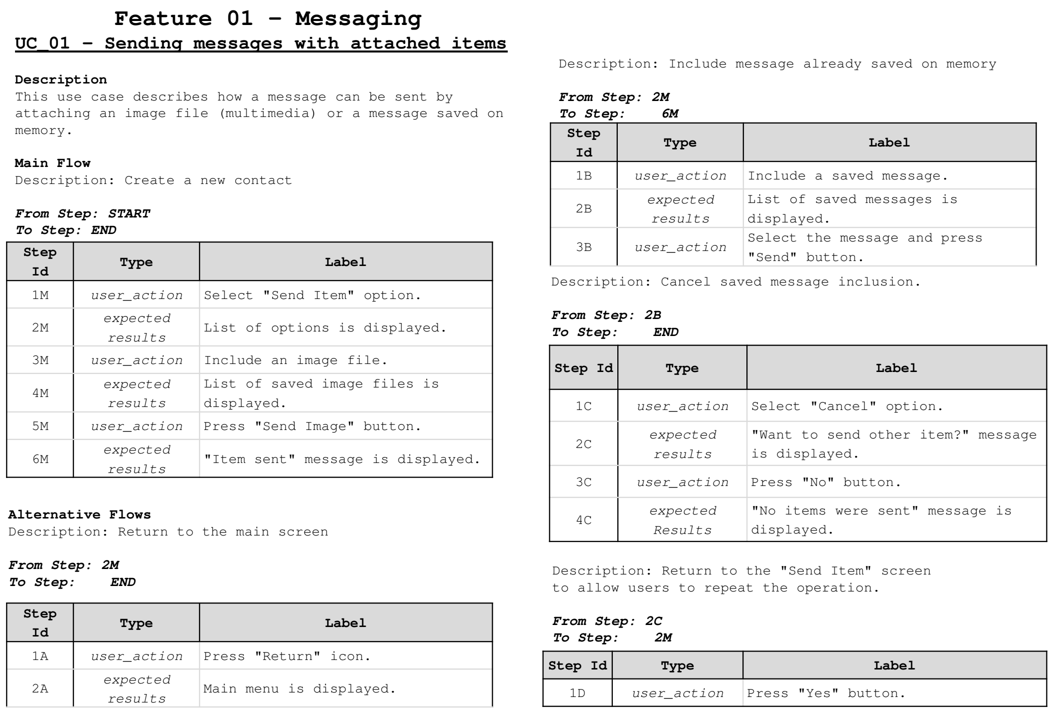
Our work uses a document template (Figure 2.2) adapted from Cabral and Sampaio [Cabral and Sampaio 2008] to describe main and alternative flows of use cases. In its original version, the use case template considered each step (of a use case) to be composed by a description combining a user action, and the corresponding system state and response. Our adaptation separates those elements as a sequence, in which a stakeholder can describe the user action and the respective system response separately. That allows more versatility in definitions of alternative flows and paths with loops, since our adaptation enables two different steps to share a common expected system response.

Then, from this use case template, we generate a Labelled Transition System (LTS) model from those use case documents to provide an intermediary model format for automatic test case generation. An LTS is defined as a 4-tuple $S = (Q; A; T_{tr}; q_0)$, where:

- Q: The set of states;
- A: A finite non-empty set of labels;
- T_{tr}: The transition relation ($T_{tr} \subseteq (Q \times A \times Q)$), where (q_a, l, q_b) indicates:
  - q_a and q_b are, respectively, a source and sink (or destination) state, and l is a label;
- q_0: The initial state.

Aside from being used as a semantic formalism for several tools (TGV [Jard and Jéron 2005], LTS-BT [Cartaxo et al. 2008], TaRGeT [Nogueira et al. 2007], UMLAUT [Jzquel et al. 1999], etc.), LTS is a simple format that is capable of representing steps of user actions and expected results, thus being a suitable candidate to our context. Unlike similar UML Diagrams, LTS can be expressed simply on a standardized text file format (Trivial Graph Format - TGF) removing dependencies to modelling tools or XMI processing libraries. On the other hand, using a generic graph format hinders applicability in real software development processes where UML models are dominant in industry.

Therefore, by choosing LTS, we can write simple yet expressive (system level) specification models, that enables early and numerous execution of our selection strategy. That simplicity also allows LTS to be an underlying semantics models for other formalism (e.g. Finite State Machines), and, consequently, it becomes easier to extend its usage to consider



Feature 01 - Messaging

UC_01 - Sending messages with attached items

Description
This use case describes how a message can be sent by attaching an image file (multimedia) or a message saved on memory.

Main Flow
Description: Create a new contact

From Step: START
To Step: END

Step Id	Type	Label
1M	*user_action*	Select "Send Item" option.
2M	*expected results*	List of options is displayed.
3M	*user_action*	Include an image file.
4M	*expected results*	List of saved image files is displayed.
5M	*user_action*	Press "Send Image" button.
6M	*expected results*	"Item sent" message is displayed.

Alternative Flows
Description: Return to the main screen

From Step: 2M
To Step: END

Step Id	Type	Label
1A	*user_action*	Press "Return" icon.
2A	*expected results*	Main menu is displayed.

Description: Include message already saved on memory

From Step: 2M
To Step: 6M

Step Id	Type	Label
1B	*user_action*	Include a saved message.
2B	*expected results*	List of saved messages is displayed.
3B	*user_action*	Select the message and press "Send" button.

Description: Cancel saved message inclusion.

From Step: 2B
To Step: END

Step Id	Type	Label
1C	*user_action*	Select "Cancel" option.
2C	*expected results*	"Want to send other item?" message is displayed.
3C	*user_action*	Press "No" button.
4C	*expected Results*	"No items were sent" message is displayed.

Description: Return to the "Send Item" screen to allow users to repeat the operation.

From Step: 2C
To Step: 2M

Step Id	Type	Label
1D	*user_action*	Press "Yes" button.

Figure 2.2: Example of a use case document used as input for our MBT process. A complete version is available on Appendix C

specification of non-determinism and timed models [Broy et al. 2005]. Furthermore, an LTS is able to visually present the system's behaviour regarding main and alternative flows of a use case, without requiring much effort in building the model.

Internal and external actions can be represented in an LTS, but since we are focusing on functional system testing, the transitions will represent interactions between the user and the system, and the system states required to execute the scenario. Annotations can be used on the LTS (Annotated LTS - ALTS) to indicate special types of interactions, for example user actions, system states, among others types of interactions [Cartaxo et al. 2007b]. Figure 2.3 presents an example of an ALTS generated from Figure 2.2. Because the ALTS is a key element in our evaluation, a more detailed description of ALTS models is provided in Chapter 5.

In order to automatically generate test cases from an LTS, we need to establish a coverage criteria such as: Requirements, paths with loops, all transitions, pairs of transitions, all states, among others mentioned in the literature [Utting and Legeard 2006]. In our approach, we use a Depth First Search (DFS) algorithm to find all paths of the LTS beginning at the initial

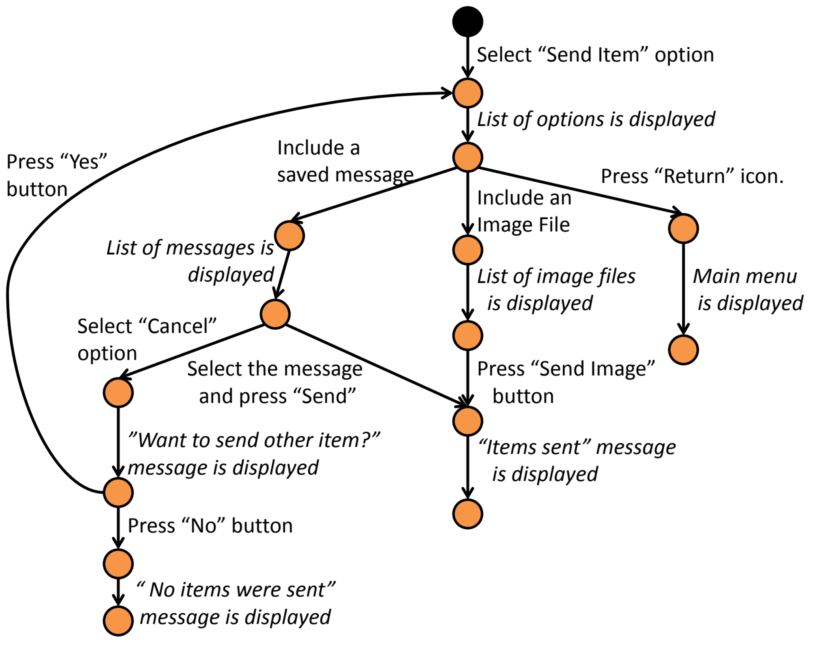

Select "Send Item" option
List of options is displayed
Press "Yes" button
Include a saved message
Press "Return" icon.
Include an Image File
List of messages is displayed
List of image files is displayed
Main menu is displayed
Select "Cancel" option
Select the message and press "Send"
Press "Send Image" button
"Want to send other item?" message is displayed
"Items sent" message is displayed
Press "No" button
" No items were sent" message is displayed

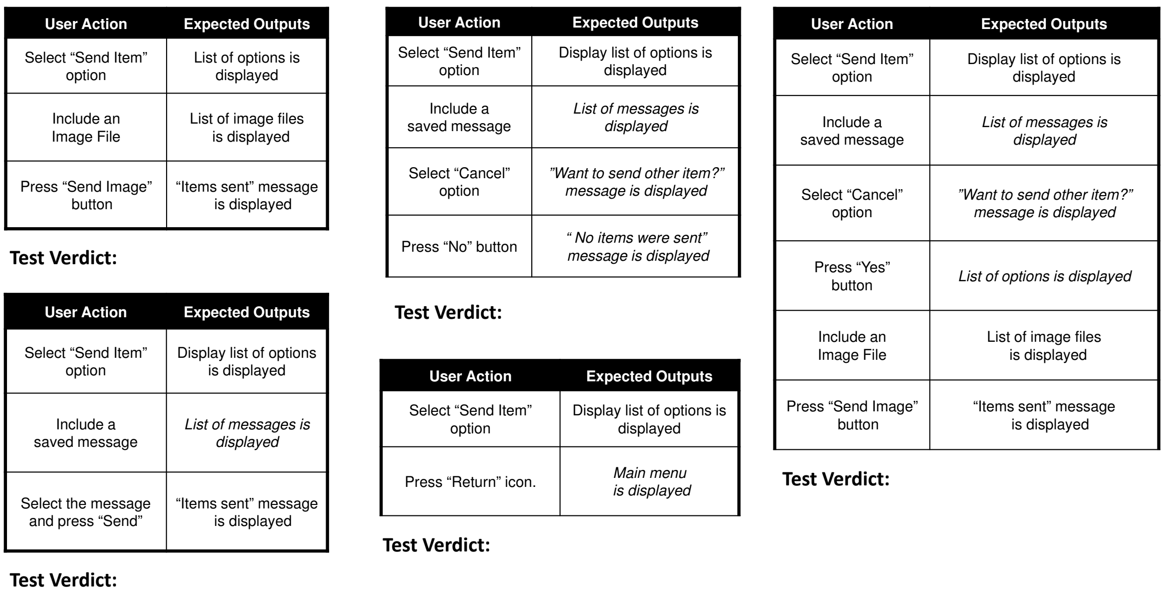
Figure 2.3: Example of an LTS generated from our use case document template.

state and ending on a state without outgoing transitions that *do not* create cycles (e.g. a leaf). In order to avoid overhead during test case generation and uncontrolled growth of our test suite, we determined that all paths with loops are traversed only once. In other words, upon finding a cycle (e.g. paths with loops), the generation continues until all transitions have been traversed at most twice or until a leaf is found.

The DFS algorithm was chosen among other techniques in literature because it is a simple algorithm, easy to implement and to execute[1] Figure 2.4 illustrates an example of a test suite generated from our toy LTS.

In our example we were able to automatically generate test cases exercising all software behaviours specified in the model. The main drawback is that the generated test cases need to be executed manually since our test suite is described in natural language. Eventually, a controlled natural language or model-driven testing approaches can be incorporated in this strategy to reduce this gap between test cases and code. Nonetheless, having test cases at early stages of software development is very beneficial for allowing early testing of software, and availability of a software specification at high level improves visualization of the system

[1]More sophisticated test case generation techniques may provide better or worse performance, however that is outside our scope of test case selection investigation.



User Action	Expected Outputs
Select “Send Item” option	List of options is displayed
Include an Image File	List of image files is displayed
Press “Send Image” button	“Items sent” message is displayed

Test Verdict:

User Action	Expected Outputs
Select “Send Item” option	Display list of options is displayed
Include a saved message	*List of messages is displayed*
Select the message and press “Send”	“Items sent” message is displayed

Test Verdict:

User Action	Expected Outputs
Select “Send Item” option	Display list of options is displayed
Include a saved message	*List of messages is displayed*
Select “Cancel” option	*”Want to send other item?” message is displayed*
Press “No” button	*“ No items were sent” message is displayed*

Test Verdict:

User Action	Expected Outputs
Select “Send Item” option	Display list of options is displayed
Press “Return” icon.	*Main menu is displayed*

Test Verdict:

User Action	Expected Outputs
Select “Send Item” option	Display list of options is displayed
Include a saved message	*List of messages is displayed*
Select “Cancel” option	*”Want to send other item?” message is displayed*
Press “Yes” button	*List of options is displayed*
Include an Image File	List of image files is displayed
Press “Send Image” button	“Items sent” message is displayed

Test Verdict:

Figure 2.4: Example of a generated test suite.

and helps stakeholders to be on the same page when developing a product.

One of the big problems with MBT in general is the size of the generated test suite. For large and complex system, the number of possible scenarios can be very big, hence impairing the execution of all generated test cases. Consequently, whenever the specification model changes, the costs of finding the test cases traversing model modifications are even bigger due to the test suite’s complexity and size. Automatic selection strategies, such as ours, can aid overcoming that difficulty and alleviate the problems of having large and redundant test suites.

2.2 Automatic Model Generation

The increasing popularity of model-based technology has led to a wide range of techniques proposed in different fields of software engineering research. Models gather valuable information regarding software and enable harness of knowledge concerning internal, external, structure, and/or behavioural interactions. Accordingly, there are several model formats to represent different types of information from software systems, such as structure (class diagrams, object diagrams) or behaviour (state machines, sequence diagrams).

Therefore, building good and consistent models is not easy and usually becomes an expensive task, consuming a lot of time and effort from the development process. As an alternative, automatic generation of models can be used, either to generate models from scratch [de Oliveira Neto et al. 2013], to extract properties of the system [Feng et al. 2007; Huselius et al. 2006; Lorenzoli et al. 2008; Deeptimahanti and Sanyal 2011], or to perform model transformation [Brottier et al. 2006; Sen et al. 2009] with specific quality attributes under control, for instance, coverage and diversity.

The Model-Driven Engineering (MDE) field has been targeting model generation from different angles. In a sense, the transformations between models can be seen as a well defined generation based on constraints established by meta-models or Object-Constraint Language (OCL) expressions [Guerra 2012]. That leads to several issues to preserve the model's information when transitioning between different types of models and levels of abstraction. A different approach could also be to then generate instances of models as input to test these transformations [Brottier et al. 2006; Sen et al. 2009]. That may resemble our approach (SBMTE), but ours is concerned in generating a space of models and finding regions of this space for optimal or near-optimal solutions, i.e. models with a positive effect on an MB technique's performance.

An example of model generation similar to ours is the technique proposed by Kanstrén [Kanstrén 2009] where a model type is defined to generate instances of that model to enable automatic test case generation. Then, behavioural patterns of the SUT are defined to extract traces of execution resulting in executable test cases. The advantage of their approach is that the generated test cases covered complex interactions that could only be seen because of the generated models. But unlike SBMTE, his approach requires an actual SUT and human interaction to edit the generated models.

In other work, Cartaxo proposed an algorithm to generate LTS models [Cartaxo 2011]. However, her generator does not address levels of realism among generated models focusing instead on generation of toy LTS. Furthermore, her generator did not perform modifications on the LTS and had different constraints and rules to guide the construction of models (e.g. choice of states and transition).

SBMTE is different from the work mentioned throughout this section because it focus on empirical evaluation by providing an underlying approach to investigate MB techniques in general. To the best of our knowledge, we could not find work generating samples of models to overcome the lack of availability of real models. The generation of models is widely used for different purpose, but the meta-heuristic search within the space of models and the concern in having realistic samples of models is a unique feature of SBMTE.

2.3 Specification-Based Regression Testing

Regression testing is performed after modifying a software system. The goal is capturing regression defects, which, in turn, are defects inserted due to a modification [Binder 1999], i.e. the test cases are executed to verify if the modifications caused the software to stop functioning as expected.

In short, let P be a baseline version of the program, and P' be the next version (i.e. delta version) of P. In turn, S and S' are, respectively, the baseline and delta specifications for P and P'. The test suite used to test P is referred to as T, and T' is the test suite used to test P'. Throughout this work, T and T' will be referred as *baseline test suite* and *delta test suite*, respectively. $P(t)/P'(t')$ stands for the execution of P/P' with test case $t \in T/t' \in T'$.

There are two main types of regression testing: *Corrective* and *progressive* regression testing. Corrective regression testing is applied when specification is not changed, for example when code refactoring is performed. Since the specification remains the same, test cases can be reused. On the other hand, in progressive regression testing the specification changes and new test cases must be designed (at least for new parts of the specification) [Tamimi and Zahoor 2011].

After each regression testing session is finished, the test cases used to test the delta version become a part of the regression test suite, and the cycle repeats each time a set of modifications are performed. Since modifications are performed frequently, the test suite size and the costs related to testing often increase significantly. Test case selection can be used to alleviate the costs incurred in regression testing. The goal is to select a (minimal) subset of test cases T_s that tests P'. Whenever the costs to select and execute the subset are smaller than the costs to execute all of the test cases (i.e. the *retest all* approach), the cost of regression testing has been reduced [Rothermel and Harrold 1996].

Ideally, T_s should contain only fault-revealing test cases, that are test cases that *will* reveal defects in P'. However, identifying this subset is impractical because there is not enough information to select only those test cases. Instead, a weaker criterion is considered and the goal becomes selecting all *modification-traversing test cases*, i.e. test cases exercising new or modified parts of P'/S', or test cases that formerly executed removed parts from P/S [Rothermel and Harrold 1996]. The subset of modification-traversing test cases is the closest approximation to the fault-revealing test cases that can be achieved without executing all test cases [Yoo and Harman 2012].

Our selection strategy (SART) focuses on *progressive* regression testing, where modifications are performed on a specification model and the goal is to select all test cases that exercise the parts of the system that has been modified in S' (compared to S). Two types of modifications are considered here, the *addition* and *removal* of model elements in the specification models.

As an example, consider that a client requests *removal* of use cases or scenarios from a system. This removal is reflected by a removal of transitions from a specification model. In turn, *addition* of new transitions represent, for example, new functionalities or scenarios. More complex modifications can be expressed as a combination of these two [Korel et al. 2002; Chen et al. 2007]. According to Leung and White [Leung and White 1989], the test cases for regression testing can be classified as:

- **Obsolete:** This class contains test cases that cannot be executed anymore due to an invalid input/output relationship, or for traversing a removed part of S or P.
- **Reusable:** This class comprises test cases exercising unmodified parts of the specification and their corresponding unmodified program construct. Since no modification is exercised, the same result is expected, meaning that they do not need to be executed during progressive regression testing.
- **Retestable:** This class includes test cases that exercise unmodified parts of the specification and may present a different result. An example of retestable test cases are scenarios exercising unchanged parts of S' but with new program constructs (e.g. boundary values).
- **New-structural:** This class contains structural test cases for new program constructs.

- **New-specification:** These test cases exercise the modified parts of the specification by executing new code in P'.

Distinguishing the classification of retestable and reusable test cases at system's specification level can be challenging. For models with lower levels of abstraction (e.g. control flow graphs built from a source code) the program construct can be easily accessed. However, that is very difficult to achieve with abstract test cases. Briand et al. adapted Leung and White's classification to consider UML designs in order to handle a higher level of abstraction. According to their definition, a retestable test case remains valid in terms of the sequence of messages to boundary objects but one or more of these messages may have changed (e.g. operation postcondition, signal class), whereas reusable test cases remains unchanged both in the sequence and the internal messages [Briand et al. 2009]. We consider a similar definition where retestable test cases are sequences of transitions that remain the same but at least one of the labels of those transitions has changed (i.e. no addition or removal of transitions happened, just changes in the label). In turn, unchanged sequences and labels will be considered as reusable test cases.

Similarly, classification and selection of obsolete test cases is challenging and yet very important. If executed, an obsolete test cases will fail not due to a regression defect, but due to an attempt to execute removed parts of the software system. Thus, maintenance to identify and remove these test cases from the test suite is required. Nonetheless, removals can also cause regression defects. For example, an inappropriate removal may cause the SUT to reach a state that should not be reached according to the new specification. But how can we test a removed part of the SUT? One solution is to exercise transitions (of an obsolete test case) that were not removed [Korel et al. 2002] based on the assumption that a region[2] (named *firewall*) around the removal can be defined where regression defects can be triggered.

Selected test cases can belong to any of the mentioned classes according to the selection goal. Specification-based regression test selection benefits from model-based techniques since most types of models support a high level representation of the system. Thus, specification-based strategies provide a more precise traceability between test cases and the

[2]For specification models, we consider *regions* to be a set of transitions and states within a small distance from the state where modifications were performed. For example direct predecessors and successors of the modified state.

modified requirements of the software, since test cases can be generated using a software's specification [Fahad and Nadeem 2008] as shown in previous sections of this chapter.

2.4 Similarity-Based Test Case Selection

There are various strategies to select test cases in literature. For example, the choice of a subset can be done manually by a tester based on her expertise, or even by randomly selecting test cases until the subset reaches the desired size. More sophisticated approaches can use probability values (or weights) to guide selection of important test cases [Prowell et al. 1999; Basanieri et al. 2002; Barbosa et al. 2007] or even specification of scenarios (named test purposes) to prune undesired scenarios and select a subset exercising a specific system's requirements [Jard and Jéron 2005; Nogueira et al. 2007].

Among proposed selection strategies in literature, similarity-based test case selection (STCS) has shown positive results for MBT approaches including some industrial cases [Cartaxo et al. 2011; Hemmati et al. 2013; Rogstad et al. 2013]. The goal is selecting the most diverse test cases based on the assumption that a diverse subset of test cases have a higher defect detection rate. This diversity is then obtained by similarity measurements among each pair of test cases. Considering that each test case is a vector of elements (e.g. code statements, model transitions, system conditions, etc.), similarity functions can be used to assign values determining the distance between two vectors. Consequently, close vectors indicate similar test cases. The challenge then becomes choosing appropriate similarity functions and encoding strategies for each test case [Hemmati et al. 2013].

For example, by considering each of our abstract test cases as a vector of steps from the LTS model, a similarity function can be used to determine which pair of test cases have similar steps. There are several similarity functions used in literature each with their own strengths and weaknesses, such as the Hamming distance and Jaccard index. Hemmati et al. [Hemmati et al. 2013] provide thorough description and examples for the different similarity functions used for test case selection.

The similarity function used by SART [de Oliveira Neto and Machado 2013] is an adapted version of the similarity function proposed by Cartaxo et al. [Cartaxo et al. 2011]. This function was chosen for presenting beneficial results in early evaluation with SART and with selection of test cases generated from our chosen type of specification model. Moreover, we decided to further explore the benefits of using a similarity function for selecting test cases based on modified specification models, before beginning to experiment with different similarity functions. Details regarding our similarity function will be presented when explaining SART in Chapter 3.

2.5 Experimental Studies in Software Engineering

In order to achieve reliable conclusions during research, we need to choose an appropriate method such as surveys, case studies and experiments and then evaluate our hypothesis. Surveys are used for exploring and understanding a population based on a sample. The analysis is often performed through forms, interviews and questionnaires, allowing researchers to explain and describe the population based on the sample drawn. In turn, case studies are conducted to investigate phenomenon within a specific time interval or industrial setting, hence observations and conclusions are often limited and hard to scale up or generalize [Wohlin et al. 2012].

The main difference between those empirical methods and an experiment is that the latter is based on a formal, rigorous and controlled investigation of variables, thus increasing confidence in obtained results. The starting point is to observe a cause and effect relationship (Figure 2.5) expressed through a hypothesis, thus we want to study the outcome (*dependent* variables) after changing the input variables (*independent* variables) to a process. Examples of experiments could be to investigate the effect of changing a software development process or testing technique (examples of independent variables) in the productivity rates of developers, time to release a product, or defect detection rate (examples of dependent variables).

Factors in an experiment, are one or more independent variables with varying values named *treatments* (or levels) that when changed will affect the dependent variables. Thus during an experiment factors assume different treatments while other independent variables (*objects* such as software artefacts and *subjects/participants* involved with the experiment) are controlled and then the effect of these changes are measured through the dependent variable for subsequent analysis. Experiments require a process in order to be properly conducted. Here, we use Wohlin's et al. process that comprise the following steps [Wohlin et al.

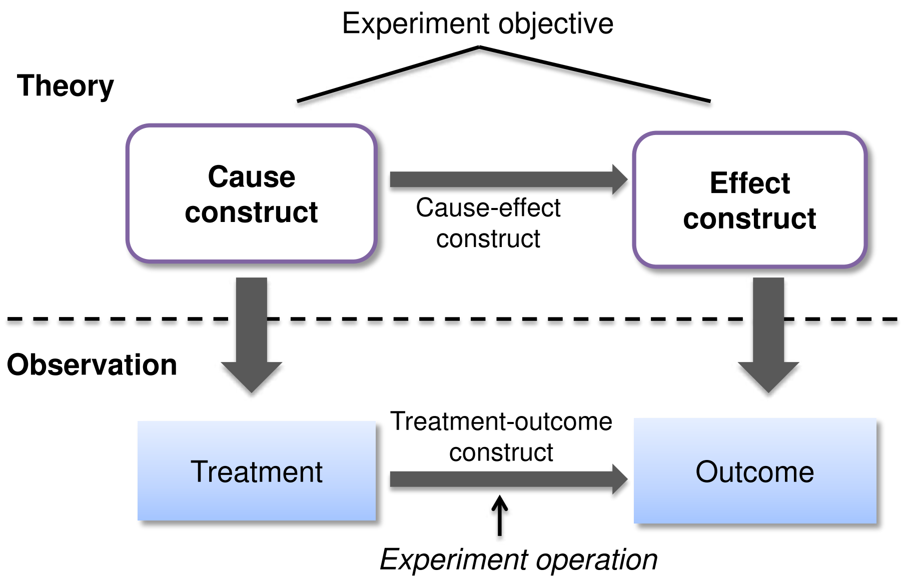


Figure 2.5: Experiment principles (adapted from Wohlin et al.[Wohlin et al. 2012])

2012]:

- **Definition:** The main concern in this step is properly defining the elements of the experiments, such as the hypothesis being investigated, context and purpose of the study.

- **Planning:** This step is the foundation of the experiment where variables and their values are defined, main activities are planned and the experimental design is determined. The latter establishes how the execution is conducted and plays a major role in enabling analysis of dependent variables with the appropriate statistical functions and tools. Also, the null and alternative hypotheses are created, and they usually represent the causation effect between treatments of a factor. Traditionally the null hypothesis indicates that the outcome is not affect by different treatments ($H_0 : \mu_a = \mu_b$) and the alternative indicate otherwise ($H_1 : \mu_a \neq \mu_b$[3]), hence the goal is often to reject the null hypothesis.

- **Operation:** Preparation and actual execution of the experiment are performed during this step, i.e. setting up tools, defining scripts and questionnaires for subjects, etc.

[3]The sign can change among the inequalities greater/less than.

- **Analysis:** The data collected during operation is organized (e.g. reduced or processed) and interpreted to draw conclusions regarding the hypothesis. Interpretation of data is mainly done by analysing descriptive statistics and perform hypothesis testing to ensure a significance level in conclusions drawn.

- **Packaging:** This step focuses on presentations and packaging of results regarding the experiment. This last activity is important since it allows other researches to reproduce the experiment based on information made available, thus it is recommended to generate research compendia with access to data, reports and the platform where the experiment was executed [González-Barahona and Robles 2012].

During this process several validity threats may appear and compromise validity of results, meaning that the results must be valid for the population being considered in order to allow generalization of results. These threats are classified as conclusion, internal, external and construct validity [Cook and Campbell 1979]. Conclusion and internal validity threats are related to the observed effect between treatment and outcome during *analysis* and *execution* respectively. For example, the former is a consequence of wrong statistical relationship and the latter is a consequence of not controlling or measuring the variables properly. In turn construct validity threats refer to properly transition from *theory to observation* where treatments and outcome indeed reflect cause and effect constructs, respectively. Last, external validity threats are concerned with *generalization* where the relationship observed during execution really implies in a general cause-effect construct relationship.

Therefore, validity threats must be identified and reported for two main reasons. First, they allow researchers to properly define what aspects of the experiment can be applied in practice in order to avoid risks or compromise the integrity of the object being studied by, for example, transferring technology to production based on wrong results. The second reason is to encourage reproducibility where, not only the threats but all information possible must be accessible, allowing other researchers to reproduce or adapt the experiment and then expand the results. Next we will discuss some basic aspects of statistical analysis to familiarize the reader with interpretation of data in an experiment.

2.6 Basic Concepts of Statistical Analysis

Statistics are powerful tools to interpret and explain data and in experiments they are used to observe the investigated cause-effect relationship. Traditionally, descriptive statistics are used to visually observe central tendencies and dispersion regarding data, and then hypothesis testing allows (or not) rejection of the experiment's null hypotheses based on a level of significance.

At early stages of analysis, interesting information regarding the data collected can be obtained through graphical representation of descriptive statistics such as mean, median, mode, variance, frequency, among others. Here we focus our discussion on arithmetic mean and variance that indicate, respectively, an estimation to the stochastic variable sampled and the dispersion of the data set around this mean [Jain 1991]. Moreover, these descriptive statistics can be plotted on commonly known types of graphic representation to allow visualization of the data set. Figure 2.6 show examples of (a) a scatter plot, (b) a histogram and (c) a boxplot.

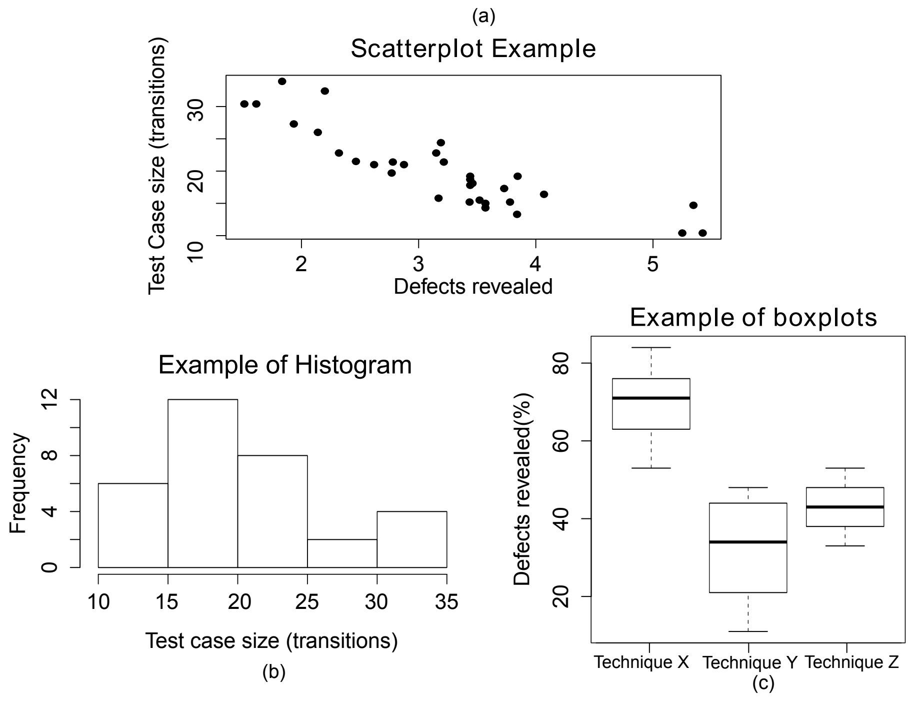


Figure 2.6: Examples of (a) scatterplot, (b) histogram and (c) boxplots.

