Universidade Estadual de Campinas
Instituto de Computação

Thaís Harumi Ussami

Incremental Tests in a Model Based Test Driven Development

Testes Incrementais em um Desenvolvimento Guiado Por Testes Baseados em Modelo

CAMPINAS
2016

Thaís Harumi Ussami

Incremental Tests in a Model Based Test Driven Development

Testes Incrementais em um Desenvolvimento Guiado Por Testes Baseados em Modelo

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientadora: Profa. Dra. Eliane Martins

Este exemplar corresponde à versão final da Dissertação defendida por Thaís Harumi Ussami e orientada pela Profa. Dra. Eliane Martins.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): CNPq, 151647/2013-5

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Us7i

Ussami, Thaís Harumi, 1990-
Incremental tests in a model based test driven development / Thaís Harumi Ussami. – Campinas, SP : [s.n.], 2016.

Orientador: Eliane Martins.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Software - Testes. 3. Teste baseado em modelos. 4. Desenvolvimento ágil de software. I. Martins, Eliane,1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Testes incrementais em um desenvolvimento guiado por testes baseados em modelo
Palavras-chave em inglês:
Software engineering
Computer software - Testing
Model-based testing
Agile software development
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Marcos Lordello Chaim
Cecilia Mary Fischer Rubira
Data de defesa: 29-02-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Thaís Harumi Ussami

Incremental Tests in a Model Based Test Driven Development

Testes Incrementais em um Desenvolvimento Guiado Por Testes Baseados em Modelo

Banca Examinadora:

- Profa. Dra. Eliane Martins
  IC/UNICAMP
- Prof. Dr. Marcos Lordello Chaim
  EACH/USP
- Profa. Dra. Cecilia Mary Fischer Rubira
  IC/UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 29 de fevereiro de 2016

Acknowledgements

I would like to firstly thank God for blessing me during this journey, providing me with faith, courage and strength, and mainly for illuminating my path.

Thank you Elza, Kazuo and Vitor for living with me the true meaning of family. Thank you for being my safe harbour, for supporting me during all circumstances, for having faith on me, for encouraging me during the hardest moments, for all your advice, and for your comprehension. Thank you very much for all your love and support, Mom, Dad, Brother.

Thank you Janito for being at my side since university, and for always being ready to help me no matter what. Thank you for calming me down, for giving me strength to carry on, for your patience, and for always listening to me and supporting me. Thank you very much for being at my side during all this journey with all your kindness and love.

Thank you Professor Eliane Martins for your trust on choosing me to be one of your advisees, your patience, and specially your dedication in supporting me. Thank you for sharing with me your knowledge and experience, which provided me with professional and personal growth. Without your support I wouldn't have been able to conclude this journey. It was an honor to meet and work with a brave woman like you.

Thank you all my friends, Maria, Angélica, Leonardo, Sandra, Bruno B., Talita, Jessica, Renan, Lucas, Bruno S., Victor, João Paulo, for always encouraging me and cheering for me, and for your brotherhood. Without your support it wouldn't have been the same.

Thank you my new friends from Unicamp, Wallace, Anderson, Juliana, Rosana, Sheila, Rodrigo, Leydi and Lucas, for all the laughs and all your help.

Thank you all the members from the DEVASSES project for all the support, and for the opportunity to be part of this project which provided me the opportunity to grow professionally. A special thank you for all UniFI colleagues, Andrea Bondavalli, Andrea Ceccarelli, Paolo, Leonardo, Andréia, Tommaso, Marco, Enrico, Nicola, Martina, Valentina, Lucia, who kindly welcomed me in Florence, and who eased my stay so far from home; I will never forget all the memorable moments that I lived with all of you. I also would like to give a special thank you for Leonardo Montechhi, who helped me very much with this work. Thank you for your dedication, insights, and support, Leo.

Thank you all professors and employees from the Computer Institute of Unicamp, for all the infrastructure and professionalism. A special thank you for Professor Cecilia Rubira and Professora Ariadne Carvalho for sharing your experience with me and for your support. I would like to also thank the grant 151647/2013-5, CNPq, for the financial support.

Resumo

O desenvolvimento de sistemas pode ser realizado seguindo diversos modelos de processo. Os métodos ágeis propõem realizar implementações iterativas e incrementais e testes antecipados, buscando uma validação antecipada do sistema. Algumas técnicas ágeis adicionam a característica de um desenvolvimento de sistema baseado em testes, como as técnicas de Desenvolvimento Baseado em Teste (do inglês *Test-Driven Development (TDD)*) e Desenvolvimento Baseado em Comportamento (do inglês *Behaviour Driven Development (BDD)*). Recentemente algumas técnicas propõem a união de técnicas ágeis de desenvolvimento baseado em testes com técnicas consolidadas da área de testes, com o objetivo principal de auxiliar na etapa de criação de testes, que serão utilizados para guiar o desenvolvimento do sistema. Um exemplo é a técnica de Desenvolvimento Guiado por Testes Baseados em Modelo (do inglês *Model-Based Test-Driven Development (MBTDD)*) que une os conceitos de Testes Baseados em Modelo (do inglês *Model-Based Testing (MBT)*) e Desenvolvimento Baseado em Teste (TDD). Portanto em MBTDD, testes são derivados de modelos que representam os comportamentos esperados do sistema, e baseado nesses testes, o desenvolvimento iterativo e incremental ocorre. Entretanto quando lidamos com processos iterativos e incrementais, surgem problemas decorrente da evolução do sistema, como por exemplo: como reutilizar os artefatos de testes, e como selecionar os testes relevantes para a codificação da nova versão do sistema. Nesse contexto, este trabalho explora um processo no qual o desenvolvimento ágil de sistema é guiado por testes baseados em modelos, com o enfoque no auxílio do reúso dos artefatos de testes e no processo de identificação de testes relevantes para o desenvolvimento de uma nova versão do sistema. Para tanto, características do processo de MBTDD são unidas com características de uma técnica que busca o reúso de artefatos de testes baseado em princípios de testes de regressão, denominada Testes de Regressão SPL Baseados em Modelo Delta (do inglês *Delta-Oriented Model-Based SPL Regression Testing*). Para realizar a avaliação da solução proposta, ela foi aplicada em exemplos existentes e comparada com a abordagem no qual nenhum caso de teste é reutilizado.

Abstract

Systems can be developed following different process models. Agile methods propose iterative and incremental implementations and anticipating tests, in order to anticipate system validation. Some agile techniques add the characteristic of development based on tests, like in Test-Driven Development (TDD) and Behaviour Driven Development (BDD). Recently some techniques proposed joining the agile techniques of development based on tests with techniques consolidated in the field of testing, with the main purpose of aiding in the test creation stage, which are used to guide the development of the system. An example is Model-Based Test-Driven Development (MBTDD) which joins the concepts of Model-Based Testing (MBT) and Test-Driven Development (TDD). Therefore in MBTDD, tests are derived from models that represent the expected behaviour of the system, and based on those tests, iterative and incremental development is performed. However, when iterative and incremental processes are used, problems appear as the consequence of the evolution of the system, such as: how to reuse the test artefacts, and how to select the relevant tests for implementing the new version of the system. In this context, this work proposes a process in which the agile development of a system is guided by model-based tests, focusing on helping with the reuse of test artefacts and on the process of identifying tests relevant to development. To achieve this goal, MBTDD process characteristics are joined with characteristics from a technique that aims to find reusability of test artefacts based on principles of regression tests, called Delta-Oriented Model-Based SPL Regression Testing. To evaluate the proposed solution, it was applied to existing examples and compared to the approach that does not reuse any test cases.

List of Figures

2.1 TDD Cycle . . . 20
2.2 Model-Based Testing Process . . . 23
2.3 Model-Based Testing Artefacts (extracted from [36]) . . . 24
2.4 Finite State Machine Example . . . 25
2.5 Model-Based Test-Driven Development Process (adapted from [47]) . . . 26
2.6 Execution Graph of the Model-Based Testing Driven Development Steps . 27

3.1 Incremental Evolution of SPL Test Artefacts (extracted from [36]) . . . 32

5.1 D-MBTDD process for the first iteration . . . 39
5.2 Core State Machine Example . . . 40
5.3 D-MBTDD process for when the test model evolves . . . 41
5.4 Regression delta example . . . 41
5.5 New test model version example . . . 42
5.6 Process of updating the test suite during the iterations that follow . . . 43
5.7 Development cycle of a new feature . . . 45
5.8 Scrum process (Adapted from [1]) . . . 46
5.9 Process of D-MBTDD with Scrum . . . 46

6.1 M2 *Automatic Power Window* core test model (extracted from [34]) . . . 53
6.2 *Automatic Power Window* with *Central Locking system* test model version (M2_D1) (extracted from [34]) . . . 55
6.3 *M2_Delta_1* of M2: *DAddAutoPWCLS* delta model (extracted from [34]) 56
6.4 CoFI core test model . . . 57
6.5 *S04_Delta_1* delta model . . . 58
6.6 S04_D1 test model version . . . 59
6.7 Bar Graph with StateMutest results . . . 72
6.8 Bar Graph with Condado results . . . 72

List of Tables

4.1 Comparison of Existing Solutions 36

6.1 Core test model information of delta case study 53
6.2 Delta model information of delta case study 54
6.3 Information of test model versions of delta case study 54
6.4 Delta models information of CoFI 58
6.5 Information of test model versions of CoFI 58
6.6 Number of valid, new, reusable and obsolete test cases in StateMutest experiments 63
6.7 Value of metrics in StateMutest experiments 64
6.8 Number of valid, new, reusable and obsolete test cases in Condado experiments 68
6.9 Value of metrics in Condado experiments 69

B.1 Transitions added or removed of each delta model 89
B.2 StateMutest examples information 90
B.3 Quantity of generated test cases for tau values 91
B.4 Condado examples information 92

List of Abbreviations and Acronyms

- **BDD** *Behaviour Driven Development*
- **CASE** *Computer Aided Software Engineering*
- **D-MBTDD** *Delta Model-Based Test-Driven Development*
- **EFSM** *Extended Finite State Machines*
- **FSM** *Finite State Machines*
- **INPE** *Brazilian Institute for Space Research*
- **MBCF** *Model-Based Control Flow*
- **MDD** *Model Driven Development*
- **MDPE** *Model Driven Performance Engineering*
- **MBT** *Model-Based Testing*
- **MBTDD** *Model-Based Test-Driven Development*
- **SPL** *Software Product Line*
- **SWPDC** *Software for a Data Collection Platform*
- **SXM** *Stream X-Machines*
- **TDD** *Test-Driven Development*
- **UML** *Unified Modelling Language*

Contents

1 Introduction **13**
- 1.1 Context . . . 13
- 1.2 Problem Identification . . . 15
- 1.3 Proposed Solution and Objectives . . . 16
- 1.4 Research Questions . . . 16
- 1.5 Contributions . . . 16
- 1.6 Document Organization . . . 17

2 Test First Development **19**
- 2.1 Test-Driven Development . . . 19
- 2.2 Behaviour Driven Development . . . 21
- 2.3 Model-Based Testing . . . 22
  - 2.3.1 Finite State Machine . . . 25
- 2.4 Model-Based Test-Driven Development . . . 25

3 Regression Testing Techniques **28**
- 3.1 Regression Testing . . . 28
  - 3.1.1 Test Cases Classification . . . 29
- 3.2 Delta-Oriented Model-Based SPL Regression Testing . . . 31

4 Related Work **33**
- 4.1 Model-Based Testing in an Agile Context . . . 33
- 4.2 Model-Based Regression Testing . . . 34
- 4.3 Reuse of Test Models . . . 35
- 4.4 Summary of the Study . . . 36

5 Proposed Solution: D-MBTDD method **37**
- 5.1 Assumptions . . . 37
- 5.2 D-MBTDD Process . . . 38
  - 5.2.1 Process for the First Iteration . . . 39
  - 5.2.2 Process for the Next Iterations . . . 40
- 5.3 D-MBTDD with Scrum . . . 45

6 Evaluation of the Proposed Solution **47**
- 6.1 Definition of the Experiment . . . 47
  - 6.1.1 Metrics . . . 48
- 6.2 Planning of the Experiment . . . 51
  - 6.2.1 Delta Case Study . . . 51
  - 6.2.2 SWPDC - Software for a Data Collection Platform . . . 56

6.3 Workflow of the Experiments . . . 59
6.3.1 Workflow for core model experiments . . . 60
6.3.2 Workflow for test model version experiments . . . 60
6.4 Controlled Experiment 1: StateMutest Experiments . . . 61
6.4.1 State Mutest . . . 61
6.4.2 Preparation . . . 61
6.4.3 Results . . . 62
6.4.4 Results Analyses . . . 66
6.5 Controlled Experiments 2: Condado Experiments . . . 67
6.5.1 Condado . . . 67
6.5.2 Preparation . . . 67
6.5.3 Results . . . 68
6.5.4 Results Analyses . . . 70
6.6 Discussion . . . 71
6.6.1 Results Common to both MBT tools . . . 71
6.6.2 Cost comparison between Regenerate-All and D-MBTDD . . . 74
6.7 Threats to the Validity of the Experiments . . . 75

7 Conclusions and Future Work **77**
7.1 Conclusions . . . 77
7.2 Limitations . . . 78
7.3 Answers to the Research Questions . . . 79
7.4 Future Work . . . 80

A Delta Information for Section 6.1.1 **87**

B Complementary Tables for Chapter 6 **89**

Chapter 1

Introduction

In a context of Model-Based Test-Driven Development, detailed in Section 1.1, Delta Model-Based Test-Driven Development (D-MBTDD) was proposed during this work to support the incremental test creation and maintenance. Section 1.2 describes how the problem was identified during this master graduate program. The proposed solution and its objectives are described in Section 1.3, and the research questions that this work evaluates are described in Section 1.4. The contributions of the proposed solution are described in Section 1.5. The document organization is detailed in Section 1.6.

1.1 Context

During traditional software development used in the 80's until the beginning of the 90's, tests were executed after the end of the system implementation, and possible problems were discovered only when the client started using the system. Therefore, disagreements between what was expected by the client and what was delivered by the developers could happen, resulting in a higher cost of maintainability and in a longer development. Analyses executed in the industry field concluded that less than 50% of the clients were satisfied with the quality of the delivered software, and only 33% were satisfied with the delivery time. The possible causes for this dissatisfaction can be the lack of interaction between the client and the developers, lack of details in the requirements definitions, management problems and/or lack of tests. Emphasizing the last one, an aggravating factor can be the lack of interaction between the clients and testers, so that testers are not updated about requirements changes [12].

Based on the problems that happen when traditional development is used, developers from the software engineering field started to propose agile approaches in the 90's. In these approaches, the software development phases (Planning, Analysis, Project, Implementation, Tests and Deliver) are executed iteratively and incrementally, with the aim of quickly delivering the system in order to include any requirements modifications in the next iterations [52].

Seventeen specialists in software development met in 2001, and motivated by these new agile approaches that aim to deliver software in time and with easier maintainability

when requirements change, created the *Agile Alliance*[1] and proposed the **Manifesto for Agile Software Development** [5], also known as **Agile Manifesto**.

The authors also proposed the "Twelve Principles of Agile Software" [6] that should be followed by developers in order to deliver a functional software. One of these principles proposes functional software delivery within small iterations, each lasting a few weeks. For each set of predefined feature implementations, the system is shown to the client in order to plan for modifications and new features for the next iteration. With these demonstrations, the client works together with the developers during the development cycle. This development based on features requires continuous code integration, therefore, tests are continually executed aiming to minimize defects in the final version of the system.

Agile development, based on iterative and incremental development, in which the tests are continuously executed, can be a solution to improve the client satisfaction with the final version of the system. Therefore, some agile approaches were created to emphasize tests. One of these techniques, proposed by Kent Beck, is Test-Driven Development (TDD) [4]. In TDD the module integrations and the tests are performed after the implementation; TDD proposes that unit tests are iteratively created before the Implementation phase. Thus, the Implementation Phase aims to execute successfully the test cases created previously.

Based on TDD, Dan North proposed Behaviour Driven Development (BDD) [41]. Differently from TDD, which focuses on system code, BDD focuses on system behaviour and on the collaboration among the people who are involved with the system. BDD proposes that business-oriented and technology-oriented people work together from the beginning of the system development, in order to anticipate the validation of the system.

However, in TDD and BDD there are difficulties with test case creation, which is usually done manually. There are no artefacts or guides to support it, therefore testers and developers report difficulties on defining which and how many test cases have to be created [41]. Moreover, during the life cycle of the system the test cases evolve and there is the need of reusability and the challenge of maintainability of the tests [42].

It is a common practice to create tests manually, in a non-systematic way. However, this practice is subject to human errors because of the repetitive process and the lack of guarantees that the system has a good test coverage, a measure that represents how complete the test set is. In order to support test case creation, the idea of generating test cases from models that represent the expected system behaviour can be used. This idea comes from Model-Based Testing (MBT) [54], in which formal test models that represent the system's behaviour are created and validated in order to automatically generate test cases from them.

Following this idea, Model-Based Test-Driven Development (MBTDD) [47] joins the concepts of MBT and TDD. Therefore, MBTDD proposes that test models are iteratively created in order to represent enough information for the current iteration, and from these models and using concepts and techniques of MBT, test cases are automatically generated. These tests guide the development of the system by using concepts and techniques of TDD.

However, in an incremental development that is model-based, first the model is changed to specify new behaviour and, then, new test cases should be derived to guide the devel-

[1] `http://www.agilealliance.org/`

opment of the new feature. Therefore, there is the problem of how to reuse test artefacts during the iterative and incremental development process. There are also the problems of identifying which test cases are reusable, which ones should be removed, which ones have to be created and which ones support the new feature development.

Regression Testing supports the analysis of how adequate the test suite is, which test cases are still valid, which ones are not, and which ones have to be created [45]. In the context of Software Product Line (SPL), Delta-Oriented Model-Based SPL Regression Testing joins concepts of Regression Testing with Model-Based Testing of SPLs and proposes an approach in which delta modelling concepts are used to express the variability between the product variants. It aims for test artefact reusability and with the support of regression deltas, that explicitly represent differences among variants, it determines which existing test cases are valid for a product variant and which ones have to be created.

There are some works that propose model-based test-driven development [11, 55, 57], however they deal only with test model reuse, they do not deal with test case reuse and with the identification of test cases that support the new feature development.

1.2 Problem Identification

This section describes how the problem was identified and a suitable solution was envisaged. During the period of this master graduate program, besides the related work, some activities supported the definition of the D-MBTDD process:

- Informal interviews with employees who use BDD in practice were performed in order to better understand the practical use of BDD and its challenges. With these interviews it was possible to confirm that there are difficulties with test case creation and maintenance.

- A six month exchange and mobility program at Resilient Computing Lab (RCL)[2] research group at the Department of Mathematics and Informatics of the University of Florence (UniFI)[3] was realized as part of DEVASSES[4] project. During this period, Professor Andrea Bondavalli and the post-doc Leonardo Montecchi collaborated on the definition of the process of D-MBTDD. Furthermore, after the return to Brazil they continued to support with the process definition.

- During a course ministered by Professor Eliane Martins, some groups of students reused test cases derived from state machines during a Model-Based test-driven development. In order to reuse the test cases, they followed a preliminary version of D-MBTDD process. With this, it was possible to identify some necessary improvements on D-MBTDD process.

[2] http://rcl.dsi.unifi.it/home
[3] www.unifi.it
[4] http://www.devasses.eu/

1.3 Proposed Solution and Objectives

This work proposes a solution based on MBTDD and Delta-Oriented Model-Based SPL Regression Testing, which supports the development of new features by means of test case reusability. Finite State Machines (FSM) are used to represent the system behaviour, and test cases are derived from them. When the system evolves, there is the reuse of the test model and the test cases. Therefore, test cases from the previous version are analysed in order to identify which ones are still valid and consequently can be reused. After that, new test cases are created from the new test model version in order to update the new test suite. The new test cases support the development of new features, while the reusable test cases are used as regression tests.

Therefore, the objectives of this work are to:

- Help with the test case creation, by using Model-Based Testing tools and techniques, and finite state machines as the behavioural model of the system.
- Help with the incremental development, by using ideas from TDD, BDD and MBTDD.
- Help with the reuse of test artefacts when the system and consequently the test model evolves, and with the identification of the test cases that will support the development of new features, by using ideas from Delta Model-Based SPL Regression Testing.

1.4 Research Questions

In order to evaluate the proposed method D-MBTDD, the following research questions were defined:

1. **RQ1:** When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require less effort for test case creation, when compared to an approach in which there is no test artefact reuse?
2. **RQ2:** When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require less effort for the identification of which test cases should guide the development of new features, when compared to an approach in which there is no test artefact reuse?
3. **RQ3:** When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require a total effort smaller than when compared to an approach in which there is no test artefact reuse?

1.5 Contributions

In this work a method which aims for reusability of test artefacts in an iterative and incremental model-based test-driven development, entitled Delta MBTDD (D-MBTDD),

was proposed. In an environment in which the test models are created iteratively and incrementally in order to represent the versions of the system, D-MBTDD contributes with not only the reuse of the test models, but also of the test cases. D-MBTDD contributes with the testers in different aspects by:

- **Reusing the test model:** the tester reuses the previous version of the test model, and only model the necessary modifications of the new version of the system.
- **Reducing the effort on the selection of test cases that guide the development of new features:** the set of new test cases guides the development of new features, and they are already identified. Therefore, effort for this task is not required from the tester.
- **Reducing the effort on transformation of abstract test cases:** because test cases from the previous version that are still valid for the new version are reused, the tester only has to transform the new abstract test cases into concrete test cases.

In addition, D-MBTDD contributes with the development of the new version by using the reusable test cases as regression tests.

1.6 Document Organization

This chapter introduced the motivations, objectives and research questions of this work. The rest of this document is organized as follows:

- **Chapter 2 - Test First Development:** presents some agile methods that propose a development based on tests: Test-Driven Development (TDD), Behaviour Driven Development (BDD) and Model-Based Test-Driven Development (MBTDD). Moreover, the concepts of Model-Based Testing (MBT) are also presented.
- **Chapter 3 - Regression Testing Techniques:** presents the fundamental concepts of Regression Testing, the test case classification and introduces Delta-Oriented Model-Based SPL Regression Testing, a Model-Based Regression Testing in the software product line context.
- **Chapter 4 - Related Work:** presents some related work that deal with model-based testing in an agile context, with model-based regression testing, and with test artefact reuse.
- **Chapter 5 - Delta MBTDD method:** presents the proposed method of this work, with its steps and processes used during the iterations of the system. Moreover, a process which joins Delta MBTDD with Scrum is presented.
- **Chapter 6 - Experiments:** presents the executed controlled experiments, in which Delta MBTDD was compared to the approach in which test cases are not reused when the system evolves. The Examples and the MBT tools used during the controlled experiments are described, as well as the results and their analyses.

- **Chapter 7 - Conclusions and Future Works:** presents the conclusions about Delta MBTDD, the answers to the research questions and the future works.

Chapter 2

Test First Development

The Test First Programming concept emerged with Extreme Programming (XP) [3], which proposed that unit tests were created before the code. With that, the developers could know what was necessary to be implemented. After the Agile Manifesto some techniques proceeded with the idea from XP and proposed software development processes based on tests. A test case succeeds when the outputs generated from the system meet the expected outputs, otherwise it fails. Some of these techniques are explained in this Chapter. Test-Driven Development(TDD) is explained in Section 2.1 and Behaviour Driven Development(BDD) is explained in Section 2.2. In Section 2.3 some concepts about Model-Based Testing(MBT) are explained, and finally in Section 2.4 a technique that combines MBT and TDD, entitled Model-Based Testing Driven Development (MBTDD), is explained.