Scatter plots are good for assessing dependence between variables and to observe outliers and data tendency whether the data is outspread or concentrated, whereas histograms consist of bars with heights representing frequency of values (or interval of values), hence providing an overview of the distribution density. In turn, boxplots are good to visualize dispersion and sample's skewness, since median and quartiles are shown.

Boxplots are also useful for comparing two or more alternatives of variables. For example, the boxplot for Technique X of Figure 2.6 (c) has a better defect detection rate than Techniques Y and Z, the same cannot be stated about Techniques Y and Z since both interval overlap, apparently indicating no statistical difference between them. Although there may not be statistical difference between them, a visual interpretation of data indicates that Technique Z is better than Technique Y since the interval of the former is tighter among higher values of defects than the latter.

That idea is related to a "*practical significance*" of data, where a calculated difference has meaningful information that affects decision making. Albeit an initial impression that overlapping intervals indicate no statistical significant difference between treatments, the experimenter needs to be aware of the practical significant difference of data, specially for experiments with software engineering where, ultimately, a stakeholder needs to decide whether a technique/method/tool should be adopted by the company, or not.

Besides visual interpretation of data and statistics, *hypothesis testing* allows researchers to verify if the null hypothesis can be rejected according to the sample distribution. Those tests are more rigorous than visual interpretation of data because a careful comparison of residuals and sample distribution is performed. There are several tests available in literature and the choice must be done carefully since the analysis depends on the experimental design used. For example, some tests do not support analysis of more than two alternatives being investigated, whereas others depend on the data distribution. The main difference begins in choosing parametric or non-parametric tests.

Parametric tests are based on models of specific distributions, thus the data must comply with assumptions of those distribution, for example being normally distributed [Jain 1991; Arcuri and Briand 2014]. Non-parametric on the other hand are less rigorous and do not make assumption regarding the data. As a consequence, non-parametric tests are more general and most of the times less powerful than parametric test. However, meeting all of a parametric test's assumptions is very difficult[4] and using the wrong test is a severe conclu-

[4]Especially in software engineering studies where new problems arise constantly and bounds or mean values are usually unknown. Besides, parametric tests assume normal distribution of data that is hardly met when

Table 2.1: Some examples of experimental designs and respective parametric and non-parametric tests.

Experimental Design	Parametric Test	Non-parametric Test
Comparing of means	Tukey	Dunn
One factor (two treatments)	*t*-test	Mann-Whitney
One factor (more than two treatments)	One-way ANOVA	Kruskal-Wallis
More than one factor	Two-way ANOVA	Friedman test

sion validity threat compromising the results [Arcuri and Briand 2014].

There is a lot of discussion in literature whether to choose parametric or non-parametric tests [Siegel and Junior 1988; Jain 1991; Arcuri and Briand 2014]. Besides the assumptions regarding the data, there are other aspects that affect that choice such as sample size, number of alternatives to compare and the purpose of the test. Table 2.1 presents a summary with several hypothesis tests and the correspondent experimental design where they can be applied.

Ultimately, the outcome of hypothesis testing is a p-value, that is compared to an established level of significance (α) to determine if the null hypothesis can be rejected. The rule of thumb is that if $p < \alpha$ then the null hypothesis can be rejected in favour of the alternative hypothesis. For example, consider that after testing hypotheses $H_0 : \text{Technique } Y = \text{Technique } Z$ and $H_1 : \text{Technique } Y \neq \text{Technique } Z$ with a Mann-Whitney test we obtained $p = 0.02 < 0.05 = \alpha$, we are allowed to claim with 95% significance level $(100\% \times (1 - \alpha))$ that Technique Y has a different effect than Technique Z on the investigated dependent variable.

investigating or comparing algorithms that are deterministic or pseudo-random.

2.7 Meta-Empirical Studies of Regression Testing Techniques

Regression test case selection is a widely researched topic, given the many possibilities of application. The community began investigating and developing different ways to evaluate these techniques, hence creating meta-empirical studies. Research on this topic seek to provide more confidence in efficiency and effectiveness of regression testing techniques by addressing cross-cutting concerns such as cost-benefit analysis and the *study of evaluation methodology*. Despite still being in early stages it is believed that its contributions will significantly help technology transfer of proposed techniques [Yoo and Harman 2012].

One of the main contributions in this field is a framework proposed by Rothermel and Harrold that has been widely used to evaluate regression test selection techniques [Rothermel and Harrold 1996] based on generality, inclusiveness, precision and efficiency. Nonetheless, the community in general needs different or complementary approaches for new issues being addressed like model-based regression testing, real-time systems or web-applications, given that these issues usually require methods and artefacts considering specific concepts and assumptions such as parallelism or non-determinism.

By clearly defining methods and artefacts used in an experiment, the results become easier to understand and reproduce [González-Barahona and Robles 2012]. For example, sharing methodologies to extract and process *datasets* can help in the comparison of results. Another example that encourages comparison of results is using the same dependent variables in different studies, like the Average Percentage of Fault Detection (APFD) used in prioritization techniques. Among the main variables that measure performance of a regression test technique, two are widely used: Rate of reduction in *size* and in *defect detection capability* [Yoo and Harman 2012].

Most of the times, assessing defect detection capability is very difficulty given that defect data is often unavailable, and without *a priori* knowledge of defects, controlled experiments are hard to perform hindering comparison of different techniques [Andrews et al. 2005]. One solution to address this issue is to use defects seeded by mutants, since studies with prioritization techniques concluded that mutation defects can be safely used when real or hand-seeded defects are not available [Do et al. 2005; Do and Rothermel 2006]. However, there is no guarantee that seeded defects are an accurate predictor of real defects.

In turn, rate of size reduction allows investigation of coverage criteria (e.g. modifications) and results from different techniques can be compared to assess the trade-off between them. Despite being an inaccurate cost measure, size reduction can be used to envision a cost reduction of the regression test. *Cost models* provide more conclusive results towards cost-effectiveness of a regression test selection technique [Leung and White 1991; Rothermel and Harrold 1997; Harrold et al. 2001b]. On the other hand, measuring costs of a regression test involves many variables that may not be available during the evaluation, such as the costs to execute the test suites, and the time needed to analyse and correct defects revealed.

Besides the dependent variables, the definition and proper usage of methods and guidelines improves on the empirical evaluation of regression test selection techniques. This is an ongoing work within the community in an attempt to overcome the current lack of empirical evaluation [Engström et al. 2008; Yoo and Harman 2012]. Eventually, methods and variables can be shared so that researchers are able to reach more confident conclusions and eventually reproduce, compare and expand existing results.

2.8 Concluding Remarks

This chapter covered the fundamental aspects of the research. Most of the discussion was focused on regression testing, test case selection and experimental studies in software engineering. Another fundamental concept used in our work is evaluation of model-based techniques through stochastic model generation [de Oliveira Neto et al. 2013], however that discussion will be found in Chapter 5 since it is also a result of this doctorate research.

Chapter 3

Similarity Approach for Regression Testing

Similarity Approach for Regression Testing (SART) is a test case selection technique to automatically identify and select test cases exercising new, modified, or affected parts of the specification model. In summary, SART compares two sets of test cases from a baseline and a delta version of the specification model. Since test cases are described through scenarios (i.e. sequences of transitions from the model), comparing the similarities enables testers to identify changes in the model. The idea is that *very similar sets* of test cases between different versions indicate that *little modifications* were performed on the model, whereas very different sets (i.e. less similar) indicate that severe modifications were performed to a point where the sequences of transitions have significantly changed.

Usage of our selection technique alone on a pre-defined set of test cases allows automatic selection of the desired subset, but when combined with automatic test case generation, the technique becomes even more powerful since comparison between test cases covering all paths can be performed automatically. In its original version SART performed test suite minimisation considering as test requirement the modifications of the model. Therefore, the reduced subset would *only* contain test cases traversing a modification. Case studies performed with the technique revealed promising results regarding the technique's percentage of size reduction [de Oliveira Neto 2010; de Oliveira Neto and Machado 2011]. However, due to transition coverage redundancy in the reduced subset, the same set of defects were being triggered lowering the defect detection rate [de Oliveira Neto and Machado 2013].

The development and improvement of the technique have continued during this doctorate research. Before presenting details regarding SART's execution, we present how the technique can be used in an MBT related test process (Figure 3.1).

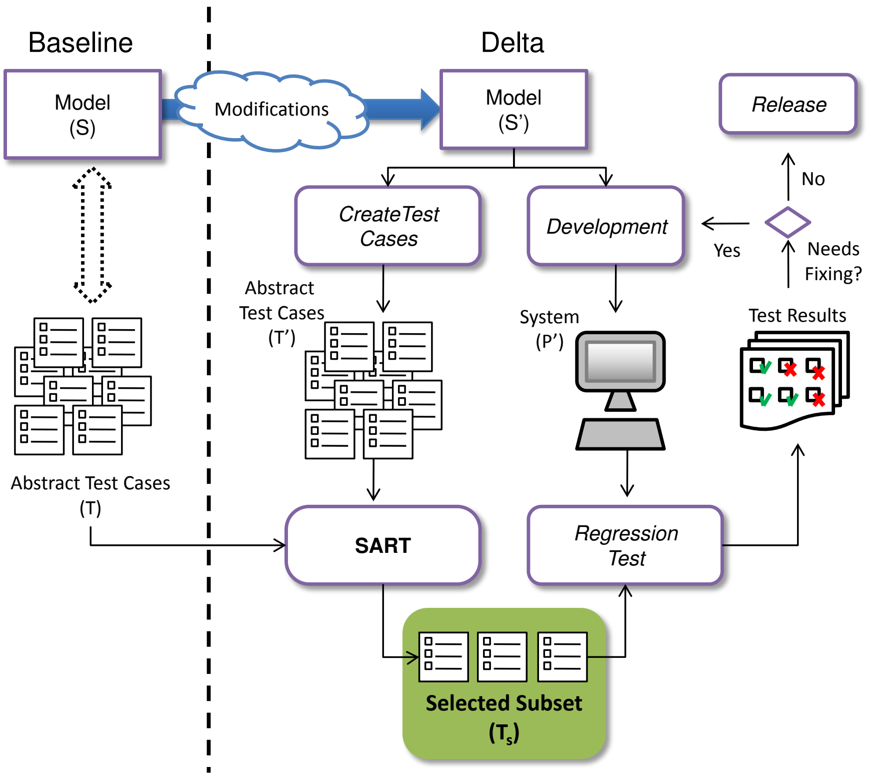
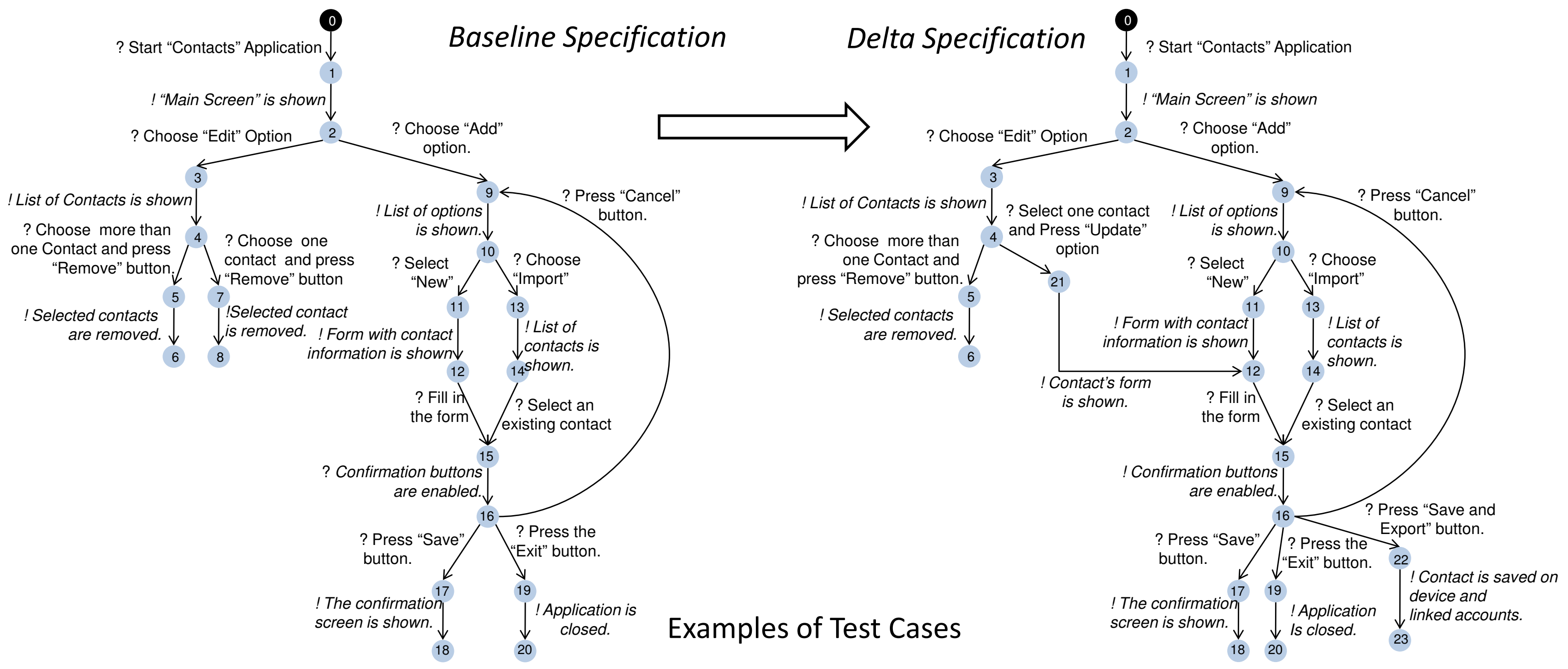


Figure 3.1: Example of a test process suitable for SART.

After changing the functionalities of the system, a new version of the specification is defined, hence a new specification model is obtained. In an MBT context, we assume that there are techniques (either manual or automatic) for creating test cases from the specification model, and since we target high level specification models we provide as input for SART sets of abstract test cases. Usually, these test suites tend to be big and redundant [Jorgensen 2002; Fraser and Wotawa 2007], specially for complex and large system models. Assuming that the resources (e.g. time, budget) are insufficient to execute the entire test suite, the tester needs to select a subset of test cases able to test the delta version of the software system and find the regression defects. In order to illustrate how SART selects test cases, we use an illustrative example.

3.1 Example

The example is a use case specification for a simple *contact list* application from a mobile phone. Figure 3.2 presents an ALTS model and examples of test cases generated from this model. The use case has two scenarios: Add or edit a contact. Editing allows removal of one or several contacts, whilst a new contact can be added by inserting the contact's information or to import it from a different source (e.g. an e-mail contact, or social media database).


Baseline Specification
Delta Specification
? Start "Contacts" Application
! "Main Screen" is shown
? Choose "Edit" Option
? Choose "Add" option.
! List of Contacts is shown
? Choose more than one Contact and press "Remove" button.
? Choose one contact and press "Remove" button
! Selected contacts are removed.
!Selected contact is removed.
! List of options is shown.
? Press "Cancel" button.
? Select "New"
? Choose "Import"
! Form with contact information is shown
! List of contacts is shown.
? Fill in the form
? Select an existing contact
? Confirmation buttons are enabled.
? Press "Save" button.
? Press the "Exit" button.
! The confirmation screen is shown.
! Application is closed.
? Select one contact and Press "Update" option
! Contact's form is shown.
! Confirmation buttons are enabled.
? Press "Save and Export" button.
! Contact is saved on device and linked accounts.
! Application Is closed.
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Examples of Test Cases

Baseline Version

User Actions	Expected Outputs
Start "Contacts" Application	*"Main Screen" is shown*
Choose "Edit" Option	*List of Contacts is shown*
Choose one contact and press Remove" button	*Selected contact Is removed.*

User Actions	Expected Outputs
Start "Contacts" Application	*"Main Screen" is shown*
Choose "Add"option.	*List of options is shown.*
Select "New"	*Form with contact information is shown*
Fill in the form.	*Confirmation buttons are enabled*
Press "Cancel" button.	*List of options is shown.*
Choose "Import"	*List of contacts is shown.*
Select an existing contact	*Confirmation buttons are enabled*
Press the "Exit" button.	*Application is closed.*

Delta Version

User Actions	Expected Outputs
Start "Contacts" Application	*"Main Screen" is shown*
Choose "Edit" Option	*List of Contacts is shown*
Select one contact and Press "Update" option	*Contact's form is shown*
Fill in the form.	*Confirmation buttons are enabled*
Press "Save" button.	*The confirmation screen is shown*

User Actions	Expected Outputs
Start "Contacts" Application	*"Main Screen" is shown*
Choose "Add"option.	*List of options is shown.*
Select "New"	*Form with contact information is shown*
Fill in the form.	*Confirmation buttons are enabled*
Press "Cancel" button.	*List of options is shown.*
Select "New"	*Form with contact information is shown*
Fill in the form.	*Confirmation buttons are enabled*
Press "Save and Export" button.	*Contact is saved on device and linked accounts.*

Figure 3.2: Examples of ALTS specification models and test cases.

Eventually, the specification is changed to incorporate three modifications: (1) The scenario for deleting only one contact has been removed. (2) An option to update a contact's information is added. (3) Export a contact's information to a different contact list (e.g. e-mail). These modifications respectively reflect on the model as following:

- Removal of transitions: (4, *"Choose one contact and press 'Remove' button"*,7) and (7, *"Selected Contacts are Removed"*,8);
- Addition of transitions: (4, *"Select one contact and press 'Update' option."*,21) and (21, *"Contact's form is shown."*,12);
- Addition of transitions: (16, *"Press 'Save and Export' buttons'."*,22) and (22, *"Contact is saved on device and linked accounts"*.,23).

Based on Korel's et al. description of interaction patterns from modifications [Korel et al. 2002; Chen et al. 2007], we consider two situations where regression defects can be triggered: First, the modified element itself can affect software behaviour, second, any behaviour (states and transitions) specified near a modification can be affected as a side-effect from modifications. Since modifications can affect states, we assume that branching states[1] are sensitive to these modifications because a defect on that branch state can cause the system to reach a different, unexpected state. Thus, the system will not produce the correspondent output for the performed user action.

In order to clarify our concept of a "modified region"[2] (Figure 3.3), consider that the addition of the "Update contact" functionality caused a defect on the functionality of "Removing a contact". During our test execution, the user action *"Select one contact and Press "Update" option"* is performed and since the functionality was successfully implemented, the test case passes. Now imagine that, when executing the next test case, the tester performs the action "Choose more than one contact and press the "Remove" button" and when checking the produced output she finds out that the contact was not removed, thus signalling a failure.

Therefore, to address these *side-effects*, we consider that *regions near modifications* comprise the modified model elements themselves and the *steps* from the same level of the mod-

[1]We refer to branching states as states with more than one outgoing transitions.

[2]States and transitions from the ALTS.

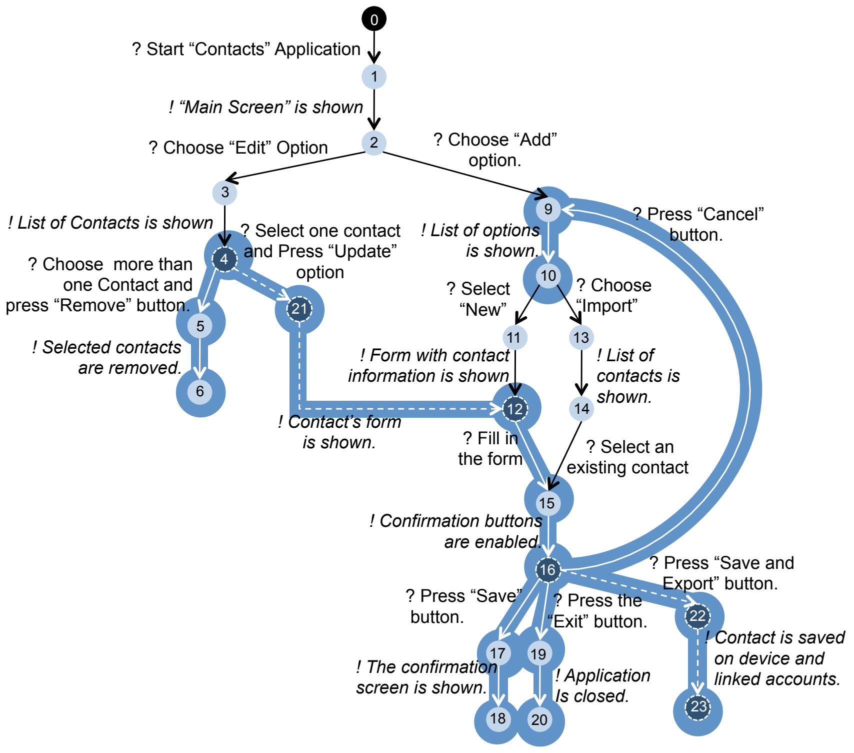

0
? Start "Contacts" Application
1
! "Main Screen" is shown
2
? Choose "Edit" Option
? Choose "Add" option.
3
! List of Contacts is shown
4
? Select one contact and Press "Update" option
? Choose more than one Contact and press "Remove" button.
5
! Selected contacts are removed.
6
21
! Contact's form is shown.
9
? Press "Cancel" button.
! List of options is shown.
10
? Select "New"
? Choose "Import"
11
13
! Form with contact information is shown
! List of contacts is shown.
12
14
? Fill in the form
? Select an existing contact
15
! Confirmation buttons are enabled.
16
? Press "Save and Export" button.
? Press "Save" button.
? Press the "Exit" button.
22
! Contact is saved on device and linked accounts.
17
19
! The confirmation screen is shown.
! Application Is closed.
18
20
23

Figure 3.3: Examples of regions and model elements affected by model's modifications.

ified element[3]. For instance the dark background highlighting states and transitions (Figure 3.3) represent the regions of the model affected by the three modifications performed. The dotted white transitions and dark states are the modified elements, whereas we assume that the white solid transitions can suffer side-effects from the performed modifications.

Based on the LTS definition presented in Chapter 2, Section 2.1 the following definition is presented regarding modified states, transitions and regions of the LTS. Let S and S' be the baseline and delta version of the LTS model, hence T_{tr}, Q, L and T'_{tr}, Q', L' is respectively, the set of transitions, states and labels from S and S'. Consider that $q_m \in (Q \cup Q')$ is a modified state, and a modified transitions ($\overrightarrow{t}_m$) can either belong to the set of added ($T_{tr:add}$) or removed ($T_{tr:rem}$) transitions. Therefore:

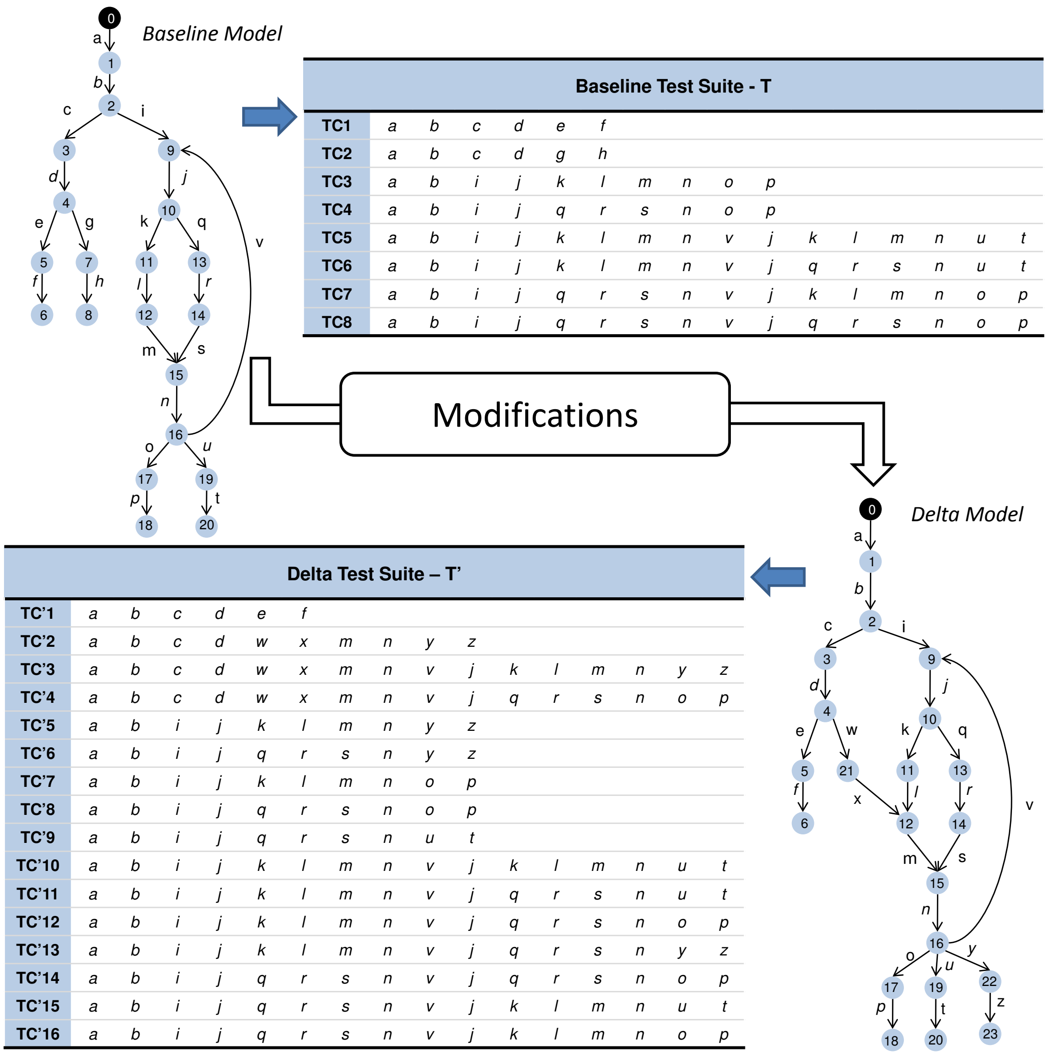
- $\overrightarrow{t}_m \in T_{tr:add} \iff \overrightarrow{t}_m \notin T_{tr} \wedge \overrightarrow{t}_m \in T'_{tr}$;
- $\overrightarrow{t}_m \in T_{tr:rem} \iff \overrightarrow{t}_m \in T_{tr} \wedge \overrightarrow{t}_m \notin T'_{tr}$;
- In order to define the set $T_{tr:reg}$ of affected regions[4] in S', consider that:

[3]The level is the longest distance between the current and the initial state.

[4]To keep the explanation simple, we decided to consider only the set of affected transitions, since affected

- $\forall \overrightarrow{t}_1, \overrightarrow{t}_2 \in T'_{tr}, \exists q_1, q_2 \in Q', \exists l_a, l_b \in L'$;
- $T_{tr:reg} = \{\overrightarrow{t}_1, \overrightarrow{t}_2 \mid \overrightarrow{t}_1 = (q_m, l_a, q_1), \overrightarrow{t}_2 = (q_1, l_b, q_2)\}$

This concept of modified and affected elements will be used to explain our selection strategy. In order to simplify the technique's step by step execution, we will change the transition's labels, generating a more compact version of the model. A summary of the test suite (defined manually by traversing the LTS models) and the model is presented in Figure 3.4.



Baseline Test Suite - T																
TC1	a	b	c	d	e	f										
TC2	a	b	c	d	g	h										
TC3	a	b	i	j	k	l	m	n	o	p						
TC4	a	b	i	j	q	r	s	n	o	p						
TC5	a	b	i	j	k	l	m	n	v	j	k	l	m	n	u	t
TC6	a	b	i	j	k	l	m	n	v	j	q	r	s	n	u	t
TC7	a	b	i	j	q	r	s	n	v	j	k	l	m	n	o	p
TC8	a	b	i	j	q	r	s	n	v	j	q	r	s	n	o	p

Delta Test Suite – T'																
TC'1	a	b	c	d	e	f										
TC'2	a	b	c	d	w	x	m	n	y	z						
TC'3	a	b	c	d	w	x	m	n	v	j	k	l	m	n	y	z
TC'4	a	b	c	d	w	x	m	n	v	j	q	r	s	n	o	p
TC'5	a	b	i	j	k	l	m	n	y	z						
TC'6	a	b	i	j	q	r	s	n	y	z						
TC'7	a	b	i	j	k	l	m	n	o	p						
TC'8	a	b	i	j	q	r	s	n	o	p						
TC'9	a	b	i	j	q	r	s	n	u	t						
TC'10	a	b	i	j	k	l	m	n	v	j	k	l	m	n	u	t
TC'11	a	b	i	j	k	l	m	n	v	j	q	r	s	n	u	t
TC'12	a	b	i	j	k	l	m	n	v	j	q	r	s	n	o	p
TC'13	a	b	i	j	k	l	m	n	v	j	q	r	s	n	y	z
TC'14	a	b	i	j	q	r	s	n	v	j	q	r	s	n	o	p
TC'15	a	b	i	j	q	r	s	n	v	j	k	l	m	n	u	t
TC'16	a	b	i	j	q	r	s	n	v	j	k	l	m	n	o	p

Figure 3.4: Compact version of the specification model, and test cases obtained from the respective models.

states can be found through each affected transition.

3.2 SART's Selection Strategy

By selecting and executing only a subset of representative test cases, we alleviate some of the cost issues when money, time or personnel are limited for testing the software system. For our regression testing context, a representative subset is the one containing test cases that exercise the modified parts of the model. Initially, our selection strategy uses a similarity function to identify the test cases exercising the modifications themselves. Then, we apply test suite minimisation techniques to minimise the set of transitions being covered by test cases and remove unnecessary transition redundancy. Last we add test cases to our subset to increase transitions coverage and complement the modifications coverage achieved in the first step. Figure 3.5 presents an overview of SART's selection strategy that will be next applied in our example.

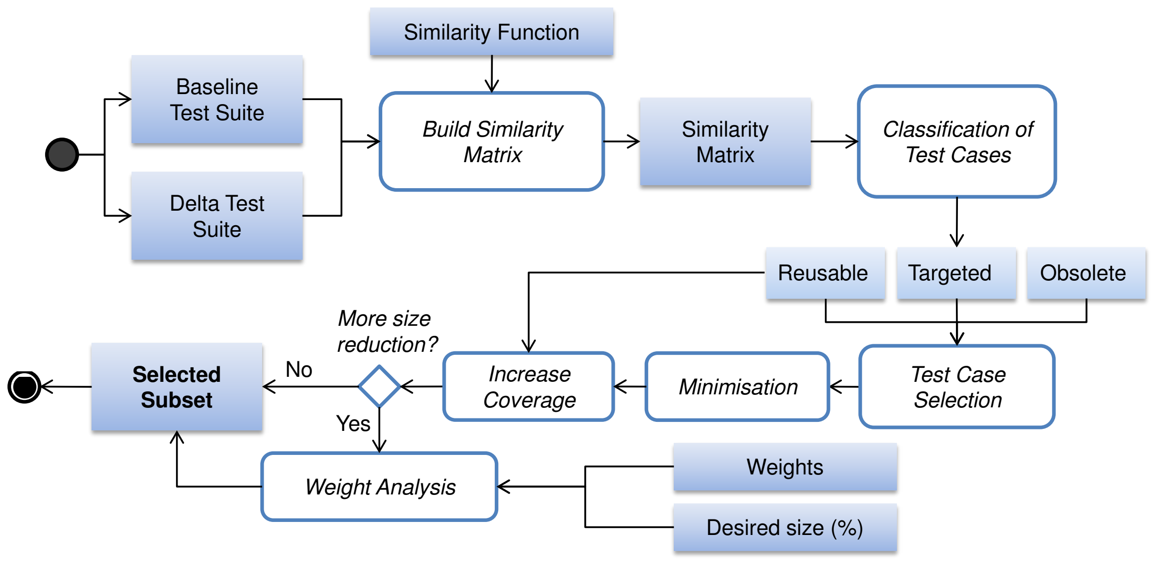


Figure 3.5: SART's selection process.

The input for SART are T and T', and the output is $T_s \subseteq T'$, hence no obsolete test cases are selected removing the need for test suite maintenance to identify and remove outdated test cases. The first step is to build the *similarity matrix*, which contains information between all pairs of test cases $(t_j, t'_i) \mid t_j \subseteq T, t'_i \subseteq T'$. The baseline test cases are placed in the columns of the matrix, while delta test cases are placed in the rows. Each position $a[i, j]$ of the matrix is filled with the similarity values calculated through Equations 3.1 and 3.2.

$$a[i,j] = \text{Similarity}(t'_i, t_j) = \frac{nit(t'_i, t_j)}{\text{AvgSize}(t'_i, t_j)} \tag{3.1}$$

$$\text{AvgSize}(t'_i, t_j) = \frac{|t'_i| + |t_j|}{2}. \tag{3.2}$$

The function *nit* counts the number of identical transitions between a test case from T and T'. Here, identical transitions is a pair of transitions with the same source and sink state, and the same label. In other words, let $\overrightarrow{t}_a = (s_{a1}, l_a, s_{a2}) \in T_{tr}$ and $\overrightarrow{t}_b = (s_{b1}, l_b, s_{b2}) \in T'_{tr}$. Then, $\overrightarrow{t}_a = \overrightarrow{t}_b \iff s_{a1} = s_{b1} \wedge l_a = l_b \wedge s_{a2} = s_{b2}$[5]. The number of identical transitions is then divided by an average of sizes (i.e. number of transitions) in order to normalize the ratings among all similarity values. As an example, we show calculation of the similarity value between $TC'6$ and $TC4$.

$$\begin{aligned}
AvgSize(TC'6, TC4) &= \frac{|TC'6| + |TC4|}{2} = \frac{10+10}{2} = 10; \\
a[6,4] &= \frac{nit(TC'6, TC4)}{AvgSize(TC'6, TC4)} \\
&= \frac{|[\overrightarrow{a}\,\overrightarrow{b}\,\overrightarrow{i}\,\overrightarrow{j}\,\overrightarrow{q}\,\overrightarrow{r}\,\overrightarrow{s}\,\overrightarrow{n}\,\overrightarrow{y}\,\overrightarrow{z}] \cap [\overrightarrow{a}\,\overrightarrow{b}\,\overrightarrow{i}\,\overrightarrow{j}\,\overrightarrow{q}\,\overrightarrow{r}\,\overrightarrow{s}\,\overrightarrow{n}\,\overrightarrow{o}\,\overrightarrow{p}]|}{10} \\
&= \frac{|[\overrightarrow{a}\,\overrightarrow{b}\,\overrightarrow{i}\,\overrightarrow{j}\,\overrightarrow{q}\,\overrightarrow{r}\,\overrightarrow{s}\,\overrightarrow{n}]|}{10} = \frac{8}{10} = 0.80;
\end{aligned}$$

The resulting value is then placed in the respective row and column of the matrix, thus 0.8 is placed at row 6, column 4 of the matrix. Furthermore, the similarity value "1" indicates that an identical sequence is found in both test suites. Therefore, all transitions are the same and no modification is exercised, being one candidate to be removed from the test suite (that can be seen by calculating the similarity between $TC1$ and $TC'1$). For the example considered, the remaining similarity values were calculated, resulting in the matrix of Table 3.2.

[5]The currently implemented version of SART considers a pair of states to be equal if they share the same label. For example, transitions $(4, g, 7)$ and $(4, w, 21)$ have the same source state (4), but different labels and sink states. Thus, in our technique, *they are not identical transitions*.

Table 3.1: Similarity matrix from the test suites in Figure 3.4.

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
TC'1	1	0.667	0.250	0.250	0.182	0.182	0.182	0.182
TC'2	0.500	0.500	0.400	0.300	0.308	0.308	0.231	0.231
TC'3	0.545	0.545	0.769	0.231	0.625	0.625	0.563	0.375
TC'4	0.364	0.364	0.538	0.692	0.438	0.625	0.688	0.688
TC'5	0.500	0.500	0.500	0.700	0.769	0.615	0.769	0.385
TC'6	0.250	0.250	0.500	0.800	0.385	0.615	0.615	0.615
TC'7	0.250	0.250	1	0.700	0.615	0.615	0.769	0.538
TC'8	0.250	0.250	0.700	1	0.385	0.615	0.769	0.769
TC'9	0.250	0.250	0.500	0.800	0.538	0.769	0.615	0.615
TC'10	0.182	0.182	0.615	0.385	1	0.813	0.688	0.500
TC'11	0.182	0.182	0.615	0.385	0.813	1	0.875	0.688
TC'12	0.182	0.182	0.769	0.769	0.688	0.875	1	0.813
TC'13	0.182	0.182	0.615	0.615	0.688	0.875	0.875	0.688
TC'14	0.182	0.182	0.538	0.769	0.500	0.688	0.813	1
TC'15	0.182	0.182	0.615	0.615	0.750	1	0.875	0.688
TC'16	0.182	0.182	0.769	0.769	0.688	0.875	1	0.813

The next step is to analyse the similarity values and classify test cases as:

- *Obsolete:* Identified through *columns* that present not a single similarity value of 1.
  - $T_{obs} = \{TC2\}$;
- *Reusable: Rows* containing at least one similarity value of 1 indicate unchanged sequences of transitions already tested in a previous version, thus a reusable test case.
  - $T_{reus} = \{TC'1, TC'7, TC'8, TC'10, TC'11, TC'12, TC'14, TC'15, TC'16\}$;
- *Targeted Test Cases:* Contains both *new specification* and test cases that were not executed before. They can be identified through *rows that do not have* a similarity value of 1.
  - $T_{targ} = \{TC'2, TC'3, TC'4, TC'5, TC'6, TC'9, TC'13\}$;

After the classification is concluded, we select test cases that exercise added (targeted test cases) and removed (obsolete) parts of the specification model. Note that an obsolete test case cannot be executed on the SUT, hence SART selects delta test cases very similar to obsolete test cases. This enables execution of similar sequences of paths where a removal has occurred, increasing the chances of revealing a regression defect caused by such a modification [Korel et al. 2002].

First, the delta test cases more similar to each respective obsolete test case are added to the subset. In this example, there is only one obsolete test case ($TC2$), thus, the highest similarity value of the respective column is obtained (0.667), resulting in the selection of $TC'1$. Notice that $TC'1$ exercises a very similar sequence to $TC2$ (both in the exercised transitions and size), specially since $TC'1$ also traverses state 4, where a transition's removal occurred.

Next, we add all targeted test cases to a subset resulting in: $T_{aux} = \{TC'1, TC'2, TC'3, TC'4, TC'5, TC'6, TC'9, TC'13\}$. As can be seen all modifications have been covered, but several test cases repeatedly cover the same transitions, hence there is still a lot of redundancy among covered transitions. The solution is applying minimisation techniques to select a reduced set of test cases covering all transitions of our current subset. There has been extensive research on usage of heuristics for test suite minimisation [Bertolino et al. 2010; Hemmati et al. 2011].

The H heuristic [Harrold et al. 1993] was chosen for our minimisation step because it showed good results[6] for revealing defects in an MBT process similar to ours [Cartaxo 2011]. The technique is to first define a cardinality table where each cardinality corresponds to the number of test cases covering a specific test requirements (TR), or in our case, a single transition from the subset. Then, the test cases covering the lowest cardinality TR are included in the reduced subset to ensure coverage of requirements being covered only by a specific test case (named *essential test case*). As test cases are included, all the respectively covered TR are marked.

After defining the traceability and cardinality tables (Table 3.2), we include the test cases covering more requirements from each cardinality set until all requirements are marked, i.e. the reduced subset covers all TR. If there is a tie among the test cases, the next cardinality is examined and so on. From our example, we begin with an empty reduced subset T_r and then investigate cardinality 1 for requirements $\overrightarrow{u}, \overrightarrow{t}, \overrightarrow{o}, \overrightarrow{p}, \overrightarrow{e}, \overrightarrow{f}$. The test cases covering TR at this cardinality are $TC'1, TC'4, TC'9$, resulting in addition of

[6]In a case study, Cartaxo showed that the H heuristic reveals more defects when compared to the Greedy (G), Greedy-Essential (GE) , and Greedy-$1to-1$ Redundancy-Essential (GRE) heuristics.

Table 3.2: (a) Traceability between test requirements and test cases, and (b) cardinality of each test requirement.

(a)

TR	Test Cases								Number of Test Cases
	TC'1	TC'2	TC'3	TC'4	TC'5	TC'6	TC'9	TC'13	
a	x	x	x	x	x	x	x	x	8
b	x	x	x	x	x	x	x	x	8
c	x	x	x	x					4
d	x	x	x	x					4
e	x								1
f	x								1
i					x	x	x	x	4
j			x	x	x	x	x	x	6
k			x		x			x	3
l			x		x			x	3
m		x	x	x	x			x	5
n		x	x	x	x	x	x	x	7
o				x					1
p				x					1
q				x		x	x	x	4
r				x		x	x	x	4
s				x		x	x	x	4
t							x		1
u							x		1
v			x	x				x	3
w		x	x	x					3
x		x	x	x					3
y		x	x		x	x		x	5
z		x	x		x	x		x	5
	6	10	14	15	10	10	10	14	

(b)

Cardinality	Test Requirements
1	e, f, o, p, t, u
3	k, l, v, w, x
4	c, d, i, q, r, s
5	m, y, z
6	j
7	n
8	a, b

$TC'4$ for covering more TR among them (last row of Table 3.2). Consequently, TRs $\vec{a}, \vec{b}, \vec{c}, \vec{d}, \vec{w}, \vec{x}, \vec{m}, \vec{n}, \vec{v}, \vec{j}, \vec{q}, \vec{r}, \vec{s}, \vec{n}, \vec{o}, \vec{p}$ are all marked as covered. Continuing with unmarked TRs at cardinality 1 test cases $TC'9$ and $TC'1$ are added to the reduced subset. The next cardinality is 3 with unmarked TRs $\vec{k}, \vec{l}$ resulting in choice of $TC'13$. After this, all TRs become marked concluding our minimisation stage with subset: $T_r = \{TC'4, TC'1, TC'9, TC'13\}$.

At this point all transitions of the subset are covered with half the number of test cases. However, some regression defects may be revealed only through interaction of transitions or, as mentioned earlier, triggered as side-effects from nearby modifications. In order to cover the side-effect regions, we fill the gaps left from removing redundant test cases with reusable test cases similar to our reduced subset. By keeping a constant similarity analysis, we ensure that our test cases are still *near* the modifications, even if not covering the modifications themselves.

The technique proceeds by calculating a new similarity matrix (Table 3.3) between T_r (rows) and T_{reus} (columns). Then, we search for the highest similarity value in each row (random choice is used for tie breaks) and then add the respective column to our final sub-

Table 3.3: Similarity matrix for the reduced subset and the reusable test cases.

	TC'7	TC'8	TC'10	TC'11	TC'12	TC'14	TC'15	TC'16
TC'1	0.25	0.25	0.18	0.18	0.18	0.18	0.18	0.18
TC'4	0.46	0.61	0.43	0.62	0.75	0.68	0.62	0.56
TC'9	0.5	0.8	0.53	0.76	0.61	0.61	0.76	0.61
TC'13	0.61	0.61	0.68	0.87	0.87	0.68	0.87	0.87

set followed by removal of that column from our new matrix in order to avoid repetitive selection of the same set of similarity values. From Table 3.3 we begin at row $TC'1$ by finding a tie (0.25) between $TC'7$ and $TC'8$, resulting in (random) selection and removal of column $TC'8$. We proceed with analysis of $TC'4, TC'9, TC'13$ resulting in selection of $TC'12, TC'15, TC'11$ respectively. At this point the size limit is reached and SART's output for our example is: $T_s = \{TC'1, TC'4, TC'8, TC'9, TC'11, TC'12, TC'13, TC'15\}$. If there were more slots to fill, the technique would return to the first row and repeat the process, until the gaps are filled or all reusable test cases are removed from the matrix. As can be seen both the modifications and regions shown in Figure 3.3 are being exercised by our selected subset, increasing the chances of revealing regression defects.

3.3 Concluding Remarks

This chapter discussed our selection strategy, and the steps for executing SART through an illustrative example. The main benefit of SART is providing automatic selection of test cases close to the specification level, instead of investigating test cases at the source code level where programming language skills and knowledge of the system's components and subsystems is required. Using this black-box approach enables, for example, stakeholders to present test cases for clients as part of an acceptance test. The major drawback, on the other hand, is requiring manual execution, albeit existing research can mitigate that problem, for example, by creating model transformations to provide executable code from high level models or to create TTCN[7] (Testing and Test Control Notation) test cases.

Furthermore, the simplicity of our similarity function provides versatility because it can be easily adapted to support different types of models such as activity diagrams or finite state

[7] `http://www.etsi.org/technologies-clusters/technologies/testing/ttcn-3`

machines, hence making our approach more independent of a model type. For example, the number of identical transitions could be used to count the number of identical activities, or guards in conditions, or messages in a sequence diagram and so forth. The classification of test cases can also alleviate maintenance of the regression test suite by helping the tester to remove obsolete test cases.

SART has come a long way since it was introduced as WSA-RT. Our similarity-based strategy has gone through major changes in order to improve in percentage of size reduction and transitions coverage, the latter being an attempt to improve defect detection rate. Unfortunately, conclusions regarding defect detection rate are hard to obtain due to limitations in evaluation methodologies, such as the lack of defect data or difficulties in tracing code defects to the model's elements. As a consequence measuring defects is challenging, and instead we are limited to measuring the number of test cases that failed[8].

Nonetheless, the evaluation strategy also proposed in this doctorate research addresses some of these limitations and allowed us to draw conclusions about usage of STCS to identify and select modification-traversing test cases. This evaluation methodology is based on stochastic generation of specification models presented in Chapter 5, but before that we will discuss some other proposed work for specification-based regression test selection and how they differ from SART.

[8]A failure can be caused by one or more defects, making it hard to precisely identify and correct each of those defects.

Chapter 4

Review on Test Case Selection for Regression Testing

Test case selection for regression testing is a widely researched topic in literature, resulting in creation of a variety of techniques, each with their own solution to select a subset of test cases, for example genetic algorithms, adaptive random selection, search-based techniques, and most of those techniques rely on artefacts from source code level [Korel et al. 2002]. Actually, the techniques were first proposed for procedural programs [Leung and White 1990; Gupta et al. 1996; Vokolos and Frankl 1997], followed by object-oriented programs [Hsia et al. 1997; Rothermel et al. 2000; Harrold et al. 2001a], but recent techniques explore several other aspects of software such as components [Zheng et al. 2006; Mao et al. 2007], database [Willmor and Embury 2005], web-services [Xu et al. 2003; Tarhini et al. 2006] and, among others, information from software models [Rothermel et al. 2000; Chen et al. 2007; Naslavsky et al. 2010].

Moreover, usage of models is not restricted only to a system's specification level. Some techniques use models to analyse source code [Wu et al. 1999; Rothermel et al. 2000; Harrold et al. 2001a; Ren et al. 2004; White et al. 2008; Mansour and Statieh 2009] and perform test case selection based on changed classes, methods, etc. Even though these techniques use a model-based approach they are not related to our research because they are mainly for white-box approaches, whereas we focus on black-box. This chapter will discuss some related work for specification-based test case selection, beginning with general aspects for selecting test cases for regression testing, followed by a section detailing some techniques.

4.1 Selection Strategies

There are several ways to select test cases, each with its own benefits and drawbacks. A strategy widely used for test case selection in general is a *random selection*. The subset is chosen by randomly adding test cases until the subset size meets the resources constraints for performing the test [Graves et al. 2001; Cartaxo et al. 2011]. In addition, some experienced testers may use their own expertise or knowledge about the SUT to select a proper set of test cases. In either cases, these strategies are risky for not relying on a formal criterion able to express representativeness of a selected subset, resulting in a proposal of more sophisticated approaches targeting specific coverage criteria.

One criterion widely used for selecting regression test cases is modification coverage through selection of *modification-traversing test cases*, since they are more likely to reveal regression defects [Rothermel and Harrold 1996; Yoo and Harman 2012]. Therefore, selection strategies developed to identify and analyse aspects of a modification can increase the defect detection capability of the selected subset, leading to one of the main strategies for selecting regression testing: *Identifying modifications*. After identifying the modified elements of a software system we select the test cases exercising these elements.

In order to identify what has been modified, information regarding different software versions is required, because then a 'simple' way to identify modified entities is comparing both versions (baseline and delta). On the other hand, this comparison can be costly depending on a software's complexity and size. One of the first techniques with that strategy was proposed by Laski and Szermer [Laski and Szermer 1992]. The goal was to compare Control Flow Graphs obtained from different versions of a source code and then identify subgraphs comprising the modified transitions and states, that in turn are mapped to code statements.

Several more recent techniques select regression test cases by identifying modifications [Sajeev and Wibowo 2003; Liang 2005; Gao et al. 2006; Naslavsky and Richardson 2007; Muccini 2007; Farooq et al. 2007; Gorthi et al. 2008; Farooq et al. 2010]; however, not all modifications are handled by these techniques. For example, some are unable to identify removed elements or more complex modifications (e.g. to replace an architectural module, or change a complex component of the software).

As mentioned before, some regression defects may be found on unmodified parts of software, triggered as side effect of a modification, such as software parts dependent of a modified element (e.g. methods, classes or transitions in a model). Therefore, identifying modifications may not be enough to cover all regression defects leading then to more sophisticated selection strategies where *dependence analysis* is required.

In dependence analysis of specification models [Chung et al. 1999; Orso et al. 2001; Korel et al. 2002; Orso et al. 2004; Chen et al. 2007], all of the model elements are investigated to identify their correspondent dependencies. Thus, if any of those elements are modified the dependencies are marked as affected leading to selection of test cases traversing these elements. Consequently, these techniques tend to be costly in practice, or limited by constraints (e.g. system's size) because complex software systems have several components often dependent among them and being able to analyse all these components and the possibilities of interactions require a lot of time and effort. On the other hand, subsets selected through dependence analysis provide more confidence because they usually increase the chances of finding regions sensitive to side effects of modifications.

In order to reduce the cost of this analysis, some techniques [Wu and Offutt 2003; Mao and Lu 2005; Chittimalli and Harrold 2008; Pasala et al. 2008; Subramaniam et al. 2009; Naslavsky et al. 2009; Naslavsky et al. 2010] perform a simpler *analysis of the models*, by defining boundaries where the modification can reach other parts of the software system. This analysis can be combined with the 'modification identification' strategy so that smaller regions of modified components are analysed (instead of analysing the entire system and test suite). Therefore, identifying modifications is the more common approach and it is usually combined with other strategies (e.g. genetic algorithms, value-based approaches or model checking) to increase the confidence of the selected subset.

Among these selection strategies, SART identifies modifications and analyses similarities between test cases to increase coverage of transitions attempting to cover modified parts of the specification model. In order to illustrate the differences between those three approaches, we present some techniques well known in literature.

4.1.1 Cluster-Based Selection

Laski and Szermer proposed a technique to automatically identify modifications on Control Flow Graphs (CFGs) obtained from baseline and delta versions of a source code [Laski and Szermer 1992]. Initially, the technique (referred in this work as the "*Cluster*" technique) identifies subgraphs, named clusters, on both CGFs and then begins traversing and marking the states that do not match. Then, the model elements belonging to a marked cluster are also marked as affected and all test cases exercising them are selected.

This idea has been adapted for specification models [Chen et al. 2002; de Oliveira Neto 2010] where the CFG would represent scenarios of a use case. As an example, consider the CFGs from Figures 4.1 (a) and (b) (baseline and delta respectively). Three modifications were performed in this example: The addition of a transition between states 16 and 12; the removal of the highlighted transition (between states 8 and 10); and the removal of state 11.

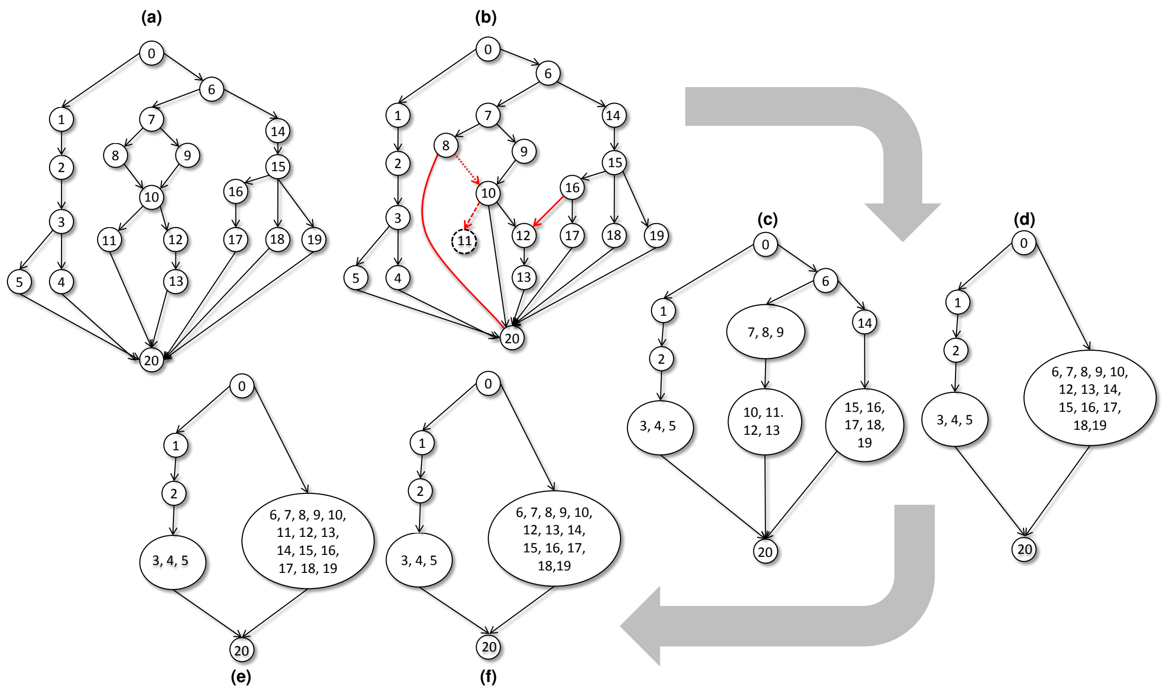


Figure 4.1: Examples of CFGs and clusters obtained from a baseline, (a), (c) and (e), and a delta version, (b), (d), (f).

By traversing the CFGs we identify the clusters, each beginning at a 'branch'[1], and end-

[1]States with more than one outgoing transitions.

ing in a 'join'[2]. Laski and Szermer propose a bottom-up approach, where small clusters are identified at first and then these smaller clusters are combined in bigger clusters. The technique stops when both CFGs are isomorph[3].

In our example, after first traversal of each CFG, 4 clusters are identified for the baseline model while only 2 are identified for the delta version. The clusters can be seen in Figures 4.1 (c) and (d). For example, the cluster named "$10, 11, 12, 13$" in Figure 4.1 (c) represents the subgraph containing states 10, 11, 12 and 13.

Since the CFGs are not isomorphic, we traverse the models once more resulting in Figures 4.1 (e) and (f). Note that the 3 clusters of Figure 4.1 (c) were combined with states 6 and 14 into one big cluster (named $6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19$). Now that both CFGs are isomorphic, the technique selects the test cases traversing the clusters where modifications were identified (i.e. test cases exercising states 6 to 19). It can be seen that no modification is found on states 1, 2 and the cluster $3, 4, 5$, hence test cases traversing these states are not to be selected.

The authors of the technique stated that numerous small clusters provide a more accurate selection. This claim was later confirmed by Rothermel and Harrold [Rothermel and Harrold 1996] in an empirical study where Cluster presented a very good defect detection capability and regression test suite size reduction. However, the study was performed for CFGs generated from source code, where several methods tend to form several small clusters.

On the other hand at the specification level, the scenarios of use cases tend to have a higher level of dependence, for instance the features of a mobile phone where a user can send a message and attach a picture taken from the camera, or send voice messages and so on. Consequently, this creates smaller numbers of *big clusters*. Similarly, more complex systems with several components tend to generate larger and more complex CFGs, in a sense that *Cluster* requires more time and effort to execute.

In conclusion, the idea of identifying the clusters to establish the entities affected by the modifications can be applied at the specification level. However, other approaches should also be used (e.g. search-based software engineering, model analysis, etc.) to improve the process of finding the clusters and selecting the test cases. This problem, on the other hand, is

[2]States where the divided flows join again

[3]The CFGs exactly corresponds in form with another.

alleviated when using SART, because our technique's input are test suites, hence comparison is done automatically through similarity functions.

4.1.2 Dependence Analysis on Extended Finite State Machines

Korel et al. [Korel et al. 2002] proposed a technique for selecting regression test cases by analysing dependencies in Extended Finite State Machine (EFSM) models. The technique uses a baseline model and a description of the modifications performed in order to generate the model's delta version. Then, from each EFSM a new model named Static Dependence Graph (SDG) is generated based on dependency analysis between the states and transitions of the EFSM.

The authors formalized several aspects of the dependencies between the EFSM's modified model elements as *interaction patterns*. Those interactions patterns are subgraphs of the SDG containing the entities of the EFSM affected by the modification. Chen et al. [Chen et al. 2007] extended the study revising and identifying more interaction patterns analysed by the technique. The next step is marking the test cases traversing a modified element of the EFSM, and then getting interaction patterns from each of those test cases. Therefore, test cases are expressed by SDG subgraphs representing the control and data dependencies among the traversed states and transitions. The technique then begins removing test cases with the same interaction pattern. Figure 4.2 illustrates this process.

The main challenge is obtaining the SDG from each EFSM, since thorough analysis of all the model elements is costly and complex. Moreover, for big test suites (a very common case in regression testing and model-based testing) identification of all redundant interaction patterns requires time. In conclusion, despite performing a very strong selection strategy [Korel et al. 2002; Chen et al. 2007], application of this technique in real industrial cases may not be practical.

In addition, most of the dependency analysis is linked to model elements of an EFSM making the technique dependent of a specific model type. That hinders the technique's versatility in an MBT process, where the entire dependence analysis needs to be adapted if a company uses different models, such as UML models. Also, the technique requires knowledge regarding modifications, because EFSM are not compared automatically, and it takes time to visually analyse models. SART, on the other hand, is fully automatic considering as

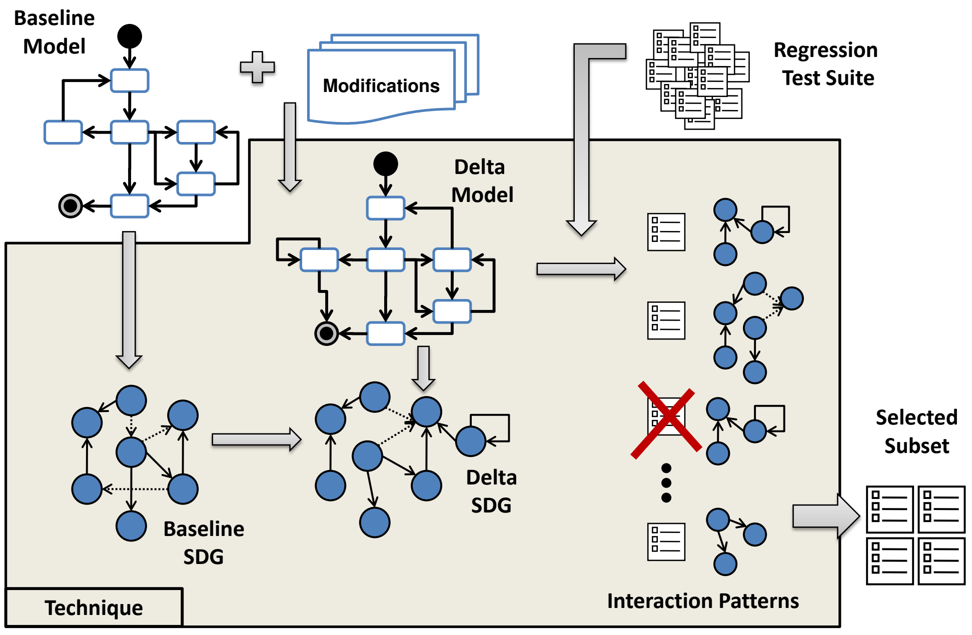


Figure 4.2: Execution of the technique proposed by Korel et al. [Korel et al. 2002].

input the test suites, hence our technique is not linked to a specific type of model.

4.1.3 Regression Test Case Selection with Risk Analysis

Chen et al. proposed a technique able to select regression test cases by identifying modifications in an UML activity diagram and performing a risk analysis [Chen et al. 2002]. In their work, the authors classify regression test cases as Targeted or Safety test cases. Targeted test cases traverse model elements affected by a modification while safety test cases traverse the unaffected model elements. The authors claim that it is important to also execute some test cases unrelated to modifications, because they may test essential functionalities or requirements of a software.

The technique is divided in two parts (Figure 4.3): The first part is to select targeted test cases, whilst the second aims at selecting safety test cases. During the first part, both versions of the activity diagram are compared to identify modifications. The second part, in turn, performs a risk analysis among the test cases not selected. The goal is to identify test cases with high risk exposure[4].

[4]The Risk Exposure (RE) is a value calculated by the technique indicating a rank that represents the probability of revealing a severe defect, and the cost for correcting this defect [Chen et al. 2002].

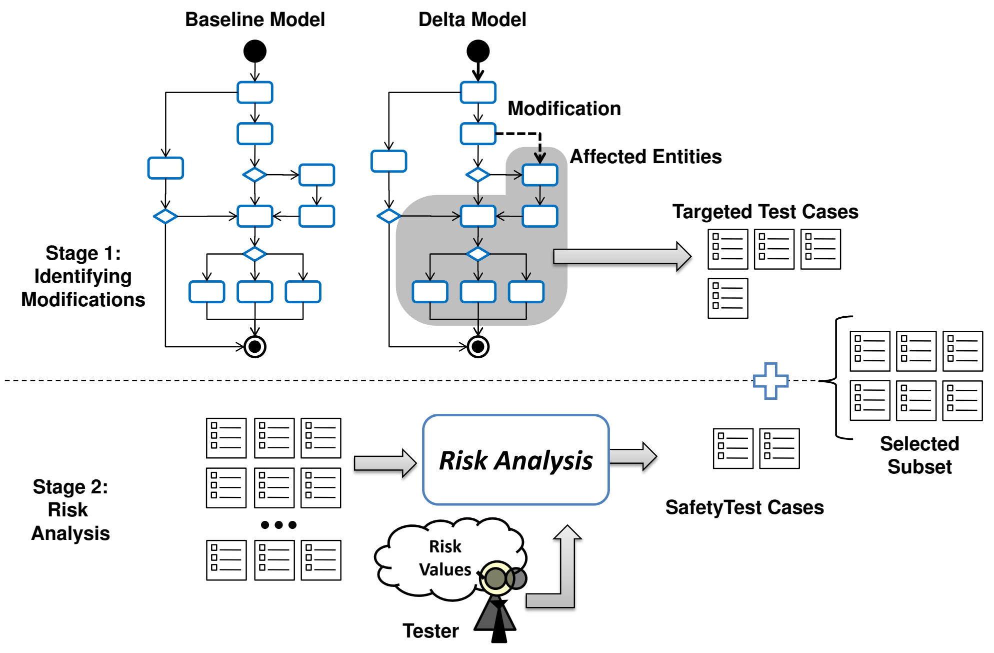


Figure 4.3: The process for selecting regression test cases proposed by Chen et al. [Chen et al. 2002]

.

For the first part the technique creates a traceability map, connecting all the test cases with each model element (i.e. activities and transitions) that it exercises. Next, both models are compared in order to identify the diagram's modifications marking all the affected entities. Chen et al. consider as affected model elements any transition or state that was modified or any descendent of a modified element. Then, the map is used to trace and select the test cases linked to affected model elements.

The next part is to perform the risk analysis, where the tester assigns values to test cases. The values represent the number of defects that a test case *can* reveal and the severity of these defects (a value from 1 to 5, where 1 is low and 5 is high severity). Based on those two values, the technique automatically calculates a risk value and ranks the test cases by a value called 'risk exposure' [Chen et al. 2002]. The test cases with the higher risk exposure values are selected as safety test cases, and then both targeted and safety test cases are included in one subset as the technique's output.

The combination of both parts benefit the technique, since test cases traversing both the affected elements and critical functionalities are executed on the SUT. Studies with the technique have shown that the risk analysis can increase the capability of revealing severe defects by automatically selecting critical test cases [Chen et al. 2002].

Moreover, the case study performed by the technique's authors only considered experienced testers, but did not address the fact that the risk analysis relies on the tester's expertise in assigning the appropriate values for test cases. Therefore, one can assume that inexperienced testers are likely to assign wrong or inaccurate risk values and then compromise the technique's performance. Despite addressing an important issue of automatic selection strategies (e.g. to ensure coverage of important parts of the model), the technique becomes sensitive to human skills. We had a similar problem detected when using old versions of SART during the weight analysis in an experiment [de Oliveira Neto 2010], and that was, in fact, a hindrance when trying to include the weight analysis in this doctorate research.