2.1 Test-Driven Development

During the development of a software there is the concern about the quality of the development process and of the product that will be delivered to the customers. Because of that, the software has to be tested in order to guarantee that the implementation fulfils all the requirements. A general software testing process involves the creation of a test suite, the execution of this test suite on the system under test (SUT) and the analysis of the outputs of the test cases. A test suite is a finite set of test cases, and a test case is composed by inputs and expected outputs[54]. Besides assisting with the system verification, the test cases can also be used to guide the development of the system, as proposed in Test-Driven Development(TDD) [4]. This technique aims to produce a "clean code", which in other words is code that is easier to be maintained. To obtain that, TDD has its main characteristic of writing tests before the implementation phase. The idea is that the tests guarantee that the requirements are implemented according to the specifications provided by the customers. And further, the tests are now part of the software development process, and they are not only executed after the code is done.

TDD is known by its cycle (Figure 2.1), composed of three phases: *Red*, *Green* and *Refactor*. During the Red phase a test case has to be written and executed before the implementation of the software starts, thus the test will fail because there is no imple-

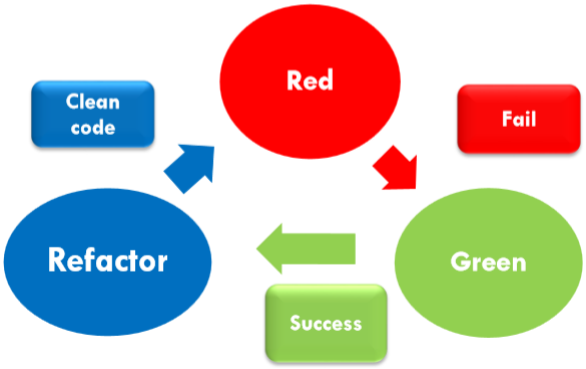

Figure 2.1: TDD Cycle

mentation yet. In the Green phase code is written that will make the test be successfully executed, not mattering if good programming practices are used. In the Refactor phase the logical duplications that may have been written during the Green phase are removed. These logical duplications can be responsible for dependencies between the tests and the code. These removals have the purpose of allowing to change both the tests and the code without affecting each other. Finally, at the end of the Refactor phase a code that is more legible and that is easier to be maintained is created [4].

The goal of TDD is to incrementally write test cases for each function, repeating the process until the software implementation is completed, or in other words until all the test cases are successfully performed [50].

Besides producing a "clean code", TDD also allows using the test cases as regression tests, encouraging the developers to confidently apply changes in the code and therefore allowing more reliable and flexible software. Because the test cases guide the development, the test cases can be used as a documentation synchronized with the code, helping with the comprehension of the code. Despite having advantages, TDD has some disadvantages that limit its use. One of them is that TDD focuses on unit tests, which focus on low level software code and do not deal with more abstract structures. But the main disadvantage of TDD is the effort to create the test cases. Because the process of test case creation is manual, writing test cases for each piece of code is time consuming and decreases the productivity of the development team [47].

Some works analysed the product quality and the productivity of the development team when applying TDD in academic or industrial projects. Maximilien and Williams performed a case study at IBM in which the use of TDD reduced 40% of the defects when compared with an ad-hoc process, and had minimum impact on the productivity of the development team [37]. Similar conclusions are presented in Saiedian and Jalote, that applied TDD to academic projects [28, 30]. However, there are some works that demonstrated the opposite. George and Williams performed a case study at AT&T in which even though the use of TDD assisted with fault detection, the programming time increased 16% when compared with traditional development [25]. Similar conclusions are presented in academic case studies performed in Nagappan et al. and Canfora et al [13, 38].

2.2 Behaviour Driven Development

When using Test-Driven Development (TDD), some developers reported some misunderstandings and a confusing experience. According to some of them, TDD does not provide the information about where to start testing, what to test, what not to test, how much to test and the reasons why the tests fail [41]. With the attempt to improve TDD, Dan North introduced a new agile software development technique entitled Behaviour Driven Development (BDD), in which instead of writing the tests in a code format like in TDD, the tests are now written with a natural language allowing an easier validation of the system by the customers [20]. The use of a natural language eases the communication between business-oriented and technology-oriented people by creating a representation in a ubiquitous language, or in other words, a common language that provides communication with no ambiguities among the analysts, the testers, the developers and the business people and that helps to specify correctly the system behaviour. Therefore, BDD provides a collaboration between the project members [51].

BDD has the characteristic of having the development guided by the expected resulting behaviour, thus the expected behaviour is considered during all the development phases of the system. Firstly, user stories are used to represent a feature of the system that was derived from the expected business outcomes [15, 51]. Using the ubiquitous language, the user stories are written following a template:

As a [role]
I want to [feature]
So that [benefit/value]

This template presents the required feature, its benefit or value for the system, and who will benefit from it. Therefore, before defining a user story it is necessary to think about the benefit or the value that the feature will produce on the system[15, 51].

There are many different contexts and expected behaviours for each feature, so for each user story some scenarios are derived to represent the expected system behaviours in specific contexts. Differently from TDD that uses unit tests to guide the development, BDD now uses acceptance tests for it, which are represented by the scenarios. The unit tests are used to test low-level SUT parts, such as classes or functions. The acceptance tests, instead, use high level SUT details and are used to check if the system requirements were correctly implemented. Therefore, the set of scenarios must facilitate the understanding of the feature and the verification that the system meets the requirements [15, 51]. Just like the user stories, the scenarios also have a template:

Given [context]
When [event]
Then [outcome]

The Given-When-Then structure of the scenarios, similar to the human concept of cause and effect, facilitates an intuitive thinking of input-process-output of the system, therefore facilitating the validation of the system by the customer [20]. Moreover, as explored by Carvalho et al., the BDD scenario structure is very similar to a description

of state transitions and therefore, the scenarios could be represented by Finite State Machines (FSM) [17, 18].

Chiavegatto et al. performed a case study to identify the benefits and disadvantages with the use of BDD. Some conclusions were that BDD eased the understanding of what had been developed, supported an easier communication between business-oriented and technology-oriented people, and eased the execution of the tests. The case study indicated that the scenarios supported a better comprehension of what is necessary to implement for all the stakeholders, since the use of a natural language helped to understand the system and the communication between the stakeholders. However, the case study affirms that there is a necessary effort with the definition of scenarios, in order to define scenarios with sufficient information to support the development process and the quality of the system [16].

Rahman and Gao indicated some requirements and challenges with the use of BDD. The system is implemented incrementally and the requirements change frequently, so in order to reflect these changes the scenarios have to be updated. Moreover, during the life cycle of the system some scenarios can be reusable and refactored, therefore there is the need of reusability and the challenge of maintainability of the scenarios. Furthermore, because testers and developers do not always work together, there is the necessity of controlling the versions of the scenarios that the testers and the developers have [42].

2.3 Model-Based Testing

Software testing is the process of analysing a system manually or automatically in order to verify if the actual behaviour meets the intended one, and to detect the differences between them. Model-Based Testing is a variant of testing that uses explicit behaviour models that represent the intended behaviours of the system [54].

Conventional test methods, such as hands-on testing, are subject to human error because of the repetitive process and the lack of guarantees that the system has a good test coverage, a measure that represents how complete the test set is and that assists with the generation of the test cases. The behaviour of the system under test is constantly changing, hence the handcrafted test cases have to be adapted to meet the new behaviour, which can be a costly process. Also, a hands-on testing produces a limited number of test cases [43]. Model-Based Testing solves these problems by using the behaviour models to represent the behaviour of the system and to assist with the test case generation [54].

The models also have to be validated in order to check if the system behaviour is correctly represented. In order to do so, the models have to be simpler than the SUT and ease the process of checking, modifying and maintenance. Otherwise, the effort of validating the model would be equal to or greater than the effort of validating the SUT. On the other hand, the models must have a sufficient level of abstraction in order to guarantee the generation of relevant test cases, which are those capable of detecting faults in the system implementation [54].

A generic model-based testing process is illustrated in Figure 2.2 and is composed of seven steps [10][54][53, p. 27-30].

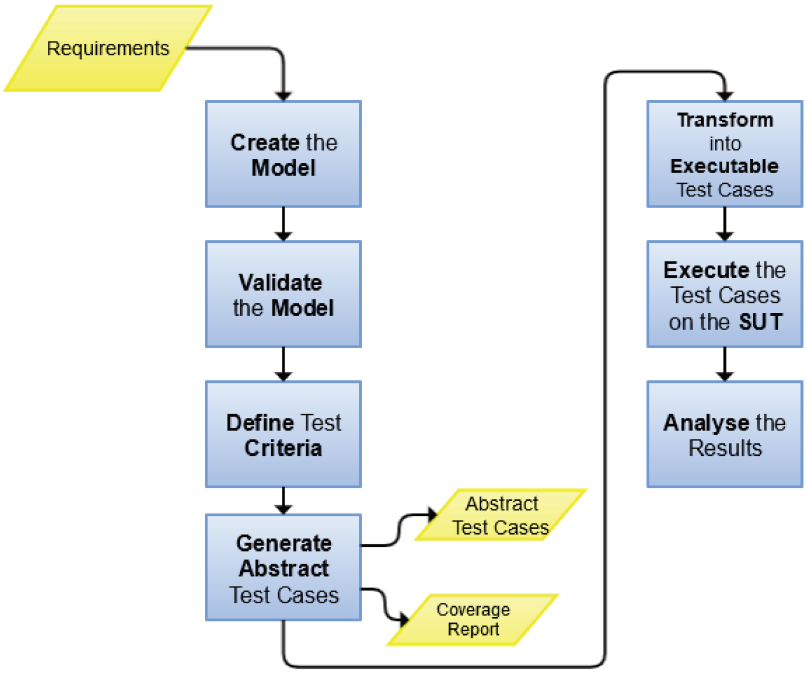

Figure 2.2: Model-Based Testing Process

Firstly a model of the SUT, which represents the intended behaviour, is built based on the system requirements. After writing the model, the second step is to validate the model in order to check if the model is consistent and if it represents the desired behaviours.

The third step is to define a test selection criterion that will indicate which test cases should be created from the model and that consequently will assist with the creation of the test suite. There are different test selection criteria, such as those that focus on a particular part of the model or those related to a particular model coverage criterion, such as all-transitions coverage or all-states coverage.

From the model and considering the test selection criterion, the abstract test cases are generated, which is the fourth step. Abstract test cases consist of a sequence of operations from the model, but they are not executable so they can not be directly executed on the SUT without a transformation. A coverage report is a possible additional output of this step. This report helps to analyse how well the generated test set exercises the complete behaviour of the model and helps to identify parts of the model that may not be well tested.

The fifth step is to transform the abstract test cases into executable test cases. After this concretion, the executable test cases are ready to be executed against the SUT. Therefore, the sixth step is to execute the executable test cases on the SUT and consequently to generate the results, which consist of the association of the executable test cases, their generated outputs and their expected outputs. These results are analysed during the seventh step, when the amount of test cases that succeed and that fail is analysed. A test case succeeds when the outputs generated from the system meet the expected outputs, otherwise it fails. Another possibility is to have an inconclusive result, which means that it is not possible to conclude something based on the available information.

When considering test case generation and test execution, a model-based testing technique can be classified as *online* or *offline*. With offline testing the test generation is performed before their execution, while with on-line testing the test generation algorithms can execute the test cases dynamically. Because offline testing generates the test cases before their execution, if the test generation process is slower than test execution, the set of test cases can be generated once and then executed many times on the system under test. It can be useful, for example, for regression testing purpose when the set of test cases is executed any time a modification is performed into the system. Besides the fact that test generation and test execution can be performed at different times, they can also be performed on different environments [54].

The fundamental test artefacts involved in model-based testing techniques are summarized in Figure 2.3. As explained before, a test model (tm) specifies the intended behaviours of the SUT. Based on a coverage criteria (CC) a finite set of test goals (TG) is selected from the test model, i.e., structural elements of tm. A test suite (TS) contains a finite set of valid test cases, which are those whose executions conform to the behaviours specified by the test model. For a test suite TS to satisfy a coverage criteria, each selected test goal $tg \in TG$ has to be covered, i.e., traversed by at least one test case $tc \in TS$ [33, 36].

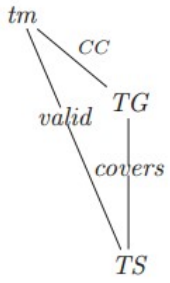

Figure 2.3: Model-Based Testing Artefacts (extracted from [36])

Software testing can be carried out at any time during the development process. However, when the tests are created based on the system code, most of the test effort occurs after the implementation phase has been completed. Instead, model-based testing approach has the advantage of allowing the models to be created in the beginning of the development cycle. With the model available sooner, the testing process can be applied in parallel with the development phase [19]. The use of models also eases the understanding of the system, helping with the requirements validation, leading to the discovery of inconsistencies in the specification and therefore avoiding errors in the system. Furthermore, the use of models to generate the test cases introduces variability in the tested behaviours and a larger number of test cases, improving the test coverage and therefore the final quality of the system [43, 46].

Another advantage is that a model-based testing approach is more efficient for systems that constantly evolve, because the model maintenance is easier than the manual test case maintenance. When the system evolves, with model-based testing it is possible to simply update the models and then generate the test cases that meet the new requirements.

However, if the test cases are transformed into executable ones manually, test cases from the previous iteration can be reused in order to reduce the effort of transforming the test cases of the current iteration. Therefore, it is possible to reuse some test artefacts during the development cycle, such as the test cases and the test models [40, 43, 46]. However, when test cases are reused there is the problem of how to identify which test cases could be reused, as well as which ones have to be deleted, or generated [19].

2.3.1 Finite State Machine

Model-Based Testing can use different test models like state machine diagrams, class diagrams, sequence diagrams, activity diagrams and other UML (*Unified Modelling Language*) models. During this work only finite state machines (FSM) were used. In the rest of this document, in order to simplify, finite state machines are also referenced as FSM or simply state machines.

Finite state machines have been widely used to model systems in different areas, such as communication protocols, real time systems and sequential circuits. A finite state machine M is a quadruple $M = (I, O, S, T)$, where I is the finite set of input actions, O is the finite set of output symbols, S is the finite set of states, and $T : S \times I \rightarrow S \times O$ is the finite set of transitions. Each transition $t \in T$ is a 4-tuple $t = (sj, i, o, sk)$ consisting of start state $sj \in S$, input symbol $i \in I$, output symbol $o \in O$ and next state $sk \in S$.

An FSM can be represented by a directed graph whose vertices are labelled as the states of the state machine and whose edges correspond to the transitions. Each edge is labelled with the input action and the output associated with the transition, and the initial state is identified with a pointer arrow, as Figure 2.4 illustrates.

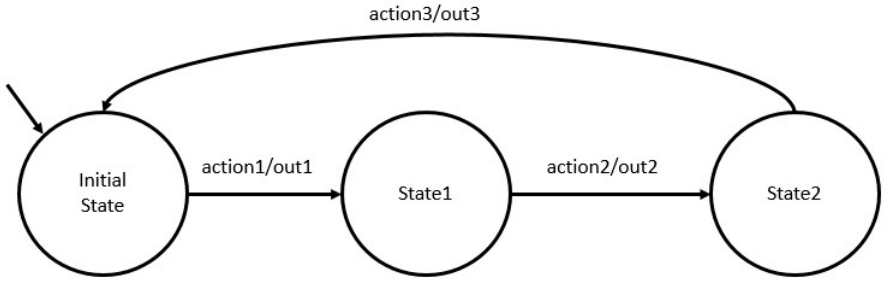

Figure 2.4: Finite State Machine Example

2.4 Model-Based Test-Driven Development

As presented in Section 2.1, a development guided by the test cases as proposed in Test-Driven Development grants a better comprehension of what should be developed, and consequently an earlier validation of the system. On the other hand, TDD has some limitations such as the focus on the low level software programming and coding details, and the time consuming task of manually creating the test cases, decreasing the productivity of the development team. As presented in Section 2.3, Model-Based Testing is a technique that aims to use behaviour models to assist with the generation of the test cases. With

the aim of improving the development guided by test cases, Sadeghi and Hosseinabadi proposed Model-Based Test-Driven Development (MBTDD), a technique that combines MBT and TDD techniques. MBTDD takes advantage of the benefits of TDD and MBT and tries to overcome the limitations of TDD [47].

In the MBTDD process, the TDD cycle is extended with MBT steps. Differently from techniques that focus on the system implementation phase, MBTDD relies on creating models that assist the generation of the test cases. Furthermore, like TDD, MBTDD aims to guarantee the possibility of earlier verification and validation of the system, so the test cases are created before the implementation phase and they are used to guide the development. The MBTDD process is composed of three main steps, as illustrated in Figure 2.5:

1. **Modelling Step:** the model that will assist the automatic generation of test cases is created.

2. **Model-Based Testing Step:** using the MBT technique, the model assists in the generation of abstract test cases and subsequently they are transformed into executable test cases.

3. **Test-Driven Programming Step:** using the TDD technique, the development is guided by the test cases. Code is written to successfully execute the test cases and consequently to satisfy the intended requirements.

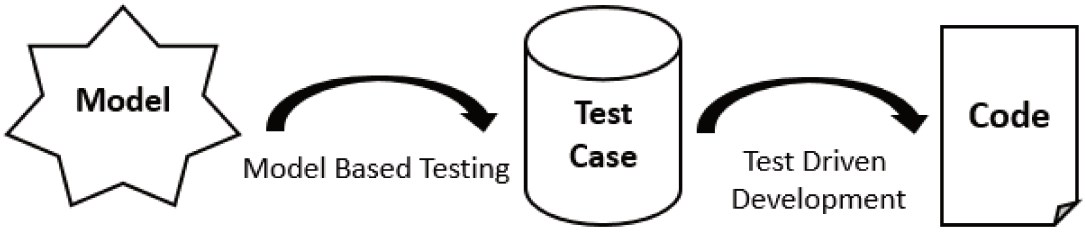

Figure 2.5: Model-Based Test-Driven Development Process (adapted from [47])

MBTDD proposes that the process is used during each abstraction level, so for example in a three-tier architecture the process should be used in the User Interface, Business Logic and Persistence layer. However, as illustrated in Figure 2.6, the order in which the MBTDD steps are executed differs. The first two steps of MBTDD (Modelling Step and Model-Based Testing Step) are executed in a top-down approach, therefore first the high level model and test cases are created and the process continues until the low level model and test cases are created. Conversely, the order of the Test-Driven Programming step execution is the opposite, a bottom-up approach. Therefore, first the necessary code to make the execution of the low level tests succeed is written and the development process continues until the corresponding code to make the high level tests succeed is written. Using the previous three-tier architecture example, the modelling step and the test case creation step start in the user interface level, pass trough the business logic level and end

in the persistence layer. However the codification is executed in the opposite order, so firstly the persistence code is written, then the business logic code and finally the user interface code.

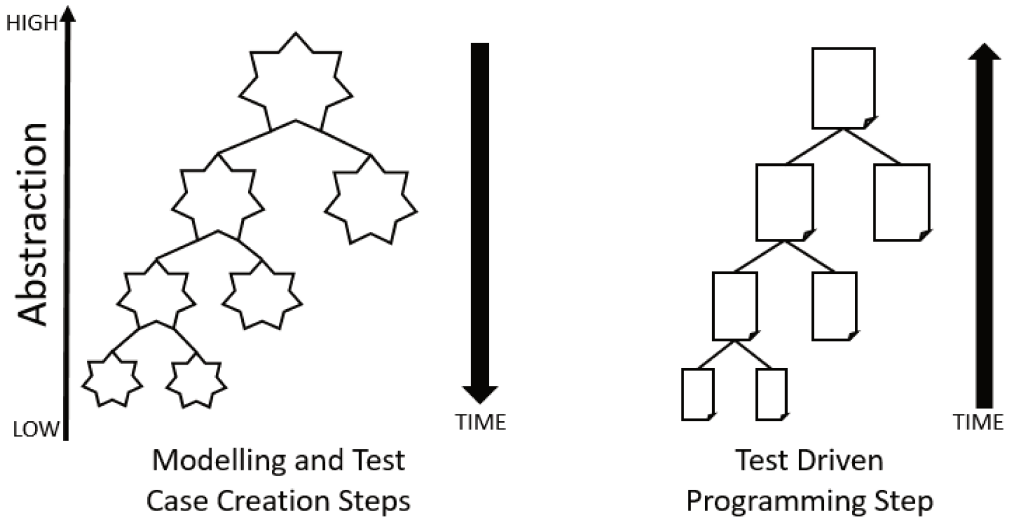

Figure 2.6: Execution Graph of the Model-Based Testing Driven Development Steps

MBTDD was evaluated in an industrial environment, in which a Human Resource Management (HRM) web application was implemented. The system was configured in a three-tier architecture and therefore user interface, business logic and persistence tests were created from their models to assist the development phases. The quality of the product and the efficiency of the production process were evaluated. The quality of the process was measured by the number of faults per kilo lines of code (KLOC), and the efficiency was evaluated by the development time, or in other words the total effort of human resources to accomplish the system. It was concluded that the quality of the product improved, while the efficiency decreased a little. This reduction was due to the extra effort of creating the models and transforming the abstract test cases into executable ones, in order to achieve the quality improvement. Nevertheless, the authors justified this extra cost with less future maintenance efforts, because of the higher quality of the product and the fewer number of faults.

Chapter 3

Regression Testing Techniques

In an iterative and incremental environment the system constantly evolves and there are frequent modifications to be made and new features to be added. In this context of evolution, regression testing is used to support the maintainance process. Regression testing is performed on a modified program to provide confidence that the changes are correct and have not adversely affected unchanged portions of the program. In Section 3.1 some basic concepts of Regression Testing are explained. Then in Section 3.2 an approach of model-based regression testing is explained.

3.1 Regression Testing

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production [1] (apud [45]). One necessary but sometimes expensive maintenance task is regression testing. It is a testing activity that is performed between two different versions of the same software in order to provide confidence that the modifications did not affect the unmodified parts of the system [61].

In agile methodologies, the software development life cycle is usually short and this imposes limitations on performing regression testing within limited resources and time [61]. Also, the development is iterative and incremental, so there are frequent changes and new features to be developed along the software life cycle. Therefore, in this context of constant modifications, it is important to try to find the limitations of the test suite for testing a modified system, and find out whether new tests might be created [45].

Most of regression testing techniques address the problem of how to guarantee confidence in a modified version of a system. Considering a program P, its modified version P', the original test set T that was used previously to test P, and the test set T' that tests the new version of the system, usually regression testing techniques have the following steps [44, 45]:

1. Select $T' \subseteq T$, the test cases to be executed on P'.

2. Apply T' on P', establishing the correctness of P' with respect to T'.

[1] S. Schach. Software Engineering. Aksen Associates, Boston, MA, 1992.

3. If necessary, create T'', a set of test cases that cover new functionalities of P'.

4. Apply T'' on P', establishing the correctness of P' with respect to T''.

5. Create T''', the new test suite and test history for P', from T, T' and T''.

A regression testing technique addresses four problems when performing the steps above. Step 1 addresses the problem of *regression test selection*, that consists on how to select a subset T' from T, which will test P'. Step 3 addresses the problem of *coverage identification*, that consists on how to identify the parts of P' that require new tests. Steps 2 and 4 address the problem of *test suite execution*, that consists on how to efficiently execute tests and check the results for correctness. And finally, Step 5 addresses the problem of *test suite maintenance*, that consists on how to update and store the test information [45].

The simplest approach of regression testing is the *retest-all*, in which all the existing test cases in the original test suite are executed ($T' = T$). However as software evolves, the test suite tends to become larger and consequently the cost to execute the entire test suite increases. Thus, the *retest-all* approach may consume excessive amounts of time and resources. Some techniques seek to reduce the resources required for regression testing in different ways. The three major branches are: minimisation, selection and prioritisation [61].