Similarly to the EFSM dependence analysis, Chen's et al. technique is dependent on a traceability between test cases and a specific type of model. Besides, selecting all descendants of a modified element is not accurate because, depending on where the modification is performed, most of the activities and transitions of the diagram will be marked and the technique is forced to select a lot of test cases. SART, on the other hand, avoids that problem by performing test suite minimisation on the subset of targeted test cases, resulting in a removal of transitions coverage redundancy and providing a smaller subset of test cases.

4.2 Concluding Remarks

This chapter discussed some work regarding specification-based test case selection and regression testing. Some selection strategies and techniques were presented to provide an overview of the context in which our proposed strategy is a part of. The original version of SART was closer to Chen's et al. technique by combining selection of affected model elements and test cases' weights. However, we decided to focus on selection of affected model elements and keep the weight analysis as an alternative for testers that want to ensure coverage of important test cases.

The current version of our selection strategy is more similar to the cluster analysis technique where affected regions of the model tend to be covered due to similarities between targeted and reusable test cases. Nonetheless, SART is more versatile than other techniques presented in this chapter, because it only requires two sets of test cases as input, instead of specific types of models.

Chapter 5

Stochastic Model Generation for Evaluation of Model-based Techniques

Evaluation of an MB technique, including SART, can only be properly done if there are samples of models large enough to enable extensive investigation of the technique's real strengths and weaknesses. However, these samples are hard to find both in industry and academia revealing the problem of *a limited availability of specification models to perform empirical evaluation of MB techniques*. Getting access to real industrial models is a problem since many companies are reluctant to share their models and usually the number of models available may not be enough to obtain sample sizes for statistically significant results. Turning to literature can also be an issue, since many publications often do not provide all artefacts used in an experiment, providing instead, just a general description about the models used, such as size or type.

Usually, researches have to rely on repositories to obtain samples, and even then there is no guarantee that the objects can be very different (or similar) to allow conclusive results based on a controlled experiment. Besides, the models should comply with assumptions and constraints established by the investigated MB technique. For example, some techniques require state-based models, while others require class diagrams, annotations, time constraints, paths with loops, etc. That aspect is not restricted to model-based techniques.

Similar problems have been reported by researches evaluating code-based techniques as well (e.g. JDolly[1] [Soares et al. 2013], UDITA [Gligoric et al. 2010]), where large samples

[1]`https://code.google.com/p/jdolly/wiki/JDollyManual`

of programs with specific characteristics are needed for experiments. Therefore, availability alone is not the problem since repositories containing samples of source code, such as open source software communities, are widely available on the Internet. In addition, a proper evaluation method requires diversity and control over the sample of objects being used with the investigated technique.

Then, how can we evaluate SART, or any other MB technique, if we cannot find a sufficient, controllable and diverse sample of models? To address this situation we have proposed to combine search based techniques and stochastic model generation with descriptive statistics of realistic models [de Oliveira Neto et al. 2013]. Combining information from actual models used in practice with stochastic model generators would allow us to generate large number of models (i.e. a space of models) that may share characteristics with industrial models. Therefore, each model within that space is a possible input and its specific characteristics will lead to a specific behaviour and thus performance of an evaluated technique.

The goal of finding models with relevant/particular characteristics leads to a search problem and search-based software engineering (SBSE) can be applied. In SBSE, classic software engineering problems are reformulated as search problems and metaheuristic search (MHS) such as genetic algorithms and simulated annealing can be used to find optimal or near-optimal solutions [Harman and Jones 2001; Clarke et al. 2003].

For example, consider that the best solutions are models where a better performance is observed when executing the investigated MB technique. Then, we are able to analyse performance on specific sets of models (regions of the space), or search for candidates where the MB technique shows especially good or bad performance. In general, search-based techniques would allow exploration and visualization of different (sub-)spaces of models as well as the difficulty for the technique over this space [Feldt 1998; Feldt 1999], enabling a kind of sensitivity analysis of the technique [Saltelli et al. 2004].

Taken together this approach is named Search Based Model Generation for Technology Evaluation (SBMTE) [de Oliveira Neto et al. 2013] and is proposed to enable empirical evaluation of MB techniques when there is a limited availability of models. In this doctorate research we focus on specification models, but the same approach can be applied to other types of models, or more generally, development artefacts for which an at least partially formal formulation can be provided.

5.1 The Model-Based Technique Evaluation Approach

The stochastic model generation is performed by a generator tool developed by the researcher. This *generator* performs automatic generation of instances of models using descriptive parameters of real models. The *parameters* are used mainly as constraints to generate specific subsets of models that are of interest, whereas stochastic choices determine which instances of models within the subset are generated on a specific invocation of the generator. Figure 5.1 presents the steps to create a stochastic model generator used in our approach. We suggest a *bottom-up* approach, where smaller units of the model are combined into bigger parts that, in turn, are combined to create a model layout.

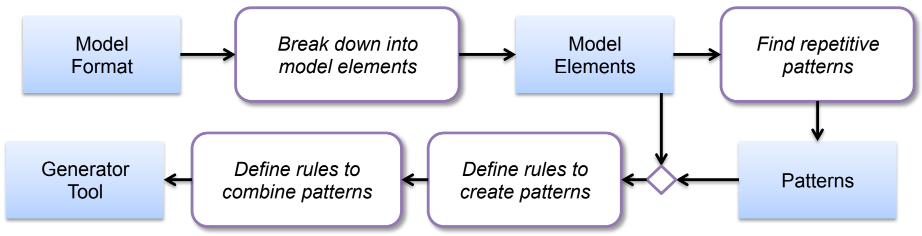

Model Format
Break down into model elements
Model Elements
Find repetitive patterns
Patterns
Define rules to create patterns
Define rules to combine patterns
Generator Tool

Figure 5.1: Steps to create a model generator for evaluation of MB techniques.

The first step in creating the generator is to choose the targeted format (or type) of model, such as Finite State Machines (FSM), Activity Diagrams, Message Sequence Charts, Sequence Diagrams, etc. After that, we break down the model into its smallest units, named ‘model elements’. Then, we search for ‘patterns’, defined as bigger combinations of model elements that are repetitively used to represent meaningful information of a model. Examples of patterns are: Decisions in an activity diagram (e.g. a combination of decision nodes, actions and activities), loops in a FSM (a transition with the same source and sink state), among others. Note that the difference between model elements and patterns is that model elements are atomic units of the model, whereas patterns are composed of one or more model elements and usually appear more than once in an instance of the model.

After defining model elements and its patterns, the next step is defining the rules to combine model elements into patterns. This is an important step regarding the generator’s validation, because the rules ensure that the generated model instances are consistent and do not violate the model’s invariant. For example, in an FSM we cannot define a state with two outgoing transitions both with the exact same guard, cause that creates non-determinism of the model. The same applies to the next step, where the rules that define the procedure to combine pattern into model layouts need to be defined. We suggest usage of stochastic decisions on how to combine pattern into model layouts, in order to achieve a higher diversity of models.

Throughout the process, especially during definition of model elements and patterns, the researcher is able to identify which pattern and model elements are representative to be used as the generator's input parameters. Imagine that the input parameter of an Activity Diagram generator could be the number of transitions and activities (model elements), or even the number of decisions/merges or concurrent activities (patterns). Thus, those decisions depend on the generator's design and the type of models targeted.

There are several elements that can help the design of a generator. Aside from experience in using and modelling using the chosen model format, the research can rely on meta-models to help definition and rules in creation of models. There is an extensive array of documentation formally defining Unified Modelling Language (UML) diagrams provided by the Object Management Group (OMG) [2], expressed in a standardized language (Meta-Object Facility — MOF). Most of the UML diagrams have their model elements defined through meta-models, and those artefacts can aid creation of a generator to the respective model format, such as definition of model elements and rules to combine the patterns.

Once the generator is defined, a desired quantity of models[3] with the specified parameters can be systematically generated. We emphasize that the generator tool must be designed to guarantee consistent instances among generated models, i.e. models that comply (definitions and rules) with the type of model and constraints defined during the specification of the generator. But, after finished, how can we use the generator to evaluate model-based techniques?

Overall, SBMTE can vary in two main dimensions: *Realism of models* and *type of search employed*. The level of realism can be related to the parameters used to generate the models, whereas the type of search is related to the purpose of evaluation. Figure 5.2 presents how the generation can be used considering both dimensions.

[2]`http://www.omg.org/spec/`

[3]Some classes of models might not allow the generation of arbitrary sizes of samples. The quantity may be limited depending on model complexity, and combinations of model elements available.

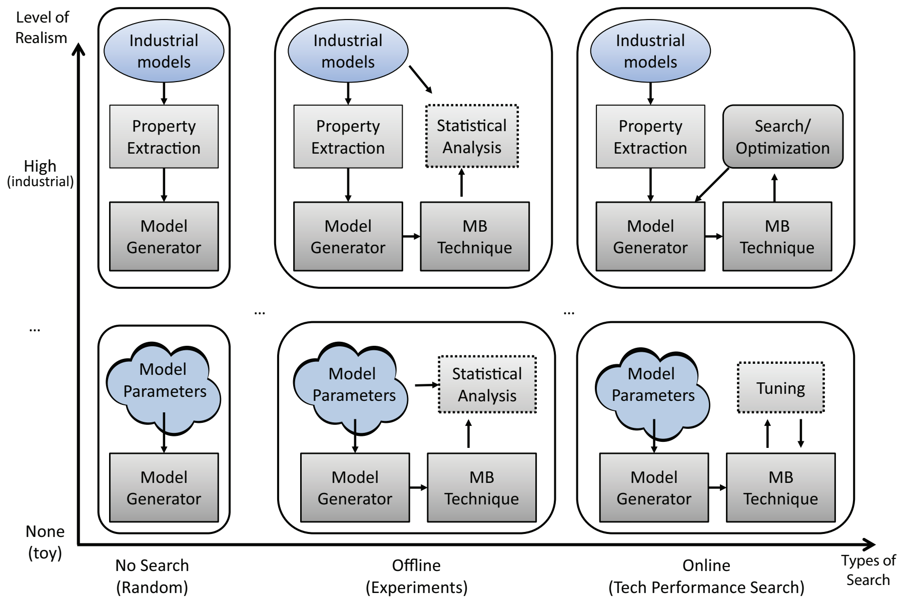


Figure 5.2: An overview of SBMTE.

High level of realism can only be achieved if the generator or its creator has access to models used in industry. The generation of models based on input from industry can help approximating the technique's performance when used outside laboratory, i.e. in industry. That reduces risks of applying the technique directly on the actual models and helps the practitioners decide if it is worthy to adopt the investigated MB technique.

However, getting access to industrial models may be a problem since companies are usually reluctant to share their models given that they often carry confidential information. Through SBMTE, researchers can develop scripts that can be executed by industry partners to get statistics regarding the models, such as an average number of states, transitions, constraints, activities, among other modelling elements. That way the researcher does not need to access the confidential models themselves, and instead, can gather the descriptive parameters to generate a similar set of models, thus avoiding or at least mitigating NDA (non-disclosure agreement) issues. We refer to that approach as *model-property extraction*.

The lower levels of realism comprise the generation based on parameters reported in literature or obtained from a few companies. Despite the generation of less realistic models, a proper type of search can still be employed to provide valid evaluation to the investigated MB technology. The search is conducted among the *space of models* and the purpose can vary from finding models matching a specific combination of parameters or finding models that cause a better or worse performance of the investigated MB technology. The idea is that the search-type can achieve at least three different levels: No/Basic, offline SBMTE and online SBMTE.

- **Basic level:** At this level, no optimization search is used. At low levels of realism, many small (toy) models are generated to ensure that the technique is feasible and properly implemented. Realistic models can be used at this level to obtain convincing evidence of the scalability of the technology to industrial systems.

- **Offline SBMTE:** The goal is to search for models that comply with specific combinations of parameters. Models are generated to enable the execution of an MB technique when industrial models (from which we possibly extracted the parameters) are not numerous enough to provide a proper evaluation. An example is to generate models to obtain samples for experiments.

- **Online SBMTE:** At this level performance measures of applying the MB technique are combined with search. The goal is to find areas among the space of models with a better (or worse) performance. With the models found, the investigated MB technique can be executed, evaluated or compared so that trade-offs are identified. This can enable a more detailed understanding of the pros and cons of the investigated technique.

Online SBMTE combined with high levels of realism provides means to perform a very strong evaluation of the investigated MB technique. By observing the models found, one can decide whether the technique should be applied in industry, based on the models of the product being developed. However, keeping the search 'in the loop' can be costly when the MB technique is very time consuming to use.

In turn, Offline SBMTE can be a very powerful asset to empirically evaluate MB techniques with limited objects (models). Depending on the descriptive parameters alone, the stochastic generation can be used to generate enough subjects for the estimated sample size, specially if statistical significance is desired. Note that the researcher needs to be careful when using the generation to target statistical significance of a sample, since the sample size must be properly estimated based on confidence intervals and variations. Furthermore, researchers can change the generator to add constraints enabling control of the generation, such as an upper bound on the sample size.

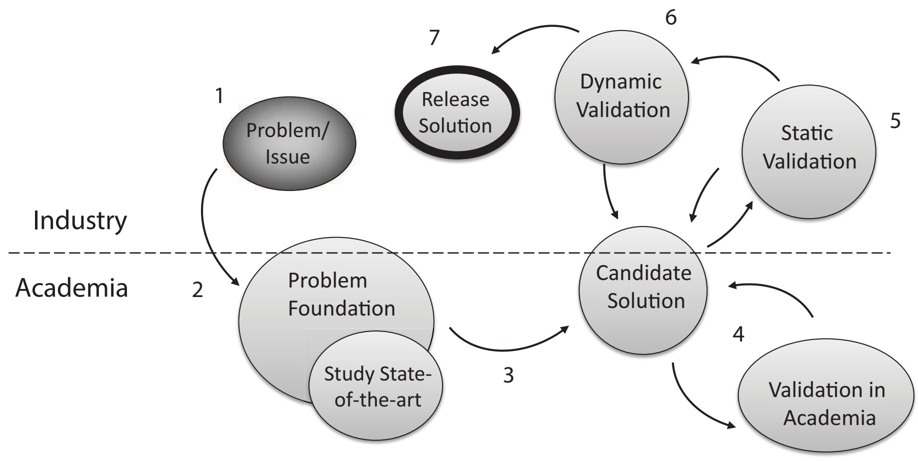


Figure 5.3: An overview of technology transfer in practice (Adapted from [Gorschek et al. 2006]).

SBMTE can also help technology transfer of MB techniques to industry. If we consider Gorschek's et al. [Gorschek et al. 2006] technology transfer model (Figure 5.3) SBMTE can be applied in Step 4. Performing experiments with a controlled generation of models allows the investigation of the technique's performance under several configurations of models. This provides a stronger evidence for when the technique is applicable and likely to be beneficial, and when it is not. With more (and better) information regarding the technique's behaviour assembled, we lower risks and increase confidence in decisions about introducing the technique in the specific development process of an industrial partner (i.e. Steps 5–6).

5.2 ALTS Model Generator

In order to develop a model generator tool according to our process presented in Figure 5.1, we began by choosing the Annotated Labelled Transitions System (ALTS) as model format. As mentioned in Chapter 2, ALTS provides a simple and easy to use model format that also allows versatility for combining simple model elements to represent software's behavioural scenarios. Next we began defining its model elements and patterns used to create the generator's rules. Here, ALTS are used as an extension to the LTS definition (presented in

Chapter 2 and Section 2.1). Therefore, the following terminology is considered, *in addition* to the standard LTS 4-tuple (Q,A,T_{tr},q_0), in order to represent states and transitions of a reactive system.

- **States:** We consider that the set of states in an LTS (Q) can be divided in two different subsets of states, representing the *user action states* and the *expected output states*, referred respectively as Q_{ua} and Q_{eo}. The first represents software's states where user actions can be performed during execution, whereas the second represents the set of states where the software system produces an output in response to that user action. Therefore, the following statement holds for all generated instances of models: $(Q = Q_{ua} \cup Q_{eo}) \ \wedge \ (Q_{ua} \cap Q_{eo} = \emptyset), \ q_o \in Q_{ua}$.

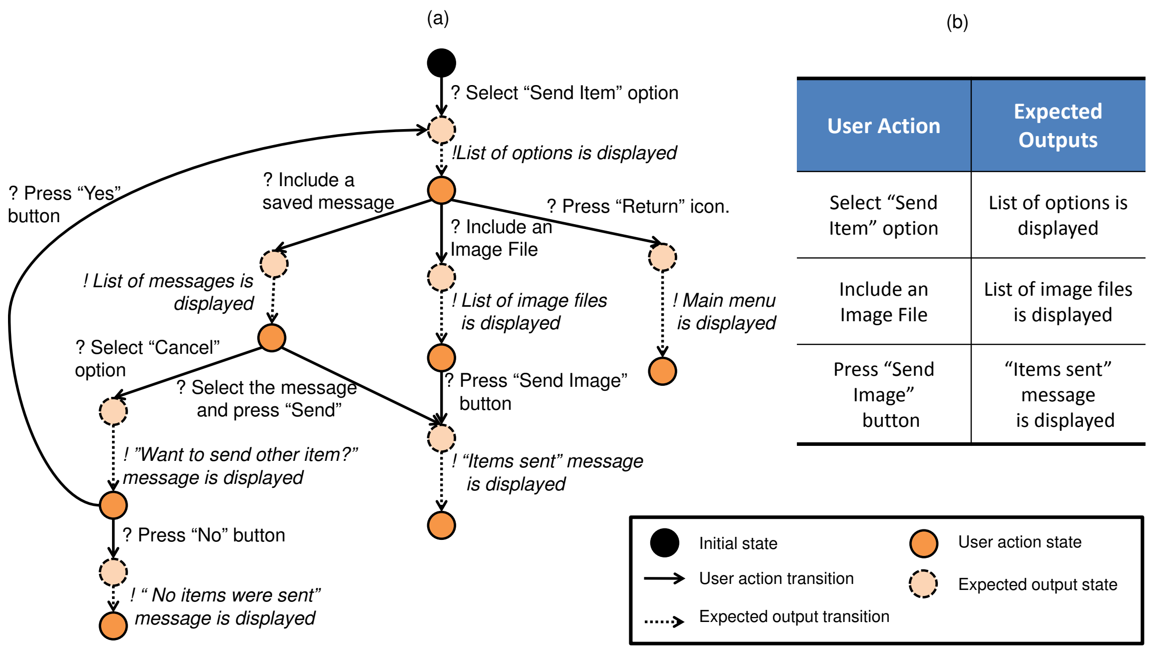
- **Transitions:** Similarly, we divided the set of transitions (T_{tr}) to allow representation of user actions and system responses. The *user action transitions* (T_{ua}) are marked with the symbol '?' at the beginning of its label and they connect a user action state to an expected output state. In turn, the *expected output transitions* (T_{eo}) connect an expected output state to a user action state, and their label are marked with the symbol '!'. They can also be defined as:
  - $T_{ua} = \{\overrightarrow{t} \in T_{tr} \mid \overrightarrow{t} = (Q_{ua} \times ? \cdot A \times Q_{eo})\}$
  - $T_{eo} = \{\overrightarrow{t} \in T_{tr} \mid \overrightarrow{t} = (Q_{eo} \times ! \cdot A \times Q_{ua})\}$

- **Step:** A step is a pair composed by one user action and the correspondent expected output, representing a user actions and the output that should be produced by the system. Or, in other words: $step_i = (\overrightarrow{t_i}, \overrightarrow{t_j}) \mid \overrightarrow{t_i} \in T_{ua} \wedge \overrightarrow{t_j} \in T_{eo}$.

The annotations above were included in the definition in order to achieve a *reasonable level of realism*, and allow generation of abstract test cases from paths of the ALTS, i.e. a non-empty finite sequence of steps (inputs and expected outputs of the system). Since we are only interested in sequences of steps, only deterministic ALTS will be considered, i.e. annotated LTS *without* internal actions (τ) and non-determinism[4] represented in the model

[4]Non-determinism is defined by: $t_i \in T_{ua}, t_j, t_k \in T_{eo}, \exists step_m = (\overrightarrow{t_i}, \overrightarrow{t_j}) \wedge step_n(\overrightarrow{t_i}, \overrightarrow{t_k}) \mid \overrightarrow{t_j} \neq \overrightarrow{t_k}$. In other words, we consider non-determinism to be represented as a user action transition ($\overrightarrow{t_i}$) followed by two different expected output transitions ($\overrightarrow{t_j} \neq \overrightarrow{t_k}$)

[Tan et al. 1997]. Figures 5.4 (a) and (b) show, respectively, an example of an ALTS and a test case.


(a)
? Select “Send Item” option
!List of options is displayed
? Press “Yes” button
? Include a saved message
? Press “Return” icon.
? Include an Image File
! List of messages is displayed
! List of image files is displayed
! Main menu is displayed
? Select “Cancel” option
? Select the message and press “Send”
? Press “Send Image” button
! ”Want to send other item?” message is displayed
! “Items sent” message is displayed
? Press “No” button
! “ No items were sent” message is displayed
(b)
Initial state
User action transition
Expected output transition
User action state
Expected output state

User Action	Expected Outputs
Select “Send Item” option	List of options is displayed
Include an Image File	List of image files is displayed
Press “Send Image” button	“Items sent” message is displayed

Figure 5.4: (a) The model elements of an ALTS and (b) a test case.

To automatically generate ALTS, states and transitions of simple structures (*patterns*) must be systematically combined. Furthermore, an automatic modification (addition and removal of transitions[5]) of the generated ALTS is also performed. In summary, the patterns are:

- **Main flow size** (S)**:** In our test process, each ALTS must specify at least one scenario of an application to be executed, i.e. the main flow in which alternative flows will branch and connect, yielding different states of the system. The length is determined by the *number of steps* in the sequence.

- **Branches** (B)**:** During the execution of the main flow, the user can perform different actions (transitions) causing the system to reach alternative flows. States with such transitions are defined as branches.

[5]States that become unreachable after deleting a transition are also removed from the ALTS. Similarly new states can be added as destination to recently added transition. For example, a completely new alternative flow can be added/removed to the main flow.

- **Joins** (J)**:** Some flows may have a common expected output for different actions. Thus, transitions leading to different actions to a common expected output state are joins.
- **Paths with loops (L):** Paths that contain a sequence of one or more transitions where the initial and final location are the same (i.e. a cycle).
- **Additions** (A)**:** The new scenarios or steps added to the specification.
- **Removals** (R)**:** The steps or entire scenarios removed from the specification.

Figure 5.5 shows: (a) The example of an ALTS automatically generated, (b) the patterns used to construct the model and, (c) a delta version after modifications are performed. The generated ALTS is isomorphic to that of Figure 5.4 illustrating that it is possible to obtain ALTS similar to real specification models.

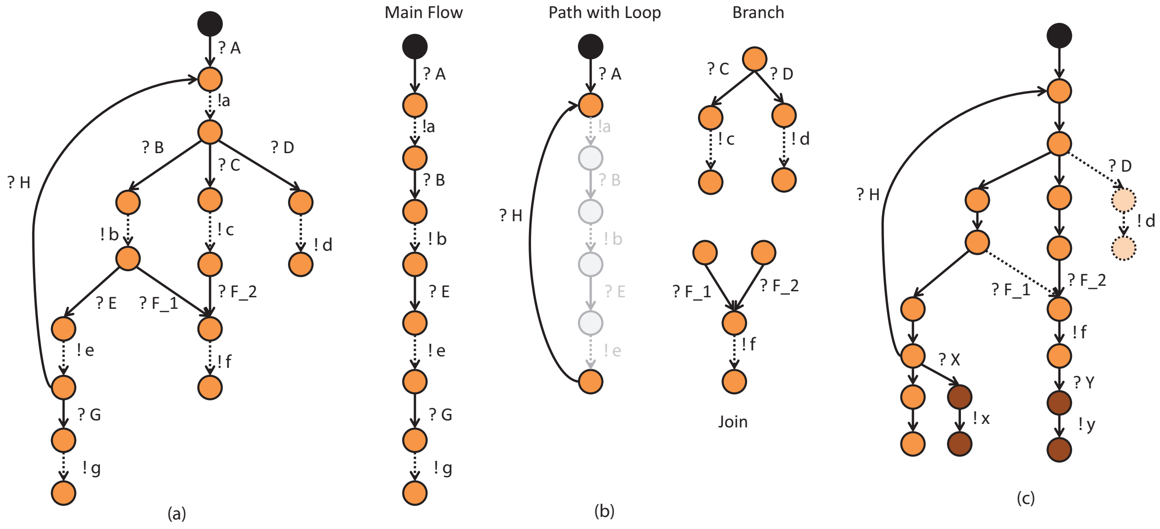


Figure 5.5: (a) Example of a generated ALTS, (b) the patterns used and (c) examples of modifications.

Once model elements and patterns are defined, we create the rules to create the patterns, and followed by the procedure of combining patterns into instances of ALTS. The definition of rules and procedures of model generation is the main aspect of the generator tool, because they ensure control in an experiment and help mitigating construct and internal validity threats. Our generator uses the following rules to create and combine the patterns:

- **Creation of the ALTS:** The creation is done iteratively so that in each iteration a branch, join or path with loop is added to the ALTS. The addition is performed in a user action state randomly selected from the current ALTS layout.

- **Eligible states:** The default selection of states is done among the user action states ($q \in Q_{ua}$), except for the initial and the last state of the initial sequence created. Note that this strategy ensures model integrity, because each pattern will begin with a user action *after* an expected output. Also, the expected output states created when adding patterns to the ALTS become eligible for the next iteration's state selection.

  – **Create main sequence:** The first step of the generation is creating a sequence of steps with the specified size. Consequently, the initial state of this sequence will also be the initial state of the ALTS (q_0). By default, patterns cannot be placed on the first and last state of this sequence (the former has an exception for branches), in order to control the main sequence length during an experiment.

  – **Add branches:** Branches are placed either on a random eligible state or the initial state of the ALTS. Two outgoing steps are then added to the chosen state.

  – **Add joins:** Joins are added on two random eligible states, so each will have an outgoing user action transition that reaches a single expected output state ($q \in Q_{eo}$) as destination.

  – **Path with Loops:** A random eligible state is chosen as source state of the loop transition, then the generator backtracks the model until it reaches the q_0, and the loop transition's destination is a random *expected output state* from that path. The created transitions are always a user action, because we assume that the user will determine the repetition of activities when interacting with the software system (e.g. provide an invalid password, or return to the main screen).

- **Modification of the ALTS:** After creating the baseline layout of the ALTS, we begin modifications to obtain a delta version. In order to keep consistence among the number of additions and removals performed, we decided to first perform all removals, and then perform the additions, otherwise the randomness of state selection can cause the generator to remove a recently added transition, hence generating a wrong ALTS with respect to the specified number of patterns.

- **Removal:** In order to avoid generation of disconnected ALTS, removals are performed on the leaves[6]. Thus, a random leaf (except for the main sequence's leaf) is selected, and the correspondent user action is also removed from the ALTS. Note that when selecting a leaf from a join, only one of the user actions is removed, since the expected output must remain as output of the remaining user action transition. An example can be seen on removal of step $(\overrightarrow{?F-1}, \overrightarrow{!f})$ from Figure 5.5 (c), where transition $\overrightarrow{!f}$ cannot be removed because the step $(\overrightarrow{?F-2}, \overrightarrow{!f})$ still remains in the ALTS.
- **Additions:** Similarly to branches, additions are performed either on eligible states or the initial state of the ALTS, but only one step is created in each addition.

```
Input: The number of ALTS to be generated (N), main sequence size (S), and the number
of branches (B), joins (J), paths with loops (L), additions (A) and removals (R).
Output: A sample with N generated ALTS models with the specified input parameters.
--------------------------------------------------------------------------------------
GenerateModels(N, S, B, J, L, A, R)
space_of_models ← ∅
for i : 1 to N do
    aux_ALTS ← generateSequence(S)
    patterns ← createCollectionPatterns(B,J,L)
    shuffle(patterns)
    for j : 1 patterns.size do
        if patterns[j] = B then addBranch(aux_ALTS)
        else if patterns[j] = J then addJoin(aux_ALTS)
        else   addLoop(aux_ALTS) //patterns[j] = L
    baseline ← aux_ALTS
    for j : 1 to A do
        addNewStep(aux_ALTS)
    for j : 1 to R do
        removeStep(aux_ALTS)
    delta ← aux_ALTS
    space_of_models ∪ {baseline,delta}
return space_of_models
```

Figure 5.6: Algorithm to generate the baseline and delta ALTS.

[6]We refer to leaf as a state without outgoing transitions. However, the ALTS *is not* a tree due to the presence of cycles.

Both processes of generating and modifying the ALTS are automatic, enabling the quick generation of a quantity N of models, specified by the generator's user. The algorithm for generation is presented in Figure 5.6. Considering that our algorithm, for the specified number N of ALTS, creates each pattern and performs all modifications, our algorithm executes in polynomial time $\mathcal{O}(N \times (S + B + J + L + A + R))$.

Initially, we begin with an empty sample of models (line 1), and iteratively include a generated ALTS. Note that each ALTS begins with a main sequence (line 3) and then patterns are attached to eligible states (lines 6–9) so that before we start modifications, we have a baseline ALTS with the specified patterns. Then, we remove steps from this baseline ALTS and then start adding transitions to it. This order is relevant, since performing the additions intertwined with removals would cause removal of a recently added transitions. Consequently, the resulting instance would not comply with the specified input patterns and modifications. After modifying the ALTS (lines 11–14), we include it in our sample of models (line 16) and then repeat the process by generating a new instance until the specified sample size is reached.

To illustrate the algorithm, Figure 5.7 presents a step by step example of a generation with parameters $N = 1$, $S = 4$, $B = 3$, $J = 2$, $L = 1$, $A = 1$ and $R = 1$. Note that the generator assigns a random order to create each patterns (except for additions and removals). For instance, the ALTS generated in this example considered the ordered patterns $[B, B, L, B, J, J]$, which means the generator creates two branches followed by a path with loop, then another branch and two joins. A different generation with the same parameters could yield a different order of patterns, for example $[L, B, J, B, J, B]$, which means that different layouts are created iteratively. That reduces the chances of having very similar ALTS.

5.3 Concluding Remarks

Alongside SART, SBMTE is one of the main contributions to this doctorate research. Therefore, while evaluating our selection strategy, we also gathered information regarding advantages and challenges of SBMTE itself. In order to verify our generator tool we used automated test scripts to verify whether the generated instances are well formed and consis-

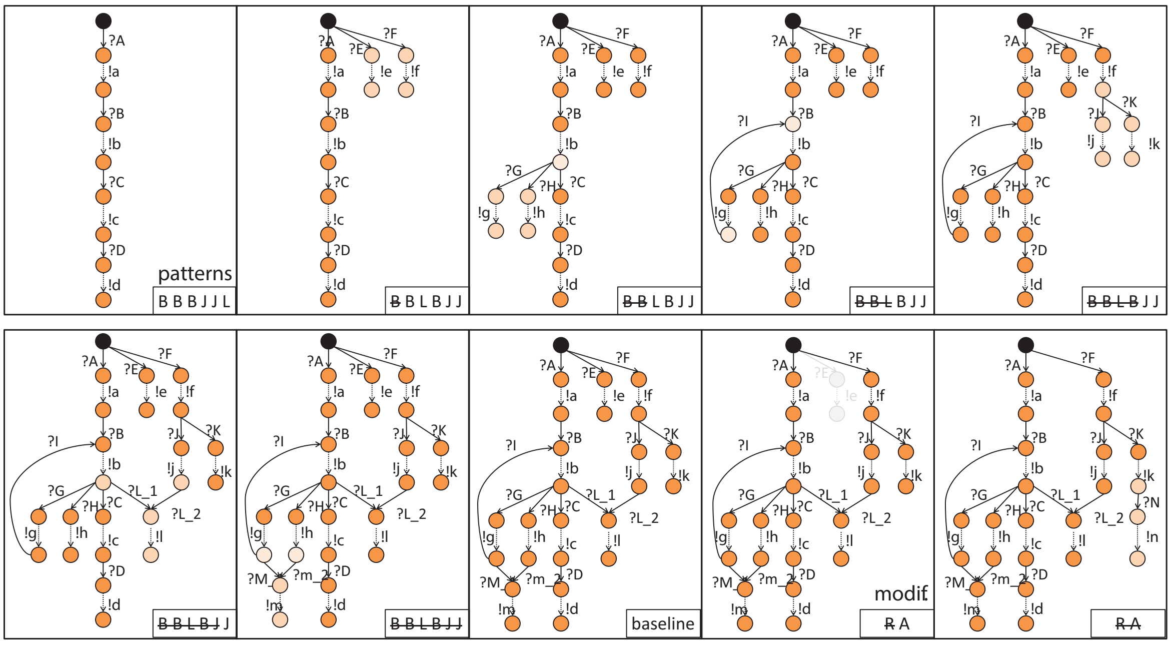


Figure 5.7: A step by step example to generate an ALTS.

tent (i.e. comply with the specified number of input parameters). Unfortunately, we were not able to define and execute an experiment to evaluate our generator tool in a controlled environment. The randomness related to construction of ALTS hinders control of our generated input sample.