Test suite minimisation techniques aim to reduce the size of the test suite by identifying and eliminating redundant test cases from the test suite, in order to reduce the number of tests to run. The redundant test cases are those that have the same input and output for a specific context. The minimisation process is also called 'test suite reduction'. However, the reduction process produces a temporary subset of the test set whereas the minimisation process permanently eliminates test cases [61].

Test case selection aims to identify and select test cases relevant to the modified parts of the software. The test case selection problem is similar to the test suite minimisation one, because both of them seek to identify a subset of test cases from the test suite. However, the difference between them is whether the focus is on a version of a system or the changes in the system. Test suite minimisation frequently is based on metrics such as coverage measured from a single version of the system. By contrast, test case selection selects the test cases relevant to the modifications between the previous and the current version of the system [61].

Test case prioritisation aims to identify the optimum ordering of test cases that maximises desirable properties, such as fault detection. It does not involve selection of test cases, and assumes that all the test cases may be executed in a specified order affected by the fact that the test execution may be terminated at any arbitrary moment [61].

3.1.1 Test Cases Classification

According to Leung and White, the test cases can be categorised. After a modification is made to the software, the test cases from the previous test suite can be classified as:

- **Reusable:** reusable test cases execute the parts of the software that remain unmodified between two versions, in other words, those parts that are common to the two versions. The reusable test cases are executed in the new version in order to verify if they will provide the same results as in the previous version. Furthermore, these test cases are classified as reusable because they may be reused for the regression testing of the future versions of P.

- **Retestable:** retestable test cases execute the parts of the software that have been modified in the new version. Thus, the retestable test cases should be executed in the new version of the system in order to test the behavior of the modified parts.

- **Obsolete:** obsolete test cases should not be executed in the new version of the system. A test case may become obsolete for different reasons, such as 1) their input/output relationships are incorrect due to a modification in the system specifications, or 2) they no longer test what they were designed to test due to modifications to the program.

After a software modification, some test cases also have to be generated in order to cover the new specifications. These test cases are classified as **new**.

It is important to note that the identification of obsolete test cases is necessary if any test case reuse is desired, independently if the test case selection approach or the retest-all approach is used. Therefore, when a modification is made on the system, it is necessary to identify the obsolete test cases in T for P' prior to performing any test case reuse approach. After identifying the obsolete test cases and removing them from T, the evaluation of the test case reuse can be performed on the remaining ones [27].

Regression testing can be categorised into progressive or corrective, based on the type of the modifications. Progressive regression testing involves modified specifications and is performed whenever new requirements are incorporated in the system. In these cases, the specification will be modified to reflect the additions, thus the new version of the system should be tested in order to verify if the new specifications are correctly implemented.

Corrective regression testing does not involve changes in the specifications, but only in design decisions and instructions of the system. In this case, most of the existing test cases can be reused, because they correctly specify the input/output relationships. Usually corrective regression testing is performed after some corrective action on the software, i.e. correction of a bug.

During regression testing, a set of test cases may be available for reuse. Regression testing reduces the cost of testing a modified system by reusing these existing tests, identifying the parts of the modified system that should be tested and creating test cases for the new parts of the system. Therefore, the use of its concepts in an iterative and incremental environment improves the maintainability process.

3.2 Delta-Oriented Model-Based SPL Regression Testing

In traditional regression testing techniques, source code is modified directly, thus their activities are supported by the code modifications analysis. On the other hand, in model-based regression testing the modifications are first done to models, rather than to source code. Therefore, regression testing is supported by model modification analysis and can be performed without source code analysis [32].

In an environment in which the test model is created iteratively and incrementally in order to represent the behaviour of the system, the reuse of test artefacts are the key in the development of different versions of the system. Software Product Lines (SPL) propose techniques for developing variant-rich software systems by means of design artefacts reuse throughout all development phases [33, 35, 36]. Each version of the system can be seem as a product variant and consequently techniques of model-based testing of SPLs can be adapted in order to be used in an iterative and incremental development of a system.

Lochau,M., et al. proposed *Delta-Oriented Model-Based SPL Regression Testing*, an approach for incremental model-based testing of SPLs based on principles of regression testing. The collection of test artefacts for a product variant p_i is a 4-tuple $ta_i = (tm_i, tg_i, ts_i, tp_i)$, in which tm_i consists of a test model, tg_i consists of a finite set of test goals in tm_i for a coverage criteria CC, ts_i consists of a test suite and tp_i consists of a test plan. A test plan consists of a subset of a test suite ($tp_i \subseteq ts_i$) containing the test cases to be (re-)tested during regression testing [33, 35, 36].

The test model tm is a state machine represented by a 4-tuple $tm = (S, s_0, L, T)$, in which S is a finite set of states, $s_0 \in S$ is the initial state, L is a set of transition labels, and $T \subseteq S \times L \times S$ is the set of transition relationships. A test case $tc = (t_0, t_1, ..., t_k) \in T^*$ of a state machine test model (tm) is a finite sequence of k transitions from tm. A test case is *valid* if its alternating sequence of states and transitions conforms to tm, i.e., the observable behaviour under the sequence of inputs conforms to the expected behaviour specified in test model [33, 35, 36].

To express the variability they use the concepts of delta modelling, which is used in SPL to explicitly specify the changes between variants, and to incrementally evolve test artefacts for product variants. In delta modelling, similar products are represented by a designated *core* product and a set of *deltas* that specify changes with respect to the core product. Considering state machines as the test models, a state machine delta specifies added and removed states and transitions from the core state machine. A modification in the label of a transition can be represented by a removal of the transition followed by the addition of a new one with the new label. In order to generate the variants of the test model, delta operations are applied to the core test model (tm_{core}). Therefore, differently from techniques that compare the previous and current version of a test model to detect the differences between them, in delta modelling the differences are explicitly specified by means of deltas [33, 35, 36].

The product test artefacts incrementally evolve via the following steps:

1. Generate an initial collection of product test artefacts ta_1 using MBT techniques,

and apply the resulting test suite ts_1 to the implementation of p_1. It is suggested to use p_{core} as p_1.

2. Incrementally evolve ta_i to ta_{i+1}, for $1 \leq i \leq n$, and apply the new (re-)test plan $tp_{i+1} \in ts_{i+1}$ to p_{i+1}.

As shown in Figure 3.1 the principle of delta modelling is applied to reason about the incremental changes between two sets of test artefacts ta to ta' by a set of sub deltas:

$$\delta_{ta,ta'} = (\delta_{tm,tm'}, \delta_{tg,tg'}, \delta_{ts,ts'}, \delta_{tp,tp'})$$

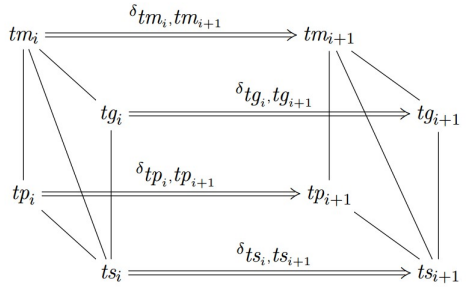

Figure 3.1: Incremental Evolution of SPL Test Artefacts (extracted from [36])

A test model delta $\delta_{tm,tm'}$ makes the differences between tm and a subsequent variant tm' explicit. Consequently, it is also called *regression delta*. A test goal delta $\delta_{tg,tg'}$ defines the evolution of the set of test goals that covers a coverage criteria in the tm'.

Sets of test cases are partitioned into subsets of reusable $ts_R \in ts_i$, obsolete $ts_O = ts_i \setminus ts_R$, and new test cases $ts_N = ts_{i+1} \setminus ts_R$. The test suite is composed of the *valid* and *obsolete* set of test cases: $ts = ts_V \bigcup ts_O$. The obsolete test cases are not discarded in the next test suite ts' because they could be reused in subsequent products.

In order to execute step 2, when the test set evolves to ts', first the *reusable* test cases are identified $ts'_R = ts_V \bigcap ts'_V$. It contains the test cases that are valid for p and for p'. In order to cover all test goals tg', other test cases may be required. A test goal $g \in tg'$ is not covered by ts'_R if either a test goal is new in p', or all test cases of p that cover g are obsolete for p'. Therefore, firstly obsolete test cases $tc \in ts_O \bigcap ts'_V$ that cover some test goals may be identified and added to ts'_R. Otherwise, new test cases are generated and compose the set TS'_N. The set ts evolves to $ts' = ts'_V \bigcup ts'_O$, via $\delta_{ts,ts'}$, to cover all the test goals tg', in which $ts'_V = ts_R \bigcup ts'_N$ and ts'_O contains the test cases from p that cover removed transitions in p'.

The test plan $t_p \subseteq ts_V$ is used to define which valid test cases from a test suite are actually executed on the product under test, where $t_p = ts_N \bigcup ts_{RT}$. New test cases $tc \in ts_N$ are applied to verify that new features are correctly implemented. From the set of reusable test cases ts_R, a retest set $ts_RT \in ts_R$ is selected to verify that the changes do not erroneously affect common behaviour covered by ts_R. For the selection of ts_{RT}, any technique can be chosen.

Chapter 4

Related Work

Works that propose approaches that deal with Model-Based Testing in an agile context, those that deal with Model-Based Regression Testing, and those that deal with test artefact reuse were analysed in order to support the definition of the proposed Delta MBTDD method. During the search for the related works, those that propose algorithms or tools were not analysed. Moreover, an attempt was made to select approaches that use any kind of state machine as the test model. However, some of them use other models such as control flow graphs. The works are separated according to their main objectives and are described in the sections bellow.

4.1 Model-Based Testing in an Agile Context

Wieczorek et al. propose a software process which joins MBT, TDD, Model Driven Development (MDD) and Model Driven Performance Engineering (MDPE). Firstly, following MDD, some models are created: structural models which identify business components, and behavioural models which describe messages between the components. Simultaneously, the models are analysed from the performance point of view. As a result, estimations are created, based on the models and consequently on the final system, minimizing possible code refactoring. From the models code stubs are automatically generated which serve as input for the unit test case generator that guide the development during the TDD cycle. MBT is used in order to generate integration tests from the models. However, even though MBT is used during the process, the main idea is to use TDD to develop business components while MBT supports the integration test generation in order to support inter-component integration and not to support the development of the system. Moreover, there is no analysis about the evolution of the system and consequently about the reuse of test artefacts.

Hametner et al. propose a method which adapts TDD for the context of the development of an automation system, in which models are used to generate the test cases. UML models are created to describe statistic and dynamic activities of the system, such as use cases, class diagrams, state machines and sequence diagrams. Even though this work proposes a method which focuses on using models to support test case creation, only the design of the models are described and the generation of test cases from these

are left as future work. Moreover, it is not explained how the test cases could support the development of the system, and there is no analysis about the evolution of the system and consequently about the reuse of test artefacts.

With the aim to improve the maintainability of regression test cases in a Scrum environment, Entin et al. propose the union of MBT and agile development, and validated it on a case study. During the process, firstly use cases are defined and, from them, user stories. After the definition of which user stories will be implemented during a specific sprint, each one of them is represented in a UML statechart. From these statecharts, regression tests are generated using MBT methods. During the next sprints an analysis is performed in order to reuse the statechart and create new ones. As results of the case study, it was observed that the modifications were represented faster when models were used, the time required for state machine creation reduces in each sprint, and many statecharts were reused. Even though this work deals with MBT in an agile development, the tests do not guide the development. Moreover, the reuse is focused on state machine reuse and not on test case reuse.

4.2 Model-Based Regression Testing

Korel et al. propose a model-based regression testing approach to reduce regression test suites, that uses an EFSM (Extended Finite State Machines) as the test model and EFSM dependence analysis in order to reduce the test suite. When a modification is performed on the EFSM, only the additions and deletions of transitions are identified between the two versions, because an addition or a deletion of a state is always associated with an addition or a deletion of a transition, respectively. For each modification, EFSM dependency analysis is performed and based on it, the regression test suite is reduced. Even though the approach is a model-based regression testing technique, it is not in an agile context and does not aim for test artefact reuse.

Farooq et al. propose a Model-Based regression testing approach which aims to classify test cases from previous versions in a context of evolving software. UML class diagrams and state machines are used, and based on the modifications on both models the test cases are classified as obsolete, reusable or retestable. The obsolete test cases cover removed transitions, the retestable ones cover changed transitions, and the reusable ones cover unchanged parts of the system. The approach is inserted in a context of an evolving software system, and consequently is suitable for an agile context. However, it aims to classify the test cases and not to generate new test cases. Moreover, the test cases do not guide the development of the system and they use only UML test models.

With the aim to support the selection of retestable test cases, i.e. test cases that traverse modified elements and have to be re-executed on the system, a selective regression testing approach based on model modifications is proposed by Naslavsky et al.. They propose creating a traceability between model elements and the corresponding test cases that traverse those elements while test cases are created from models. Firstly UML sequence diagrams are transformed into Model-Based control flow graph (*MBCFG*), which are an alternative view to the sequence diagrams that are used to support abstract test

case generation. During this transformation, the traceability information is stored in the traceability model. From *MBCFG*, a test hierarchy is generated and another traceability model is created. When a second version of the sequence diagram is created, both versions are compared and a UML model of the differences is created. The new sequence diagram is transformed into a *MBCFG*, and based on it and on the traceability models it is possible to perform an analysis in the test hierarchy. The analysis supports the classification of the test cases from the previous version and consequently the selection of retestable test cases. The approach is inserted in a context of an evolving software system, and consequently is suitable for an agile context. However, even though it aims for test case reuse, the selected test cases are those that have to be re-executed on the system and do not guide the system development.

Motivated by the effort required for the MBT step of transforming abstract test cases into executable ones in order to test an embedded system, Blech et al. propose to reuse test cases on different levels of abstraction. However, the approach of test case reuse is limited to a specific environment, and consequently it uses a specific model for this context. Moreover, the approach is not in an agile context and the test cases do not guide the development.

4.3 Reuse of Test Models

Weißleder et al. propose an approach for automatic test suite derivation based on reusable UML (*Unified Modelling Language*) state machine test models. In the context of software product lines (SPL), state machines can be used to model the behaviour of product variants and based on them test suites can be automatically generated using MBT techniques. The context of a UML state machine is described in a context class. Therefore, instead of creating one state machine per product variant, in this work it is proposed to create one state machine that contain all features, which is associated with a context class. This context class is the superclass of all product variants. Since specialized classes contain the same operations, attributes and associations of the general class, but their behaviour change depending on the value of the pre and/or postconditions or attributes, each product variant is expressed by using different values of pre and/or postconditions or attributes of the context class. In this work there is the reuse of test artefacts, as the behaviour model is inherited by the subclasses. However, only the test model is reused, the test cases are not reused. Moreover, the approach is not intended to guide agile testing.

Dranidis et al. propose a just-in-time on-line Model-Based regression testing approach focusing on the reuse of test models, in which Stream X-machines (SXM) are used as test models. SXM extend finite state machines by including a memory structure, and from them the test cases are derived with the support of an MBT method and tool. When the test model evolves through compositions, it is proposed to reduce the SXM for the composition context and generate test cases only for the reduced SXM. Therefore, a reduced test set is created in order to improve the execution of on-line tests. The approach is inserted in a context of an evolving software system, and consequently is suitable for an agile context. Since their goal is online testing, an important concern is the performance

of test execution, and as such, it is essential to reduce the number of irrelevant test cases, that is, test cases that do not cover modified parts. In this sense, the authors propose to reduce the test model, so that it contains only the parts that are relevant for the applied modification. However, from the reduced test model all test cases are generated without any test case reuse and they do not guide the development of the system.

4.4 Summary of the Study

Table 4.1 summarizes the existing solutions according to some characteristics: which kind of test model is used; if the approach was proposed in an agile context; if the tests generated from test model guide the development of the system, like in MBTDD; if a Model-Based regression testing method was used or proposed; if there was test artefact reuse; and if test cases were reused. The last line contains Delta MBTDD, the approach proposed in this work.

Table 4.1: Comparison of Existing Solutions

Solution	**Model**	**MBT in Agile Context**	**MBTDD**	**Model-Based Regression Testing**	**Test Model Reuse**	**Test Case Reuse**
Wieczorek et al. (2008)	Not specified	X	-	-	-	-
Hametner et al. (2010)	UML models	X	-	-	-	-
Entin et al. (2012)	UML statechart	X	-	-	X	-
Korel et al. (2002)	EFSM	-	-	X	-	-
Farooq et al. (2007)	UML state machine	X	-	X	X	X
Naslavsky et al. (2010)	MBCFG	X	-	X	X	X
Blech et al. (2012)	Specific Model	-	-	X	X	X
Weißleder et al. (2008)	UML state machine	-	-	-	X	X
Dranidis et al. (2010)	SXM	X	-	X	X	-
D-MBTDD	**FSM**	**X**	**X**	**X**	**X**	**X**

Chapter 5

Proposed Solution: D-MBTDD method

In order to support the incremental test creation and maintenance in a Model-Based Test-Driven Development, the Delta Model-Based Test-Driven Development (D-MBTDD) method was proposed during this work. D-MBTDD supports the development of new features by means of test case and test model reusability. Test cases from previous versions that are valid for a current iteration are reused and new test cases that cover new features are created. The development of new features is guided by the new test cases, while the reusable test cases are used as regression tests. In order to use the D-MBTDD methods, there are some assumptions described in Section 5.1. The D-MBTDD process is described in Section 5.2, and in Section 5.3 a process which joins D-MBTDD and Scrum is presented.

5.1 Assumptions

D-MBTDD method relies on certain assumptions to guarantee its use and results:

- Even though any kind of behavioural test model could be used with D-MBTDD, during this work only **Finite State Machines (FSM)** are used as test models.
- Before starting the test case generation, the finite state machines were **already created and validated by specialists**. Furthermore, when the system evolves, the *regression deltas* with the modifications necessary to obtain a new version of the state machine were also created and validated by specialists. Therefore, the representation of the system behaviour as a state machine is not in the scope of D-MBTDD.
- The **MBT tool** used to support the offline test case creation is based on a **test purpose**, i.e., it is possible to generate test cases that cover a specific coverage set.
- When the system evolves, the **system implementation** of the unmodified elements of the test model have **not** been changed in order to guarantee that no accidental or intentional changes were introduced.

5.2 D-MBTDD Process

With the same motivation as Blech et al., D-MBTDD aims to reduce the effort required for the transformation of abstract test cases into executable ones by reusing test cases from previous versions. Moreover D-MBTDD proposes an approach in an agile context of iterative and incremental development, based on *Model-Based Test-Driven Development* (Section 2.4). Because the test cases generated from test models guide the development of the system, D-MBTDD uses an offline MBT approach in order to generate and execute the test cases.

D-MBTDD proposes some modifications in *Model-Based Test-Driven Development* by adding some characteristics of *Delta-Oriented Model-Based SPL Regression Testing* (Section 3.2) in order to support incremental tests in MBTDD. This technique was chosen since it aims to reuse test artefacts by means of analyses of differences between different versions of a test model. Even though it was proposed for a SPL context, each product can be seen as one version of the system. When the system evolves, the modifications are discussed with the client and consequently the differences are already known before the beginning of the development cycle. Therefore, like in *Delta-Oriented Model-Based SPL Regression Testing*, these differences are represented in *regression deltas*.

The same collection of artefacts is used, but because D-MBTDD is not in the context of product lines, instead of having one collection per product variant, in D-MBTDD each system version has a collection. Therefore, each test model represents one version of the system behaviour.

Differently from Delta-Oriented Model-Based SPL Regression Testing, D-MBTDD assumes a sequential evolution of the test models in which the *regression deltas* are applied to the previous version of the test model and not just to the core test model.

When the system evolves, D-MBTDD proposes a process based on *Delta-Oriented Model-Based SPL Regression Testing* process in order to support the incremental tests. D-MBTDD classifies test cases from the previous version in order to identify the reusable test cases, and generates new test cases in order to complete the new test suite. D-MBTDD also does not discard the obsolete test cases in the next test suite because they could be reused in subsequent versions.

D-MBTDD performs these tasks in order to support the development of new features, in which new test cases guide the development and the reusable test cases are used as regression tests after the implementation. Therefore, besides the classification of test cases from the previous version, D-MBTDD aims for the selection of test cases that guide the development of new features. Even though Naslavsky et al. and Farooq et al. propose Model-Based regression testing approaches for evolving systems, they do not focus on the generation of new test cases that cover new features and they do not deal with development guided by test cases.

The test artefacts of D-MBTDD follow the definitions used in *Delta-Oriented Model-Based SPL Regression Testing*, however in D-MBTDD the *test plan* artefact is not composed of *new* and *retestable* test cases. In D-MBTDD no technique is used to select the set of retestable test cases from the reusable set $ts_R T \in ts_R$, and the entire reusable set is included in the test plan. Therefore, in D-MBTDD the test plan is composed of the *new*

and *reusable* test cases. Moreover, in *Delta-Oriented Model-Based SPL Regression Testing* the regression deltas contain information about additions and deletions of transitions and states. However, because D-MBTDD deals with FSMs, like in Korel et al. regression deltas contain only the additions and deletions of transitions.

The proposed development cycle is iterative and incremental, so the test artefacts and the system are iteratively and incrementally created. Each iteration involves all the phases: design, coding and testing. At the end of the iteration a working product is demonstrated to the stakeholders. During the first iteration process, described in Subsection 5.2.1, the initial test artefacts and the first version of the system are created. And then during the process for the iterations that follow, described in Subsection 5.2.2, the test artefacts evolve in order to include new functionalities and changes.

5.2.1 Process for the First Iteration

During the first iteration, the initial collection of test artefacts and the first version of the system are created following the flowchart illustrated in Figure 5.1. Each step of the process is explained below.

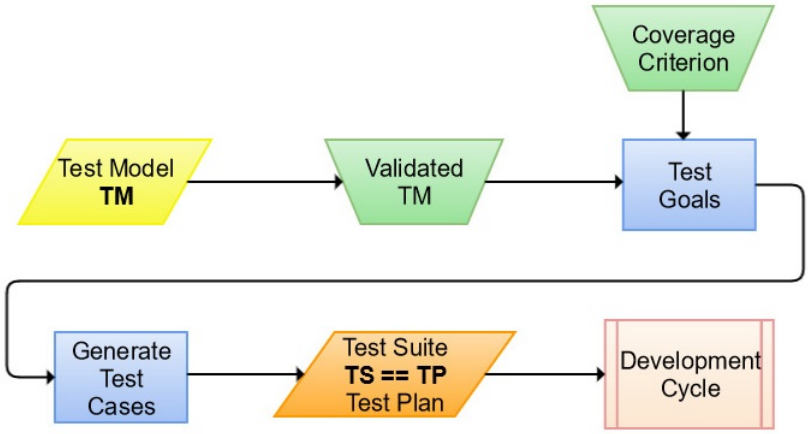

Figure 5.1: D-MBTDD process for the first iteration

1. Create the Test Model: First the *test model (TM)* is created by specialists in order to represent the system behaviour.

2. Validate Test Model: The *test model (TM)* is *validated with a specialist.* This validation is not formal, it is a validation in which the goal is to check if the behaviour represented in the state machine is in accordance to his needs.

3. Test Goals Definition: To generate the test cases from the state machine, first a *coverage criterion* that the test suite has to reach is agreed upon between the client, the testers and the developers, in other words, the stakeholders. Based on this criterion, the *test goals* that the test suite has to cover are derived.

For example, considering the core state machine illustrated in Figure 5.2 and a coverage criterion of 100% transition coverage, the test goals are: $tg = \{t0...t15\}$.

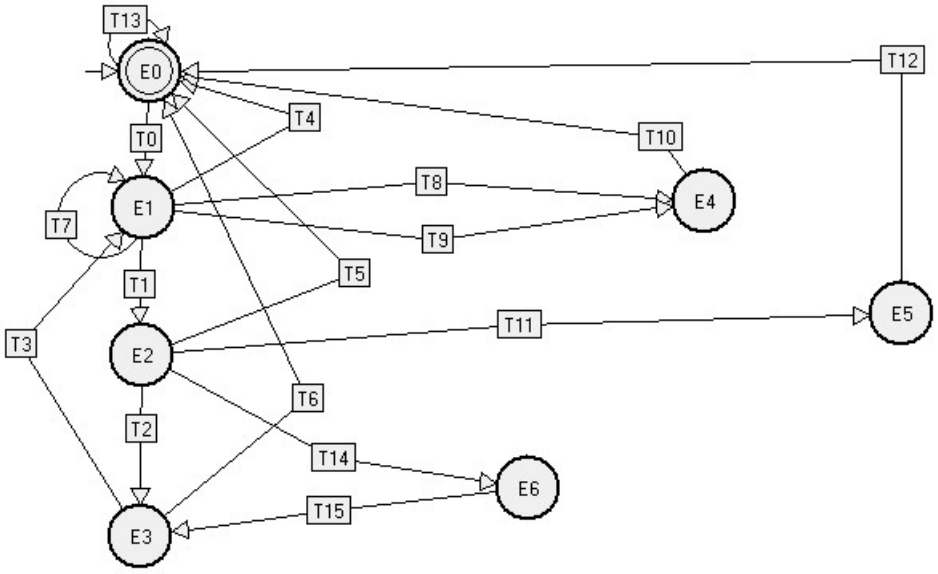

Figure 5.2: Core State Machine Example

4. Generate Test Cases: Based on the test goals, the *test cases* that cover all of them are generated. In order to support this task, an MBT tool is used.

For example, for the core state machine of Figure 5.2, 25 test cases are created using Condado tool, which is described in Subsection 6.5.1, in order to cover all test goals.

5. Test Suite and Test Plan Definition: Because in the first iteration there are no test cases to be reused, the test plan is equal to the test suite.

6. Development Cycle: The *test plan* guides the development, during the development cycle. The *Development Cycle* is based on the principle of Test-Driven Development and follows the same cycle explained in Subsection 2.1 and illustrated in Figure 2.1.

5.2.2 Process for the Next Iterations

Because the environment is iterative and incremental, in the iterations that follow, we will have new requirements and some changes to do. The steps of the iterations that follow are illustrated in a flowchart (Figure 5.3) and they are described below.

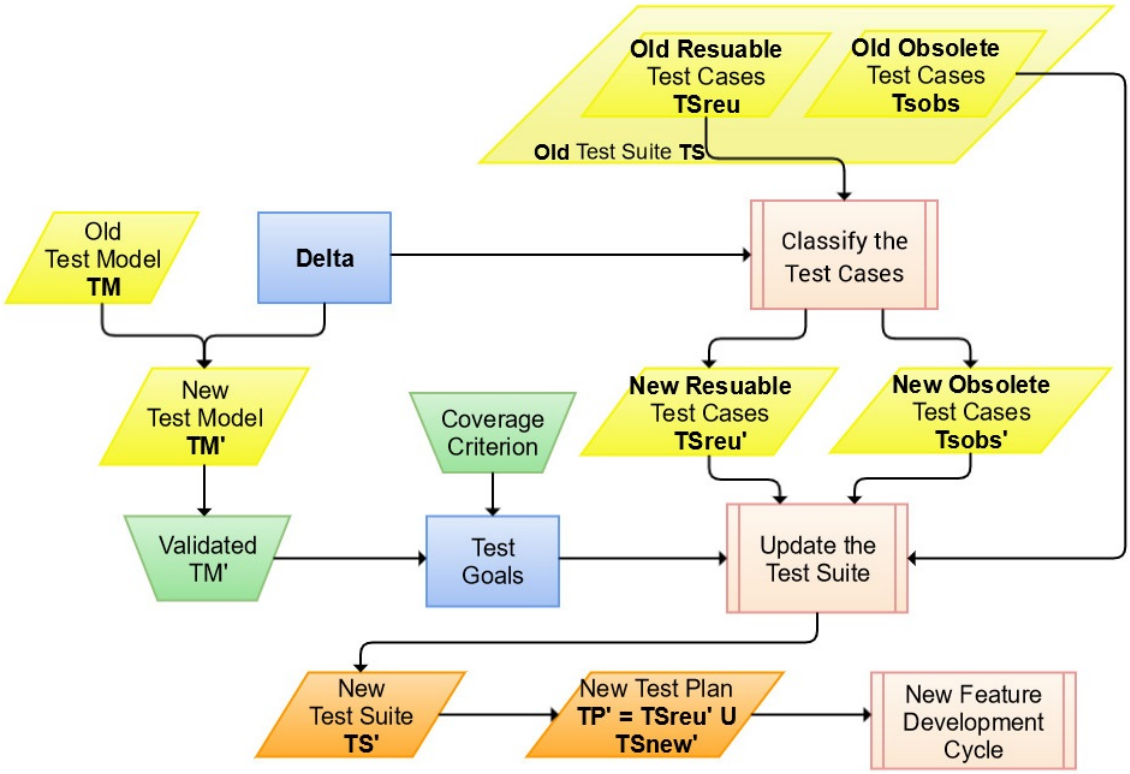

Figure 5.3: D-MBTDD process for when the test model evolves

1. New Test Model Creation and Validation: First, the modifications discussed with the client in order to obtain a new version of the system are represented in a *regression delta*. This *regression delta* with the modifications is applied to the *previous test model version (TM)* in order to obtain the *new test model version (TM')*. Like in the first iteration process, this new version is *validated* in order to verify it.

For example, the regression delta illustrated in Figure 5.4 removes transition $t5$, illustrated with dashed lines, and adds transitions $t16$ and $t17$, illustrated with double lines. When the regression delta is applied to the test model previously illustrated in Figure 5.2, the new test model illustrated in Figure 5.5 is obtained.

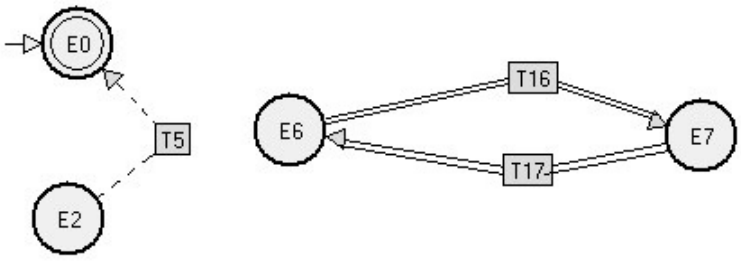

Figure 5.4: Regression delta example

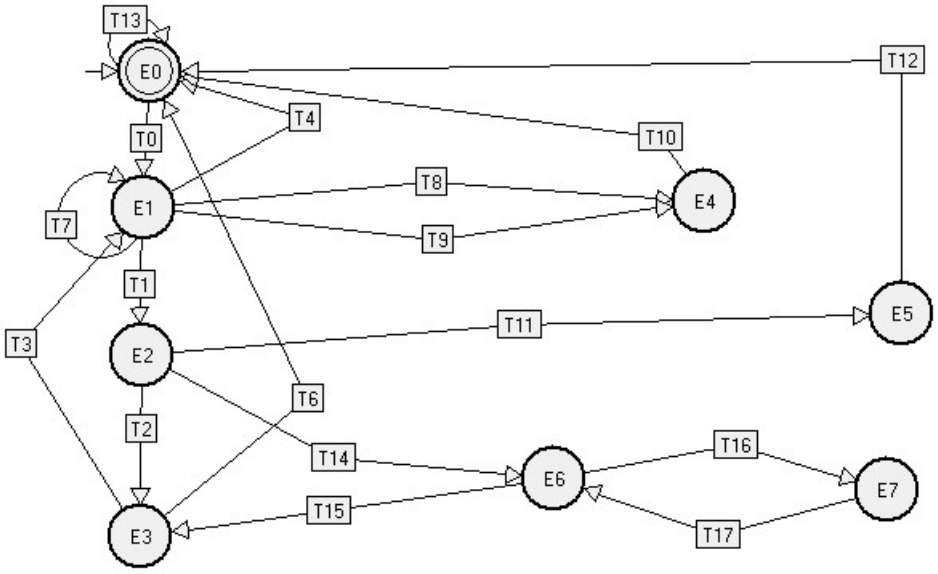

Figure 5.5: New test model version example

2. Update Test Goals: Based on a *coverage criterion*, which can optionally be the same from the previous iteration, and based on the new test model, the test goals are updated.

For example, for the new test model of Figure 5.5 and using the same coverage criterion, the test goals are updated to: $tg = \{t0...t4, t6...t17\}$.

3. Revalidation of the Test Suite: The set of reusable test cases from the previous test suite ($TSreu$) are analysed in order to revalidate the test suite. As stated in Section 3.1, a test case might be valid according to the core model, but becomes invalid for the new version as the transition has been removed in the path exercised by the test case. Therefore, with this analysis the test cases from the previous version are *classified* as *reusable* if they remain valid, or *obsolete* if not. After that, the set of reusable ($TSreu'$) and obsolete ($TSobs'$) test cases of the new version is created. In this step the states and transitions covered by each test case are identified, in order to have the traceability between the state machine, the test cases and the test goals.

In our example, because the old test model (Figure 5.2) was the first version, there were no obsolete test cases ($TSobs = \emptyset$) and therefore all test cases are part of $TSreu$. $TSreu$ had the 25 test cases and when analysed, 22 remained valid in the new version and 3 became obsolete. Therefore after the classification of the test cases from the previous test suite, $TSreu'$ has 22 test cases and $TSobs'$ 3 test cases.

4. Update the Test Suite: Using the sets of classified test cases ($TSreu'$ and $TSobs'$) and the set of obsolete test cases from the previous test suite ($TSobs$), the test suite is updated in order to cover all the test goals. D-MBTDD also does not discard the obsolete test cases in the next test suite because they could be reused in subsequent versions. In order to execute this task, some steps illustrated in the flowchart represented in Figure 5.6 are performed, and they are described below.

4.1. Update the New Test Suite: Firstly, the new test suite TS' is updated in order to

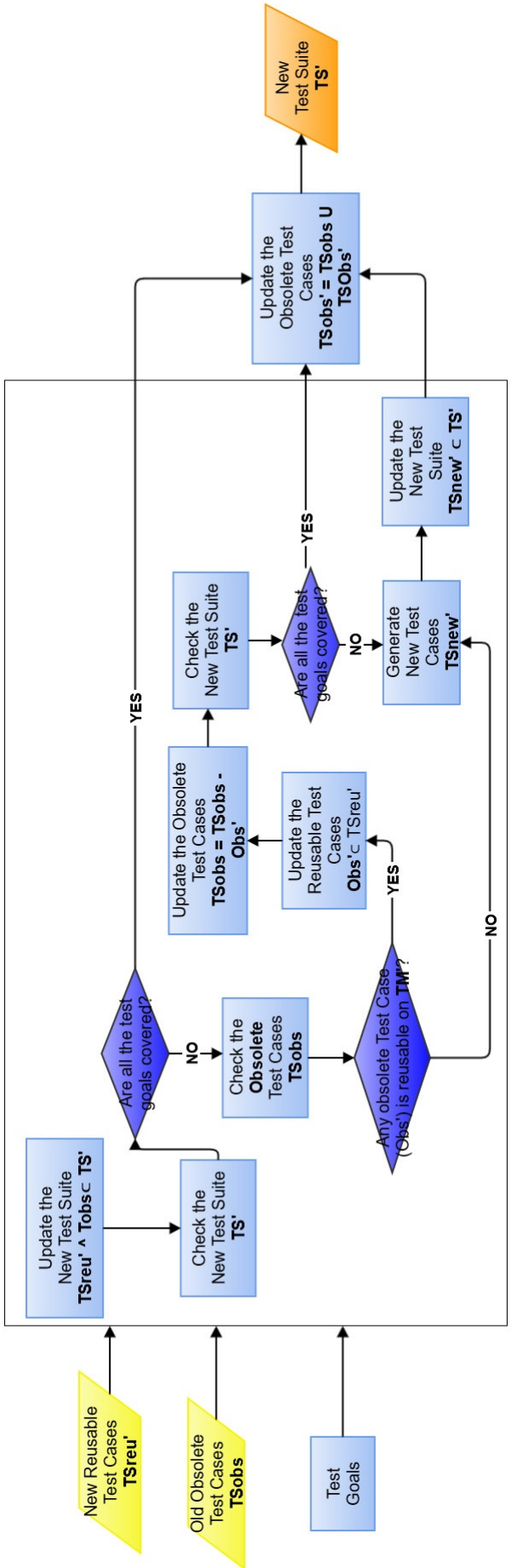

Figure 5.6: Process of updating the test suite during the iterations that follow

include the new test set of reusable test cases ($TSreus'$) and the test set of obsolete test cases from the previous version ($TSobs$).

In our example, after the execution of step 4.1. TS' has only the 22 reusable test cases from $TSreus'$, because $TSobs = \emptyset$.

4.2. Check the New Test Suite: After that, it is verified if the new test suite covers all test goals. In the positive case, the next step is *4.6 Update the Obsolete Test Cases*, otherwise it is *4.3. Check the Obsolete Test Cases.*

In our example, TS' does not cover all test goals, therefore step 4.3. has to be executed.

4.3. Check the Obsolete Test Cases: In case the test suite does not cover all test goals, the set of obsolete test cases from the previous version ($TSobs$) is verified in order to check if some of them become valid for the new version. It can happen for example if after a few iterations a previously removed behaviour was included again in the system. In the positive case, the next step is *4.4 Update the Reusable Test Cases*, otherwise it is *4.5 Generate New Test Cases.*

In our example, there is no obsolete test case from $TSobs$ which become valid in the new version, therefore step 4.5 has to be executed and step 4.4 is not executed.

4.4. Update the Reusable and Obsolete Test Cases: If some test cases (Obs') from the previous version becomes valid for the new version, they are included in the reusable test set $TSreu'$ and removed from the obsolete test set $TSobs$. After that, it is verified if the new test suite covers all test goals. In the positive case, the next step is *4.6 Update Obsolete Test Cases*, otherwise it is *4.5. Generate New Test Cases.*

4.5. Generate New Test Cases: In order to cover all test goals, new test cases are generated with the support of an MBT tool. The set of new test cases ($TSnew'$) is included in the new test suite TS'.

In our example, 10 new test cases are generated in order to cover all test goals and make up the test suite $TSnew'$. The new test suite TS' now contains the 33 test cases: 22 from $TSreus'$ and the 10 generated in $TSnew'$.

4.6. Update the Obsolete Test Cases: After the new test suite TS' covers all test goals, the obsolete test cases are updated in order to include the set of obsolete test cases from the previous version ($TSobs$) with the set of obsolete test cases from the new version ($TSobs'$). Finally the process of Updating the Test Suite is finished with the new test suite containing three sets of test cases: reusable ($TSreus'$), new ($TSnew'$) and obsolete ($TSobs'$).

In our example, $TSobs'$ continues with the 3 obsolete test cases classified as such in step 3, because $TSobs = \emptyset$. Therefore, the new test suite is updated resulting in 36 test cases: 22 from $TSreus'$, 10 from $TSnew'$ and 3 from $TSobs'$.

5. New Test Plan: After the New Test Suite is updated in order to cover all test goals of the new version, the new test plan (TP') is created with only the valid test cases. Therefore, TP' includes the sets of reusable ($TSreus'$) and new test cases ($TSnew'$) of the new version.

In our example, TP' is created with 36 test cases: 22 from $TSreus'$ and 10 from $TSnew'$.

6. New Feature Development Cycle: The *test plan* guides the development. Because now only the new features have to be developed, a new development cycle is

proposed based on the TDD cycle and it is illustrated in Figure 5.7.

The test plan is composed of the set of reusable test cases ($Treu'$) and new test cases ($Tnew'$). The classification into new and reusable test cases will assist development so that it is focused on the new features. Firstly, the new test cases will assist the development of new features in order to obtain the new version of the system (S'). After that, the reusable test cases are applied to the new version of the system in order to guarantee confidence in the modified version of the system, so they are used as regression tests. If some test cases fail, the fixes are performed and the test cases are again applied. These steps are repeated until all the test case executions succeed and, therefore, the new version of the system is successfully implemented.

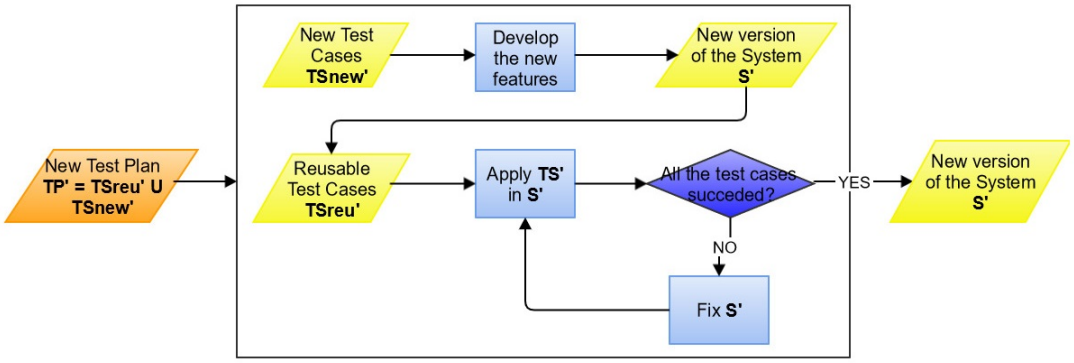

Figure 5.7: Development cycle of a new feature

5.3 D-MBTDD with Scrum

Scrum is a management framework for incremental software development, which uses a few artefacts and a self-adaptive team. Scrum uses fixed-length iterations, called *Sprints*, which typically last two to four weeks, during which the system is planned, developed, and demonstrated to the client. Scrum uses a *Product Backlog* to represent a list of desired functionalities ranked by their priorities, which are defined between the stakeholders. At the beginning of each Sprint, the Product Owner and the team hold a *Sprint Planning Meeting* to negotiate which Product Backlog Items will be worked on during the Sprint. These negotiated items compose the *Sprint Backlog*. During the Sprint, a *Daily Scrum Meeting* is realized every day in which all members of the team report their activities and impediments. At the end of the Sprint, a *deliverable* of the system is created and demonstrated to the client during the *demo* [48, 49]. Figure 5.8 presents a simplified version of Scrum process.

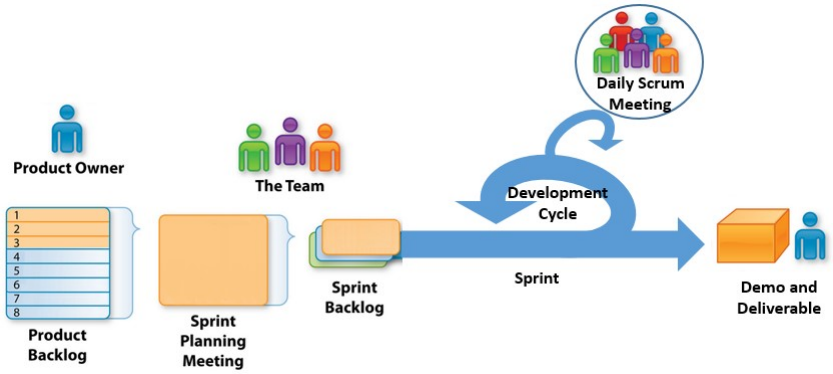

Figure 5.8: Scrum process (Adapted from [1])

Because Scrum is widely used nowadays, a process of how to use D-MBTDD with Scrum was proposed and is illustrated in Figure 5.9. Because now the development is guided by test cases derived from test model, the process adds a **D-MBTDD cycle** before the development cycle during a Sprint. During the D-MBTDD cycle the steps of the test model design and all the necessary steps to generate the test suite are performed. The generated test cases are used in the development cycle in which the new test cases guide the development of new features and the reusable test cases are used as regression tests. After the execution of the development cycle, a deliverable is created and it is presented to the client during the *demo*.

Furthermore, a **test specialist** role is proposed in the Scrum team. This new role is responsible for supporting all the steps of D-MBTDD. Therefore, the test specialist creates a representation of the desired behaviour in the test models, supports the test suite generation, and supports the team to solve possible test problems during the Sprint.

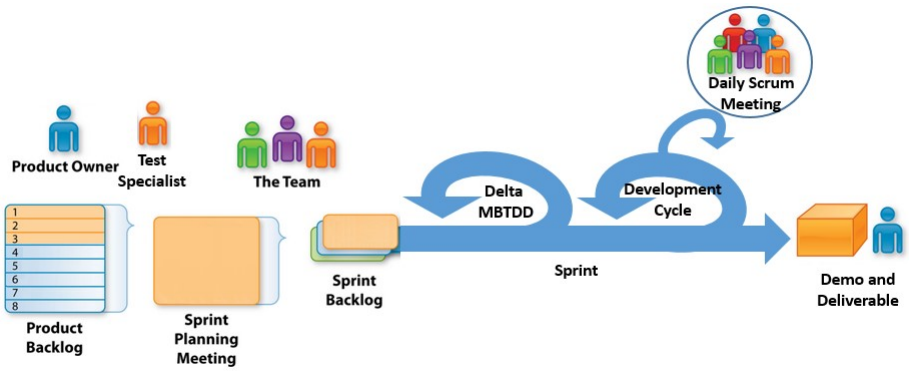

Figure 5.9: Process of D-MBTDD with Scrum

Chapter 6

Evaluation of the Proposed Solution

In software engineering, controlled experiments are used in situations where the simulation is manipulated directly and systematically. It can be used to simulate the real-world behaviour of different approaches applied to objects, in order to compare them. Variables are used to support this evaluation and their outcomes are analysed [58]. Therefore, in order to evaluate the applicability of the proposed solution, two controlled experiments were performed during this work, each one with the support of a different MBT tool. Section 6.1 describes the approach Regenerate-All, used for comparison with D-MBTDD, as well as the metrics used to support the analyses of the results. The objects used during the experiments are described in Section 6.2. The experiments were performed in both MBT tools following a process described in Section 6.3. The first controlled experiment used StateMutest as the MBT tool. The StateMutest tool, the preparation for conducting the experiment with it, the results and the analyses are described in Section 6.4. The second controlled experiment used Condado as the MBT tool. The details about the second controlled experiments are described in Section 6.5, similarly to how the details of the first experement were described. The discussion about all the results of the controlled experiments and a comparison between Regenerate-All and D-MBTDD are described in Section 6.6. The threats to the validity of the experiments are described in Section 6.7.

6.1 Definition of the Experiment

In an iterative and incremental environment, when the system evolves and so does the test model, in order to update the test suite for the new version, it is possible to perform two different approaches: regenerate all test cases, or reuse some test cases from the previous version. The Regenerate-All method follows the first approach, while D-MBTDD follows the second one.

In Regenerate-All, the test model is updated and from this new version all **executable test cases** are generated without using any information from test cases created for the previous version. Regenerate-All aims to reuse and maintain only the test model, and it regenerates the test suite any time the test model evolves, discarding previously generate test cases. Binder justifies this approach by affirming that updating a test model requires less effort than maintaining a test suite, since the size and complexity of test models grows

more slowly than the test suite [7, 8].

Usually MBT tools do not generate platform specific test cases. Therefore, even though Regenerate-All assumes the regeneration of executable test cases, they still have to be adapted in order to be executable for a specific platform. If the system is executed in different platforms, e.g. web and mobile, the test cases have to be adapted for each platform. Thus, even though executable test cases are generated from the MBT tool, effort is still required to adapt them for the specific platform.

In order to generate test cases, the same coverage criterion is used with Regenerate-All and D-MBTDD. Regenerate-All generates all test cases based on the complete model. Consequently depending on the coverage criterion, the test suite will be composed of test cases that traverse modified elements, but also of test cases that traverse only unmodified elements. Therefore, in order to support the development of new features, it will be necessary to identify those that cover modified elements. Differently from Regenerate-All, D-MBTDD generates test cases based on the missing test goals and consequently based on the modified elements of the state machine. Ideally, when D-MBTDD is used all generated test cases traverse at least one modified element, so that all test cases traverse modified elements.