However, future work comprise a thorough execution and analysis of our ALTS generator using similarity measures among graphs [Singh et al. 2007; Zager and Verghese 2008] to mathematically observe the differences among the generated instances of ALTS. So far, our evaluation was done manually by generating a few hundreds of models and visually observing whether the ALTS are similar among each other.

The discussion in this chapter covered the general aspects of SBMTE, such as the methodology to introduce automatic model generation into an evaluation process, followed by presentation of our model generator tool. The benefits, challenges and answers to research questions were obtained after applying SBMTE in our evaluation process, leaving the remainder of discussion together with the experiment's results (Chapter 7).

Chapter 6

Evaluation

Next, we present details of the evaluation methodology used to explore and evaluate SART's potential. Our evaluation is divided in three different studies. First and foremost we investigate SART's capabilities through an experiment. Second, we perform a different experiment (re-using some artefacts from the first experiment) in order to compare SART with two other test case selection techniques: A traditional approach for similarity-based test case selection [Cartaxo et al. 2011], and a random selection of test cases. Last, but not least, we conduct an exploratory case study where SART is used on industrial artefacts.

The goal with the second experiment and the exploratory case study is to complement information regarding SART's performance obtained in the first experimental study. Therefore, the main investigation towards SART is based on results from the first experimental study. The goal with our comparative study is to show whether our adaptation of the counting function [Cartaxo et al. 2011] affected the technique's performance under the specific regression testing context, i.e. if the technique is able to handle specific regression test requirements such as covering modifications and triggering regression defects. In turn, comparison with random selection allows us to observe the benefits and drawbacks of adding a specific coverage criteria to our selection technique.

In turn, the exploratory case study with industrial artefacts provide valuable insight towards practical applicability of SART in an MBT process. Unfortunately, the limited size of our sample of industrial specification models hindered execution of an experimental study with practitioners. Since the definition and execution of our case study is small, we decided to describe details regarding the definition, planning, artefacts and results in Chapter 7

(Section 7.3).

Similarly, we decided to provide all information for our second experiment in Chapter 7 (Section 7.2), because most of its definition and planning is similar to the first (for example, the artefacts and tools). Consequently, this chapter will explain the main elements of our first (and main) experimental study, such as: Our goals, hypotheses, factors, dependent variables and tools.

6.1 Experiment

The first direction to explore SART's potential and robustness is by investigating its capability of selecting representative[1] subsets regardless of the input provided. Since our test process begins with a modified specification model, we conjectured if our selection strategy is affected by specific types of models, e.g. with specific quantities or layout of states and transitions. Consequently, we gather information to assist a tester in deciding whether to apply SART or not.

For example, the addition of several alternative flows in a document describing the scenarios of a use case can be represented as models with several additions of branches. As a consequence, there will be more paths to be covered and perhaps the investigated technique will show an improved size reduction or reveal more defects. Or it might have problems handling those (or other) specific types of models.

Therefore, we investigate the informal hypothesis that *the performance of SART is strongly affected by the type of model used to design/generate the test cases*. Similarly to SART, some MB techniques use test cases that are very close to the specification model, i.e. they can be mapped to paths of the model [Bertolino et al. 2008; Cartaxo 2011; de Araújo et al. 2012; Coutinho et al. 2013; Ouriques et al. 2013]. So this experiment can be adapted to include other MB techniques to observe if those techniques are robust or dependent of the type of models used as input. For now, we focus our experiment on SART, under the following definition template ([Wohlin et al. 2012]):

[1]It was determined in the introduction of this thesis that representativeness, in our research, is expressed through model modifications being covered and probability of triggering regression defects

Analyse **SART**
for the purpose of **detecting strengths and weaknesses**
with respect to its **performance**
from the point of view of the **tester**
in the context of **progressive regression testing**.

Given that we could not find a sufficient sample of industrial specification models, we decided to use SBMTE to generate a large sample of specification models to execute our MBT process. Besides enabling the evaluation of SART, by actually applying SBMTE we gather information about issues, challenges and advantages regarding usage of stochastic model generation to evaluate MB techniques.

Considering that our goal is to analyse an automatic test case selection technique, there is no human interaction in our experiment, hence *no subjects participate*. Instead, we use objects representing artefacts of the considered test process, such as: Specification models, the modifications performed, mutation on the models, test suites and test cases. Since it can be costly to design test cases manually from a large sample of specification models, we will use automatic test case generation based on a DFS algorithm (as described in Chapter 2) to obtain all paths traversing only one loop, since the coverage of more loops would represent a costly time overhead in test case generation. An overview of the experiment is presented in Figure 6.1.

SART selects from test cases obtained from instances of models, that are, in turn, defined by the input parameters given to the generator, i.e. the patterns: Main sequence size (S), branches (B), joins (J), paths with loops (L), additions (A) and removals (R). Thus, S, B, J, L, A and R are *factors* in our design. Considering that the generation of models is affected by isolated and combined patterns, we will vary each factor to a high and low value ("+" and "-" respectively) and investigate all interactions. Hence a 2^6 *factorial design* is defined for this experiment, with 64 combinations of treatments. During the next paragraphs, each combination will be referred as *configuration*.

Despite defining random locations and orders to create the patterns, the generation is not completely random. Also, since constraints and a desired number of patterns guide the generation, the developed generator is pseudo-random. Nonetheless, observing the results

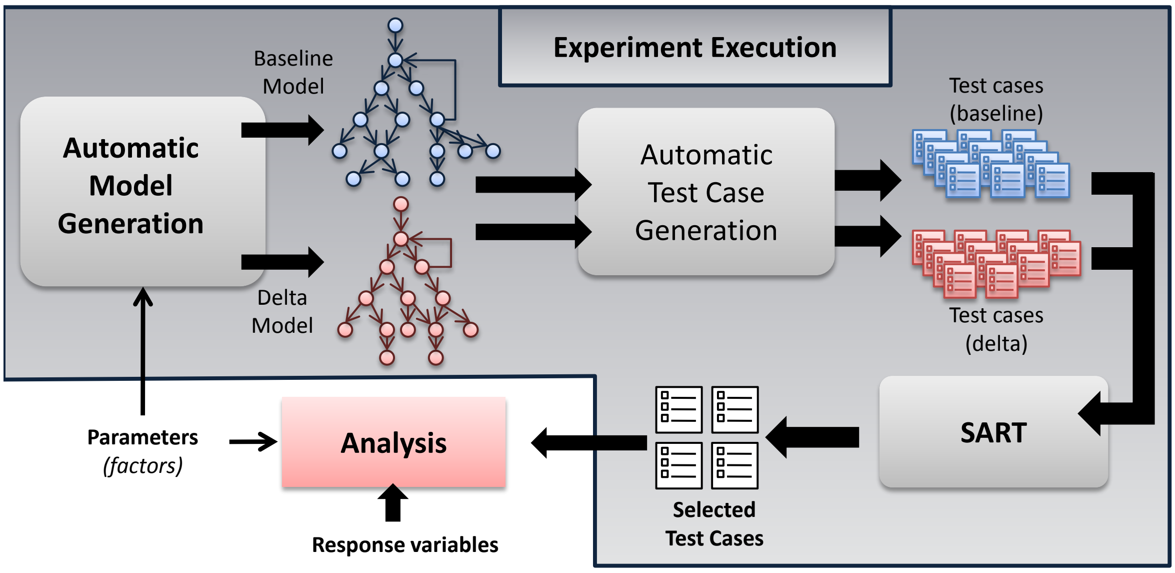


Figure 6.1: Overview of the experiment.

from a single generated ALTS would provide inaccurate results. Thus, a number[2] $N = 10$ ALTS is generated for each configuration, resulting in 64 mean values. We replicate the experiment 3 times in order to address possible outliers or residual errors due to execution issues. In summary, SART was executed $1,920$ times $(64 * 10 * 3)$ providing nearly 2000 data points to be statistically analysed.

6.1.1 Response Variables

SART's performance is measured based on two widely used response variables in regression testing literature [Yoo and Harman 2012]: Percentage of size reduction and effectiveness of defect detection. By analysing these two variable we observe how much size reduction the technique can achieve and whether this reduction pays off by still revealing defects in a significant way. The percentage of size reduction, referred as *SizeRed* in this work, is calculated by Equation 6.1. Remember that T' is the test suite automatically generated from the delta version of the model and T_s is the selected subset of test cases. In practice, each

[2]The number of generated ALTS (10) was chosen after a prior power analysis where we considered $N = 100$ ALTS. This prior analysis revealed that $N \leq 10$ would be sufficient to achieve statistical significance in our analysis.

test case may cost differently (e.g. each may require a different amount of time to execute). Since such information is not available in our experimental setting, we consider that each test case has the same cost, for example, to be executed in the SUT.

$$\text{SizeRed} = (1 - \frac{|T_s|}{|T'|}) \times 100 \tag{6.1}$$

Measuring effectiveness of defect detection, on the other hand, is challenging for our experiment. One or more defects are revealed when test cases *fail*, that is when the output produced by the SUT differs from the expected output specified in the test case [Binder 1999; IEEE 2013]. After observing a failure, the defect is searched in the source code. Traditionally, effectiveness of defect detection is measured by counting the defects revealed by the complete test suite and then compare it with the number of defects revealed when executing only the selected subset [Yoo and Harman 2012]. Our experiment is not suitable for the traditional approach because the models are automatically generated and there are no real implementation or defects to measure. Even if a source code were available, SART selects abstract test cases and measuring defects at the natural language level is very inaccurate since the gap between the specification and the code hinders the traceability between the failure observed at the high level test case and the defect in the code. Ultimately, the information visible at our level of abstraction are the triggered failures.

Given these limitations, we decided to measure the *probability of observing a failure* after reducing the size of the test suite. Basically, each subset will trigger (or not) at least one failure, i.e. 1 means that at least one failure is observed, and 0 otherwise. By observing the result for quantity N of ALTS generated for each configuration, we have a binomial distribution yielding the probability that the selected subset will trigger a failure. We will refer to this probability as *PFail* (Equation 6.2).

$$\text{PFail} = \frac{\sum_{i=1}^{N} x_i}{N} \text{ where } x_i = \begin{cases} \text{1,if and only if } T_s \text{ reveals at least one failure during trial } i. \\ \text{0,otherwise.} \end{cases} \tag{6.2}$$

In order to observe failures, this work combines fault models and mutation of the specification model. The fault models are constructed based on different fault hypotheses regarding defect representation [Binder 1996]. Thus, we will have insight about SART's performance

Table 6.1: Null and alternative hypotheses for SizeRed and PFail.

$H_{0\ sr:S}$: $S_+(SizeRed) = S_-(SizeRed)$	$H_{0\ pf:S}$: $S_+(PFail) = S_-(PFail)$
$H_{1\ sr:S}$: $S_+(SizeRed) \neq S_-(SizeRed)$	$H_{1\ pf:S}$: $S_+(PFail) \neq S_-(PFail)$
$H_{0\ sr:B}$: $B_+(SizeRed) = B_-(SizeRed)$	$H_{0\ pf:B}$: $B_+(PFail) = B_-(PFail)$
$H_{1\ sr:B}$: $B_+(SizeRed) \neq B_-(SizeRed)$	$H_{1\ pf:B}$: $B_+(PFail) \neq B_-(PFail)$
$H_{0\ sr:J}$: $J_+(SizeRed) = J_-(SizeRed)$	$H_{0\ pf:J}$: $J_+(PFail) = J_-(PFail)$
$H_{1\ sr:J}$: $J_+(SizeRed) \neq J_-(SizeRed)$	$H_{1\ pf:J}$: $J_+(PFail) \neq J_-(PFail)$
$H_{0\ sr:L}$: $L_+(SizeRed) = L_-(SizeRed)$	$H_{0\ pf:L}$: $L_+(PFail) = L_-(PFail)$
$H_{1\ sr:L}$: $L_+(SizeRed) \neq L_-(SizeRed)$	$H_{1\ pf:L}$: $L_+(PFail) \neq L_-(PFail)$
$H_{0\ sr:A}$: $A_+(SizeRed) = A_-(SizeRed)$	$H_{0\ pf:A}$: $A_+(PFail) = A_-(PFail)$
$H_{1\ sr:A}$: $A_+(SizeRed) \neq A_-(SizeRed)$	$H_{1\ pf:A}$: $A_+(PFail) \neq A_-(PFail)$
$H_{0\ sr:R}$: $R_+(SizeRed) = R_-(SizeRed)$	$H_{0\ pf:R}$: $R_+(PFail) = R_-(PFail)$
$H_{1\ sr:R}$: $R_+(SizeRed) \neq R_-(SizeRed)$	$H_{1\ pf:R}$: $R_+(PFail) \neq R_-(PFail)$

for different situations in which a defect can occur. In order to keep the discussion within the definition and planning of the experiment, we will only present the details concerning the fault models later in this chapter (Section 6.2).

Based on the response variables, we defined null and alternative hypothesis (Table 6.1) to investigate the effect of changing the factor's treatments on *SizeRed* and *PFail*. Besides statistical hypothesis testing, we will use a table of contrast and liner regression model to observe the coefficients and assess each factor's effect on SART's mean *SizeRed* and *PFail*.

6.1.2 The Experiment Environment and Execution

SART, the generator and the experiment are implemented in the LTS-BT tool [Cartaxo et al. 2008], and the experiment execution is designed to be fully automatic and easily configured. In its current version, LTS-BT is written in Java 1.7 enabling cross-platform execution through a *jar* file, and provides support to execute different techniques for test case generation [de Araújo et al. 2012], prioritization [Ouriques et al. 2013], selection [de Oliveira Neto and Machado 2011; de Oliveira Neto and Machado 2013] and test suite minimisation [Coutinho et al. 2013]. The jar file is executed using the terminal and providing keywords as parameters, and LTS-BT's output is an XML file that can be imported by the TestLink tool.

LTS-BT defines interfaces for all of its techniques, thus adding a new technique is fairly easy. Both SART and the ALTS generator can be executed independently of the experiment, that in turn is configured through text files provided as input for the tool that allows modification of factor's treatments. In order to support automatic statistical analysis, we wrote R scripts, by which data is processed, plotted and tested generating graphic and textual information regarding distribution, intervals, and p-values. Figure 6.2 illustrates the environment for our experiment's execution.

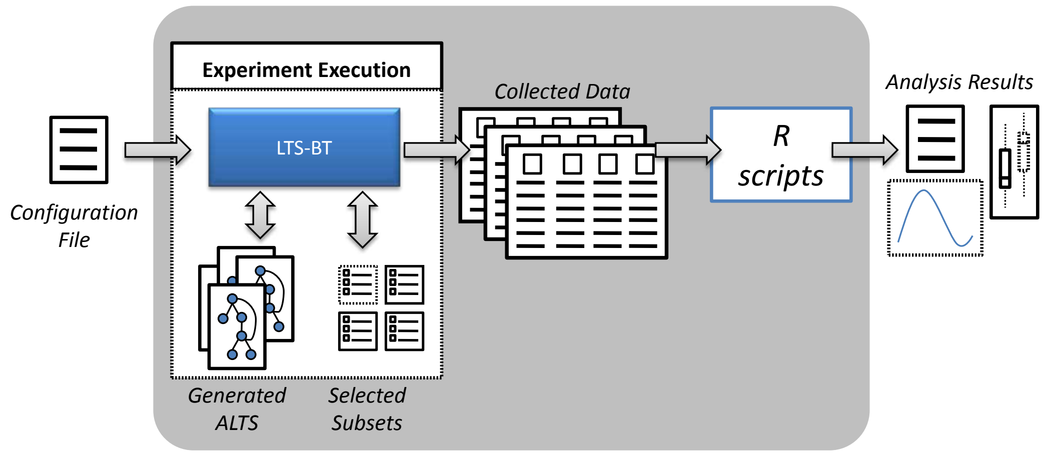


Figure 6.2: Platform where the experiment is executed.

In order to define the factor's treatments for generating the ALTS, a search for repositories of specification models was performed. However, none of the repositories found had a sufficient database of ALTS. Therefore, the values were chosen based on industry's technical reports describing dimensions of use cases [Smith 2003], because the description of main and alternative flows in the ALTS considered in this study is similar to scenarios specified in use cases for test case generation [Cabral and Sampaio 2008]. The author states that a total of 300 scenarios (which will lead to approximately 300 test cases) is a reasonable quantity of scenarios to be described at the system level's use cases. This was the starting point to define our treatments.

Considering our ALTS generator, a test suite with approximately 300 test cases can be

Table 6.2: Table with treatments (or levels) for factors.

Factors	Main Size	Branches	Joins	Path with loops	Additions	Removals
ID	S	B	J	L	A	R
High level (+)	*20*	*150*	*50*	*8*	*10*	*10*
Low Level (-)	*10*	*75*	*25*	*4*	*5*	*5*

obtained if our model has, approximately 150 branches. Based on that number of branches, we started simulations[3] with our ALTS generator to create a sample of models able to yield a test suite containing approximately 300 test cases. Then, we divided the value for each treatment by 2, in order to obtain a big (yet reasonable) distance among dimensions of our models. Ultimately, the values chosen are presented in Table 6.2.

6.2 Analysing Failure Coverage through Fault Models

Fault models represent defects and can be used to validate a technique or gather information regarding the defect detection capability of a test suite, when real defects are not available for the tester [Binder 1996]. Tan et al. introduced [Tan et al. 1997] the concepts of a fault model $F(m)$ for LTS models as a set of all defective LTS implementation of the specification considered. An instance of a fault model $M \in F(m)$ can be obtained by performing mutations, i.e. modifications on the specification models to obtain different behaviour. Here we consider a fault model where a single mutation is performed, that can be either an 'output defect' (an output of a transition is wrong) or a 'transfer defect' (the transition leads to a different state than expected) [Bochmann and Petrenko 1994].

The goal by using mutations *is not* to assess the generated test cases, but to simulate a testing phase with automatically generated artefacts and obtain data to allow the analysis of model-based techniques. We will refer to the process of changing an element of the ALTS implementation model as a *mutation of the model*[4].

To support the fault models, a *fault hypothesis* (FH) is created based on: An extrapolation

[3]By varying the remaining factors: Main sequence size, joins, paths with loops, additions and removals

[4]Note that this approach is not related to traditional mutation testing, instead we refer to mutants as changes in the model since it is conceptually similar to mutation testing.

from past experiences; an assumption that a defect is related to specific circumstances of the specification (e.g. boundary values on conditions) and; an argument (or evidence) on possible errors and the defects they could yield [Binder 1996]. Complementary information such as an expert's opinion, or an operational profile can improve the fault model, and help in estimating more defects.

We will consider an instance M of a fault model as a *faulty implementation* of the delta specification S'. This faulty implementation model (FIM) represents the execution of the SUT, given that the actual implementation of the specification model is not available in this study. As mentioned by Tan et al., any faulty implementation of $F(m)$ must be detected by failing at least one test case.

In order to find failures, we will traverse M collecting traces of execution (i.e. sequence of transitions) to represent the behaviour of the SUT from executing the test cases generated from the specification. Each configuration of factors will need a number N generated delta ALTS to execute SART and consequently a quantity N of FIM. Each selected subset may trigger a failure on the FIM (1 or 0) and the sum of these events divided by N results in *PFail* for each configuration (see Equation 6.2). This process is illustrated by Figure 6.3.

6.2.1 Mutating a Model

In order to be traversed, the mutant needs to be placed in a transition of the ALTS. In practice, estimation for defect detection rely on analysis from the expert involved with the model being used. Here, SART is continuously executed, thus, the process of choosing the location of the mutant needs to be automatic as well. Four different fault hypotheses were defined, resulting in four different fault models. These fault hypotheses were chosen to represent a variety of situations where a defect can occur, considering the modifications performed in a model and possible defects during implementation of the model.

FH1: An output may present an unexpected output due to the wrong interaction of steps performed.

FH2: A code defect causes a step to reach a different state, thus executing a different flow.

FH3: The modification causes a defect resulting in a different output 'near' the modification.

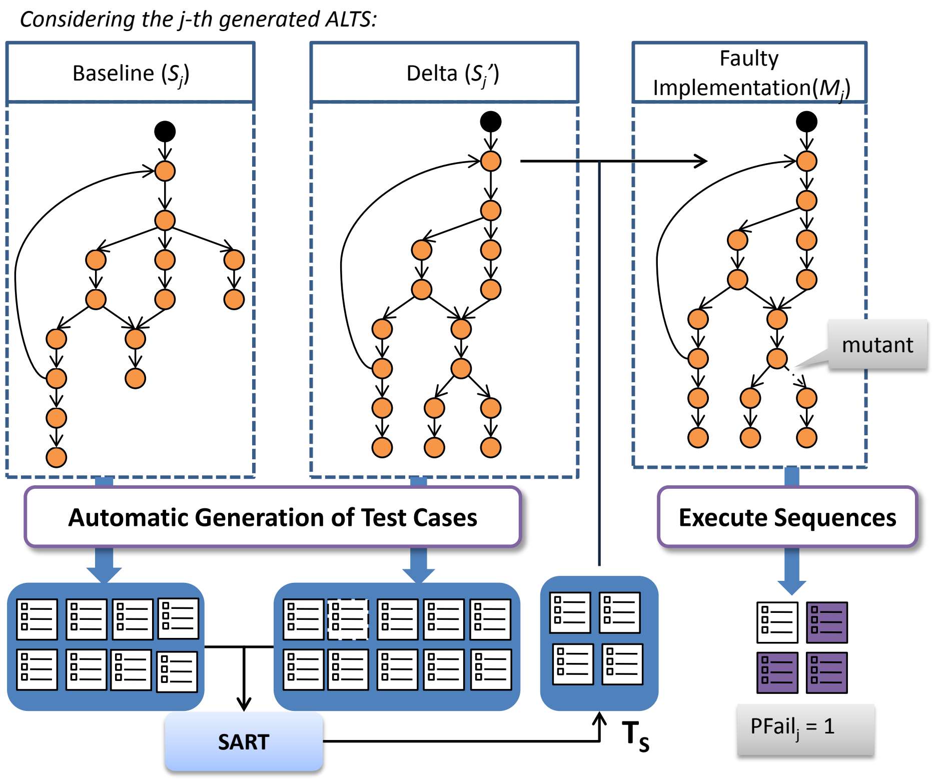

Considering the j-th generated ALTS:
Baseline (S_j)
Delta (S_j')
Faulty Implementation(M_j)
mutant
Automatic Generation of Test Cases
Execute Sequences
SART
T_S
PFail$_j$ = 1

Figure 6.3: Usage of a faulty implementation model (FIM) to measure *PFail*.

FH4: The state modified caused a defect producing a different output in the modified state itself.

Fault hypotheses FH1, FH3 and FH4 are represented by mutations in the output transitions (e.g. by changing of labels to signal the difference), whereas FH2 represents a modification in the flows of the ALTS (for example, an unspecified sequence in the delta model). The goal with these four fault hypotheses is to verify if SART is able to reveal a failure despite the size reduction obtained. FH1 and FH2 are related to the layout of the ALTS (i.e. states and transitions), whilst FH3 and FH4 are related to the location of the modification performed.

In order to find regions of the ALTS that are more likely to trigger defects (i.e. perform a mutation), we rely on the assumption that at the system specification level, a state with several outgoing transitions (branching states) are more likely to cause errors during implementation. When thinking at the code level, these different scenarios can represent different parts of the code that need to interact in order to provide the specified functionality. For example, when writing a text message in a smartphone the user can decide to attach pictures taken from the camera or from a gallery of saved images, hence activating different features of the device. Failures can be caused by wrong interactions when invoking these features and then the system produces an unexpected output, thus we assume that branching states are more likely to reveal failures. Furthermore, more transitions represent more interactions yielding even greater chances of revealing a failure.

To also consider the chance of having a defect anywhere on the ALTS, a roulette wheel is used. However, the wheel is modified so that states that fit the above assumption have bigger slices of the wheel (i.e. a biased wheel), hence increasing their chances of being chosen for mutation. In order to identify and group these branching states we use the *longest path* of all states from the initial state (q_0), defined by $l\_path(q)$ of a state $q \in Q$ is the *number of transitions* in the longest $(q_0, q) - path$ of the ALTS[5].

- $l\_path(q_0) = 0$;
- $l\_path(q_i) = max(l\_path(q_{j1}), ..., l\_path(q_{jn})) + 1$; where $\forall q_j \exists q_i | \exists (q_j, \alpha, q_i) \in T$;

Whenever a state q divides the flow of execution, more states will share the same l_path from the initial state. Those states can then be clustered. Finally, the size of the clusters (i.e. its number of states) will determine the slice of the wheel and, hence, the probability of choosing a random state from that cluster. An example can be seen in Figure 6.4.

Both ALTS of Figure 6.4 were generated with the same configuration. However, the different layout of flows implies that both implementation should have different distribution of mutants. The model is divided into clusters according to the $l\_path(q)$, and these clusters are then ordered by size. Bigger clusters have bigger slices of the wheel. For Figure 6.4 (a), most division of flows happen near the leaves of the ALTS, yielding bigger clusters for $l\_path(q)$ equal to 9 and 10. On the other hand Figure 6.4 (b) has a more balanced distribution of flows, which reflects a more balanced wheel than Figure 6.4 (a). After the wheel spins, a random state from the chosen cluster (slice) is selected. The result of this process is a state where the mutation will be performed.

[5]To avoid infinite values, the loops are not traversed when calculating the path

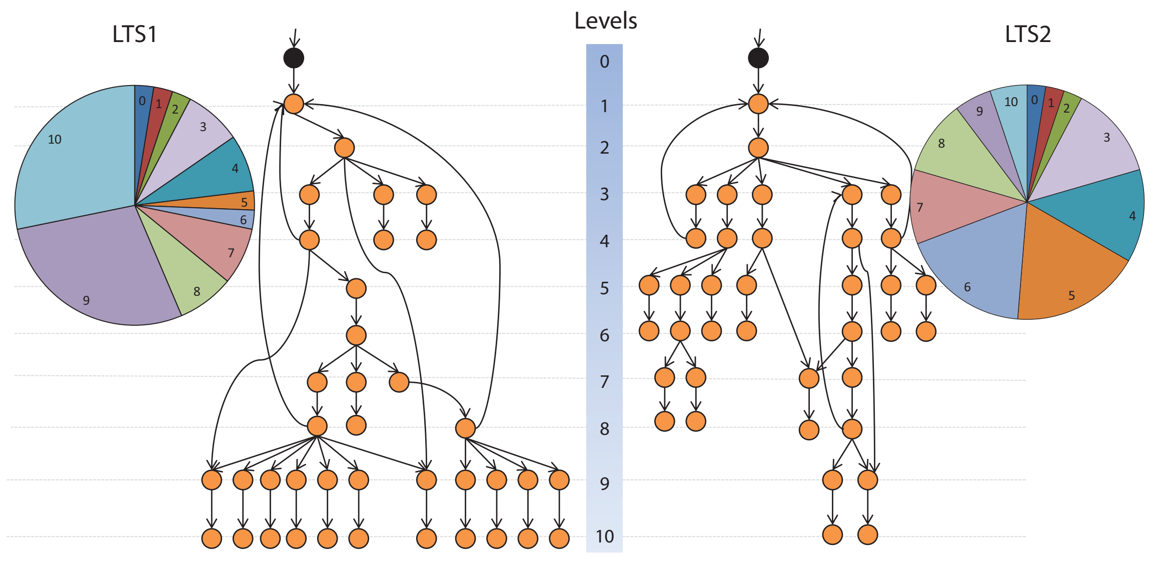


Figure 6.4: Roulette wheels obtained from different graph configurations.

6.2.2 Fault Models

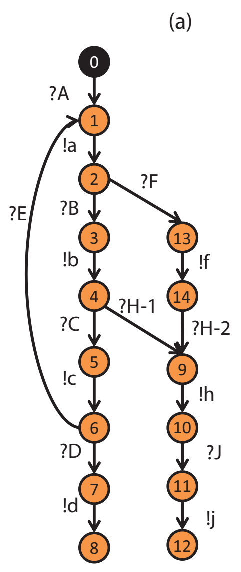
Each fault model will be explained through the example provided in Figure 6.5. Four different fault models were considered in this experiment (one for each fault hypothesis) and the mutants are indicated by the transitions with labels changed to '$!x$' (Figure 6.5 (c)).

FM1: Binder's Fault Model

According to Binder, a defect is observed at the system specification level through the failure that exposed it [Binder 1999]. The tester usually signals a failure when the output produced by the system differs from what was expected. Therefore, based on FH1 we change the label of a chosen expected output to represent the defect. The choice of state is based on the biased wheel, noting that only output transitions are eligible for the mutation.

As an example, consider that the biased wheel chooses state 5 resulting in creation of FM1 of Figure 6.5 (c). The test suite from Figure 6.5 (b) shows that 4 test cases will not have paths in FM1 (TC_1, TC_4, TC_5, TC_6). Therefore, if SART selects *any* of those 4 test cases, the failures will be triggered, since the system would produce output '$!x$' instead of the expected '$!c$'.

FM2: Single-State-Transition Fault Model



(b)

Action	Output
?A	!a
?B	!b
?C	!c
?D	!d

TC_1

Action	Output
?A	!a
?B	!b
?H-1	!h
?J	!j

TC_2

Action	Output
?A	!a
?F	!f
?H-2	!h
?J	!j

TC_3

Action	Output
?A	!a
?B	!b
?C	!c
?E	!a
?B	!b
?C	!c
?D	!d

TC_4

Action	Output
?A	!a
?B	!b
?C	!c
?E	!a
?B	!b
?H-1	!h
?J	!j

TC_5

Action	Output
?A	!a
?B	!b
?C	!c
?E	!a
?F	!f
?H-2	!h
?J	!j

TC_6

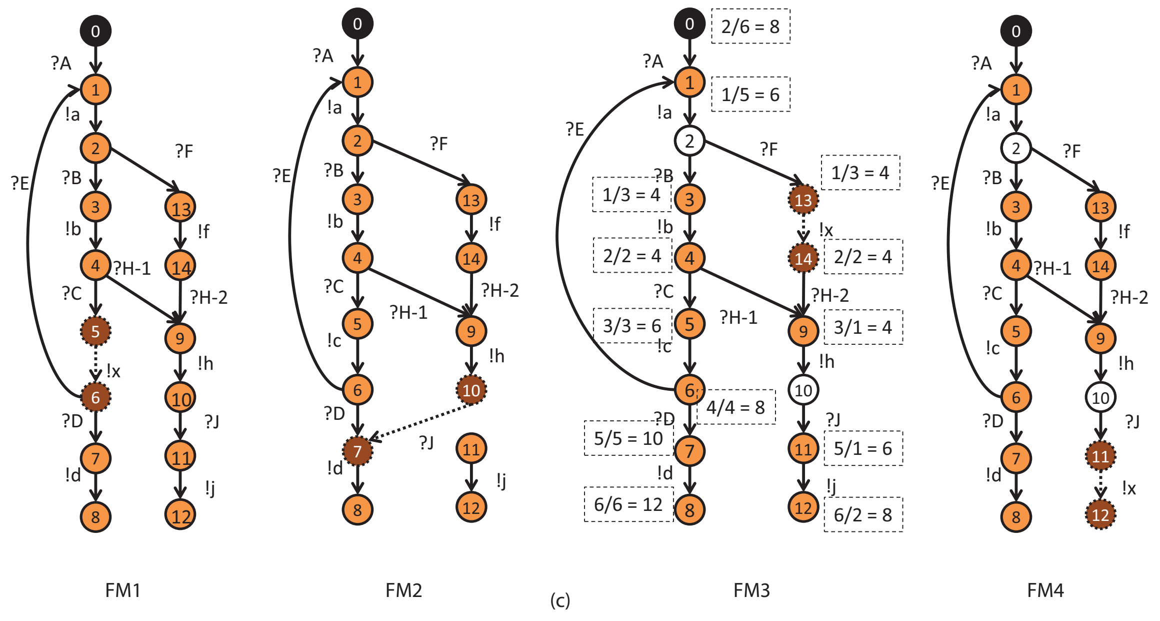


Figure 6.5: (a) Example of a generated ALTS, (b) the respective generated test suite and (c) fault models for each fault hypothesis.