During this work two controlled experiments were performed with the main objective of comparing D-MBTDD with the Regenerate-All approach, in order to verify the gain of reusing test cases in an iterative and incremental environment, in which the test cases guide the development. Each controlled experiment used a different MBT tool to support the test case generation: StateMutest and Condado. These tools were chosen because during test case generation it is possible to define test purposes that have to be reached, which according to Grabowski et. al. [26], could be a sequence of events, or a set of states or transitions. The choice of the test purpose is made by the test specialist together with the client, as it indicates which part of the specification is interesting to exercise. This characteristic is necessary in order to execute the creation of new test cases that cover missing test goals, a step of the D-MBTDD methodology.

Even though Regenerate-All proposes to regenerate executable test cases, during this work the scenario in which **abstract test cases** are regenerated and transformed into executable test cases was analysed. Whenever the system was evolved to create a new version, the two approaches were performed in the resulting version. The gain was analysed from the perspective of the testers, who have to select and transform the test cases that guide the development.

6.1.1 Metrics

The evaluation focuses on comparing the gains in reusing the test cases. For that purpose, some research questions were defined and previously described in Section 1.4. In order to support the analyses required to obtain the answers to these questions, some metrics were defined for Regenerate-All and D-MBTDD.

The metrics were defined considering:

- *TSO*: the size of the *Old Test Suite*;

- TSN: the size of the *New Test Suite*;
- N: number of *new test cases* generated by D-MBTDD;
- $c(re)$: cost to *revalidate* the old test suite;
- $c(ge)$: cost to *generate* the test cases;
- $c(id)$: cost to *identify* the test cases that guide the development of new features;
- $c(tr)$: cost to *transform* the abstract test cases into executable ones

In order to support the analyses of Research Question **RQ1**, the following metrics were defined:

- **Generated Test Cases (GenTC):** represents the number of generated test cases by each approach.

  When Regenerate-All is used, the new test suite is composed only of the generated test cases, therefore:

  $$\boldsymbol{GenTC(RA) = TSN} \tag{6.1}$$

  When D-MBTDD is used, only new test cases are generated per iteration, therefore:

  $$\boldsymbol{GenTC(DMBTDD) = N} \tag{6.2}$$

- **Effort for test case creation per iteration (E_Cr):** represents the effort for creating test cases per iteration.

  In Regenerate-All, the effort for creating test cases is equal to the effort to generate the test cases, therefore:

  $$E\_Cr(RA) = GenTC*c(ge)$$

  Considering equation 6.1:

  $$\boldsymbol{E\_Cr(RA) = TSN*c(ge)} \tag{6.3}$$

  In D-MBTDD, before generating the test cases it is necessary to revalidate the test cases from the old test suite, therefore:

  $$E\_Cr(DMBTDD) = TSO*c(re) + GenTC*c(ge)$$

  Considering equation 6.2:

  $$\boldsymbol{E\_Cr(DMBTDD) = TSO*c(re) + N*c(ge)} \tag{6.4}$$

In order to support the analyses of Research Question **RQ2**, the following metrics were defined:

- **Not Modification Test Cases (NotModTC):** represents the number of test cases that do not traverse any modified element of the state machine, i.e., that traverse only the unmodified parts.

- **Modification Test Cases (ModTC):** represents the number of test cases that traverse at least one modified element of the sate machine. This value is calculated as:

$$\boldsymbol{ModTC = GenTC - NotModTC} \tag{6.5}$$

- **Focus:** measures how focused the generated test cases are on the modified elements. This metric measures the ability of the technique to generate more modification traversal test cases than not modification traversal test cases in each resulting set of generated test cases, thus it is calculated as:

$$\boldsymbol{Focus = \frac{ModTC}{GenTC}} \tag{6.6}$$

- **Effort for identifying modification traversal test cases (E_Id):** represents the effort for identifying test cases that guide the development of new features, i.e., the modification traversal test cases.

  In Regenerate-All, it is necessary to identify the modification traversal test cases from all the generate test cases, therefore:

$$E\_Id(RA) = GenTC*c(id)$$

  Considering the equation 6.1:

$$\boldsymbol{E\_Id(RA) = TSN*c(id)} \tag{6.7}$$

  In D-MBTDD, the modification traversal test cases are already identified in the set of new test cases, therefore no effort is required for this task:

$$\boldsymbol{E\_Id(DMBTDD) = 0} \tag{6.8}$$

During the controlled experiments, in order to calculate the metric *Not Modification Test Cases (NotModTC)* for each experiment, a *shell script* was created and is available on `https://goo.gl/MXywuW`.

In order to support the analyses of Research Question **RQ3**, the following metrics were defined:

- **Total effort to use the approach (E_Total):** represents the effort for using the approach. It is composed of the effort to generate the test cases, to identify the modification traversal test cases, and to transform abstract test cases into executable test cases.

  Considering equations 6.3 and 6.7, and that all generated test cases have to be transformed into executable test cases in Regenerate-All, the total effort is:

$$E\_Total(RA) = TSN*c(ge) + TSN*c(id) + GenTC*c(tr)$$

Considering equation 6.1:

$$\boldsymbol{E\_Total(RA) = TSN*c(ge) + TSN*c(id) + TSN*c(tr)} \quad (6.9)$$

Considering equations 6.4 and 6.8, and that only the new test cases have to be transformed into executable test cases in D-MBTDD, the total effort is:

$$\boldsymbol{E\_Total(DMBTDD) = TSO*c(re) + N*c(ge) + N*c(tr)} \quad (6.10)$$

6.2 Planning of the Experiment

The controlled experiments were conducted by the author of this work, and the objects were selected according to their characteristics. During the controlled experiments an agile environment was simulated in which test models were created iteratively and incrementally, moreover only finite state machines were used as test models. Therefore, the selected objects were finite state machines which had at least two versions that represented test models created incrementally, and whose differences were represented as state machines, i.e. *regression deltas*.

The objects used during the controlled experiments were extracted from two different sources. The first one, described in Subsection 6.2.1, is a case study used by the authors of the *Delta-Oriented Model-Based SPL Regression Testing* (Subsection 3.2) method, which inspired this proposal. The other, described in Subsection 6.2.2, is a real-world space application developed by an industrial partner. In total there were 9 test models with 20 different versions.

In order to simplify the state machines, the input and output of all transitions were omitted and the transitions were labelled with an identifier (ID). Following the idea from *Delta-Oriented Model-Based SPL Regression Testing* (Subsection 3.2), a modification of an input or output was represented as a removal of the transition followed by the addition of a new one with a new identifier (ID) label.

The objects followed the idea of *Delta-Oriented Model-Based SPL Regression Testing* and was composed of: a *core state machine test model*, which will be referred to from now on as **core model**; different *state machine test model versions*, which will be referred to from now on as **test model versions**; and *state machine delta models*, which will be referred to from now on as **delta models**.

6.2.1 Delta Case Study

The case study is from the automotive domain, a simplified Body Comfort System (BCS) including the following functionalities [34, 36]:

- *Power Window* with *Finger Protection*
- Electric and heatable *Exterior Mirror*

- *Alarm System* with *Interior Monitoring*
- *Central Locking System* with *Automatic Locking*
- *Remote Control Key* with *Safety Function*, *Alarm System Control*, and *Power Window Control*

Automatic Locking allows the *Central Locking System* to provide the automated locking of the doors when the car is driving. *Safety Function* allows the *Remote Control Key* to provide the automated locking of the car after a specific timeout, i.e., the car is unintentionally unlocked. The *Power Window* has two alternatives: the *Manual Power Window*, which moves up/down when pressing and holding the button for the window movement, and the *Automatic Power Window*, which moves up/down when pressing the button for the window movement once.

Although the authors applied Delta-Oriented Model-Based SPL regression testing to different UML models, we only use the **state machine** ones [34, p. 103-148]. There are some core test models that specify the complete behaviour, i.e., there is no version changing the behaviour of the core, and therefore there is no correspondent delta model. From the 21 core test models, just those which have at least one delta model were selected. Thus, 8 core test models were selected:

- *Manual Power Window*, which specifies the behaviour of the manual power window movement;
- *Automatic Power Window*, which specifies the behaviour of the automatic power window movement;
- *Remote Control Key*, which specifies the behaviour of the remote control key controller reacting to remote signals;
- *Central Locking System*, which specifies the activation/deactivation of the central locking system;
- *Human Interface Component*,which specifies the behaviour of the human machine interface reacting to the interaction with the driver;
- *LED Automatic Power Window*, which specifies the turning on/off of the LED if the automatic power window moves down/up;
- *Alarm System*, which specifies the behaviour of the activation/deactivation of the alarm system as well as the enabling/disabling of the alarm monitoring;
- *Exterior Mirror*, which specifies the behaviour of the exterior mirror position adjustment.

Table 6.1 summarizes the number of states, transitions and delta models for each core test model selected; the name of each delta model was extracted from the original source.

Furthermore, there is an identifier (ID) for each model that will be used to reference each one of them from here.

Table 6.1: Core test model information of delta case study

ID	Model	States	Transitions	Deltas
M1	*Manual Power Window*	8	13	1
M2	*Automatic Power Window*	15	19	1
M3	*Remote Control Key*	3	4	3
M4	*Central Locking System*	4	4	2
M5	*Human Interface Component*	7	12	3
M6	*LED Automatic Power Window*	7	8	1
M7	*Alarm System*	10	13	2
M8	*Exterior Mirror*	21	48	2

As an example, Figure 6.1 illustrates the core test model M2 (*Automatic Power Window*). All the selected models are available on `https://goo.gl/I3T9Sf`.

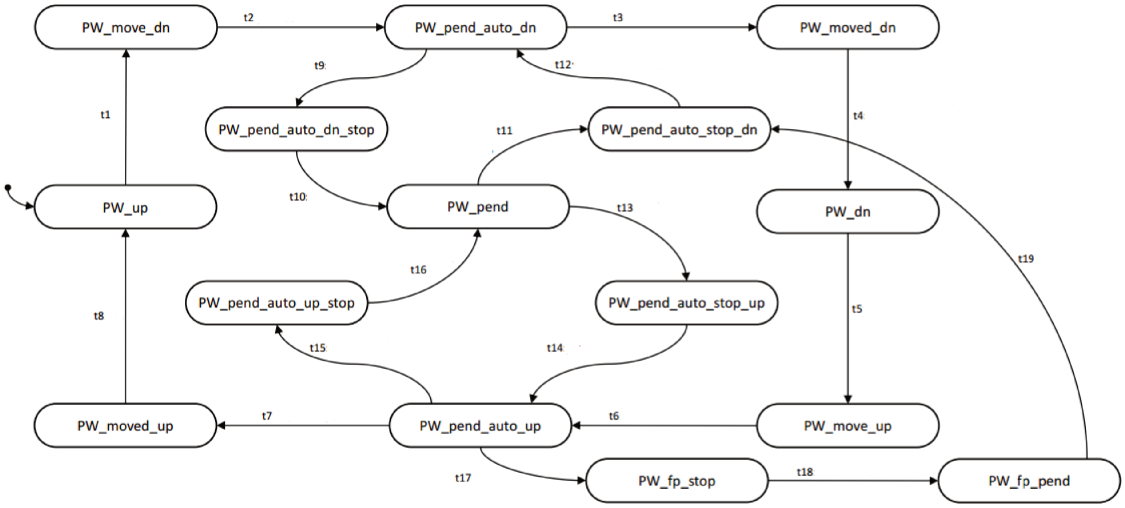

Figure 6.1: M2 *Automatic Power Window* core test model (extracted from [34])

In total, there are 15 delta models, which applied to the correspondent core test model result in test model versions. The description of each delta model is on Appendix A.

Table 6.2 describes for each delta model: the number of states and transitions added (labelled with a + (plus)) and/or removed (labelled with a − (minus), and an identifier (ID) that will be used to reference it from here. Table B.1 in Appendix B presents which transitions were added or removed for each delta model.

Table 6.2: Delta model information of delta case study

ID	Delta	States	Transitions
M1_Delta_1	*DAddManPWCLS*	+5	+14
M2_Delta_1	*DAddAutoPWCLS*	+8	+16
M3_Delta_2	*DAddRCKSF*	+2	+4
M3_Delta_1	*DAddRCKCAP*	+4 −1	+7 −2
M3_Delta_3	*DAddRCKCAPSF*	+4	+8
M4_Delta_1	*DAddCLSAL*	+3	+4
M4_Delta_2	*DAddCLSRCK*	0	+2
M5_Delta_1	*DAddHMIAS*	+2	+4
M5_Delta_2	*DAddHMILEDAS*	+1	+2
M5_Delta_3	*DAddHMILEDManPW*	+1	+3
M6_Delta_1	*DAddLEDAutoPWCLS*	+6	+9
M7_Delta_1	*DAddASCAS*	0	+3
M7_Delta_2	*DAddASIM*	+3 0	+6 −2
M8_Delta_1	*DAddEMHeating*	+18	+36
M8_Delta_2	*DAddEMLEDEM*	+24 0	+48 −24

Table 6.3 describes how the test model versions were obtained. For each test model version, the table shows which test model version was used as the base, acting as a previous version, and what delta model was applied to that base test model in order to obtain the new version. Moreover, the table shows the number of states and transitions for each test model version, and an identifier (ID) that will be used to reference it from here.

Table 6.3: Information of test model versions of delta case study

ID	Previous test model ID	Delta ID	States	Transitions
M1_D1	M1	M1_Delta_1	13	27
M2_D1	M2	M2_Delta_1	23	35
M3_D2	M3	M3_Delta_2	5	8
M3_D2_D1	M3_D2	M3_Delta_1	8	13
M3_D2_D1_D3	M3_D2_D1	M3_Delta_3	12	21
M4_D1	M4	M4_Delta_1	7	8

Continued on next page

Table 6.3 – continued from previous page

ID	Previous test model ID	Delta ID	States	Transitions
M4_D1_D2	M4_D1	M4_Delta_2	7	10
M5_D1	M5	M5_Delta_1	9	16
M5_D1_D2	M5_D1	M5_Delta_2	10	18
M5_D1_D2_D3	M5_D1_D2	M5_Delta_3	11	21
M6_D1	M6	M6_Delta_1	13	17
M7_D1	M7	M7_Delta_1	10	16
M7_D1_D2	M7_D1	M7_Delta_2	13	20
M8_D1	M8	M8_Delta_1	39	84
M8_D1_D2	M8_D1	M8_Delta_2	63	108

As an example, we obtain the test model version M2_D1 for the component *Automatic Power Window* with *Central Locking system*, illustrated in Figure 6.2, after applying *M2_Delta_1* to the core *M2* model, illustrated in Figure 6.3. All the delta models follow the notation of a + (plus) to represent addition of an element, and a − (minus) for removals. Furthermore, the elements from the previous model are represented by dashed borderlines. All the delta models and the test model versions are available on `https://goo.gl/I3T9Sf`.

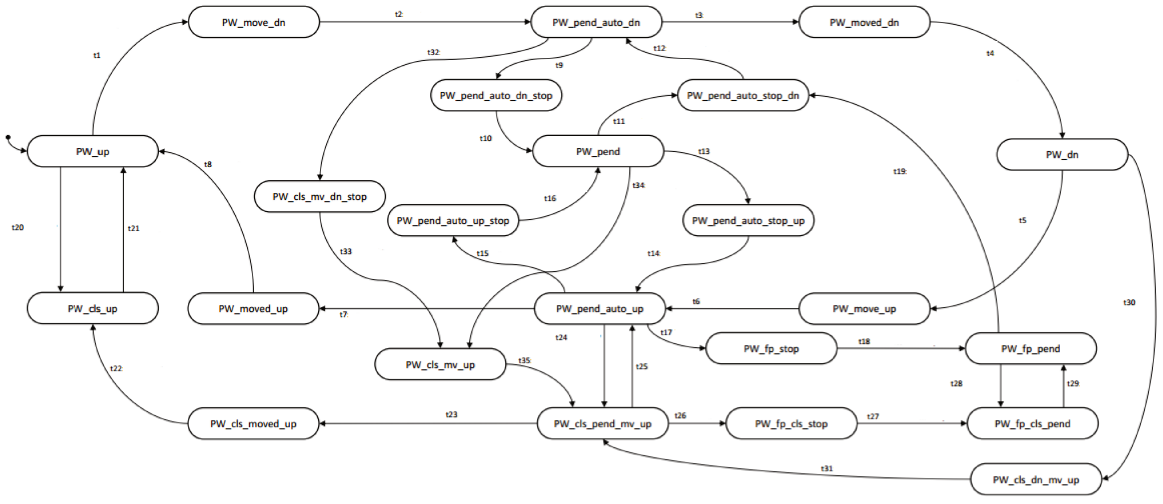

Figure 6.2: *Automatic Power Window* with *Central Locking system* test model version (M2_D1) (extracted from [34])

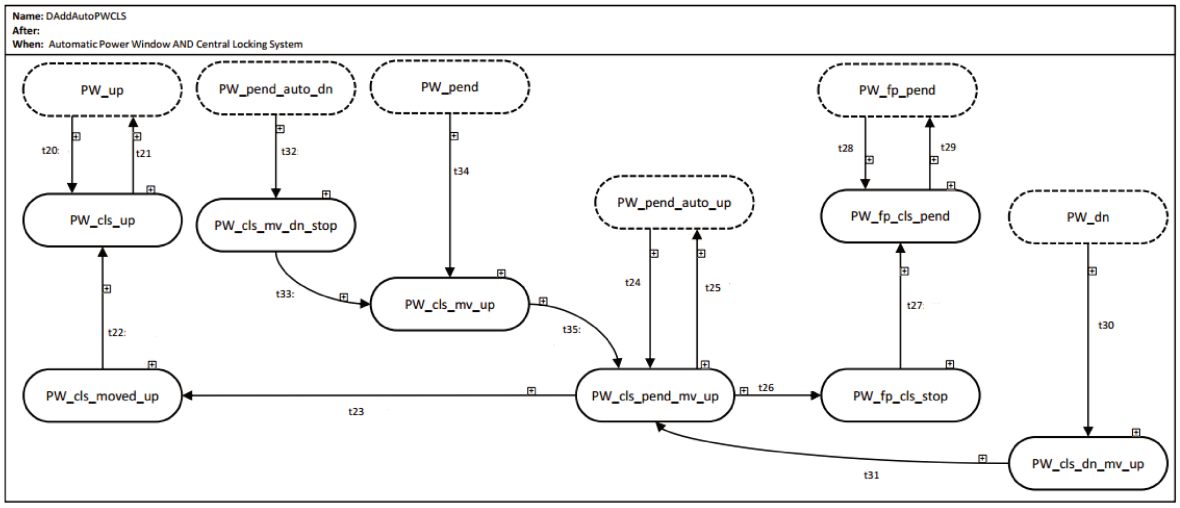

Figure 6.3: *M2_ Delta_ 1* of M2: *DAddAutoPWCLS* delta model (extracted from [34])

6.2.2 SWPDC - Software for a Data Collection Platform

The other object was a space application, the Software for a Data Collection Platform (SWPDC), from INPE[1], the Brazilian Institute for Space Research. The state machine test models were modelled together with an industrial partner and follow the CoFI (Conformance and Fault Injection) methodology. CoFi is a Model-Based testing methodology that addresses, among others, conformance and fault injection testing systematization of embedded software in space missions, with the aim of having automatic test case generation. To fulfill this aim, CoFI uses a set of Finite State Machines (FSM). For each service provided by the system under test, its behaviour is modelled in different viewpoints: Normal, Specified Exception, Fault Tolerance and Sneak Path (correct inputs occurring at the wrong moments). For each viewpoint one or more FSM are created [24].

Because each state machine represents a viewpoint of the application, when comparing the different state machines, it is noticed that they have few elements in common. That is explained because the state machines were not modelled incrementally. Therefore, each state machine do not represent neither a variant of a test model nor a version of a test model.

Even though state machines from different viewpoint were not created incrementally, it was possible to simulate an incremental development using the provided artefacts. Therefore, some adaptations were performed in order to use them in the controlled experiments. The first FSM in the Normal viewpoint was considered as the core test model, and new versions were created by adding states and transitions from the other state machines.

The set of services was modelled as FSM test models. However, in order to use them as an example of incremental development, only the services that had at least three FSMs, counting those of the Normal viewpoint and of the Specified Exception viewpoint, were analysed. In total 8 services were analysed. From these services, only one was used in the controlled experiments due to time restrictions.

The core test model of the selected example was composed of **16 states** and **20**

[1] www.inpe.br

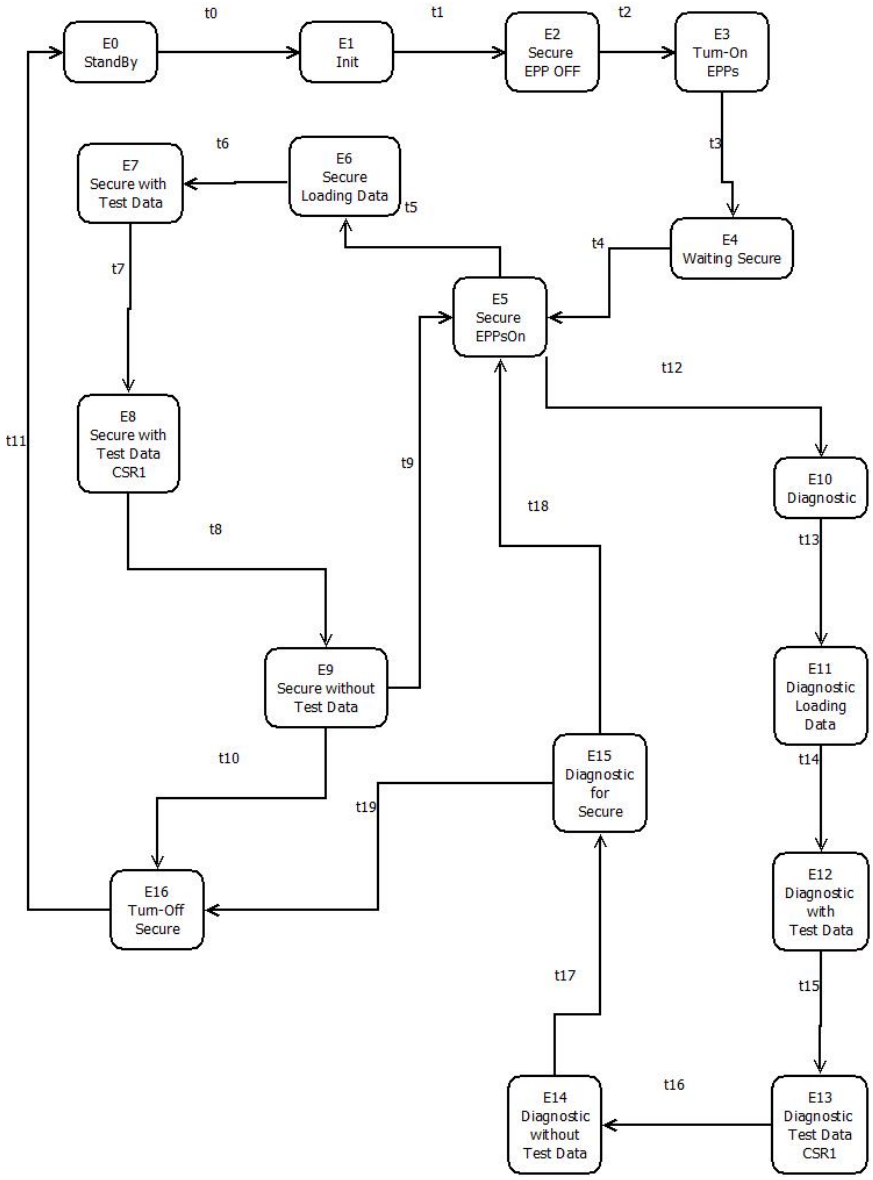

Figure 6.4: CoFI core test model

transitions and it is illustrated in Figure 6.4. There was another FSM of the Normal type and 4 FSM of the Specified Exception, composing **5 deltas** to be applied to the core test model. This core test model will be referenced with an identifier (ID) of S04.