Failures can also be triggered when the execution reaches an unexpected state, thus an output specified in the system (i.e. from another state) is produced but differs from the expected output of the test case [Cheng and Jou 1990]. For example, imagine that a transition removed from the ALTS did not have its correspondent code part removed properly and is still traversed when executing the SUT [Korel et al. 2002].

In our case, the FIM is obtained by spinning the biased wheel and changing the destination of one input transition of the chosen state (thus named single-state-transition). For example, consider that state 11 was chosen resulting in the FIM labeled FM2 from Figure 6.5 (c). The destination of transition $?J$ is modified to state 7 (randomly chosen) and when executed with FM2, any of TC_2, TC_3, TC_5 and TC_6 will signal a different output, hence triggering a failure.

FM3: Mutation Near Modifications

Based on FH3 we defined a fault model to simulate defects near the modifications performed (algorithm of Figure 6.6). To find regions near the modifications, the distances between each modified state and the remaining states are calculated (line 5). In order to calculate the shortest distance, we traverse the graph ignoring the direction of transitions.

For example, the distance between states 2 and 1 is defined by $shortestDistance(2, 1) = shortestDistance(1, 2) = 1$. Otherwise, we would not be able to reach all ancestors of a modified state, leading to an invalid distance value. In addition, if the same distance is found among states (lines 12 and 13), we decide randomly which state is chosen. In the end, the state with the *smallest total distance* is chosen.

Note that we traverse all states once for each modification performed (lines 3–5), and each of these $|Q| \times |Q_m|$ times the method $shortestDistance(m, q)$ is called, in which a simple DFS search can be performed to determine the shortest distance between states m and q. Worst case scenario $shortestDistance(m, q)$ executes in polynomial time with complexity $O(|T'_{tr}|)$. Therefore, our algorithm in Figure 6.6 takes time $O(|Q| \times |Q_m| \times |T'_{tr}|)$.

In our example of Figure 6.5 (a), two modifications were performed: A removal in state 2 and an addition to state 10. The calculated distance and their total sum (dotted boxes near each state) is presented in FM3 of Figure 6.5 (c). Since states $3, 4, 9, 13, 14$ have the same total distance ($d = 4$), a random state among them is chosen. Considering that State 13 is

```
Input  : The FIM (e.g. FM3) and the set of modified states $Q_m \subseteq Q$
Output : The nearest state to all modifications performed.

     FM NEAR MODIFICATIONS($F$, $Q_m$)
 1   For all $q \in Q$ do
 2       $d[q] \leftarrow 0$
 3   For all $m \in Q_m$ do
 4       For all $q \in Q - Q_m$ do
 5           $D[q] \leftarrow d[q]$ + shortestDistance(m,q)
 6   chosenState $\leftarrow q_0$ //Begins the search with the initial state
 7   minDistance $\leftarrow d[q_0]$
 8   For all $q \in Q - Q_m$ do
 9       if $d[q]$ < minDistance then
10           minDistance $\leftarrow d[q]$
11           chosenState $\leftarrow q$
12       else if $d[q]$ = minDistance then
             //Tie breaks between equal distances
             //   are decided randomly
13           chosenState $\leftarrow$ RANDOM(chosenState, $q$)
14   return chosenState
```

Figure 6.6: Algorithm to find the state nearest to all the other modified states.

the output for our algorithm, then we perform a mutation on its output transition. In the end, the test cases TC_3 and TC_6 will trigger a failure when executed with FM3.

FM4: Mutation at the Modifications

In a more optimistic scenario one would assume that the defects are found in the modification themselves. The problem in handling this assumption is that we cannot directly map transitions to their respective code part(s). Nevertheless, when considering the goal of covering all modifications with the selected subset, FH4 becomes a hypothesis worthy of investigation. Therefore, we defined an algorithm presented in Figure 6.7 to create instances of faulty implementation models where the mutant is on a modified state of the ALTS.

Representing an optimal scenario, the mutation is performed on one of the modified states that, in turn, is chosen randomly (line 2). For instance, consider that State 10 is chosen, we then obtain FM4 of Figure 6.5 (c) and change its output transition. Consequently, the test cases that will trigger a failure become TC_2, TC_3, TC_5 and TC_6.

```
Input :The FIM (e.g. FM4) and the set of modified states $Q_m \subseteq Q$
Output:One of the modified states, chosen randomly

    FM AT MODIFICATIONS($F$,$Q_m$)
    //Chooses a random state among the set of modified states
    chosenState ← RANDOM($Q_m$)
    return chosenState
```

Figure 6.7: Algorithm that randomly chooses one of the modified state.

6.3 Concluding Remarks

This chapter discussed the evaluation methodology for our proposed selection strategy. We presented the definition, planning and execution of an experimental study to analyse SART's performance based on percentage of size reduction (*SizeRed*) and probability to trigger failures when executing the selected subset (*PFail*). There are two main challenges when designing this experiment, both due to the lack of specification models for our test process and due to the lack of defect data to measure rate of defect detection.

The former was solved through SBMTE, where our ALTS generator creates large samples of models to provide as input to our test process. Consequently, the values chosen to define the number of patterns generated are the factors for this experiment, and the goal is investigating if specification of different patterns significantly affects SART's performance, i.e. if the technique is able to select test cases exercising modified regions regardless of the type of model provided as input (e.g. big models, with several branches, small additions).

In turn, we used mutants to overcome lack of defect data and allow visualization of SART's behaviour when handling four different scenarios where a defect can be triggered. A full factorial design is used to enable analysis of all interactions between factors, yielding a thorough investigation of SART's selection capability. Despite the large number of possible combination of factors, the experiment executes automatically in a platform integrated with other tools (LTS-BT and R).

The integrated and automatic execution allows other researches to reproduce our experiment, or execute it with different treatments for factors by simply editing a configuration file. In fact, that allowed us to execute a simple comparative study between SART and two other test case selection strategies: A technique proposed by Cartaxo et al. [Cartaxo et al. 2011] and a random test case selection. By comparing SART with each technique, we can observe if our adaptation of the similarity function is better than the other two when targeting specific regression testing needs, such as coverage of modifications and regression defects.

Chapter 7

Results and Analysis for SART

After definition and planning of the experimental study, we executed the experiment and collected the data for analysis. Details regarding the results (graphics, p-values, the effects of each factor, etc.) are presented in this chapter along answers to our research questions. In addition, we performed an exploratory study where SART is executed with industrial specification models, thus yielding access to information regarding test case execution and defects found. That allows comparison between our automatic selection of test cases and a traditional approach where a subject (e.g. a tester) manually selects test cases.

7.1 Experimental Study

Our hypothesis in the experiment states that there is a strong relationship between the model type used to generate test cases and SART's performance. Therefore, the analysis is focused on the effect that a factor has on the dependent variables. Figure 7.1 presents the results for the (a) effectiveness of reduction in size (*SizeRed*) and (b) probability of revealing failures (*PFail*). First, the intervals for *SizeRed* and *PFail* are presented and discussed, then we take a closer look at p-values and the effects of main and confounding of factors. As mentioned in the previous chapter, we will use S, B, J, L, A and R to respectively address the factors main size, branches, joins, loops, additions and removals. Also, the plots and intervals for interactions of the factors (e.g. SBJ, $BLAR$, etc.) were removed from figures, since the observed effect for them was small.

Note that our percentage of size reduction is high, reducing up to 96% of the test suite

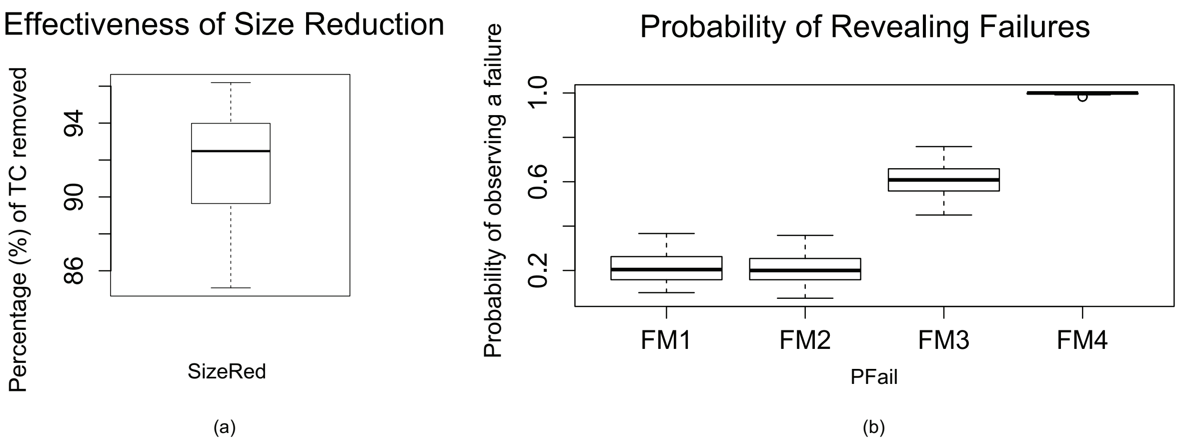


Figure 7.1: Boxplot for measuring (a) *SizeRed* and (b) *PFail*.

size. Despite the variation, the results indicate that an average of 92% (standard-deviation of 3.11%) of size reduction can be achieved by executing SART. The intervals for each main effects presented in Figure 7.2 show that the variation of the treatments in almost all configurations would not significantly increase the size of the selected subset (i.e. the intervals overlap). The only exception is the branches factor and since the difference is small (roughly 1.5%) we conclude that SART's size reduction capability is *robust*.

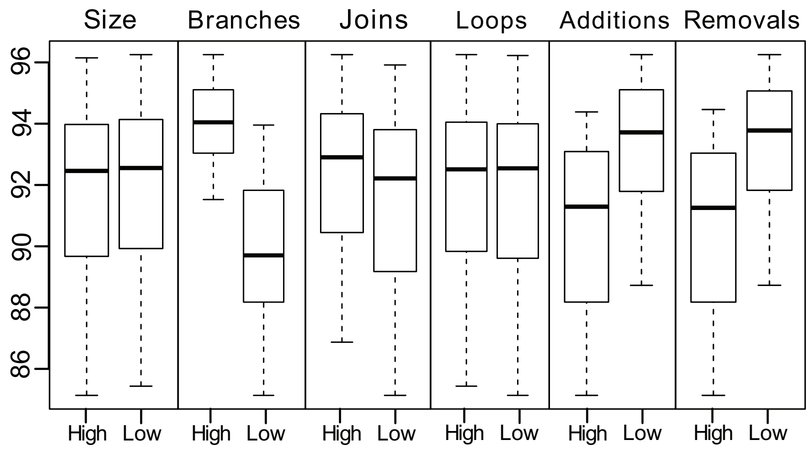
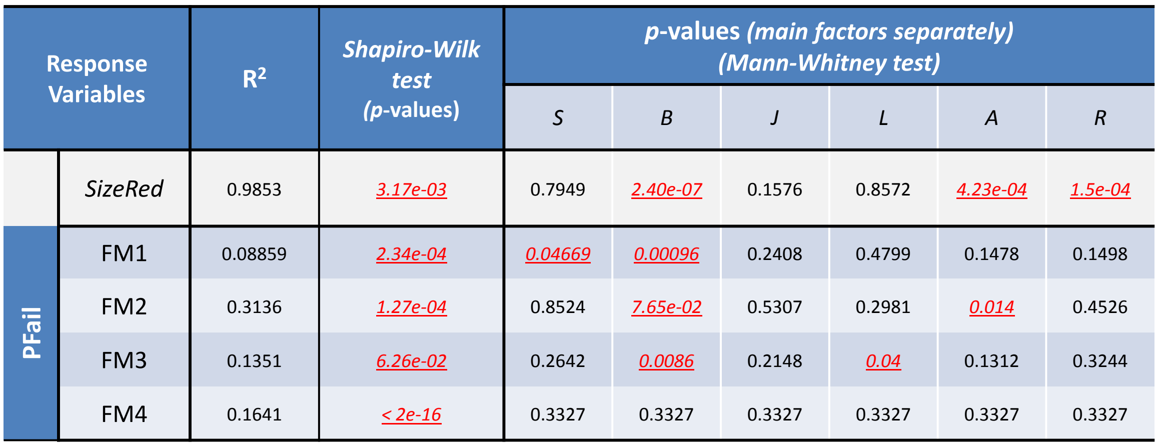


Figure 7.2: Main effects of each factor's treatment on *SizeRed*.

Figure 7.1 (b) presents the results for *PFail* representing the probability that SART will select a subset that triggers failures. The main aspect to remember of the four fault models is that $FM1$ and $FM2$ are based on the distribution of states and transitions among the delta

Table 7.1: Table with p-values and correlation (R^2) between factors and the response variables. The p-values smaller than $\alpha = 0.05$ are underlined.



Response Variables		R²	Shapiro-Wilk test (p-values)	p-values (main factors separately) (Mann-Whitney test)					
				S	B	J	L	A	R
	SizeRed	0.9853	3.17e-03	0.7949	2.40e-07	0.1576	0.8572	4.23e-04	1.5e-04
PFail	FM1	0.08859	2.34e-04	0.04669	0.00096	0.2408	0.4799	0.1478	0.1498
PFail	FM2	0.3136	1.27e-04	0.8524	7.65e-02	0.5307	0.2981	0.014	0.4526
PFail	FM3	0.1351	6.26e-02	0.2642	0.0086	0.2148	0.04	0.1312	0.3244
PFail	FM4	0.1641	< 2e-16	0.3327	0.3327	0.3327	0.3327	0.3327	0.3327

model, whereas $FM3$ and $FM4$ are based on the modifications performed on the baseline model. $FM1$ and $FM2$ indicate that SART has a low probability (around 21%) of triggering failures when defects are more likely to appear in branches, joins and loops.

However, $FM3$ and $FM4$ have an increased probability of triggering failures since both $FM3$ and $FM4$ consider the modifications performed in the ALTS (unlike the mutants in $FM1$ and $FM2$). Whenever the mutant is near a modification ($FM3$), SART has a probability of 60% of triggering a failure. Despite illustrating an unrealistic scenario, $FM4$ indicates that SART covers all the modifications. In some cases of removed transitions (outliers in $FM4$ presented in Figure 7.1), SART did not cover the mutated transition, but instead covered the state where the removal happened. That happens when several test cases are equally similar to an obsolete test case, leading to a random choice during tie breaks.

Now we will investigate the p-values and the effects of the factors to assert the hypothesis of our experiment. We used R scripts to obtain a linear regression model (*lm*) based on the full factorial design and then began analyzing the resulting coefficients and p-values. Table 7.1 shows a summary of the *lm* model.

Initially, we turned to the adjusted coefficient of determination (R^2) to see how well the data points fit the models. In our case, none of the *PFail* results present a high R^2 indicating that the respective *lm* models fit the data poorly. That makes sense since it is hard to find accurate coefficients to predict *PFail* given that the mutants can be very unpredictable due to the biased wheel ($FM1$, $FM2$) or the random choice of modifications ($FM3$, $FM4$).

In summary, it is risky to assert the hypotheses with respect to *PFail* by considering the coefficients of the *lm* model, hence limiting the conclusions of *PFail* to the evidence that SART has an increased probability of revealing failures caused by modifications. As a consequence, we will then focus our investigation on *SizeRed* since it showed a high R^2. First, a Shapiro-Wilk test to verify normality of samples revealed that none of the results were normally distributed. Consequently, we performed a Mann-Whitney non-parametric test to assert the *SizeRed* hypotheses (presented initially in Table 6.1).

By considering a level of significance $\alpha = 0.05$ we are able to reject null hypotheses $H_{0\ sr:B}$, $H_{0\ sr:A}$, $H_{0\ sr:R}$, implying that variations on branches, additions and removals significantly affect SART's percentage of size reduction. On the other hand, we could not reject the null hypotheses for the remaining factors ($H_{0\ sr:S}$, $H_{0\ sr:J}$, $H_{0\ sr:L}$) leading to the conclusion that SART achieves significant size reduction regardless of the model's number of joins, loops and main sequence size.

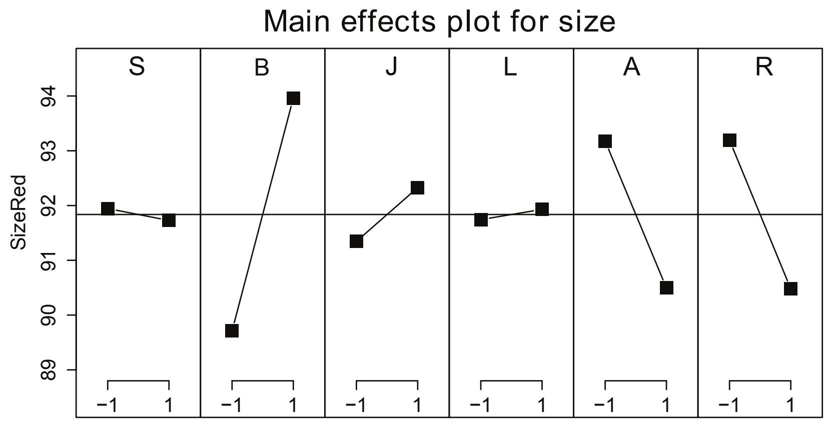


Figure 7.3: Mean for each of the main factors and their treatments.

A closer look at the visual data strengthens confirmation drawn from hypotheses testing, where some of those factors have a higher effect than others. Figure 7.3 show the effect sizes for each of the main factors. As can be seen, the most important factor is Branches (B) followed by Addition and Removals (A and R). Also, the lowest treatments (-1) of A and R provided a bigger size reduction. In turn, we concluded that S, J and L do not have a significant impact on the results because their coefficients are close to 0. The importance of Branches can also be seen on the interaction plots of Figure 7.4.

The diagonal spaces in the grid plot indicates *(i)* the factors associated with the x-axis along the same column and *(ii)* two lines in the same row (each line indicating a treatment of respective row factor). The key aspect to observe is whether the solid black and dotted red lines are parallel to each other. For example, the plot labelled $S \times B$ contains averages for low and high treatment of the B factor (x-axis) for both treatments of S. The red dotted line represents the low treatment of S, whilst the solid black line represents its high treatment. For $S \times B$, the lines show that *SizeRed* is basically the same for both low and high treatments of B (i.e. the lines are parallel). Thus, the interaction $S \times B$ does not have a significant impact on *SizeRed*.

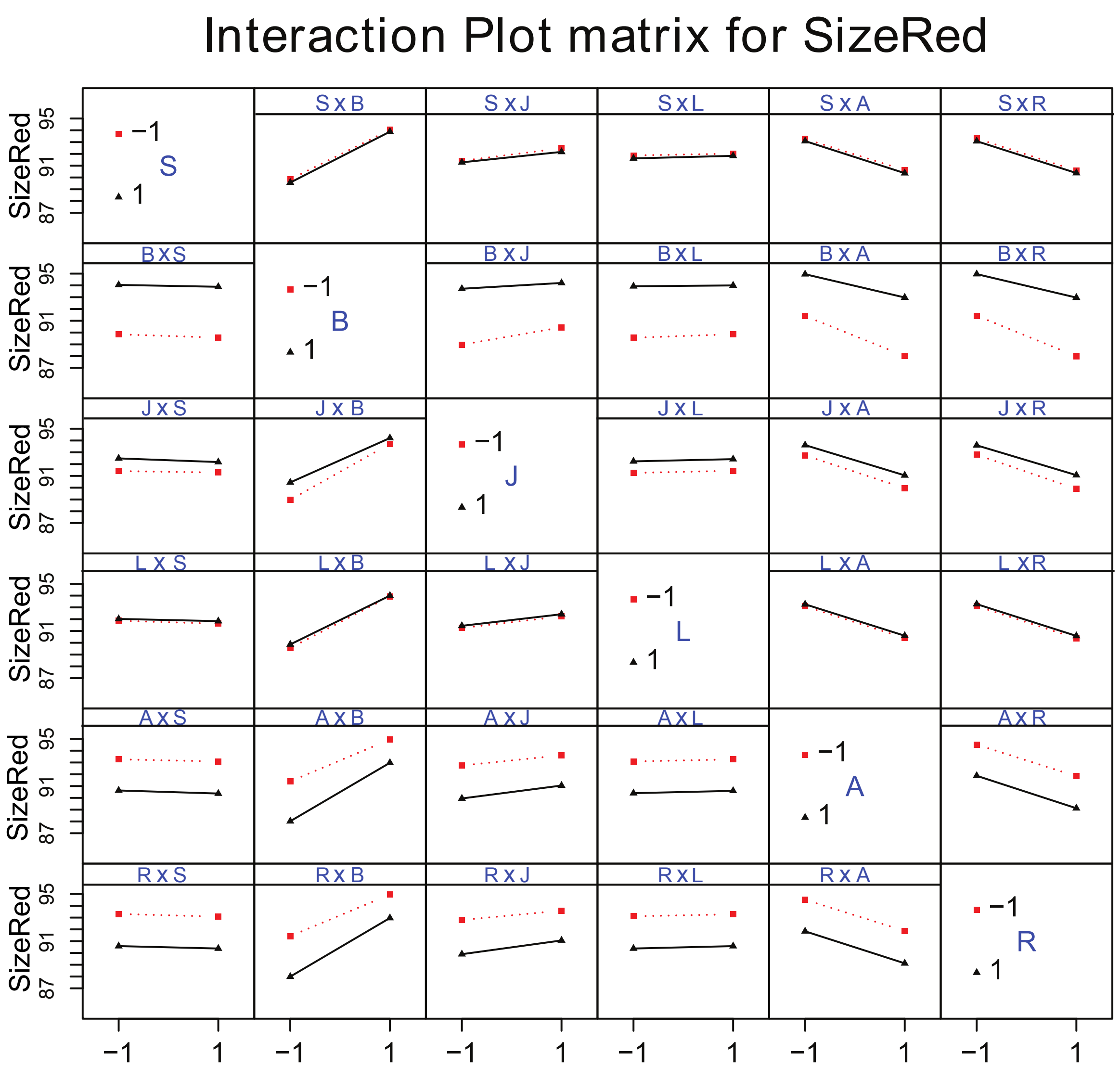


Figure 7.4: Means for interacting 2-factors.

On the other hand, for column B, the dotted and solid lines for most interactions (J, A, R) *are not parallel* to each other, indicating a strong effect from the interacting factors. Moreover, observing row B closely (i.e. the second row in the grid) we see that a high treatment of B in all interactions results in a higher size reduction, indicating that SART shows a good size reduction for test suites exercising several alternative flows that were modified. A similar conclusion can be obtained from factors A and R (rows A and R from Figure 7.4). But unlike factor B, higher treatments of A and R indicate a smaller size reduction. Therefore, SART achieves better size reduction when *less* modifications (additions and removals) are performed on the model.

The effect on average *SizeRed* for branches is bigger because each branch indicates that there is one new scenario to be covered by a test case. The higher the number of branches the bigger the test suite becomes, hence allowing greater size reduction. In turn, the effect sizes for additions and removals is explained by the fact that SART selects test cases until all modifications are covered, thus performing less modifications results in a smaller subset.

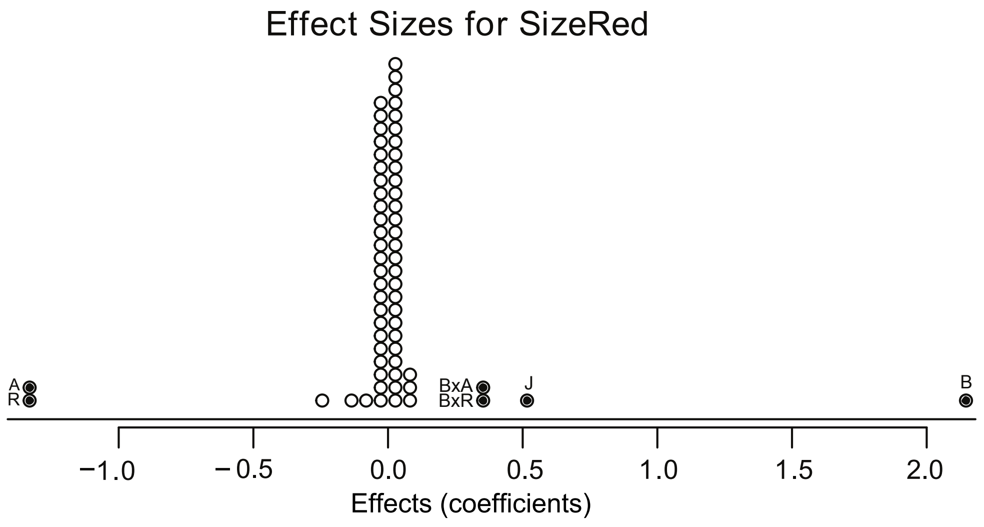


Figure 7.5: Plot of all effects for a full-factorial 2^6 design. Most important effects are B, A and R. The values for each effect can be seen in Appendix B.

By plotting the effects obtained for all 64 interactions (Figure 7.5), one can see how factors B, A and R stand out when compared with all the other interactions (including $3, 4, 5$ and $6 - factors$ interaction). The analysis of all p-values (see Appendix B) indicates that interactions with more than 2 factors are not significant, unless involving the factor B (e.g.

$B \times A$ and $B \times R$).

In summary, the statistical evidence shows (with a level of confidence $\alpha = 0.05$) that there is a strong relationship between the generated models and SART's percentage of size reduction, more specifically the number of branches and modifications on the model. In addition, the confidence intervals and the low standard deviation compared to the average $SizeRed$ indicate that the size reduction is also robust. Unfortunately, the hypothesis could not be asserted with respect to *PFail*, but the analysis reveals that the technique is more likely to reveal failures triggered by defects related to model modifications, showing that SART still reveals failures despite the significant size reduction.

In conclusion, SART is able to achieve significant size reduction and still cover all model modifications and reveal failures. However, the results from $FM3$ and $FM4$ still indicates that the selection can be improved to increase *PFail*. During this experiment, SART required an average of 200 milliseconds in each execution (to reduce hundreds of test cases). Given that this work targets a higher level of abstraction of the system, most test cases are manually executed, which implies that execution of all test cases is a tedious and time-consuming task. The size reduction provided by SART can reduce this time when each test case has the same cost of execution. However, further studies using cost models are recommended to precisely assess the cost efficiency of SART in practice.

7.2 Comparative Study

Next, we present a study to compare the results from adapting the traditional counting function (CF) proposed by Cartaxo et al. for our regression testing context (RQ4 presented in Chapter 1). In other words, we wanted to investigate whether it was worthwhile to change the CF to analyse test suites from different software versions, or would a straightforward application of CF on T' result in a representative subset for regression testing. We also compared SART with a random selection (RDM) technique because a random selection tend to present satisfactory transition coverage whenever size reduction is high (below 80%) [Cartaxo et al. 2011].

Therefore, our goal is to assess whether CF and RDM are better than SART in targeting regression test issues, such as coverage of new and modified parts of the system's specification. Figure 7.6 presents an overview of this second experiment. We used the same set of generated ALTS from the previous experiment as input for this study. SART, CF and RDM were executed to select subsets from the automatically generates test suite. Thus, this experiment has the same input and instruments used in the previous experiment (Section 7.1). The main difference lies in our analysis.

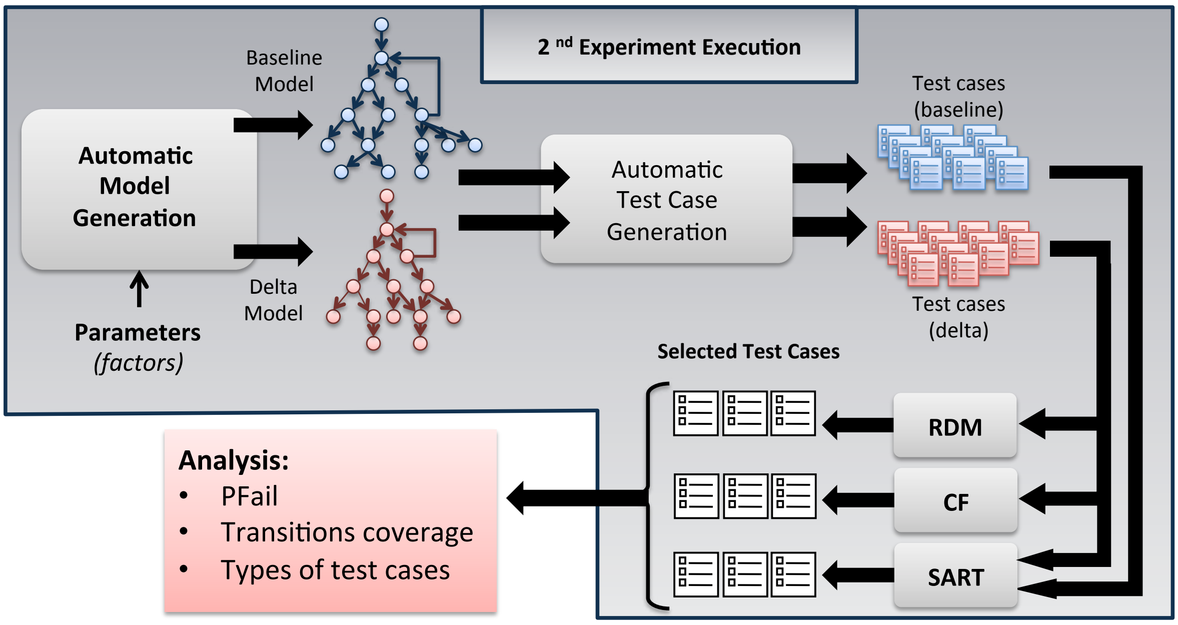


Figure 7.6: Overview of our comparative study.

We analyse all three techniques with respect to *PFail*, *coverage of model's transitions* and *selected classes of test cases* (reusable and targeted). Analysis with *SizeRed* is not suitable for this study because, when using CF and RDM, the tester establishes the desired size ration, since the goal with both techniques is to select a subset of test cases suitable for time and budget constraints. For instance, if the budget available allows execution of only 40% of the entire test suite, both techniques will produce a subset with only 40% of the test cases. SART on the other hand, first establishes the minimum subset size to cover all modified regions of the model, and then if the resulting subset is still big, the tester can apply other strategies to reduce even more the number of test cases. In order to obtain subsets of the same size and provide a fair coverage comparison, the size ratio criteria provided for CF and RDM was the size of the subset provided by SART.

Figure 7.7 presents the results for transitions coverage, indicating that all subsets have

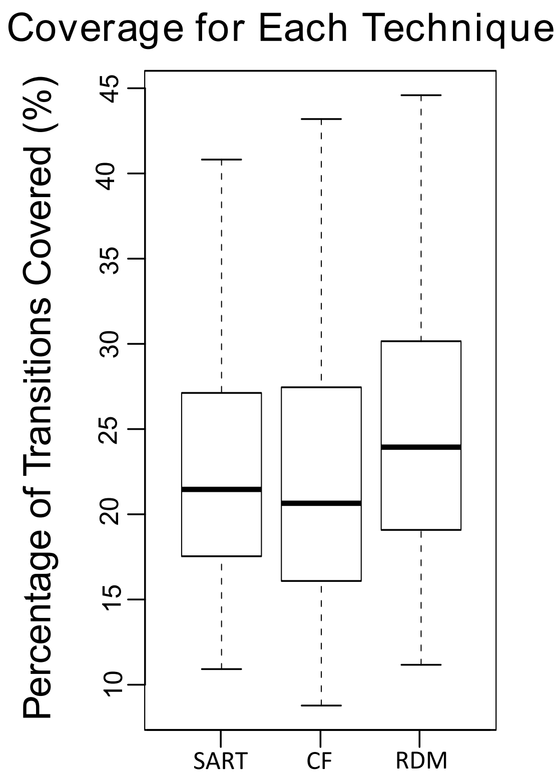


Figure 7.7: Percentage of ALTS's transitions covered by each selected subset.

a very similar percentage of transitions coverage, i.e. the transitions of the reduced subsets from SART, CF and RDM correspond, respectively, to 22.42%, 22.18% and 24.86% of the delta ALTS model. As expected, RDM presents the highest transitions coverage due to the high percentage of size reduction (around 92%). For RDM, the chances of selecting the same transition is smaller if *any path* of the model is a candidate, whereas SART and CF have specific selection goals leading to less options, hence a more restrictive coverage.