Table 6.4 describes for each delta model: the number of states and transitions added (labelled with a + (plus)), and an identifier (ID) that will be used to reference it from here. Table B.1 in Appendix B presents which transitions were added for each delta model.

Table 6.4: Delta models information of CoFI

ID	Delta	States	Transitions
S04_Delta_1	*Normal 2*	+2	+4
S04_Delta_2	*Specified Exception 1*	+4	+13
S04_Delta_3	*Specified Exception 2*	0	+6
S04_Delta_4	*Specified Exception 3*	+1	+8
S04_Delta_5	*Specified Exception 4*	+1	+4

Table 6.5 describes for each test model version: the relationship between the core test model and which delta(s) model(s) are applied in order to obtain the test model version, the number of states and transitions, and an identifier (ID) that will be used to reference it from here.

Table 6.5: Information of test model versions of CoFI

ID	Model_ID	Delta_ID	States	Transitions
S04_D1	S04	S04_Delta_1	18	24
S04_D1_D2	S04_D1	S04_Delta_2	22	37
S04_D1_D2_D3	S04_D1_D2	S04_Delta_3	22	43
S04_D1_D2_D3_D4	S04_D1_D2_D3	S04_Delta_4	23	51
S04_D1_D2_D3_D4_D5	S04_D1_D2_D3_D4	S04_Delta_5	24	55

As example, the test model version S04_D1, illustrated in Figure 6.6, is obtained after applying *S04_Delta1*, illustrated in Figure 6.5, to the core test model. The added elements are represented with dashed lines. All the delta models and the test model versions are available on `https://goo.gl/WRhMs6`.

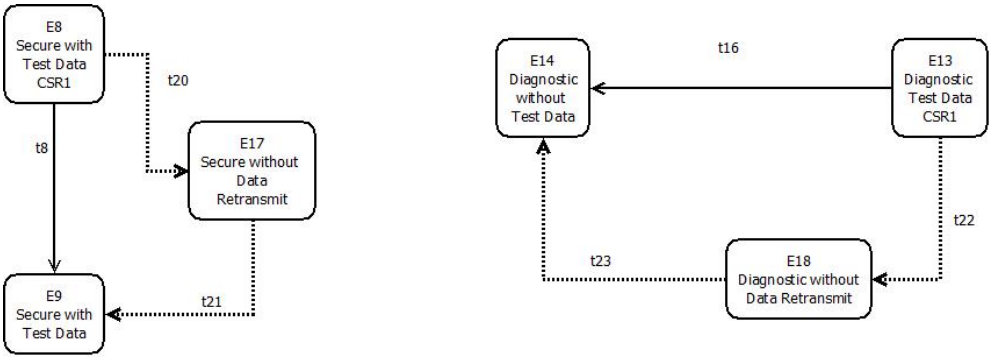

Figure 6.5: *S04_Delta_1* delta model

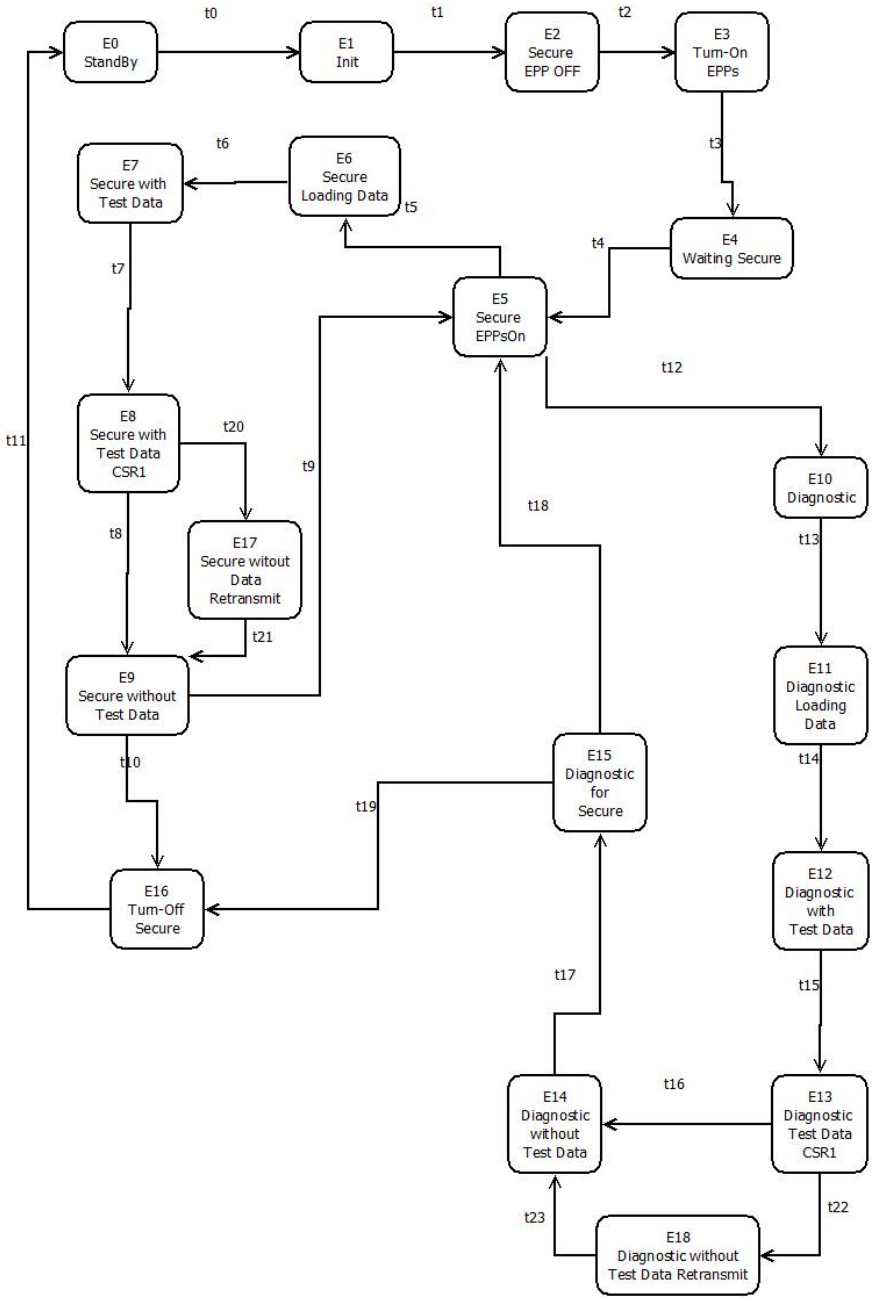

Figure 6.6: S04_D1 test model version

6.3 Workflow of the Experiments

During the controlled experiments, the workflow described in Subsection 6.3.1 was followed in order to execute the experiments with the first version of each state machine. For each test model version, the process described in Subsection 6.3.2 was followed in order to update the test suite.

Even though D-MBTDD uses a *test plan* and a development cycle in order to support the development of the system, these steps were not performed because the objectives are related to the test suite update. Therefore, the system was not implemented and the experiments stopped on the test suite creation/update.

6.3.1 Workflow for core model experiments

Following the process specified in Subsection 5.2.1, for each core test model of each project of both MBT tools, the following steps were performed:

1. **Create the state machine.**
2. **Define the Test Goals** based on a coverage criterion of **100% transition coverage**.
3. **Configure the parameters** of the MBT tool.
4. **Generate the test suite** that covers all the test goals.

6.3.2 Workflow for test model version experiments

After the test suite creation for the core test model, its versions were created. During the controlled experiment, D-MBTDD and Regenerate-All processes were performed for each test model version. In order to generate test cases, the same criterion of 100% of transition coverage was used with Regenerate-All and D-MBTDD.

In the Regenerate-All process, when the system evolves only the test model is reused and new test cases are created from scratch. Therefore, the steps performed to create a new version are the same as described in Subsection 6.3.1.

Differently from Regenerate-All, D-MBTDD aims to reuse test cases from the previous test suite, besides the test model reuse. Thus, according to the process specified in Subsection 5.2.2, for each version of each core test model, the following steps were performed:

1. **Create the new version** of the state machine.
2. **Update the Test Goals** based on a coverage criterion of **100% transition coverage**.
3. **Revalidate the test cases** from the previous test suite based on the delta model.
4. **Configure the parameters** of the MBT tool.
5. **Update the Test Suite**.
   - 5.1. Check if any obsolete test cases from the old test suite become valid.
   - 5.2. Generate new test cases to cover all test goals, based on the modified elements.

In order to automate the execution of steps 2. and 3. a *shell script* was created and it is available on `https://goo.gl/MXywuW`.

The examples used during the controlled experiments did not have a situation in which an obsolete test case becomes valid. Therefore, the step 5.1. was not executed in any experiment.

6.4 Controlled Experiment 1: StateMutest Experiments

All the objects described in Section 6.2 were created in the StateMutest tool. A description of the tool is present in Subsection 6.4.1. A setup process was necessary in order to generate the test cases and it is described in Subsection 6.4.2. The results obtained from the core test model M2 of the delta case study, illustrated in Figure 6.1, was *t8.* with the experiments, and the results of some metrics defined in 6.1.1 are shown in Subsection 6.4.3. The analyses of the results valid only for StateMutest experiments are described in Subsection 6.4.4.

6.4.1 State Mutest

StateMutest [14] is an MBT tool developed by a partnership between researchers from the University of Campinas (UNICAMP) and the Federal University of São Carlos (UFSCar), and it uses FSM or Extended Finite State Machines (EFSM) as its models. Besides other features, StateMutest automates the offline test case generation based on models by means of the MOST (Multi-Objective Search-based Testing) algorithm, proposed in Yano. MOST uses the concepts of Model-Based testing and search based testing, thus it generates sequences of multi-objective tests by means of a meta-heuristic algorithm. It uses a multi-objective optimization approach, which searches for a balance between two objective functions: the minimum length of the input sequence and the test purpose coverage.

To generate the test cases, some parameters have to be configured, and three of them are the most relevant for the controlled experiment:

- **The value of Tau** is related to the determinism of the search algorithm. The determinism tends to increase with greater values. According to Yano [59], the range of values that maximizes the algorithm efficiency is between 1 and 5.
- MOST evaluates a configured **Maximum Number of Evaluations (maxNumEval)** of the objective functions. However, the algorithm resets its search after $(maxNumEval/numRes)$ evaluations, in which **numRes** represents the **number of reinitialization** of the algorithm.

MOST uses a set of transitions to be covered as the test purpose, which is called the **coverage set**. It is also necessary to define a **target transition** that all test cases must cover. As output StateMutest generates abstract test cases, which contain, among other information, a sequence of reached transitions [14, 59, 60].

6.4.2 Preparation

As explained in 6.4.1, StateMutest has an option for selecting the *coverage set* and which *target transition* has to be covered during test case creation. For this study, a transition that leads back to the initial state was chosen as the target transition. The reason for that was to generate test cases that represent a cycle which starts and finishes in the

same state. For example, the target transition of the core test model M2 of the delta case study, illustrated in Figure 6.1 was *t8*.

In order to perform step 3 of the workflow for the core model experiments (Subsection 6.3.1), the target transition was defined and the coverage set was configured as all transitions minus the target transition. Because the coverage criterion was 100% transition coverage, the coverage set for all core test models included all the transitions of the state machine minus the target transition; for example, for the core test model M2 the coverage set was all transitions minus *t8*.

In order to perform step 4. of the workflow for the test model versions experiments (Subsection 6.3.2), D-MBTDD and Regenerate-All had different configurations. In order to obtain a 100% transition coverage when D-MBTDD was used, the coverage set was composed of missing transitions, because part of the transitions were already covered with the reusable test cases. However, when Regenerate-All was used, in order to obtain the same coverage criterion the coverage set was composed of all the transitions of the state machine minus the target transition. For example, for the test model version M2_D1 (Figure 6.2) the target transition was $t21$ and the coverage set was $t20..t35$ when using D-MBTDD, and all transitions minus $t21$ when using Regenerate-All.

The relationships among all the state machines, their tau value, their target transition, and their coverage set in the StateMutest projects are shown in Table B.2 in B.

As explained in Section 6.4.1, there are some parameters that have to be configured in order to generate the test cases in StateMutest: the *tau value*, the *maximum number of evaluations (maxNumEval)* and the *number of reinitialization (numRes)*. There are no standard parameter values that work well on every problem, therefore it is required to find suitable values for the parameters of the search algorithm. Specifically, the *tau value* must be tuned for each model, otherwise, the algorithm performance will suffer. For that reason, it is necessary to perform some experiments to fine tune the tau value. The performance here is measured in terms of the number of test cases generated: the greater this number, the better.

To search for a suitable tau value, for each state machine the test case generation process was executed five times varying the tau value from 1 to 5. The value that generated the largest quantity of test cases was selected. If after the preliminary executions there were two or more values of tau which generated the largest quantity of test cases, an analysis of the generated test cases was performed in order to define the most adequate value. For each state machine, the relationship between the value of tau and the quantity of generated test cases is represented in Table B.3 in Appendix B. The greater the value of *maxNumEval* is, the more accurate is the result, but the slower the execution is. Therefore, during the preliminary executions, the value of *maxNumEval* was set to 10^5 and the value of *numRes* was set to 10, and after the definition of the tau value, the value of *maxNumEval* was increased to 10^6 and the value of *numRes* to 100.

6.4.3 Results

Table 6.6 shows the number of valid test cases for each model and each approach in the experiments that used the StateMutest tool. Furthermore, when applicable, i.e. when D-

MBTDD was used, it also shows the number of reusable, obsolete and new test cases. In these cases, the number of valid test cases was the sum of the value of new and reusable test cases, while in the Regenerate-All experiments the number of valid test cases was equal to the value of generated test cases. When a determined information was not applicable, a minus (−) signal is displayed.

Table 6.6: Number of valid, new, reusable and obsolete test cases in StateMutest experiments

Model ID	**Approach**	**Valid TCs**	**New TCs**	**Reusable TCs**	**Obsolete TCs**
M1	-	8	-	-	-
M1_D1	Regenerate-All	13	-	-	-
M1_D1	D-MBTDD	15	7	8	0
M2	-	8	-	-	-
M2_D1	Regenerate-All	17	-	-	-
M2_D1	D-MBTDD	15	7	8	0
M3	-	2	-	-	-
M3_D2	Regenerate-All	5	-	-	-
M3_D2	D-MBTDD	5	3	2	0
M3_D2_D1	Regenerate-All	4	-	-	-
M3_D2_D1	D-MBTDD	5	2	3	2
M3_D2_D1_D3	Regenerate-All	8	-	-	-
M3_D2_D1_D3	D-MBTDD	11	6	5	0
M4	-	1	-	-	-
M4_D1	Regenerate-All	5	-	-	-
M4_D1	D-MBTDD	2	1	1	0
M4_D1_D2	Regenerate-All	6	-	-	-
M4_D1_D2	D-MBTDD	4	2	2	0
M5	-	8	-	-	-
M5_D1	Regenerate-All	7	-	-	-
M5_D1	D-MBTDD	10	2	8	0
M5_D1_D2	Regenerate-All	7	-	-	-
M5_D1_D2	D-MBTDD	12	2	10	0
M5_D1_D2_D3	Regenerate-All	10	-	-	-
M5_D1_D2_D3	D-MBTDD	14	2	12	0
M6	-	3	-	-	-
M6_D1	Regenerate-All	12	-	-	-
M6_D1	D-MBTDD	10	7	3	0
M7	-	8	-	-	-
M7_D1	Regenerate-All	11	-	-	-
M7_D1	D-MBTDD	11	3	8	0

Continued on next page

Table 6.6 – continued from previous page

Model ID	Approach	Valid TCs	New TCs	Reusable TCs	Obsolete TCs
M7_D1_D2	Regenerate-All	13	-	-	-
M7_D1_D2	D-MBTDD	15	10	5	6
M8	-	12	-	-	-
M8_D1	Regenerate-All	10	-	-	-
M8_D1	D-MBTDD	20	8	12	0
M8_D1_D2	Regenerate-All	11	-	-	-
M8_D1_D2	D-MBTDD	11	9	2	18
S04	-	5	-	-	-
S04_D1	Regenerate-All	6	-	-	-
S04_D1	D-MBTDD	7	2	5	0
S04_D1_D2	Regenerate-All	6	-	-	-
S04_D1_D2	D-MBTDD	13	6	7	0
S04_D1_D2_D3	Regenerate-All	12	-	-	-
S04_D1_D2_D3	D-MBTDD	19	6	13	0
S04_D1_D2_D3_D4	Regenerate-All	13	-	-	-
S04_D1_D2_D3_D4	D-MBTDD	24	5	19	0
S04_D1_D2_D3_D4_D5	Regenerate-All	10	-	-	-
S04_D1_D2_D3_D4_D5	D-MBTDD	27	3	24	0

For each **test model version**, the values of Generated Test Cases (GenTC), Not Modification Test Cases (NotModTC), Modification Test Cases (ModTC) and Focus obtained with StateMutest are shown in Table 6.7.

Table 6.7: Value of metrics in StateMutest experiments

Model ID	Approach	GenTC	NotModTC	ModTC	Focus (%)
M1_D1	Regenerate-All	13	2	11	84.62
M1_D1	D-MBTDD	7	0	7	100
M2_D1	Regenerate-All	17	0	17	100
M2_D1	D-MBTDD	7	0	7	100
M3_D2	Regenerate-All	5	0	5	100
M3_D2	D-MBTDD	3	0	3	100
M3_D2_D1	Regenerate-All	4	0	4	100

Continued on next page

Table 6.7 – continued from previous page

Model ID	Approach	GenTC	NotModTC	ModTC	Focus (%)
M3_D2_D1	D-MBTDD	2	0	2	100
M3_D2_D1_D3	Regenerate-All	8	0	8	100
M3_D2_D1_D3	D-MBTDD	6	1	5	83.33
M4_D1	Regenerate-All	5	0	5	100
M4_D1	D-MBTDD	1	0	1	100
M4_D1_D2	Regenerate-All	6	1	5	83.33
M4_D1_D2	D-MBTDD	2	0	2	100
M5_D1	Regenerate-All	7	0	7	100
M5_D1	D-MBTDD	2	0	2	100
M5_D1_D2	Regenerate-All	7	0	7	100
M5_D1_D2	D-MBTDD	2	0	2	100
M5_D1_D2_D3	Regenerate-All	10	0	10	100
M5_D1_D2_D3	D-MBTDD	2	0	2	100
M6_D1	Regenerate-All	12	3	9	75
M6_D1	D-MBTDD	7	1	6	85.71
M7_D1	Regenerate-All	11	2	9	81.82
M7_D1	D-MBTDD	3	0	3	100
M7_D1_D2	Regenerate-All	11	6	5	45.45
M7_D1_D2	D-MBTDD	10	3	7	70
M8_D1	Regenerate-All	10	2	8	80
M8_D1	D-MBTDD	8	1	7	87.50
M8_D1_D2	Regenerate-All	10	3	7	70
M8_D1_D2	D-MBTDD	9	2	7	77.78
S04_D1	Regenerate-All	6	1	5	83.33
S04_D1	D-MBTDD	2	0	2	100
S04_D1_D2	Regenerate-All	6	0	6	100

Continued on next page

Table 6.7 – continued from previous page

Model ID	Approach	GenTC	NotModTC	ModTC	Focus (%)
S04_D1_D2	D-MBTDD	6	0	6	100
S04_D1_D2_D3	Regenerate-All	12	2	10	83.33
S04_D1_D2_D3	D-MBTDD	6	0	6	100
S04_D1_D2_D3_D4	Regenerate-All	13	4	9	69.23
S04_D1_D2_D3_D4	D-MBTDD	5	1	4	80
S04_D1_D2_D3_D4_D5	Regenerate-All	10	7	3	30
S04_D1_D2_D3_D4_D5	D-MBTDD	3	1	2	66.67

6.4.4 Results Analyses

StateMutest uses a multi-objective algorithm that is not deterministic, thus it has some characteristics:

- Each execution of a StateMutest example can generate different test cases. Thus, executing two times the algorithm with the same configuration (model, test purposes, and so on), the test suite generated will not necessarily be the same.
- When D-MBTDD was used, the missing transitions were specified as the coverage set during the preparation phase, therefore it was expected that all generated test cases traversed modified elements. However the multi-objective algorithm searches for a balance between the minimum length of the input sequence and the test purpose coverage. Consequently, even though in the coverage set there were only modified elements, it was possible to generate some test cases that traverse only the unmodified parts of the state machine when D-MBTDD was used. For example, it happened with the M3_D2_D1_D3, M6_D1, M7_D1_D2 experiments in Table 6.7. However, these results could be improved if a different coverage set was used.

Therefore, some findings were only identified on StateMutest examples and they are explained below.

As Table 6.6 shows, the number of valid test cases and, consequently, the size of the test suite tends to increase as the state machine evolves, when D-MBTDD was used. When the number of valid test cases generated with D-MBTDD is compared to the number of Regenerate-All, it can be greater when D-MBTDD is used. For example, while Regenerate-All had 10 test cases with M5_D1_D2_D3 experiment, D-MBTDD had 14 valid test cases. This happens because with Regenerate-All there is no reuse of previous test cases, therefore the test suite is composed only of the newly generated test cases. Differently from Regenerate-All, in D-MBTDD the test suite is composed of reusable and

new test cases, therefore because the number reusable test cases tends to increase with the evolution of the model, the same happens with the size of the test suite. Moreover, it tends to increase because there is no use of a selection testing technique for the reusable test cases aiming to select the retestable test cases. If a selection testing technique were used, the size of the test suite could increase more gradually. However, one more step would be necessary to obtain the test suite.

6.5 Controlled Experiments 2: Condado Experiments

All the objects described in Section 6.2 were created in the Condado tool. A description of the tool is present in Subsection 6.5.1. A setup process was necessary in order to generate the test cases and it is described in Subsection 6.5.2. The results obtained with the experiments, and the results of each metric defined in 6.1.1 are shown in Subsection 6.5.3. The analyses of the results valid only for Condado experiments are described in Subsection 6.5.4.

6.5.1 Condado

Condado is an MBT tool developed among researchers from the University of Campinas (UNICAMP) and the Brazilian Institute for Space Research (INPE), and it uses FSM as its models. Condado automates the offline test case generation based on models by means of an exhaustive algorithm like depth-first search. Condado implements the all-transition pairs as test criterion. The all transition-pairs generates test cases that covers all combinations of pairs of transitions, therefore it is liable to combinatorial explosion. There is also the possibility to generate test cases that only cover some specified transitions, instead of covering all transitions. As output Condado generates abstract test cases, which contain, among other information, a sequence of reached transitions [2].

6.5.2 Preparation

As explained in 6.5.1, Condado exhaustively combines all transitions to derive the test cases, which starts in the initial state and ends in a specified final state. For the controlled experiments, the final state was defined as the initial state. For example, for the core test model M2 (6.1) the final state was *pwUp*. The relationship between all the state machines and their final states in the Condado projects are shown in Table B.4 in Appendix B.

Condado generates a test suite that covers all transitions, i.e. has 100% transition coverage. Therefore, in order to perform step 3 of the workflow for the core model experiments (Subsection 6.3.1), only the final state was configured for the core test models.

In order to perform step 4. of the workflow for the test model versions experiments (Subsection 6.3.2), D-MBTDD and Regenerate-All had different configurations. In order to obtain 100% transition coverage when D-MBTDD was used, the coverage set was composed of missing transitions, because part of the transitions were already covered with the reusable test cases. As explained in 6.5.1, it is possible to generate test cases that cover only some specified transitions with Condado. In order to do that, these transitions

are described in a setup file. Therefore, for the test model versions when D-MBTDD was used the missing transitions were specified in the setup file. In order to obtain the same coverage criterion when Regenerate-All was used only the final state was configured.

6.5.3 Results

Table 6.8 shows the number of valid test cases for each model and each approach in the experiments that used the Condado tool. Furthermore, when applicable, i.e. when D-MBTDD was used, it shows the number of reusable, obsolete and new test cases. In these cases, the number of valid test cases was the sum of the value of new and reusable test cases, while in the Regenerate-All experiments the number of valid test cases was equal to the value of generated test cases. When a determined information was not applicable, a minus (−) signal is displayed.

Because Condado uses an exhaustive algorithm that combines all transitions to generate the test cases, for some models there was an explosion on the number of test cases: M1_D1; M8; M8_D1; M8_D1_D2; S04_D1_D2; S04_D1_D2_D3; S04_D1_D2_D3_D4; S04_D1_D2_D3_D4_D5. Therefore, they are not present in the results.

Table 6.8: Number of valid, new, reusable and obsolete test cases in Condado experiments

Model ID	**Approach**	**Valid TCs**	**New TCs**	**Reusable TCs**	**Obsolete TCs**
M1	-	304	-	-	-
M2	-	24	-	-	-
M2_D1	Regenerate-All	4278	-	-	-
M2_D1	D-MBTDD	4278	4254	24	0
M3	-	2	-	-	-
M3_D2	Regenerate-All	4	-	-	-
M3_D2	D-MBTDD	4	2	2	0
M3_D2_D1	Regenerate-All	6	-	-	-
M3_D2_D1	D-MBTDD	6	3	3	1
M3_D2_D1_D3	Regenerate-All	38	-	-	-
M3_D2_D1_D3	D-MBTDD	38	32	6	0
M4	-	1	-	-	-
M4_D1	Regenerate-All	2	-	-	-
M4_D1	D-MBTDD	2	1	1	0
M4_D1_D2	Regenerate-All	5	-	-	-
M4_D1_D2	D-MBTDD	5	3	2	0
M5	-	6	-	-	-
M5_D1	Regenerate-All	8	-	-	-
M5_D1	D-MBTDD	8	2	6	0

Continued on next page

Table 6.8 – continued from previous page

Model ID	Approach	Valid TCs	New TCs	Reusable TCs	Obsolete TCs
M5_D1_D2	Regenerate-All	9	-	-	-
M5_D1_D2	D-MBTDD	9	1	8	0
M5_D1_D2_D3	Regenerate-All	11	-	-	-
M5_D1_D2_D3	D-MBTDD	11	2	9	0
M6	-	2	-	-	-
M6_D1	Regenerate-All	9	-	-	-
M6_D1	D-MBTDD	9	7	2	0
M7	-	5	-	-	-
M7_D1	Regenerate-All	17	-	-	-
M7_D1	D-MBTDD	17	12	5	0
M7_D1_D2	Regenerate-All	37	-	-	-
M7_D1_D2	D-MBTDD	37	32	5	12
S04	-	10	-	-	-
S04_D1	Regenerate-All	52	-	-	-
S04_D1	D-MBTDD	52	42	10	0

For each **test model version**, the values of Generated Test Cases (GenTC), Not Modification Test Cases (NotModTC), Modification Test Cases (ModTC) and Focus obtained with Condado are shown in Table 6.9.

Table 6.9: Value of metrics in Condado experiments

Model ID	Approach	GenTC	NotModTC	ModTC	Focus (%)
M2_D1	Regenerate-All	4278	24	4254	99.44
M2_D1	D-MBTDD	4254	0	4254	100
M3_D2	Regenerate-All	4	2	2	50
M3_D2	D-MBTDD	2	0	2	100
M3_D2_D1	Regenerate-All	6	3	3	50
M3_D2_D1	D-MBTDD	3	0	3	100
M3_D2_D1_D3	Regenerate-All	38	6	32	84.21
M3_D2_D1_D3	D-MBTDD	32	0	32	100
M4_D1	Regenerate-All	2	1	1	50
M4_D1	D-MBTDD	1	0	1	100
M4_D1_D2	Regenerate-All	5	2	3	60
M4_D1_D2	D-MBTDD	3	0	3	100
M5_D1	Regenerate-All	8	6	2	25

Continued on next page

Table 6.9 – continued from previous page

Model ID	Approach	GenTC	NotModTC	ModTC	Focus (%)
M5_D1	D-MBTDD	2	0	2	100
M5_D1_D2	Regenerate-All	9	8	1	11.11
M5_D1_D2	D-MBTDD	1	0	1	100
M5_D1_D2_D3	Regenerate-All	11	9	2	18.18
M5_D1_D2_D3	D-MBTDD	2	0	2	100
M6_D1	Regenerate-All	9	2	7	77.78
M6_D1	D-MBTDD	7	0	7	100
M7_D1	Regenerate-All	17	5	12	70.59
M7_D1	D-MBTDD	12	0	12	100
M7_D1_D2	Regenerate-All	37	5	32	86.49
M7_D1_D2	D-MBTDD	32	0	32	100
S04_D1	Regenerate-All	52	10	42	80.77
S04_D1	D-MBTDD	42	0	42	100

6.5.4 Results Analyses

Condado uses a deterministic algorithm that exhaustively combines all transitions to derive test cases. In the cases where the number of test cases did not explode, no matter how many times it was executed, all executions generated the same test cases. In Condado, when no transitions are specified in the setup file, the test suite of both approaches are the same. However, for D-MBTDD it is useful to indicate the desired transitions during setup, in order to filter the generated test set. The following observations were seen only in the Condado experiments:

1. In Table 6.8, the number of *valid* test cases when Regenerate-All is used is equal to the sum of *new* and *reusable* test cases when D-MBTDD is used. Therefore, the value of *valid* test cases when Regenerate-All was used is the same as the value when D-MBTDD was used.

2. In Table 6.9, the value of *NotModTC* when Regenerate-All was used is the same value of *reusable* test cases from Table 6.8 when D-MBTDD was used.

3. In Table 6.9, the value of *NotModTC* when D-MBTDD was used is always 0.

4. In Table 6.9, the value of *Modification Test Cases* is the same independently of the usage of either Regenerate-All or D-MBTDD.

5. In Table 6.9, the value of *Focus* when using D-MBTDD is always 100%, and the value when Regenerate-All was used is always smaller than 100%.

Items 1, 2 and 4 are explained by the fact that the test cases that traverse the unmodified parts and the modified parts are the same for both approaches. Moreover, for D-MBTDD all generated test cases traverse a modified part, which explains item 3. Consequently, the focus when using D-MBTDD was always 100% (item 5). The focus when Regenerate-All was used was always smaller than 100% (item 5), because at least one Not Modification Traversal Test Case was genenerated (item 2).

6.6 Discussion

Analyses were conducted in order to compare the Regenerate-All and the D-MBTDD approach applied to the same test model version. Ideally, it was expected that when compared to Regenerate-All results, D-MBTDD would provide an equal or greater value of *Focus* because it better supports the development of new features. Furthermore, it was expected that D-MBTDD experiments would provide an equal or smaller value of *Modification Traversal Test Cases* than Regenerate-All experiments, so that there would be less abstract test cases to be transformed into executables in order to support the development of new features.

The results common to both StateMutest and Condado experiments are present in Subsection 6.6.1. A comparison between the effort of using Regenerate-All and D-MBTDD is discussed in Subsection 6.6.2.

6.6.1 Results Common to both MBT tools

When analysing Tables 6.6 to 6.9, some results were common to both tools. The results of the metrics were plotted in bar charts in order to support the analyses. The results evaluated with StateMutest were plotted in Chart 6.7, and the ones evaluated with Condado were plotted in Chart 6.8. The results of the experiments with M2_D1 were omitted in Chart 6.8 in order to simplify it, because they were outliers, generating a greater value of test cases when compared to the other experiments. The **total size** of each bar represents the value of *Generated Test Cases*. Moreover, each bar was split into two parts: **a blue bar filled with little squares**, and **an orange bar filled with circles**. The blue bar represents the value of *Modification Traversal Test Cases*, and the orange bar represents the value of *Not Modification Traversal Test Cases*.

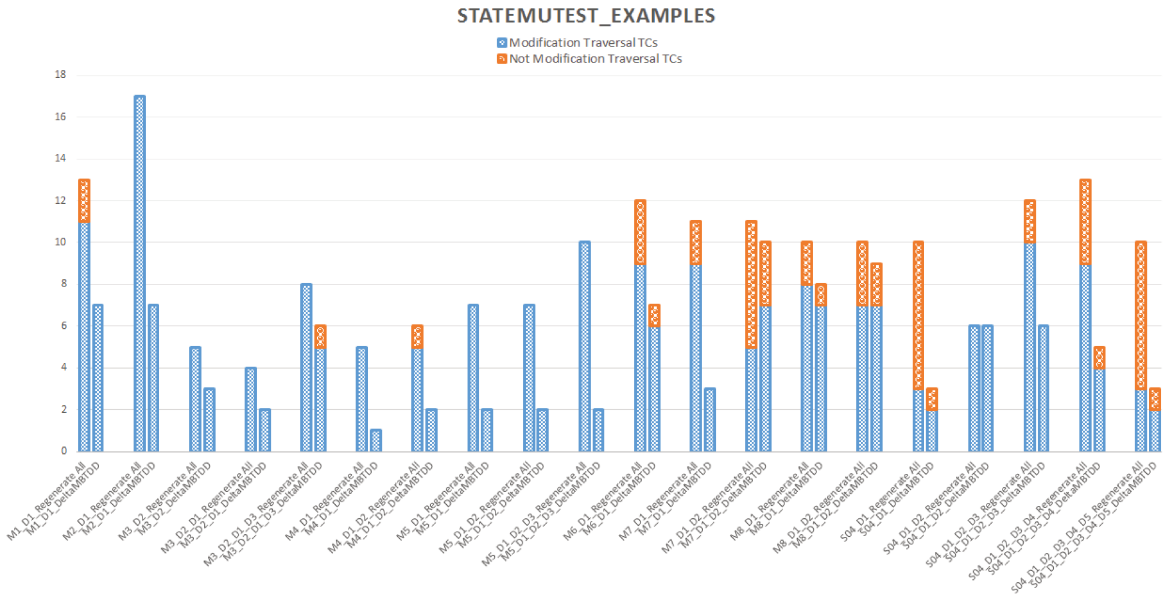

Figure 6.7: Bar Graph with StateMutest results

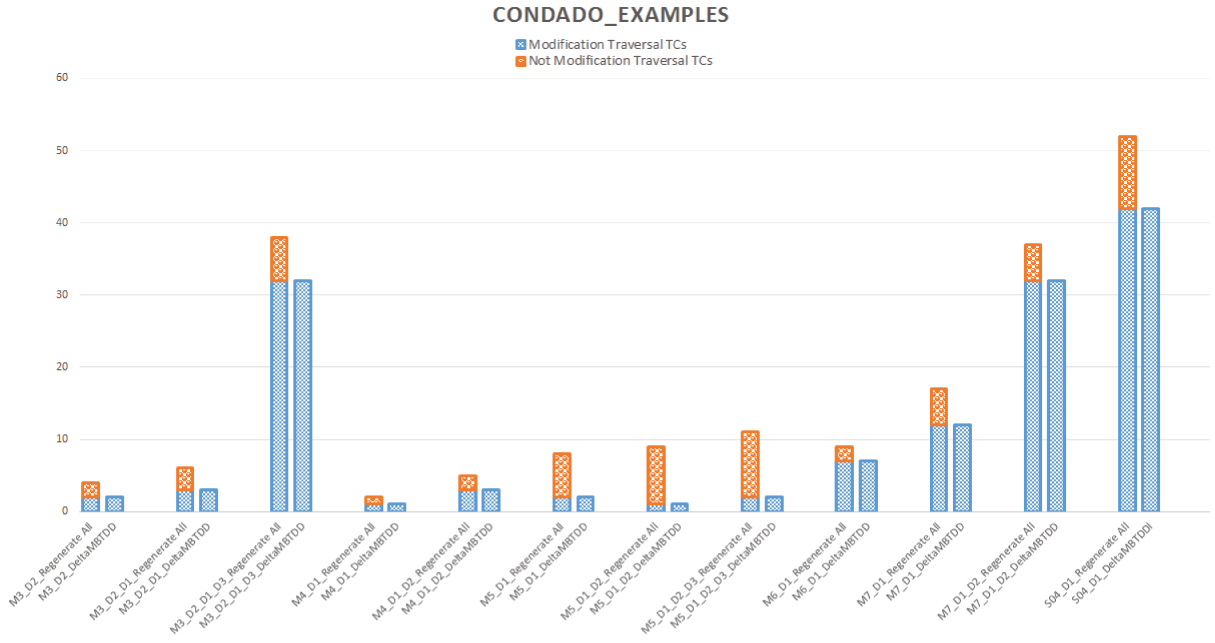

Figure 6.8: Bar Graph with Condado results

These common results are presented below and a brief discussion about them is presented afterwards.

1. When there are only additions in the delta model, the valid test cases from the previous version remain reusable in the new test model version when D-MBTDD is used. However, when the delta model has at least one deletion, part of the valid test cases from the previous version becomes obsolete in the test model version.

2. In all experiments, the value of *Generated Test Cases* when D-MBTDD was used is equal to or smaller than the value when Regenerate-All was used.

3. In 96.97% of the experiments, the value of *ModTC* when D-MBTDD was used is equal to or smaller than the value when Regenerate-All was used. The only exception was the M7_D1_D2 example using the StateMutest tool, in which the opposite happened.

4. In 96.97% of the experiments, the value of *NotModTC* when D-MBTDD was used is equal to or smaller than the value when Regenerate-All was used. The only exception was the M3_D2_D1_D3 example using the StateMutest tool, in which the opposite happened. Remembering that these values could be improved if more experiments with different coverage sets are used.

5. In 96.97% of the experiments, the value of *Focus* when D-MBTDD was used is equal to or greater than the value when Regenerate-All was used. The only exception was the M3_D2_D1_D3 example using the StateMutest tool, in which the opposite happened.

Item 1 can be illustrated with two examples present in Table 6.6 and Table 6.8. It can be seen in Table 6.2 that delta model M7_Delta1 only adds elements and M7_Delta2 deletes two transitions. Thus, when M7_Delta1 was applied to M7 in order to obtain the test model version M7_D1, all valid test cases from M7 remained reusable for M7_D1. And when M7_Delta2 was applied to M7_D1 in order to obtain the test model version M7_D1_D2, not all valid test cases from M7_D1 remained reusable for M7_D1_D2, because some of them became obsolete.

With item 2 it is possible to conclude that, as expected, D-MBTDD generates less test cases per execution when compared to Regenerate-All. Thus, if all generated test cases have to be transformed into executable test cases, D-MBTDD requires less effort with this task when compared to Regenerate-All. This analysis is illustrated by the fact that the total size of the D-MBTDD bars is smaller than the Regenerate-All bars in both Charts.

Item 3 supports the previous conclusion with the fact that in the majority of the examples, D-MBTDD generated an equal or smaller value of modification traversal test cases than Regenerate-All. It is illustrated by the fact that the blue bars of D-MBTDD are equal to or smaller than the Regenerate-All bars, except for the M7_D1_D2 StateMutest example. Moreover, with the Condado experiments the blue bars of both approaches have the same size, which illustrates analysis 4 of Subsection 6.5.4.

Moreover, item 4 illustrates that even though some not modification traversal test cases were generated when D-MBTDD was used, this value was smaller than when Regenerate-All was used. It is illustrated by the fact that the orange bars of D-MBTDD are equal to or smaller than the Regenerate-All bars, except for the M3_D2_D1_D3 StateMutest example. Moreover, with Condado experiments the orange bars of D-MBTDD do not exist, which illustrates analysis 3 of Subsection 6.5.4.

From expression 6.5 and 6.6, the value of *Focus* is inversely proportional to the value of *Not Modification Traversal Test Cases*. Therefore, the value of focus of D-MBTDD is equal to or greater than the value for Regenerate-All, which illustrates analysis 5.

When analysing Tables 6.2, 6.4, 6.7 and 6.9, it was verified that the difference between the number of generated test cases of a test model version when Regenerate-All was used

and when D-MBTDD was used, was impacted by the delta model which was applied in order to obtain this version. Because the experiments used the coverage criterion of 100% transition coverage, only the modifications of the transitions were analysed in the delta models. It was verified that when the delta model had many modifications, the difference between the generated test cases when Regenerate-All was used and when D-MBTDD was used is smaller than when the applied delta model did not contain many modifications. For example, in Table 6.2 it is shown that delta M5_Delta_3 adds 3 transitions and M7_Delta_2 adds 6 transitions and removes 2 transitions. In Table 6.7 it is shown that when M5_Delta_3 was applied in order to obtain M5_D1_D2_D3, the difference between the generated test cases when Regenerate-All was used and when D-MBTDD was used was 8, while when M7_Delta_2 was applied in order to obtain M7_D1_D2, the difference was 1. It is also possible to verify this relationship analysing the differences between the total size of the bar when Regenerate-All was used and D-MBTDD was used, with the modifications of the applied delta model.

6.6.2 Cost comparison between Regenerate-All and D-MBTDD

In the context of a Model-Based test-driven development, the total effort to use an approach is impacted mainly by the following costs: the cost of generating test cases; the cost of identifying test cases that guide the development of new features; the cost of transforming abstract test cases into executable test cases. A comparison between the Regenerate-All and D-MBTDD costs is discussed bellow.

Cost of generating test cases

As equations 6.3 and 6.4 show, Regenerate-All requires less effort to generate test cases per iteration when compared to D-MBTDD because there is no need to classify test cases from the previous test suite. However, this is not true if $(TSO + N) < TSN$ and $c(re) < c(ge)$.

Cost of identifying test cases that guide the development of new features

As present in equation 6.7, in order to support the development of new features, the test cases that traverse modified elements have to be identified. With D-MBTDD there is no such cost to identify these test cases, as described in equation 6.8, because they are already identified in the set of new test cases. With D-MBTDD, the test cases that cover the modified elements and those that cover only unmodified elements are present in the set of new and reusable test cases respectively.

Moreover, based on the controlled experiments, D-MBTDD is more focused on generating test cases that traverse modified elements, when compared to Regenerate-All. In 96.97% of the experiments, the value of *focus* was equal to or greater than when D-MBTDD was used.

Cost of transforming abstract test cases into executable ones

It is not possible to affirm that the generated test cases when D-MBTDD was used is always smaller than the number when Regenerate-All was used, because it depends on

the MBT tool and the coverage criterion used. However, in all experiments executed during this work, $Gen(DMBTDD)$ was equal to or smaller than $Gen(RA)$. Therefore, if the MBT tool generates abstract test cases and the transformation into executable test cases is performed manually, the cost of transforming the generated abstract test cases into executable test cases with D-MBTDD is smaller than with Regenerate-All. However, if the new version has almost nothing in common with the previous, the numbers of $Gen(DMBTDD)$ and $Gen(RA)$ will be more similar, and consequently the cost of transforming the generated abstract test cases into executable test cases with D-MBTDD will be similar to transforming them with Regenerate-All.

Total Effort

Even though Regenerate-All has a smaller cost to generate test cases, in order to support the development of new features it is necessary to identify the test cases that traverse modified elements and transform all generated abstract test cases into executable test cases. D-MBTDD, instead, has an additional cost of revalidating the previous test suite, but does not have the cost of identifying the test cases that guide the development of new features.

As equations 6.3 and 6.4 show, the total effort for using Regenerate-All is impacted by the value of TSN of Regenerate-All, and by the values of TSO and N of D-MBTDD. According to the evolution of the test model, in order to cover an specific criterion, the number of generated test cases when Regenerate-All is used ($Gen(RA)$) tends to increase, and according to equation 6.1 consequently the size of the new test suite (TSN). With D-MBTDD, in order to cover the same criterion, the generated test cases ($Gen(DMBTDD)$), and according to equation 6.2 the value of N, tends to be smaller than $Gen(RA) = TSN$ because there are reusable test cases that already cover parts of the criterion.

If we consider:

- $TSO \approx TSN$, i.e., the size of the old and new test suites are approximately the same, and
- $N < TSN$, the number of generated test cases with D-MBTDD is smaller than with Regenerate-All as discussed above.

Then two conditions improve the effectiveness of D-MBTDD over Regenerate-All:

- $c(tr) > c(re)$, i.e. the cost of transforming abstract test cases into executable test cases is greater than the cost of revalidating the old test suite; and
- $c(id) > c(re)$, i.e. the cost of identifying the test cases that guide the development of new features is greater than the cost of revalidating the old test suite.

6.7 Threats to the Validity of the Experiments

When performing an experiment, there are some threats to the validity to be concerned about [58]

- The metrics measured during the controlled experiments may not be the most adequate and they were defined by the author of the experiments, representing a threat to the construct validity.
- The results were analysed by the same person who performed the controlled experiments, representing a threat to the external validity.
- The models and the number of deltas used during the experiments represent threats to external validity since their complexity, their size and their domain can affect the results.
- The MBT tool and the definition of a target to generate test cases represent threats to internal validity, because they can affect the relationship between the used approach and the obtained results.

Chapter 7

Conclusions and Future Work

This work proposes the D-MBTDD method, an iterative and incremental method based on Model-Based Test-Driven Development (MBTDD) which adds concepts of Delta-Oriented Model-Based SPL Regression Testing to support test artefact reuse along the evolution of the system. D-MBTDD follows the same main steps of MBTDD, thus test cases are generated from test models using Model-Based Testing (MBT) techniques, and these test cases guide the development during the development cycle based on Test-Driven Development. The conclusions of this work are described in Section 7.1, and the limitations of D-MBTDD are described in Section 7.2. The answers to the research questions are detailed in Section 7.3, and the future work in Section 7.4.

7.1 Conclusions

The main contribution of D-MBTDD concerns the reuse not only of the test model, but also of the test cases when the system evolves. The motivation of this work was not only to help testers in generating test cases to guide the development, but also in reducing the tester's effort when new increments are developed. This is achieved in two ways: by reusing the test model, which is modified according to the requirements of the new increment. And also, by reusing test cases, as the tester only needs to create test cases to exercise the modified parts. In this way, the effort to transform the abstract test cases into executable test cases is spent only on new test cases.

After modifying the test model in order to represent the new version of the system, D-MBTDD revalidates the previous test suite in order to identify which test cases are still valid and consequently can be reused, and which ones have to be created. During the development cycle, D-MBTDD proposes that the new test cases support the development of new features and the reusable ones are used as regression tests.

In addition, this work proposes a process in which D-MBTDD is used together with Scrum. In this case, a new cycle is included before the development cycle, in which the test model and the test cases are created in order to guide the development during the development cycle. Moreover, a test specialist role is included in the Scrum team in order to support the test activities during all the Sprint.

To assess whether D-MBTDD can be useful, controlled experiments were performed.

The goal was to determine whether it could be a more cost-effective approach from the point of view of test case generation, w.r.t. the Regenerate-All approach, which does not reuse test cases. Therefore when the test model evolved both approaches were used in order to update the test suite. To support test case generation two MBT tools were used: StateMutest and Condado, and all the experiments were executed with both MBT tools.

Based on the results obtained with the experiments, when the test model of the system is created iteratively and incrementally, D-MBTDD did not generate inferior results when compared to Regenerate-All. For the objects of the experiments, D-MBTDD had greater possibility of generating test cases that traverse at least one modified element and which support the development of new features. The analyses concluded that even though D-MBTDD requires a revalidation of the old test suite, it requires less effort to support the development of new features because the test cases that cover the modified elements are already identified and present in the set of new test cases. Therefore, when the system evolves only the set of new test cases have to be transformed into executable test cases. Moreover, the test cases from the previous version that are still valid, i.e. the reusable test cases, are applied to the system after the implementation in order to verify that the changes did not affect the unmodified parts.