These results become clearer as we analyse *PFail* for each technique (Figure 7.8). Note that all techniques have similar performance for FM1, FM2 and FM3 due to the randomness involved in determining the mutants in these three fault models. However, the benefits of using a selection criterion based on modifications becomes evident when analysing FM4, where RDM and CF have an average probability of 33.8% and 37% (respectively) of triggering defects at the modifications, whereas SART's probability is 100%.

Figure 7.9 presents the average percentage of targeted[1] and reusable test cases selected by each technique. SART has a balanced proportion where the test suite is composed by

[1]Targeted test cases refer to both new-specification and retestable test cases.

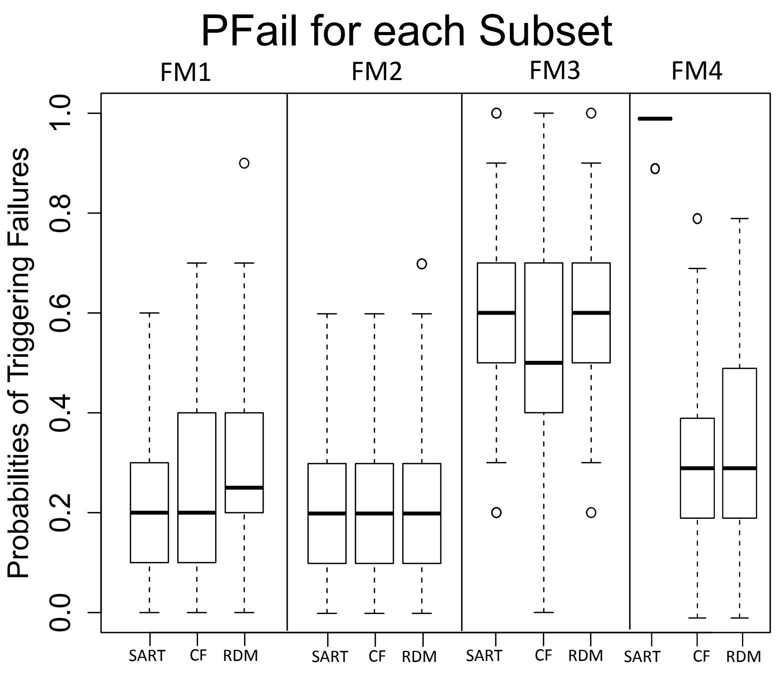


Figure 7.8: *PFail* for each fault model and selected.

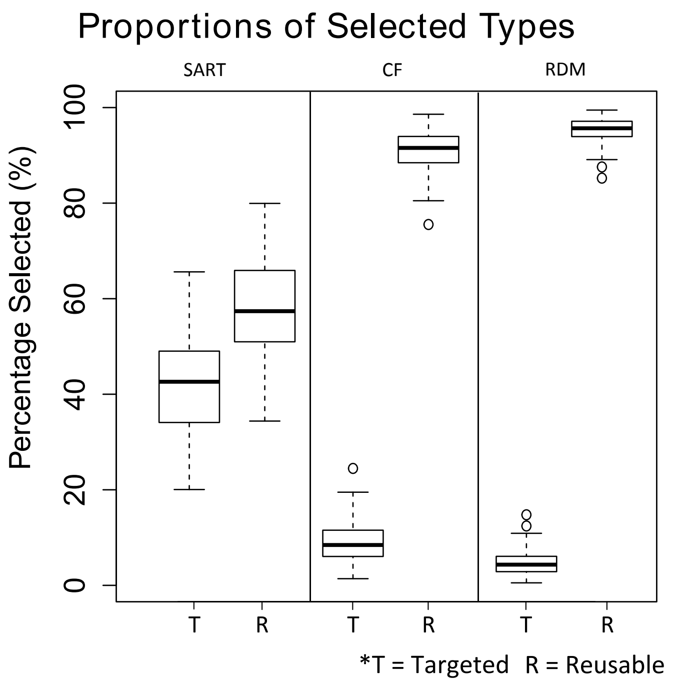


Figure 7.9: Percentage of targeted and reusable test case selected by each technique.

an average of 48% targeted and 52% reusable test cases. CF and RDM, on the other hand, have very low percentages of targeted test cases (respectively, 8% and 4%), indicating that coverage of modifications is lower. Considering that regression testing should be executed with modification-traversing test cases, the evidence shows that SART is more suitable than CF and RDM.

Therefore, despite better transition coverage, the analysis shows that it is very risky to use RDM for regression testing instead of SART, that, in turn, guarantees modifications coverage and increased chances of triggering regression defects. Moreover, the results show that using the CF alone on a regression testing context does not address its specific needs. The time required to execute all techniques was less than 1 second, in fact each execution took between 50 to 200 milliseconds to reduce hundreds of test cases, hence comparison between execution time was unnecessary. In conclusion, our adaptation of the counting function gave SART the boost to achieve an automatic selection of test cases exercising the modified regions of a specification model, providing more confidence whenever less test cases are necessary to meet resource constraints of a progressive regression test process.

7.3 Evaluation with Industrial Specification Models

In complement to the evidence gathered on both experiments, we need to observe benefits of using SART in an industrial context. Thus, we developed an exploratory case study to use SART with industrial artefacts. This study is part of a collaboration between practitioners and researchers in academia where an MBT process is used to test a software system that collects and processes biometrics information. Unfortunately, only a limited number of specification models were available to our research, hindering execution of experiments. Albeit only a small investigation could be performed, the results are promising.

Figure 7.10 presents an overview of our exploratory case study. A total of four specification models were modified to meet new requirements, and the test cases were automatically generated from each specification model. Given that there was not enough time to run all generated test cases, a participant from our research group used her own expertise and knowledge about the SUT to select, and, subsequently, execute the test cases. We then applied SART to compare the pros and cons of testing the modified use cases.

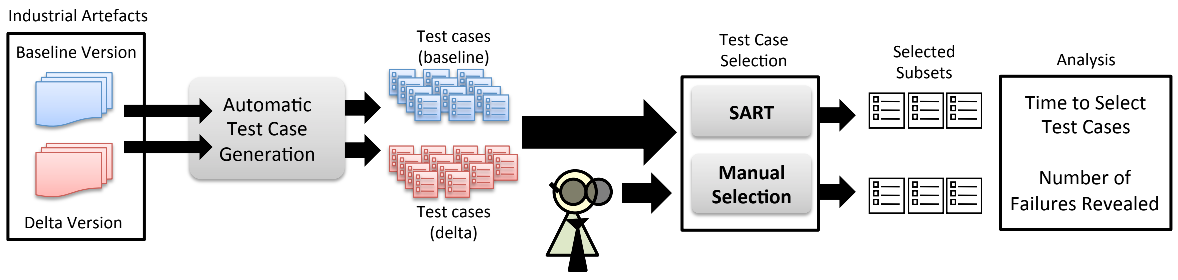


Figure 7.10: Overview of our case study with industrial artefacts.

Due to confidentiality agreements, we are not able to present the industrial models used. Instead, we present the number of states, transitions and modifications in each generated ALTS to illustrate their size (Table 7.2). Model 1 and Model 2 are small, whereas Model 3 and Model 4 are bigger and have more complex interactions (e.g. more branches, loops and joins in the ALTS).

A summary of data regarding test cases' selection is presented in Table 7.3. As can be seen, SART was able to select a test suite that is either equal or smaller than the one selected by the tester. Also note that SART was able to select the test cases that failed when testing the SUT, thus the same defects were found.

Consequently, SART was able to perform better than the manual approach, since less test cases were selected and still all the failures were observed during the test. Moreover, the main benefit is related to the selection process. The subject required a significant amount of time (approximately 4 hours) to select a subset of test cases, and SART, on the other hand, required an average of 200 *millisecond* to select each subset.

Besides being a very time consuming process, the manual selection is laborious and te-

Table 7.2: Industrial artefacts characteristics (size and number of modifications).

Specification Models	Baseline Version		Delta Version		Modifications		
	States	*Transitions*	*States*	*Transitions*	*Removals*	*Additions*	***Total***
Model 1	11	11	10	9	3	1	**4**
Model 2	22	24	20	21	5	2	**7**
Model 3	32	33	28	28	6	1	**7**
Model 4	32	38	22	25	13	0	**13**

Table 7.3: Comparison between SART and a manual selection using industrial artefacts.

Specification Models	Number of Generated Test Cases	Selected Subset Size		Number of Test Cases that Failed	
		Manual	*SART*	*Manual*	*SART*
Model 1	3	3	3	0	0
Model 2	14	10	6	1	1
Model 3	16	8	5	5	5
Model 4	67	7	7	4	4

dious, since most test cases have very similar sequences that can be bewildering. In addition, relying on a tester's expertise can be risky, since human factors such as experience, motivation, etc. can compromise the outcome leading to an error prone selection. Usage of an automatic strategy yields more consistent results, whereas the effort lies in deciding whether the technique would be appropriate for the model being used. For example, imagine that we are interested in branch coverage, instead of modifications. In that case, a different technique would be more appropriate instead of SART.

7.4 Threats to Validity

In order to overcome the difficulties related to our evaluation methodology, several threats to validity had to be addressed. One of the main concerns is the construct validity of our first experiment, given that its execution relies on automatic generation of models. However, the definition of patterns and the constraints to generate consistent ALTS hold the integrity of the generated models. Also, the model type used for model generation and the resulting test suites comply with similar formats used in literature [Bertolino et al. 2008; Cabral and Sampaio 2008; Cartaxo 2011].

One concern regarding internal validity is the choice of parameters to generate models, in turn related to: *(i) The representativeness of the sample* (i.e. space of models) obtained, and *(ii) the generalization of the results*. To obtain a fairly general and representative sample, it would be necessary to obtain parameters from a wide sample of real models. The choice of values for parameters in this study was based on industry reports describing dimensions (sizes, number of steps and branches) of use cases [Smith 2003]. Thus, the sample was able to achieve a representativeness and is general to the ALTS specification models. Our methodology and generator are also applicable in other techniques, since graph models are widely used for regression test case selection [Biswas et al. 2011; Tamimi and Zahoor 2011; Yoo and Harman 2012]. Eventually, results from different selection strategies in graph models (control/program/system dependencies, clusters, graph walk, etc.) can be compared by using our methodology.

Another difficulty with this study is the evaluation of SART's performance regarding defect detection capability. Mutations of the model, fault hypotheses and fault models were used to address the lack of implementations and defect data experienced in many empirical studies [Andrews et al. 2005]. Despite hindering definitive conclusions regarding SART's defect detection capability, the analysis provides statistical evidence that the technique selects test cases exercising the modifications of the specification models, regardless of their location in the model.

Regarding external validity, the size of the sample used, the p-values for $SizeRed$ and our comparative study allows generalization, to similar contexts, of our results regarding SART's percentage of size reduction (unlike the results with *PFail*). In order to take generalization to the next level, an experiment with specification-based regression test selection should be performed to compare performance of different selection strategies, such as dependence analysis [Korel et al. 2002; Chen et al. 2007] and model comparison [Leung and White 1991; Chen et al. 2002]. However, the complexity involved in controlling all these techniques (each one uses a different type of model) demands effort and time beyond the scope of this research.

Despite providing important information regarding SART's applicability, our study with industrial models has some limitations as well, because the specification models used are reasonably small and simple. Nonetheless, the subject participating in our case study emphasized that it is very difficult and tedious to perform manual selection of diverse test cases exercising the modifications. In turn, the selection performed by SART was much faster, easier and able to reveal defects. Eventually, we intend to investigate further modifications on other industrial specification models and apply other techniques to compare with our results.

Note that many of the threats to validity are side effects from attempting to overcome challenges found in empirical evaluations of MB techniques (for example, the small availability of models) and regression test case selection (e.g. the lack of real fault data). This highlights the importance of this work as a proposal of an evaluation framework for similar techniques. Thus, with this study we want to encourage other researchers to use SBMTE and stimulate the empirical evaluation of proposed MB techniques.

7.5 Challenges and Rewards with SBMTE

Samples of models are hard to find hindering empirical evaluation of MB techniques. To address this lack of availability we decided to use stochastic model generation with descriptive statistics of realistic models [de Oliveira Neto et al. 2013]. This approach, named SBMTE, enabled the generation of a large number of models that share characteristics with industrial models.

Besides properly defining the role of the generator tool in an experiment, the experimenter needs to be able to generate realistic instances of models, i.e. the generated models should have characteristics typical of models used in organizations developing software (the larger and the more complex, the better). Otherwise, it is hard to translate the conclusions obtained from analysed data to the decision of adopting an investigated MB technique. Those characteristics can be expressed through parameters of the model such as size, type, dependencies, among others.

Consequently, the main challenges in using SBMTE are: Choosing an appropriate model format and defining a good set of parameters to generate representative models. We chose ALTS for being simple to use and easy to adapt, and the level of realism was obtained through rules to ensure consistent input and output relationships among transitions and states. For example, Finite State Machines (FSM) share characteristics with LTS models, such as states, transitions and labels. Thus, we believe that similar concepts can be applied in developing an FSM generator.

As SBMTE is used, generators for different model formats can be shared, making it easier to generate models used by different techniques being investigated. Regarding the challenge of choosing parameters, we suggest using model property extraction from repositories, industry partners, or by referring to literature.

The advantage of SBMTE is providing a platform for strong empirical evaluation of model-based techniques when large samples of models are not available. Nonetheless, the experimenter needs to design the experiment carefully to consider the model generation, or else the observed causation may be invalid. Many empirical evaluations of regression test selection technique present contradictory results, because some evaluations are done in a specific context [Engström et al. 2008]. Definition of experiments in terms of SBMTE allows comparison of MB techniques on the same platform, for example, dimensions of models, scalability, transitions coverage, among others.

Ultimately, our objective with SBMTE was to first validate SART through an experiment and obtain conclusive results, hence industry partners were not involved and we cannot really argue towards technology transfer of SART, specifically. However, based on Gorsheck's et al. model [Gorschek et al. 2006], we believe that thorough investigation of possible scenarios (e.g. executing any technique with a wide range of model types) increases confidence and encourages technology transfer of any MB technique, including SART.

7.6 Concluding Remarks

Usually, proposed research on MB techniques relies on model elements such as transitions, states or classes to get information from software to achieve a specific goal (e.g. model transformation or test suite prioritization). In our work, *Offline SBMTE* enabled observation of the effects that a variety of instances of models have on an MB technique (e.g. scalability, versatility, weaknesses). For example, the generation of thousands of models allowed us to confirm a relationship between branches in an ALTS model and SART's *SizeRed*, otherwise unseen in case studies with smaller samples [de Oliveira Neto and Machado 2011; de Oliveira Neto et al. 2013].

Greater potential is expected when including meta-heuristics to identify regions of the space of models that yields better (or worse) results. By dynamically finding better/worse models and providing them as input to experiments, the MB technique can be better understood, challenged and thus improved. Eventually, when model generation and meta-heuristics are included and SBMTE can be executed "in the loop" of a development process (*Online SBMTE*), then each organization will be able to develop and constantly evaluate its own MB technique.

In conclusion to our evaluation, we have gathered the data collected throughout the research and summarize them below in order to answer our research questions:

RQ1: How to use similarity functions to identify modifications?
A very common approach is considering transitions coverage as a criterion to determine if test cases are similar [Cartaxo et al. 2007a; Cartaxo et al. 2011; Hemmati et al. 2011]. Instead of analysing similarities among test cases belonging to the same test suite, we analyse *similarities between test cases of different versions* of a software, which made it possible to identify test cases traversing modified parts of a model. Therefore, very different pairs of test cases indicate sequences of transitions that were changed.

RQ2: How to select test cases based on the identified modifications?
Regression testing literature suggests classification of test cases based on their different roles (see Chapter 2) when performing regression test [Leung and White 1991; Briand et al. 2009]. Based on the similarity values from rows and columns of the matrix, we are able to classify the test cases as obsolete, reusable and targeted (i.e. retestable or new-specification) and select the ones traversing modified regions of the model.

RQ3: How to address redundancy issues occurring in test suites from both regression test suites and MBT approaches?
Since regression test suites tend to grow with each new version, redundancy scales up significantly. Our concern here is transitions coverage, and our solution is to apply test suite minimization techniques on the selected subset of targeted test cases. Subsequently, the modifications being repeatedly traversed by selected test cases are replaced by similar reusable test cases to ensure coverage of transitions nearby modified parts of the model.

RQ4: Is our selection strategy beneficial for regression testing when compared to plain application of similarity functions on a test suite?
Based on the comparative study, we concluded that there is no indication of an improvement in transitions coverage when using SART because of the significant size reduction achieved.

On the other hand, our usage of STCS is better in achieving coverage of modifications and selection of test cases for regression testing, when compared to the original use of the counting function on a set of test cases and the random selection of test cases.

Chapter 8

Conclusions

Specification models have been used as an important asset for software development process by providing useful information about the product being developed. Moreover, effective and efficient test generation techniques based on models are becoming available, giving leverage to automated traceability between design models and test cases. For regression testing this represents an automated mechanism to handle modifications at the specification level [Harrold and Orso 2008]. Consequently, research about specification-based test case selection for regression testing keeps growing in order to fill the gap of selecting test cases based on modifications of a model [Tamimi and Zahoor 2011; Yoo and Harman 2012]. This doctorate research presents a new selection strategy to address this issue, by mitigating existing problems in handling high level specification elements, such as abstract test cases.

Our research is part of a model-based testing process where specification models representing the system are used to design test cases and execute the SUT. For addressing the regression testing context, we assume that these models undergo modifications representing addition and removal of a software's functionalities. In order to address the problem of accrued test suites commonly found in MBT and regression testing processes [Fraser and Wotawa 2007; Harrold and Orso 2008; Bertolino et al. 2010; Yoo and Harman 2012] we developed a selection strategy named Similarity Approach for Regression Testing (SART) that uses a similarity function to analyse pairs of abstract test cases from different versions of software and determine which test cases traverse modified regions of the specification model.

Our goal was to investigate similarity-based test case selection (STCS) based on the assumption that searching among similarity values allows the technique to identify test cases covering severe modifications (low similarities) and remove those covering unmodified parts (high similarities). This investigation was performed through an experiment that confirmed our assumption that not only the technique is able to cover modifications performed on a specification model, but also provide a robust size reduction in a test suite. Furthermore, comparison with a STCS technique (counting function) [Cartaxo et al. 2011] and a random test case selection technique provided evidence that SART is able to address regression testing needs, being a more appropriate choice for selecting test cases in a progressive regression testing context than the other two techniques.

In order to obtain conclusive results, we used a large sample of specification models provided as input to our test process. To account for more representativeness and diversity of models, we proposed an approach, named Search-Based Model Technology Evaluation (SBMTE) to evaluate model-based techniques through stochastic generation of realistic models. Studies with model-based (MB) techniques often struggle with availability of sample of models to obtain statistically significant conclusions [de Oliveira Neto et al. 2013], and through SBMTE we address that problem by using generator tools to create instances of models based on characteristics of real industry models, such as size and layout. That enables early validation of MB techniques still in laboratory, mitigating risks before applying the technique in practice and encouraging empirical evaluation of existing techniques by alleviating the challenges of obtaining industrial models.

Besides SART and SBMTE, our research provides many other contributions detailed in publications achieved during our research [de Oliveira Neto and Machado 2011; de Oliveira Neto and Machado 2013; de Oliveira Neto et al. 2013]. In general, smaller, yet significant, contributions of this doctorate research are:

- **The experiment:** All artefacts and methods of our experiment are detailed in this document and available on the Internet[1] so that other researches can reuse its parts or reproduce it completely. In addition, the entire experiment is automatically executed in a tool allowing quick and easy reproduction in case other researchers decide to

[1]Details of the data and tables are available for download: `https://sites.google.com/site/fgonetosite/home/downloads`

investigate our results or expand them by using different treatments for factors.

- **Analysis based on fault models:** We define a method to determine four different fault models to represent defects on implementation that would trigger failures in turn represented by mutants at the specification model. Those mutants are changes of model's elements and would signal a failure when traversed by a test case. Consequently we can investigate if a technique is able to achieve an intended coverage. For example, one of our fault models represented defects triggered when traversing a model modification. Another example using our approach is to create a fault model to include mutants in loop transitions to assess loop coverage, and so on.

- **Tool support:** One of the main concerns throughout development of this research was to provide a platform for automatic execution of our proposed and investigated techniques. SART, our generator tool and the experiment platform is executable through the LTS-BT tool [2], and the statistical analysis is performed with the assistance of R[3] scripts. Besides alleviating the effort in using a full factorial experimental design, this decision allowed us to enhance reproducibility of our research.

In addition to all contributions we would in particular like to highlight the benefits provided by both SART and SBMTE. First, most work with similarity-based test case selection do not address specific issues of regression testing and use similarity functions to determine similarity among test cases of a single test suite or software version, whereas we use it as tool to investigate similarities between different software versions. More importantly, our proposal of SBMTE is a unique contribution to the MBT community for allowing the evaluation of MB techniques.

Nevertheless, the endeavour to overcome the obstacles of empirical evaluation of model-based techniques yields threats to validity to our methodology, hence limiting some of our conclusions. More specifically, the lack of parameters from real industry models limit generalization of our study to the type of model used (ALTS), instead of applying our results to specification models in general. Also, we believe that the randomness in assigning mutants through a biased wheel ($FM1$ and $FM2$) and breaking ties (when choosing states in $FM3$)

[2]`https://sites.google.com/a/computacao.ufcg.edu.br/lts-bt/`

[3]`http://www.r-project.org/`

hindered our conclusions with respect to our failure analysis.

Moreover, our selection strategy has been used for a general type of state-based or behavioural model, hence we cannot draw conclusions regarding usage of our strategy for test cases generated from structural diagrams such as UML class, component and object diagrams. These diagrams contain different information from software, usually from lower software levels such as code statements and functions/methods. That would require a new experiment using a different set of variables and hypotheses.

Also, usage of ALTS in our current implementation of SART limit the technique's applicability because most companies use UML diagrams to represent their models. However, LTS-BT supports creation of ALTS models from more popular diagrams such as activity and sequence diagrams [Cartaxo et al. 2007b; de Oliveira Neto 2010]. In the mean time, we use the document template to describe use case and scenarios (presented in Chapter 2) as primary input format, and ALTS are used as intermediary format to generate test cases. Additionally, the output is an XML file that can be imported to TestLink[4].

That leads us to the discussion of future work regarding our research. Now that the experiment provides evidence regarding the benefits of using SART to identify modifications on a test suite, we expect that usage of SART with different similarity functions presented in literature can yield different results that can be better or worse than SART's current version. In an experiment Hemmati et al. investigated 320 techniques [Hemmati et al. 2013] for STCS that could, eventually, be incorporated with SART to obtain better results. Furthermore, we could also investigate different minimisation heuristics to achieve more effective minimisation of the targeted test cases set. Accordingly, there are numerous possibilities to combine different similarity function and minimisation heuristics in an experiment (possibly using SBMTE) to determine which of them can enhance SART's performance.

Furthermore, SBMTE can be executed with techniques similar to SART that have been proposed for test case prioritization and minimization [Ouriques et al. 2013; Coutinho et al. 2013]. Those strategies can benefit from SBMTE, given the appropriate adaptation to address their respective issues and assumptions. For example, prioritization techniques could benefit from measuring Average Percentage of Faults Detected (APFD) of the resulting set of test cases. Moreover, other specification-based selection techniques could be executed with the

[4]`http://testlink.org/`

same space of models to compare rate of size reduction.

Besides the several options of using SBMTE in empirical evaluations, future work also comprise improvement of SBMTE itself. The generation can be engineered to target specific types of models. For example, the model generator could be engineered to receive meta-model specifications and then automatically generate instances of a desired model format. Moreover, some models are not available to the academic community due to concerns from industrial partners regarding proprietary information included in models. Those concerns hinder publication of results from experiments performed with real models. With SBMTE, researchers can write extraction scripts that industrial partners can apply on their models to gather data and avoid confidentiality issues. In turn, realistic models are generated at large scales allowing execution and publishing of experiments.

In conclusion, we produced a new selection strategy and proposed an evaluation methodology. The latter is also an attempt to encourage and disseminate some of the experimenting practices such as awareness to validity threats, control and definition of variables, conclusions based on statistical tests and reproducibility. Those practices raise the bar on confidence and significance required to achieve satisfactory results in any research, yielding a more rigorous reviewing process for existing work in literature and consequently an improvement on feedback and research provided to the academic community.

Appendix A

Example using the Weighted Similarity Analysis

The size of the selected subset provided by SART depends mainly on the number of modifications that must be covered. In some cases, the size of the selected subset may not comply with the testing resources of a development process. Since more size reduction is required, we believe that it is up to the tester to determine which test cases are more important to be tested, among those traversing modifications of the specification. Then, we decided to include an additional strategy based on a value-based approach to select important test cases. But what is a value-based approach?

Traditionally, each test case has an equal value, also known as *weight*, often indicating that the scenarios being tested in a System Under Test (SUT) are equally important, for example, to the client. But usually, in industry, specific parts of the SUT have different values, and thus, test cases exercising critical or “valuable” parts should have different weights [Boehm 2006].

In a value-based approach, weights are assigned to artefacts in order to represent their importance with respect to the product being produced. As an example, consider a medical application software where a patient’s symptoms are provided as input to obtain a list of diseases as output. Naturally, testing the parts responsible for analysing the symptoms becomes more important than, for example, testing formatting of strings in exported reports. Defects on the first testing criteria can lead to a wring medication or even the patient’s death, whereas defects on the second would lead to less severe consequences.

The challenge then becomes how to properly define and assign the weights to software artefacts. Usually, weights can represent operational profiles, scenarios likely to reveal defects, and are often defined by humans with appropriate expertise on the subject, e.g. an experienced tester, manager or even the client. SART's strategy regarding weight analysis is based on Bertolini et al. strategy, named Weighted Similarity Approach, or simply WSA [Bertolino et al. 2008; Cartaxo 2011]. This section presents and overall description and example of WSA selection strategy. Further details regarding the algorithm of WSA can be found in Cartaxo's thesis [Cartaxo 2011].

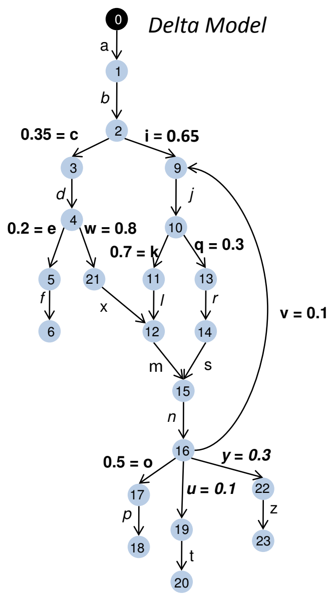
WSA ensures that the *least important* and *most similar* test cases are automatically removed (i.e. test cases with low weights and exercise similar sequences of transitions). For considering a similarity analysis on specification models, WSA becomes a suitable candidate to be used alongside SART. Therefore, we chose WSA to provide usage of a value-based approach when executing SART, enabling selection of important test cases traversing changed parts of the specification.

In WSA, the weights are assigned only to transitions and they represent an operational profile of the delta specification model (S') since the delta version P' is the one delivered to the system's users. Each transition has a value defined as $p(\overrightarrow{t_i})$ that indicates the probability that the system's user will execute transitions $\overrightarrow{t_i}$. The idea is that once the user reaches a certain location of the scenario (i.e. state), $p(\overrightarrow{t_i})$ represents the probability that the user will perform a specific step among all steps yielding from the current specification state. That is defined by equations A.1 and A.2, considering that State $q \in Q$ has $i = 1, 2, \cdots, n$ successors A successor is a state that can be reached through an outgoing transition of the current state.

$$T_{outgoing}(q) = \{\overrightarrow{t_i} \mid \exists q_i, \overrightarrow{t_i} = (q, q_i) \in T_{tr} \tag{A.1}$$

$$\sum_{i=1}^{n} p(\overrightarrow{t_i}) = 1, \overrightarrow{t_i} \in T_{outgoing}(q) \tag{A.2}$$

In other words, for each state in the ALTS, the sum of all $p(\overrightarrow{t_i})$ regarding its outgoing transition must be 1. Therefore, states with a single outgoing transition will have a transition with probability $p(\overrightarrow{t_i}) = 1$, whereas branching states will have its probability distributed among all outgoing transitions.



(a)

Weights	Delta Test Suite – T'																
0,070	**TC'1**	*a*	*b*	*c*	*d*	*e*	*f*										
0,084	**TC'2**	*a*	*b*	*c*	*d*	*w*	*x*	*m*	*n*	*y*	*z*						
0,006	**TC'3**	*a*	*b*	*c*	*d*	*w*	*x*	*m*	*n*	*v*	*j*	*k*	*l*	*m*	*n*	*y*	*z*
0,004	**TC'4**	*a*	*b*	*c*	*d*	*w*	*x*	*m*	*n*	*v*	*j*	*q*	*r*	*s*	*n*	*o*	*p*
0,137	**TC'5**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*y*	*z*						
0,059	**TC'6**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*y*	*z*						
0,228	**TC'7**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*						
0,098	**TC'8**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*o*	*p*						
0,020	**TC'9**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*u*	*t*						
0,003	**TC'10**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*v*	*j*	*k*	*l*	*m*	*n*	*u*	*t*
0,001	**TC'11**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*v*	*j*	*q*	*r*	*s*	*n*	*u*	*t*
0,007	**TC'12**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*v*	*j*	*q*	*r*	*s*	*n*	*o*	*p*
0,004	**TC'13**	*a*	*b*	*i*	*j*	*k*	*l*	*m*	*n*	*v*	*j*	*q*	*r*	*s*	*n*	*y*	*z*
0,003	**TC'14**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*v*	*j*	*q*	*r*	*s*	*n*	*o*	*p*
0,001	**TC'15**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*v*	*j*	*k*	*l*	*m*	*n*	*u*	*t*
0,007	**TC'16**	*a*	*b*	*i*	*j*	*q*	*r*	*s*	*n*	*v*	*j*	*k*	*l*	*m*	*n*	*o*	*p*

(b)

Figure A.1: (a) Example of an ALTS model with weights assigned to branches, and (b) the weights for each test case. The shaded test cases represent the subset selected in Chapter 3.

In order to illustrate this process, we assigned weights to the transitions of the specification used in Chapter 3, resulting in Figure A.1. So, for instance $p(\overrightarrow{k}) = 0.7$ indicates that, after reaching State 10 the user has a 70% probability to execute action $\overrightarrow{k}$, and 30% probability of executing action $\overrightarrow{q}$. In order to improve visualisation, all weights $p(\overrightarrow{t_i}) = 1$ have been omitted.

After the operational profile is defined, the weight of each test case $(weight(t_i), t_i \in T_s)$ is automatically calculated by *multiplying* the weights of its correspondent transitions. For instance, the test case $TC'8$ (with weight 0.097) is the more important test case of our selected subset, and represents the scenario of importing and saving a contact's information on a phonebook. After obtaining weights for each test case, we begin to execute WSA (Figure A.2) using, as input, SART's selected subset. From Chapter 3, the input for our example is $T_s = \{TC'1, TC'4, TC'8, TC'9, TC'11, TC'12, TC'13, TC'14\}$.

Initially, we generate the weighted similarity matrix W that, in turn, differs from our similarity matrix in two aspects: Only *one test suite* is considered in the analysis, and the

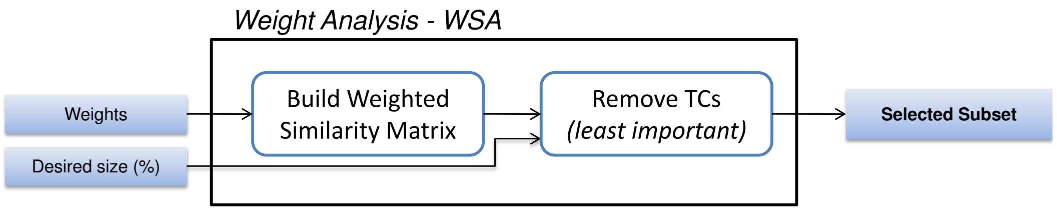

Weight Analysis - WSA
Weights
Desired size (%)
Build Weighted Similarity Matrix
Remove TCs (least important)
Selected Subset

Figure A.2: Steps to execute WSA given, as input: A test suite with weighted test cases, and the number of test cases that should be removed.

similarity *values are divided* by a weight. Therefore, based on Equations A.3 and A.4 we generate each element $w[i,j]$ of the matrix, remembering that each index of the matrix represent a test case. Since we only use one test suite as input, and all pairs of test cases are analysed, resulting in a square matrix $W_{|T_s|\times|T_s|}$, where each $t \in T_s$ is placed on a respective row *and* column of W.

$$WSA(t_i,t_j) = \frac{nit(t_i,t_j)}{\text{AvgSize}(t_i,t_j)},\ t_i,t_j \in T_s \tag{A.3}$$

$$w[i,j] = \frac{WSA(t_i,t_j)}{weight(t_i)} \tag{A.4}$$

Mathematically, the baseline value is the similarity value $WSA(t_i,t_j)$[1] between the selected test cases; then we divide this value by a test case's weight $0 < weight(t_i) \leq 1$. In other words, each similarity value obtained is divided by the weight of the test cases of the correspondent matrix *row*, leading to a significant increase in $w[i,j]$ of the least important test cases (i.e. low weights). As a result, the unwanted test cases (very similar and least important) can be found by searching for the higher values in W.

In each turn, and after finding the highest $w[i,j]$, we remove the respective row and column of the chosen test case (for a tie break we randomly choose one of the candidates), and repeat the process until the user of the technique (e.g. a tester) decides that the size obtained is suitable. For instance, in our example of Figure A.1, we calculate the W matrix for T_s resulting in Table A, and the tester wants to remove 3 test cases.

First, we find $w[TC'15, TC''11] = w[TC'11, TC'15] = 732.6$ meaning that $TC15$ and $TC11$ are both very similar and less important test cases, thus we (randomly) choose $TC15$

[1] $AvgSize$ and nit are the same functions described in Chapter 3, where the first calculates the average size between both test cases, and the second retrieves the number of identical transitions between both test cases.

Table A.1: Weighted similarity matrix from the test suite of Figure A.1.

	TC'1	TC'4	TC'8	TC'9	TC'11	TC'12	TC'13	TC'15
TC'1	0.00	5.19	3.57	3.57	2.57	2.57	2.60	2.57
TC'4	85.71	0.00	145.24	128.21	147.62	178.57	133.93	133.33
TC'8	2.56	6.26	0.00	8.21	6.31	7.89	6.26	6.31
TC'9	12.82	27.59	41.03	0.00	38.97	31.28	31.54	31.28
TC'11	131.87	454.21	450.55	556.78	0.00	641.03	637.36	732.60
TC'12	26.37	109.89	112.67	89.38	128.21	0.00	127.47	128.21
TC'13	43.96	137.24	148.96	150.18	212.45	212.45	0.00	212.45
TC'15	131.87	410.26	450.55	446.89	732.60	641.03	637.36	0.00

to be removed from T_s and then remove its row and column from W in order to proceed with the technique. The next removal is $TC'11$ due to $w[TC'11, TC'12] = 641.03$ followed by removal of $TC'13$ ($w[TC'13, TC'12] = 212.4$). Therefore, the resulting subset becomes $T_s = \{TC'1, TC'4, TC'8, TC'9, TC'12\}$. Note that $TC'11, TC'12, TC'13, TC'15$ are very similar among themselves and the technique was able to keep the more important test case — $weight(TC'12) = 0.0068$ — and although $TC'4$ has a smaller weight, it traverses a very unique sequence of transitions when compared to the other ones in T_s, thus it should not be removed from the subset.

The main benefit of WSA is providing the automatic algorithm to strike a balance between weights and similarity values to ensure removal of redundant and least important test cases [Cartaxo 2011]. Despite the benefits, appropriate evaluation and safe application of WSA is hindered by the process of assigning weights. That is, to rely on a subject's expertise to express an operational profile can compromise the technique's performance, since the selection is guided by a subjective value (e.g. an operational profile, a distribution of estimated defects) defined by a human being. On the other hand, that can be beneficial if the information provided by the subject is accurate.

In summary, we recommend usage of the weight analysis in SART whenever the selected subset, obtained after identifying modifications, does not comply with available resources and more size reduction is necessary. Otherwise, the weight analysis can be avoided, since the focus of regression testing is still finding regression defects and WSA analysis does not consider modifications of the specification model in its selection process.

Appendix B

Tables detailing coefficients and p-values.

This appendix contains data regarding p-values and coefficients obtained from all interactions of factors reported in our experiment (Chapter 7). We present p-values regarding the linear regression model for each coefficient obtained for the main factors and all possible interactions (Tables B.1, B.2 and B.3). Below, the "Indicator" column signals whenever a p-value is smaller than the considered level of significance ('$***, \alpha = 0.001$'; '$**, \alpha = 0.01$'; '$*, \alpha = 0.05$'; '$., \alpha = 0.01$'). R scripts were used to obtain the data and p-values.

Table B.1: Data and p-values for all main factors and 2-way interactions of factors.

Main factors	**Coefficient**	t **value**	p**-value**	**Indicator**
S	-0.1279406	-4.718	6.13e-06	***
B	22.233729	81.996	< 2e-16	***
J	0.5423573	20.002	< 2e-16	***
L	0.0107323	0.396	0.69291	
A	-13.664448	-50.393	< 2e-16	***
R	-14.033688	-51.755	< 2e-16	***
2**-way interactions**	**Coefficient**	t **value**	p**-value**	**Indicator**
S:B	0.0505292	1.863	0.06469	.
S:J	-0.0838510	-3.092	0.00244	**
B:J	-0.2650833	-9.776	< 2e-16	***
S:L	-0.0365198	-1.347	0.18042	
B:L	0.0011375	0.042	0.96660	
J:L	0.0507594	1.872	0.06349	.
S:A	0.0047510	0.175	0.86119	
B:A	0.3311833	12.214	< 2e-16	***
J:A	0.0692448	2.554	0.01183	*
L:A	-0.0012344	-0.046	0.96376	
S:R	-0.0043500	-0.160	0.87280	
B:R	0.3659969	13.498	< 2e-16	***
J:R	0.1281167	4.725	5.97e-06	***
L:R	0.0345563	1.274	0.20483	
A:R	-0.0011458	-0.042	0.96636	

Table B.2: Data and p-values for all 3-way interactions of factors.

3-way interactions	Coefficient	t value	p-value	Indicator
S:B:J	0.0159646	0.589	0.55706	
S:B:L	0.0210000	0.774	0.44009	
S:J:L	0.0207948	0.767	0.44456	
B:J:L	-0.0421271	-1.554	0.12275	
S:B:A	-0.0498771	-1.839	0.06817	.
S:J:A	0.0349198	1.288	0.20014	
B:J:A	-0.0694729	-2.562	0.01156	*
S:L:A	-0.0694740	-2.562	0.01156	*
B:L:A	-0.0213979	-0.789	0.43149	
J:L:A	-0.0534573	-1.971	0.05083	.
S:B:R	0.0134344	0.495	0.62113	
S:J:R	-0.0117813	-0.434	0.66467	
B:J:R	-0.0546135	-2.014	0.04610	*
S:L:R	-0.0003688	-0.014	0.98917	
B:L:R	0.0061802	0.228	0.82007	
J:L:R	0.0034917	0.129	0.89774	
S:A:R	-0.0179938	-0.664	0.50814	
B:A:R	-0.0324573	-1.197	0.23352	
J:A:R	0.0365562	1.348	0.17999	
L:A:R	-0.0097833	-0.361	0.71884	

Table B.3: Data and p-values for all 4,5 and 6-way interactions of factors.

4-way interactions	Coefficient	t value	p-value	Indicator
S:B:J:L	-0.0256229	-0.945	0.34646	
S:B:J:A	-0.0537500	-1.982	0.04959	*
S:B:L:A	0.0326688	1.205	0.23051	
S:J:L:A	0.0245656	0.906	0.36666	
B:J:L:A	-0.0085917	-0.317	0.75187	
S:B:J:R	-0.0054969	-0.203	0.83968	
S:B:L:R	-0.0206385	-0.761	0.44798	
S:J:L:R	-0.0455833	-1.681	0.09519	.
B:J:L:R	0.0252865	0.933	0.35281	
S:B:A:R	0.0412885	1.523	0.13030	
S:J:A:R	0.0236250	0.871	0.38524	
B:J:A:R	-0.0261135	-0.963	0.33734	
S:L:A:R	0.0099625	0.367	0.71392	
B:L:A:R	-0.0339823	-1.253	0.21240	
J:L:A:R	0.0276813	1.021	0.30925	
5-way interactions	**Coefficient**	**t value**	**p-value**	**Indicator**
S:B:J:L:A	-0.0141250	-0.521	0.60332	
S:B:J:L:R	0.0082135	0.303	0.76245	
S:B:J:A:R	-0.0060719	-0.224	0.82317	
S:B:L:A:R	-0.0194385	-0.717	0.47476	
S:J:L:A:R	0.0106604	0.393	0.69487	
B:J:L:A:R	-0.0253177	-0.934	0.35222	
6-way interactions	**Coefficient**	**t value**	**p-value**	**Indicator**
S:B:J:L:A:R	0.0110552	0.408	0.68417	

Appendix C

An Example of a Complete Use Case Document

Feature 01 - Messaging

UC_01 - Sending messages with attached items

Description

This use case describes how a message can be sent by attaching an image file (multimedia) or a message saved on memory (e.g. received or forwarded message).

Main Flow

Description: Create a new contact
***From Step:* START** ***To Step:* END**

Step Id	Type	Label
1M	*user_action*	Select "Send Item" option.
2M	*expected_results*	List of options is displayed.
3M	*user_action*	Include an image file.
4M	*expected_results*	List of saved image files is displayed.
5M	*user_action*	Press "Send Image" button.
6M	*expected_results*	"Item sent" message is displayed.

Alternative Flows

Description: Return to the main screen
***From Step:* 2M** ***To Step:* END**

Step Id	Type	Label
1A	*user_action*	Press "Return" icon.
2A	*expected_results*	Main menu is displayed.

Description: Include message already saved on memory (e.g. messages received, previous conversations or forwarded to inbox).
***From Step:* 2M** ***To Step:* 6M**

Step Id	Type	Label
1B	*user_action*	Include a saved message.

2B	*expected_results*	List of saved messages is displayed.
3B	*user_action*	Select the message and press "Send" button.

Description: Cancel saved message inclusion.
From Step: 2B ***To Step: END***

Step Id	Type	Label
1C	*user_action*	Select "Cancel" option.
2C	*expected_results*	"Want to send other item?" message is displayed.
3C	*user_action*	Press "No" button.
4C	*expected_results*	"No items were sent" message is displayed.

Description: Return to the "Send Item" screen to allow users to repeat the operation.
From Step: 2C ***To Step: 2M***

Step Id	Type	Label
1D	*user_action*	Press "Yes" button.

Bibliography

[Agrawal et al. 1993] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. (1993). Incremental regression testing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, DC, USA. IEEE Computer Society.

[Andrews et al. 2005] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA. ACM.

[Arcuri and Briand 2014] Arcuri, A. and Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

[Barbosa et al. 2007] Barbosa, Daniel, L., Lima, H. S., Machado, P. D. L., Figueiredo, J. C., Jucá, M. A., and Andrade, W. L. (2007). Automating functional testing of components from UML specifications. *International Journal of Software Engineering and Knowledge Engineering*.

[Basanieri et al. 2002] Basanieri, F., Bertolino, A., and Marchetti, E. (2002). The cow suite approach to planning and deriving test suites in UML projects. In *Proceedings of the 5th International Conference on The Unified Modeling Language (UML 02)*, UML'02, pages 383–397, London,UK. Springer-Verlag.

[Beizer 1990] Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

[Bertolino et al. 2008] Bertolino, A., Cartaxo, E. G., Machado, P. D. L., and Marchetti, E.

(2008). Weighting influence of user behavior in software validation. In *19th International Workshop on Database and Expert Systems Application*, pages 495–500.

[Bertolino et al. 2010] Bertolino, A., Cartaxo, E. G., Machado, P. D. L., Marchetti, E., and Ouriques, J. a. F. S. (2010). Test suite reduction in good order: Comparing heuristics from a new viewpoint. In *The 22nd IFIP International Conference on Testing Software and Systems (ICTSS'10)*, pages 13–18.

[Binder 1996] Binder, R. (1996). Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6:125–252.

[Binder 1999] Binder, R. V. (1999). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Biswas et al. 2011] Biswas, S., Mall, R., Satpathy, M., and Sukumaran, S. (2011). Regression test selection techniques: A survey. *Informatica*, 35(3):289–321.

[Bochmann and Petrenko 1994] Bochmann, G. V. and Petrenko, A. (1994). Protocol testing: Review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 109–124, New York, NY, USA. ACM.

[Boehm 2006] Boehm, B. (2006). Value-based software engineering: Overview and agenda. In *In book*, pages 3–14. Springer Verlag.

[Briand et al. 2009] Briand, L. C., Labiche, Y., and He, S. (2009). Automating regression test selection based on UML designs. *Information and Software Technology*, 51(1):16–30. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[Briand et al. 2002] Briand, L. C., Labiche, Y., and Soccar, G. (2002). Automating impact analysis and regression test selection based on uml designs. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 252, Washington, DC, USA. IEEE Computer Society.

[Brottier et al. 2006] Brottier, E., Fleurey, F., Steel, J., Baudry, B., and Le Traon, Y. (2006). Metamodel-based test generation for model transformations: an algorithm and a tool. In

Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on, pages 85–94.

[Broy et al. 2005] Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., and Pretschner, A., editors (2005). *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer.

[Cabral and Sampaio 2008] Cabral, G. and Sampaio, A. (2008). Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188.

[Cartaxo 2011] Cartaxo, E. G. (2011). *Estratégias para Controlar o Tamanho da Suíte de Teste Gerada a partir de Abordagens de Teste Baseado em Modelos*. PhD thesis, Federal University of Campina Grande.

[Cartaxo et al. 2008] Cartaxo, E. G., Andrade, W. L., de Oliveira Neto, F. G., and Machado, P. D. L. (2008). LTS-BT: A tool to generate and select functional test cases for embedded systems. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 1540–1544, New York, NY, USA. ACM.

[Cartaxo et al. 2007a] Cartaxo, E. G., de Oliveira Neto, F. G., and Machado, P. D. L. (2007a). Automated test case selection based on a similarity function. In *Proceedings of MOTES07 - Model-based Testing - Workshop in conjunction with the 37th Annual Congress of the Gesellschaft fuer Informatik*, volume 110 of *Lecture Notes in Informatics (LNI)*, pages 381–386.

[Cartaxo et al. 2007b] Cartaxo, E. G., de Oliveira Neto, F. G., and Machado, P. D. L. (2007b). Test case generation by means of UML sequence diagrams and labeled transition systems. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics, 2007 (SMC'07)*, pages 1292–1297.

[Cartaxo et al. 2011] Cartaxo, E. G., Machado, P. D. L., and de Oliveira Neto, F. G. (2011). On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100.

[Carver and Tai 1998] Carver, R. H. and Tai, K.-C. (1998). Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transanctions on Software Engineering*, 24(6):471–490.

[Chen et al. 2002] Chen, Y., Probert, R. L., and Sims, D. P. (2002). Specification-based regression test selection with risk analysis. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press.

[Chen et al. 2007] Chen, Y., Probert, R. L., and Ural, H. (2007). Regression test suite reduction using extended dependence analysis. In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 62–69, New York, NY, USA. ACM.

[Cheng and Jou 1990] Cheng, K.-T. and Jou, J.-Y. (1990). A single-state-transition fault model for sequential machines. In *IEEE International Conference on Computer-Aided Design*, pages 226–229.

[Chittimalli and Harrold 2008] Chittimalli, P. K. and Harrold, M. J. (2008). Regression test selection on system requirements. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 87–96, New York, NY, USA. ACM.

[Chung et al. 1999] Chung, I. S., Kim, H. S., Bae, H. S., Kwon, Y. R., and Lee, D. G. (1999). Testing of concurrent programs after specification changes. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 199–, Washington, DC, USA. IEEE Computer Society.

[Clarke et al. 2003] Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. (2003). Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175.

[Cook and Campbell 1979] Cook, T. D. and Campbell, D. T. (1979). *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company.

[Coutinho et al. 2013] Coutinho, A. E. V. B., Cartaxo, E. G., and Machado, P. D. L. (2013). Test suite reduction based on similarity of test cases. In *7th Brazilian Workshop on Systematic and Automated Software Testing*, volume 7, Brasilia.

[Dalal et al. 1999] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. (1999). Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, New York, NY, USA. ACM.

[de Araújo et al. 2012] de Araújo, J. D. S., Cartaxo, E. G., de Oliveira Neto, F. G., and Machado, P. D. L. (2012). Controlando a diversidade e a quantidade de casos de teste na geração automática a partir de modelos com loop. In *6th Brazilian Workshop on Systematic and Automated Software Testing*, volume 6, Natal, RN, Brazil.

[de Oliveira Neto 2010] de Oliveira Neto, F. G. (2010). Investigação e avaliação experimental de técnicas de re-teste seletivo para teste de regressão baseado em especificação. Master's thesis, Federal University of Campina Grande.

[de Oliveira Neto et al. 2013] de Oliveira Neto, F. G., Feldt, R., Torkar, R., and Machado, P. D. L. (2013). Searching for models to evaluate software technology. In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, pages 12–15.

[de Oliveira Neto and Machado 2011] de Oliveira Neto, F. G. and Machado, P. D. L. (2011). WSA-RT: uma técnica para a seleção de casos de teste de regressão baseados na especificação do sistema. In *5th Brazilian Workshop on Systematic and Automated Software Testing*, volume 5, São Paulo, SP, Brazil.

[de Oliveira Neto and Machado 2013] de Oliveira Neto, F. G. and Machado, P. D. L. (2013). Seleção automática de casos de teste de regressão baseada em similaridade e valores. *Revista de Informática Teórica e Aplicada*, 20:139–154.

[Deeptimahanti and Sanyal 2011] Deeptimahanti, D. K. and Sanyal, R. (2011). Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 165–174, New York, NY, USA. ACM.

[Do et al. 2005] Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435. 10.1007/s10664-005-3861-2.

[Do and Rothermel 2006] Do, H. and Rothermel, G. (2006). An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 141–151, New York, NY, USA. ACM.

[El-Far 2001] El-Far, I. K. (2001). Enjoying the perks of model-based testing. In *In Proceedings of the Software Testing, Analysis, and Review Conference*.

[Engström et al. 2008] Engström, E., Skoglund, M., and Runeson, P. (2008). Empirical evaluations of regression test selection techniques: a systematic review. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 22–31, New York, NY, USA. ACM.

[Fahad and Nadeem 2008] Fahad, M. and Nadeem, A. (2008). A survey of UML based regression testing. In Shi, Z., Mercier-Laurent, E., and Leake, D., editors, *Intelligent Information Processing IV*, volume 288 of *IFIP Advances in Information and Communication Technology*, pages 200–210. Springer Boston.

[Farooq et al. 2007] Farooq, Q., Iqbal, M. Z. Z., Malik, Z. I., and Nadeem, A. (2007). An approach for selective state machine based regression testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, New York, NY, USA. ACM.

[Farooq et al. 2010] Farooq, Q., Iqbal, M. Z. Z., Malik, Z. I., and Riebisch, M. (2010). A model-based regression testing approach for evolving software systems with flexible tool support. In *Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 41–49, Washington, DC, USA. IEEE Computer Society.

[Feldt 1998] Feldt, R. (1998). Generating diverse software versions with genetic programming: an experimental study. *Software, IEE Proceedings-*, 145(6):228–236.

[Feldt 1999] Feldt, R. (1999). Genetic programming as an explorative tool in-early software development phases. *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, pages 11–19.

[Feng et al. 2007] Feng, T. H., Wang, L., Zheng, W., Kanajan, S., and Seshia, S. A. (2007). Automatic model generation for black box real-time systems. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6.

[Fraser and Wotawa 2007] Fraser, G. and Wotawa, F. (2007). Redundancy based test-suite reduction. In Dwyer, M. and Lopes, A., editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin Heidelberg.

[Gao et al. 2006] Gao, J., Gopinathan, D., Mai, Q., and He, J. (2006). A systematic regression testing method and tool for software components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, COMPSAC '06, pages 455–466, Washington, DC, USA. IEEE Computer Society.

[Gligoric et al. 2010] Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., and Marinov, D. (2010). Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 225–234, New York, NY, USA. ACM.

[González-Barahona and Robles 2012] González-Barahona, J. M. and Robles, G. (2012). On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1-2):75–89.

[Gorschek et al. 2006] Gorschek, T., Wohlin, C., Carre, P., and Larsson, S. B. M. (2006). A model for technology transfer in practice. *Software, IEEE*, 23(6):88–95.

[Gorthi et al. 2008] Gorthi, R. P., Pasala, A., Chanduka, K. K., and Leong, B. (2008). Specification-based approach to select regression test suite to validate changed software. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 153–160, Washington, DC, USA. IEEE Computer Society.

[Graves et al. 2001] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. (2001). An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering Methodology*, 10(2):184–208.

[Guerra 2012] Guerra, E. (2012). Specification-driven test generation for model transformations. In *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg.

[Gupta et al. 1996] Gupta, R., Harrold, M. J., and Soffa, L. (1996). Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6:83–112.

[Harman and Jones 2001] Harman, M. and Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43:833–839.

[Harrold et al. 1993] Harrold, M. J., Gupta, R., and Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions Softwware Engineering Methodology*, 2(3):270–285.

[Harrold et al. 2001a] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. (2001a). Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 312–326, New York, NY, USA. ACM.

[Harrold and Orso 2008] Harrold, M. J. and Orso, A. (2008). Retesting software during development and maintenance. In *Frontiers of Software Maintenance (FoSM 2008)*, pages 99–108, Beijing, China.

[Harrold et al. 2001b] Harrold, M. J., Rosenblum, D., Rothermel, G., and Weyuker, E. (2001b). Empirical studies of a prediction model for regression test selection. *Software Engineering, IEEE Transactions on*, 27(3):248–263.

[Hemmati et al. 2011] Hemmati, H., Arcuri, A., and Briand, L. (2011). Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *4th ICST 2011*.

[Hemmati et al. 2013] Hemmati, H., Arcuri, A., and Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42.

[Hsia et al. 1997] Hsia, P., Li, X., Kung, D. C., Hsu, C.-T., Li, L., Toyoshima, Y., and Chen, C. (1997). A technique for the selective revalidation of oo software. *Journal of Software Maintenance*, 9(4):217–233.

[Huselius et al. 2006] Huselius, J., Andersson, J., Hansson, H., and Punnekkat, S. (2006). Automatic generation and validation of models of legacy software. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 342 –349.

[IEEE 2013] IEEE (2013). IEEE Standard for Software and systems engineering - Software testing - Part 1: Concepts and definitions. *IEEE Std. 29119-1:2013(E)*, pages i –56.

[Jain 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley.

[Jard and Jéron 2005] Jard, C. and Jéron, T. (2005). TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal Software Tools for Technology Transfer*, 7(4):297–315.

[Jorgensen 2002] Jorgensen, P. C. (2002). *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA.

[Jzquel et al. 1999] Jzquel, J.-m., Ho, W. M., and Guennec, A. (1999). Pennaneac 'h. UMLAUT: an extendible UML transformation framework. In *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE.*

[Kanstrén 2009] Kanstrén, T. (2009). Behaviour pattern-based model generation for model-based testing. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, pages 233–241.

[Korel et al. 2002] Korel, B., Tahat, L. H., and Vaysburg, B. (2002). Model based regression test reduction using dependence analysis. In *Proceedings of the International Con-*

ference on Software Maintenance (ICSM '02), pages 214–223, Washington, DC, USA. IEEE Computer Society.

[Laski and Szermer 1992] Laski, J. and Szermer, W. (1992). Identification of program modifications and its applications in software maintenance. In *ICSM '92: Proceedings of the Conference on Software Maintenance*, pages 282–290. IEEE Computer Society.

[Leung and White 1989] Leung, H. K. N. and White, L. (1989). Insights into regression testing. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69.

[Leung and White 1990] Leung, H. K. N. and White, L. (1990). A study of integration testing and software regression at the integration level. In *Proceedings of the International Conference on Software Maintenance (ICSM'90)*, pages 290–300.

[Leung and White 1991] Leung, H. K. N. and White, L. (1991). A cost model to compare regression test strategies. In *In Proceedings of International Conference on Software Maintenance*, pages 201–208.

[Liang 2005] Liang, H. (2005). Regression testing of classes based on TCOZ specification. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '05, pages 450–457, Washington, DC, USA. IEEE Computer Society.

[Lorenzoli et al. 2008] Lorenzoli, D., Mariani, L., and Pezzé, M. (2008). Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, New York, NY, USA. ACM.

[Ma et al. 2005] Ma, X.-y., He, Z.-f., Sheng, B.-k., and Ye, C.-q. (2005). A genetic algorithm for test-suite reduction. In *IEEE International Conference on System, Man and Cybernetics*, pages 133–139.

[Mansour and Statieh 2009] Mansour, N. and Statieh, W. (2009). Regression test selection for c# programs. In *Advances in Software Engineering*, ASE '09.

[Mao and Lu 2005] Mao, C. and Lu, Y. (2005). Regression testing for component-based software systems by enhancing change information. In *Proceedings of the 12th Asia-*

Pacific Software Engineering Conference, pages 611–618, Washington, DC, USA. IEEE Computer Society.

[Mao et al. 2007] Mao, C., Lu, Y., and Zhang, J. (2007). Regression testing for component-based software via built-in test design. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1416–1421, New York, NY, USA. ACM.

[Muccini 2007] Muccini, H. (2007). Using model differencing for architecture-level regression testing. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '07, pages 59–66, Washington, DC, USA. IEEE Computer Society.

[Naslavsky and Richardson 2007] Naslavsky, L. and Richardson, D. J. (2007). Using traceability to support model-based regression testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 567–570, New York, NY, USA. ACM.

[Naslavsky et al. 2009] Naslavsky, L., Ziv, H., and Richardson, D. J. (2009). A model-based regression test selection technique. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 515–518.

[Naslavsky et al. 2010] Naslavsky, L., Ziv, H., and Richardson, D. J. (2010). MbSRT2: Model-based selective regression testing with traceability. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 89–98, Washington, DC, USA. IEEE Computer Society.

[Nogueira et al. 2007] Nogueira, S., Cartaxo, E. G., Torres, D., Aranha, E., and Marques, R. (2007). Model based test generation: An industrial experience. In *1st Brazilian Workshop on Systematic and Automated Software Testing*.

[Orso et al. 2001] Orso, A., Harrold, M. J., Rosenblum, D., Rothermel, G., Do, H., and Soffa, M. L. (2001). Using component metacontent to support the regression testing of component-based software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 716–, Washington, DC, USA. IEEE Computer Society.

[Orso et al. 2004] Orso, A., Shi, N., and Harrold, M. J. (2004). Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 241–251, New York, NY, USA. ACM.

[Ouriques et al. 2013] Ouriques, J. F. S., Cartaxo, E. G., and Machado, P. D. L. (2013). On the influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing. In *27th Simposio Brasileiro de Engenharia de Software*, volume 27, Brasilia.

[Pasala et al. 2008] Pasala, A., Fung, Y. L. H. L. Y., Akladios, F., and Gorthi, A. R. G. R. P. (2008). Selection of regression test suite to validate software applications upon deployment of upgrades. In *Proceedings of the 19th Australian Conference on Software Engineering*, ASWEC '08, pages 130–138, Washington, DC, USA. IEEE Computer Society.

[Prowell et al. 1999] Prowell, S. J., Trammell, C. J., Linger, R. C., and Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley.

[Ren et al. 2004] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: A tool for change impact analysis of java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448. ACM Press.

[Rogstad et al. 2013] Rogstad, E., Briand, L. C., and Torkar, R. (2013). Test case selection for black-box regression testing of database applications. *Information and Software Technology*, 55(10):1781 – 1795.

[Rothermel and Harrold 1996] Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22:529–551.

[Rothermel and Harrold 1997] Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology*, 6(2):173–210.

[Rothermel et al. 2000] Rothermel, G., Harrold, M. J., and Dedhia, J. (2000). Regression test selection for C++ software. *Software Testing, Verification & Reliability*, 10.

[Sajeev and Wibowo 2003] Sajeev, A. S. M. and Wibowo, B. (2003). Regression test selection based on version changes of components. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, APSEC '03, Washington, DC, USA. IEEE Computer Society.

[Saltelli et al. 2004] Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M. (2004). *Sensitivity analysis in practice: a guide to assessing scientific models*. John Wiley & Sons.

[Sen et al. 2009] Sen, S., Baudry, B., and Mottu, J.-M. (2009). Automatic model generation strategies for model transformation testing. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 148–164, Berlin, Heidelberg. Springer-Verlag.

[Siegel and Junior 1988] Siegel, S. and Junior, N. J. C. (1988). *Nonparametric Statistics for The Behavioral Sciences*. McGraw-Hill.

[Singh et al. 2007] Singh, R., Xu, J., and Berger, B. (2007). Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology*, RECOMB'07, pages 16–31, Berlin, Heidelberg. Springer-Verlag.

[Smith 2003] Smith, J. (2003). The estimation of effort based on use cases. Technical report, IBM Corporation.

[Soares et al. 2013] Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162.

[Subramaniam et al. 2009] Subramaniam, M., Xiao, L., Guo, B., and Pap, Z. (2009). An approach for test selection for efsms using a theorem prover. In *TESTCOM '09/FATES '09: Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, pages 146–162, Berlin, Heidelberg. Springer-Verlag.

[Tamimi and Zahoor 2011] Tamimi, S. and Zahoor, M. (2011). Analysis of model based regression testing approaches. In *Proceedings of the 10th WSEAS International Confer-*

ence on Communications, Electrical; Computer Engineering, ACELAE'11, pages 65–70, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS).

[Tan et al. 1997] Tan, Q., Petrenko, A., and Bochmann, G. V. (1997). Checking experiments with labeled transition systems for trace equivalence. In Kim, M., Kang, S., and Hong, K., editors, *Testing of Communicating Systems*, IFIP The International Federation for Information Processing, pages 167–182. Springer US.

[Tarhini et al. 2006] Tarhini, A., Fouchal, H., and Mansour, N. (2006). Regression testing web services-based applications. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, AICCSA '06, pages 163–170, Washington, DC, USA. IEEE Computer Society.

[Utting and Legeard 2006] Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Vokolos and Frankl 1997] Vokolos, F. I. and Frankl, P. G. (1997). Pythia: A regression test selection tool based on textual differencing. In *IFIP 3rd Internatinal Conference on on Reliability, Quality and Safety of Software-intensive Systems*, ENCRESS '97, pages 3–21, London, UK, UK. Chapman & Hall, Ltd.

[White et al. 2008] White, L. J., Jaber, K., Robinson, B., and Rajlich, V. (2008). Extended firewall for regression testing: an experience report. *Journal of Software Maintenance*, 20(6):419–433.

[Willmor and Embury 2005] Willmor, D. and Embury, S. M. (2005). A safe regression test selection technique for database driven applications. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society.

[Wohlin et al. 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

[Wu et al. 1999] Wu, Y., Chen, M.-H., and Kao, H. M. (1999). Regression testing on object-oriented programs. In *Proceedings. 10th International Symposium on Software Reliability Engineering*, pages 270–279.

[Wu and Offutt 2003] Wu, Y. and Offutt, J. (2003). Maintaining evolving component-based software with uml. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, Washington, DC, USA. IEEE Computer Society.

[Xu et al. 2003] Xu, L., Xu, B., Chen, Z., Jiang, J., and Chen, H. (2003). Regression testing for web applications based on slicing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pages 652–, Washington, DC, USA. IEEE Computer Society.

[Yoo and Harman 2012] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120.

[Zager and Verghese 2008] Zager, L. A. and Verghese, G. C. (2008). Graph similarity scoring and matching. *Applied Mathematics Letters"*, 21(1):86–94.

[Zheng et al. 2006] Zheng, J., Robinson, B., Williams, L., and Smiley, K. (2006). Applying regression test selection for COTS-based applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 512–522, New York, NY, USA. ACM.