The total effort of Regenerate-All and D-MBTDD is impacted mostly by the number of generated test cases per iteration. According to the evolution of the test model, Regenerate-All tends to generate more test cases than D-MBTDD in order to cover the same criterion. Moreover, Regenerate-All has to identify the test cases that will support the development of new features from the set of generated test cases. Therefore, when the cost of transforming abstract tests into executable tests is greater than the cost of revalidating the old test suite, D-MBTDD requires less effort than Regenerate-All. Moreover, when the cost of Regenerate-All test identification is greater than the revalidation of D-MBTDD, D-MBTDD also requires less effort than Regenerate-All.

7.2 Limitations

D-MBTDD has some limitations and in some cases it is not the best option when compared to Regenerate-All:

- The benefit of D-MBTDD when compared to Regenerate-All is impacted by the test model design and how different the new version is from the previous one. If the test model is created for example, with only one transition that reaches the target state, when modifications are made in the test model in order to obtain a new version of the system, all or the majority of the test cases from the previous version became obsolete. The same scenario happens if the regression delta contains many modifications and therefore the new version has almost nothing in common with the previous one. Therefore, if all or the majority of the test cases become obsolete when a regression delta is applied, D-MBTDD and Regenerate-All will produce similar quantity of generated test cases.

- If the cost to revalidate the old test suite of D-MBTDD is greater than the cost of identifying the test cases that guide the development of Regenerate-All, and if

the cost to transform abstract test cases into executable is not significant, e.g. executable test cases are already generated with the MBT tool, the effort of Regenerate-All could be smaller than of D-MBTDD.

7.3 Answers to the Research Questions

Some research questions were defined and previously described in Section 1.4. Their answers are described bellow:

1. **RQ1: When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require less effort for test case creation, when compared to an approach in which there is no test artefact reuse?**

   No, because before creating test cases, D-MBTDD performs a revalidation of the previous test suite in order to identify the test cases that are still valid and which ones have to be created. Regenerate-All, instead, generate test cases without analysing any previous test cases. Therefore, D-MBTDD requires more effort for test case creation when compared to Regenerate-All.

2. **RQ2: When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require less effort for the identification of which test cases should guide the development of new features, when compared to an approach in which there is no test artefact reuse?**

   Yes. When the system and the test model evolve, test cases that traverse modified elements support the development of new features. With D-MBTDD these test cases are already identified and present in the set of new test cases. When no test cases from the previous version are reused, i.e. when Regenerate-All is used, an analysis in all generated test cases in order to identify those that traverse modified elements has to be performed before the implementation phase. Moreover, based on the controlled experiments, D-MBTDD is more focused on generating test cases that traverse modified elements when compared to Regenerate-All. Therefore, D-MBTDD requires less effort for the identification of which test cases should guide the development of new features, when compared to Regenerate-All.

3. **RQ3: When the system and consequently the test model evolves in an iterative and incremental development based on tests, does D-MBTDD require a total effort smaller than when compared to an approach in which there is no test artefact reuse?**

   It depends. The total effort of Regenerate-All and D-MBTDD is impacted mostly by the number of generated test cases per iteration. According to the evolution of the test model, in order to cover the same criterion, Regenerate-All tends to generate more test cases than D-MBTDD. Therefore, when the cost of transforming abstract tests into executable tests is greater than the cost of revalidating the old test

suite, D-MBTDD requires less effort than Regenerate-All. Moreover, Regenerate-All has to identify the test cases that will support the development of new features from the set of generated test cases. Therefore, when the cost of Regenerate-All test identification is greater than the revalidation of D-MBTDD, D-MBTDD also requires less effort than Regenerate-All.

7.4 Future Work

With the conclusions of this work, some points were identified that open up the possibility of future works. These points mainly allow for future improvements of D-MBTDD method, regarding the current limitations of the method and its validation.

- **Use different test models:** Any type of behavioural test model can be used with D-MBTDD, however during this proof of concept only finite state machine test models were considered. Even though finite state machines are widely used, the problem of state explosion can happen. Therefore, as a future work the use of other types of test models such as extended finite state machines (EFSM) should be analysed.

- **Execute the experiments with more examples:** The experiments had 9 core test models and 20 delta models in total, therefore more examples should be used in order to obtain more results. Furthermore, the examples should have more regression delta state machines with different types of modifications; more examples with many removals should be explored, and examples that explore situations in which obsolete test cases become valid.

- **Execute the experiment with different MBT tools:** The same experiment should be executed with different MBT tools in order to verify if the same conclusions can be made. Remembering that the MBT tool should use the test purpose to generate test cases.

- **Execute an experiment with D-MBTDD and Scrum:** An experiment should be performed in a case study in which a real system is implemented in a Scrum environment. Therefore, it should verify the benefits and limitations of D-MBTDD in a real development, and it should analyse the proposed process in which D-MBTDD is used together with Scrum.

- **Include traceability models:** Usually the context of a state machine's behaviour is defined in a class diagram. Therefore, traceability models should be created between class diagrams and test models, and between test models and test cases. Consequently, if modifications occur in a class diagram element the corresponding state machine element is updated, and vice-versa, by using the traceability model information.

Bibliography

[1] Agile For All. Into to agile. `http://www.agileforall.com/intro-to-agile` Accessed January 10,2016.

[2] Ana M. Ambrosio, Maria de Fátima M. Francisco, and Eliane Martins. Atifs: A testing toolset with software fault injection. In *Proceedings of York Computer Science Yellow Report 2003 - Workshop Softest: UK Testing Research*, York, UK, 2003.

[3] Kent Beck. *Extreme Programming Explained: Embrance Change*. Addison-Wesley Professional, us edition, 10 1999. ISBN 978-0201616415.

[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321146530.

[5] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Marting Fowler, James Granning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. `http://www.agilemanifesto.org` Acccessed January 10,2016.

[6] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Marting Fowler, James Granning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Twelve principles of agile software, 2001. `http://www.agilemanifesto.org/principles.html` Acccessed January 10,2016.

[7] Robert Binder. Model-based testing: Taking bdd/atdd to the next level, 2014. Slides from presentation at the Chicago Quality Assurance Association, February 25, 2014 `http://pt.slideshare.net/robertvbinder/taking-bddtothenextlevel?next_slideshow=1` Acccessed January 10,2016.

[8] Robert V. Binder. How to ice the testing backblog, 2013. `http://robertvbinder.com/how-to-ice-the-testing-backblob/` Accessed January 10,2016.

[9] Jan Olaf Blech, Dongyue Mou, and Daniel Ratiu. Reusing test-cases on different levels of abstraction in a model based development tool. In *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012.*, pages 13–27, 2012. doi: 10.4204/EPTCS.80.2.

[10] Jonas Boberg. Early fault detection with model-based testing. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 9–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: 10.1145/1411273.1411276.

[11] Konstantinos Bratanis, Dimitris Dranidis, and Anthony J. H. Simons. An extensible architecture for run-time monitoring of conversational web services. In *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*, MONA '10, pages 9–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0422-1. doi: 10.1145/1929566.1929568.

[12] Campgemine and Sogeti. Maximizing the value of good testing practice in an agile environment - delivering on time, in scope, on budget and at the right level of quality. White Paper, 2010. `http://www.capgemini.com/resource-file-access/resource/pdf/Maximizing_the_Value_of_Good_Testing_Practice_in_an_Agile_Environment.pdf` Acessado em 09 de março de 2014.

[13] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. Evaluating advantages of test driven development: A controlled experiment with professionals. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 364–371, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: 10.1145/1159733.1159788.

[14] Wallace Felipe Francisco Cardoso. Statemutest: an extended state model based test support tool (original title in portuguese: Statemutest: uma ferramenta de apoio ao teste baseado em modelos de estado estendidos). Master's thesis, Unicamp, SP, 2015.

[15] Álvaro Carrera, CarlosA. Iglesias, and Mercedes Garijo. Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Information Systems Frontiers*, 16(2):169–182, 2014. ISSN 1387-3326. doi: 10.1007/s10796-013-9438-5.

[16] Rafael B. Chiavegatto, Lidiane V. da Silva, Andréia Vieira, and William R. Malvezzani. Behaviour driven development with automated tests with jbehave and selenium (original title in portuguese: Desenvolvimento orientado a comportamento com testes automatizados utilizando jbehave e selenium. In *Regional Meeting of Information Computer and System*, Aug 2013.

[17] Rogério Atem de Carvalho, Fernando Luis de Carvalho e Silva, and Rodrigo Soares Manhães. Mapping business process modeling constructs to behavior driven development ubiquitous language. *CoRR*, abs/1006.4892, 2010.

[18] Rogério Atem de Carvalho, Rodrigo Soares Manhães, and Fernando Luis de Carvalho e Silva. Filling the gap between business process modeling and behavior driven development. *CoRR*, abs/1005.4975, 2010.

[19] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Pro-*

ceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-880-0. doi: 10.1145/1353673.1353681.

[20] Melanie Diepenbeck, Mathias Soeken, Daniel Grose, and Rolf Drechsler. Behavior driven development for circuit design and verification. In *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pages 9–16, Nov 2012.

[21] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. *Towards a Service-Based Internet: Third European Conference, ServiceWave 2010, Ghent, Belgium, December 13-15, 2010. Proceedings*, chapter Enabling Proactive Adaptation through Just-in-Time Testing of Conversational Services, pages 63–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17694-4.

[22] Vladimir Entin, Mathias Winder, Bo Zhang, and Stepahn Christmann. Introducing model-based testing in an industrial scrum project. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 43–49, June 2012.

[23] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I Malik, and Aamer Nadeem. 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, pages 44–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-850-3. doi: 10.1145/1291535.1291540.

[24] Fatima Mattiello Francisco, E. Villani, Eliane Martins, and Ana Maria Ambrosio. An experience on the technology transfer of cofi methodology to automotive domain. In *LADC2013 Sixth Latin-American Symposium on Dependable Computing*, Rio de Janeiro, RJ, BR, 2003.

[25] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342, 2004. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2003.09.011. Special Issue on Software Engineering, Applications, Practices and Tools from the {ACM} Symposium on Applied Computing 2003.

[26] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. Test case generation with test purpose specification by mscs, 1993.

[27] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001. ISSN 1049-331X. doi: 10.1145/367008.367020.

[28] Atul Gupta and Pankaj Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *Empirical Software Engineering and*

Measurement, 2007. ESEM 2007. First International Symposium on, pages 285–294, Sept 2007. doi: 10.1109/ESEM.2007.41.

[29] Reinhard Hametner, Dietmar Winkler, Thomas Ostreicher, Stefan Biffl, and Alois Zoitl. The adaptation of test-driven software processes to industrial automation engineering. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 921–927, July 2010.

[30] David S. Janzen and Hossein Saiedian. On the influence of test-driven development on software design. In *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*, pages 141–148, April 2006. doi: 10.1109/CSEET.2006.25.

[31] Bogdan Korel, Luay H. Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 214–223, 2002. doi: 10.1109/ICSM.2002.1167768.

[32] Hadar Ziv Leila Naslavsky and Debra J. Richardson. A model-based regression test selection technique. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 515–518, Sept 2009. doi: 10.1109/ICSM.2009.5306338.

[33] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based spl regression testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering*, PLEASE '12, pages 53–56, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1751-1.

[34] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. Delta-oriented software product line test models - the body comfort system case study. Technical Report 2012-07, TU Braunschweig, 2013.

[35] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *Proceedings of the 6th International Conference on Tests and Proofs*, TAP'12, pages 67–82, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30472-9. doi: 10.1007/978-3-642-30473-6_7.

[36] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based integration testing of large-scale systems. *J. Syst. Softw.*, 91: 63–84, May 2014. ISSN 0164-1212. doi: 10.1016/j.jss.2013.11.1096.

[37] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569, May 2003. doi: 10.1109/ICSE.2003.1201238.

[38] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Softw. Engg.*, 13(3):289–302, June 2008. ISSN 1382-3256. doi: 10.1007/s10664-008-9062-z.

[39] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. Mbsrt2: Model-based selective regression testing with traceability. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 89–98, April 2010.

[40] Arilo C. D. Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. Characterization of model-based software testing approaches. *Technical ReportES-713/07, PESC-COPPE/UFRJ. Available at http://www. cos. ufrj. br/uploadfiles/1188491168. pdf*, 2007.

[41] Dan North. Introducing BDD. *Better Software Magazine*, March 2006. URL `http://dannorth.net/introducing-bdd/`.

[42] Mazedur Rahman and Jerry Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*, pages 321–325, March 2015.

[43] Steven Rosaria and Harry Robinson. Applying models in your testing process. *Information and Software Technology*, 42(12):815 – 824, 2000. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/S0950-5849(00)00125-7.

[44] Gregg Rothermel and Mary J. Harrold. A framework for evaluating regression test selection techniques. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 201–210, May 1994. doi: 10.1109/ICSE.1994.296779.

[45] Gregg Rothermel and Mary J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, Aug 1996. ISSN 0098-5589. doi: 10.1109/32.536955.

[46] Benhard Rumpe. Agile test-based modeling. In *Proceedings of the International Conference on Software Engineering Research & Practice*, SERP'2006, USA, 2006. CSREA Press.

[47] Alireza Sadeghi and Seyed-Hassan Mirian-Hosseinabadi. Mbtdd: Model based test driven development. *International Journal of Software Engineering and Knowledge Engineering*, 22(08):1085–1102, 2012.

[48] Ken Schwaber. Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995.

[49] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130676349.

[50] Odd P. N. Slyngstad, Jingyue Li, Reidar Conradi, Harald Ronneberg, Einar Landre, and Harald Wesenberg. The impact of test driven development on the evolution of a reusable framework of components - an industrial case study. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, pages 214–223, Oct 2008.

[51] Carlos Solis and Wang Xiaofeng. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387, Aug 2011.

[52] Ian Sommerville. *Software Engineering (original title in Portuguese: Engenharia de software)*, chapter Agile Software Development (original title in Portuguese: Desenvolvimento rápido de software). Addison Wesley, São Paulo, 8 edition, 2007. ISBN 8588639289.

[53] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123725011, 9780080466484.

[54] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012. ISSN 0960-0833.

[55] Stephan Weißleder and Dehla Sokenou. Parteg - a model-based testing tool. *SoftwaretechnikTrends, 30(2)*, 2010.

[56] Stephan Weißleder, Dehla Sokenou, and Holger Schlingloff. Reusing state machines for automatic test generation in product lines. In Thomas Bauer, Hajo Eichler, Axel Rennoch, editor, *MoTiP '08: Model-Based Testing in Practice*. Fraunhofer IRB Verlag, 2008.

[57] Sebastian Wieczorek, Alin Stefanescu, Mathias Fritzsche, and Joacquim Schnitter. Enhancing test driven development with model based testing and performance analysis. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, pages 82–86, Aug 2008.

[58] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.

[59] Thaise Yano. *A multi-objective evolutive approach for test cases automation from state machines (original title in Portuguese: Uma abordagem evolutiva multiobjetivo para geração automática de casos de teste a partir de máquinas de estados)*. PhD thesis, Unicamp, SP, 2011.

[60] Thaise Yano, Eliane Martins, and Fabiano L. de Sousa. Most: A multi-objective search-based testing from efsm. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 164–173, March 2011.

[61] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012. ISSN 0960-0833. doi: 10.1002/stv.430.

Appendix A

Delta Information for Section 6.1.1

This Appendix describes the 15 delta models of the Delta Case Study described in Section 6.2.1.

- **Delta_1 of M1:** *DAddManPWCLS* applied to M1 results in a test model version for the component *Manual Power Window* with *Central Locking system*. In addition to the core model, the window movement of the manual power window is blocked/unblocked based on an activation/deactivation of the central locking system.
- **Delta_1 of M2:** *DAddAutoPWCLS* applied to M2 results in a test model version for the component *Automatic Power Window* with *Central Locking system*. The version specifies a similar behaviour to the previous one, but in this case the power window is automatic instead of manual.
- **Delta_1 of M3:** *DAddRCKSF* applied to M3 results in a test model version for the component *Remote Control Key Controller* with *Safety Function*. In addition to the core model, the remote control key controller re-locks the car after occurring a timeout representing the situation that the car was unintentionally unlocked.
- **Delta_2 of M3:** *DAddRCKCAP* applied to the previous M3 version results in a test model version for the component *Remote Control Key Controller* with *Control Automatic Power Window* and *Safety Function*. In addition to the previous version, the remote control key controller controls the upwards/downwards movement of the window via the remote key.
- **Delta_3 of M3:** *DAddRCKCAPSF* applied to the previous M3 version results in a second version of the test model version for the component *Remote Control Key Controller* with *Control Automatic Power Window* and *Safety Function*.
- **Delta_1 of M4:** *DAddCLSAL* applied to M4 results in a test model version for the component *Central Locking System* with *Automatic Locking*. In addition to the core model, the version specifies that the central locking system locks the doors without blocking the window when the car is driving.

- **Delta_2 of M4:** *DAddCLSRCK* applied to the previous M4 version results in a test model version for the component *Central Locking System* with *Remote Control Key* and *Automatic Locking*. In addition to the previous version, the central locking system gets activated/deactivated via a remote key.

- **Delta_1 of M5:** *DAddHMIAS* applied to M5 results in a test model version for the component *Human Interface Component* with *Alarm System*. In addition to the core model, the version specifies that the human machine interface enables the activation/deactivation of the alarm system via interaction with the driver.

- **Delta_2 of M5:** *DAddHMILEDAS* applied to the previous M5 version results in a test model version for the component *Human Interface Component* with *Alarm System* and *LED Alarm System*. In addition to the previous version, the human machine interface enables the confirmation of the silent alarm.

- **Delta_3 of M5:** *DAddHMILEDManPW* applied the previous M5 version results in a test model version for the component *Human Interface Component* with *Alarm System*, *LED Alarm System*, *Manual Power Window* and *LED Power Window*. In addition to the previous version, the human machine interface provides information about the release of the window buttons for the corresponding LEDs.

- **Delta_1 of M6:** *DAddLEDAutoPWCLS* applied to M6 results in a test model version for the component *LED Automatic Power Window* with *Automatic Power Window*, *LED Power Window* and *Central Locking System*. In addition to the core model, there is a new LED turning on/off if the automatic power window moves up while the central locking system is active.

- **Delta_1 of M7:** *DAddASCAS* applied to M7 results in a test model version for the component *Alarm System* with *Control Alarm System*. In addition to the core model, the alarm monitoring of the alarm system is additionally enabled/disabled by a remote key.

- **Delta_2 of M7:** *DAddASIM* applied to the previous M7 version results in a test model version for the component *Alarm System* with *Control Alarm System* and *Interior Monitoring*. In addition to the previous version, the alarm of the alarm system is triggered by the interior monitoring.

- **Delta_1 of M8:** *DAddEMHeating* applied to M8 results in a test model version for the component *Exterior Mirror* with *Heatable*. In addition to the core model, the exterior mirror is heatable if the outside temperature is too low.

- **Delta_2 of M8:** *DAddEMLEDEM* applied to the previous M8 version results in a test model version for the component *Exterior Mirror* with *Heatable* and *LED Exterior Mirror*. In addition to the previous version, the exterior mirror sends the information of its current position to the corresponding LEDs.

Appendix B

Complementary Tables for Chapter 6

This Appendix presents some tables with complementary information for Chapter6. Table B.1 presents which transitions were added or removed for each delta model.

As described in Subsection 6.4.2 and Subsection 6.5.2, there are some setup processes needed in order to perform the experiments with StateMutest and Condado Tool. For the StateMutest examples, Table B.2 represents the relationships among all the state machines, their tau value, their target transition, and their dependency transitions, and Table B.3 represents the relationship between the value of tau and the quantity of generated test cases. For the Condado exaples, Table B.4 represents the relationship between all the state machines and their final states.

Table B.1: Transitions added or removed of each delta model

ID	Transition Added	Transition Removed
M1_Delta_1	t14..t27	-
M2_Delta_1	t20..t35	-
M3_Delta_2	t5..t8	-
M3_Delta_1	t9..t15	t3,t4
M3_Delta_3	t16..t23	-
M4_Delta_1	t7..t10	-
M4_Delta_2	t5,t6	-
M5_Delta_1	t13..t16	-
M5_Delta_2	t17,t18	-
M5_Delta_3	t19..t21	-
M6_Delta_1	t9..t17	-
M7_Delta_1	t14..t16	-
M7_Delta_2	t17..t22	t10,t13
M8_Delta_1	t49..t84	-
M8_Delta_2	t85..t132	t1..12,t17,t18, t23,t24,t29,t30, t35,t36,t45..t48

Continued on next page

Table B.1 – continued from previous page

Model ID	Transition Added	Transition Removed
S04_Delta_1	t20..t23	-
S04_Delta_2	t24..t36	-
S04_Delta_3	t37..t42	-
S04_Delta_4	t43..t50	-
S04_Delta_5	t51..t54	-

Table B.2: StateMutest examples information

Model ID	Approach	Value of Tau	Target transition	Dependence transitions
M1	-	3	$t9$	$AllTransitions - t9$
M1_D1	Regenerate-All	2	$t9$	$AllTransitions - t9$
M1_D1	D-MBTDD	2	$t9$	$t14..t27$
M2	-	3	$t8$	$AllTransitions - t8$
M2_D1	Regenerate-All	3	$t8$	$AllTransitions - t8$
M2_D1	D-MBTDD	3	$t9$	$t20..t35$
M3	-	3	$t4$	$AllTransitions - t4$
M3_D2	Regenerate-All	2	$t8$	$AllTransitions - t8$
M3_D2	D-MBTDD	2	$t8$	$t5..t7$
M3_D2_D1	Regenerate-All	4	$t15$	$AllTransitions - t15$
M3_D2_D1	D-MBTDD	4	$t15$	$t2, t9..t14$
M3_D2_D1_D3	Regenerate-All	2	$t15$	$AllTransitions - t15$
M3_D2_D1_D3	D-MBTDD	2	$t15$	$t16..t23$
M4	-	3	$t4$	$AllTransitions - t4$
M4_D1	Regenerate-All	1	$t10$	$AllTransitions - t10$
M4_D1	D-MBTDD	1	$t10$	$t7..t10$
M4_D1_D2	Regenerate-All	2	$t4$	$AllTransitions - t4$
M4_D1_D2	D-MBTDD	2	$t4$	$t5, t6$
M5	-	2	$t12$	$AllTransitions - t12$
M5_D1	Regenerate-All	1	$t16$	$AllTransitions - t16$
M5_D1	D-MBTDD	1	$t10$	$t13..t15$
M5_D1_D2	Regenerate-All	1	$t18$	$AllTransitions - t18$
M5_D1_D2	D-MBTDD	1	$t18$	$t17$
M5_D1_D2_D3	Regenerate-All	2	$t21$	$AllTransitions - t21$
M5_D1_D2_D3	D-MBTDD	2	$t21$	$t19, t20$
M6	-	2	$t8$	$AllTransitions - t8$

Continued on next page

Table B.2 – continued from previous page

Model ID	Approach	Value of Tau	Target transition	Dependence transitions
M6_D1	Regenerate-All	4	$t8$	$AllTransitions - t8$
M6_D1	D-MBTDD	4	$t8$	$t9..t17$
M7	-	1	$t6$	$AllTransitions - t6$
M7_D1	Regenerate-All	3	$t6$	$AllTransitions - t6$
M7_D1	D-MBTDD	3	$t6$	$t14..t16$
M7_D1_D2	Regenerate-All	3	$t6$	$AllTransitions - t6$
M7_D1_D2	D-MBTDD	3	$t6$	$t7..t9, t11, t12, t16..t22$
M8	-	2	$t44$	$AllTransitions - t44$
M8_D1	Regenerate-All	2	$t44$	$AllTransitions - t44$
M8_D1	D-MBTDD	2	$t44$	$t49..t84$
M8_D1_D2	Regenerate-All	2	$t44$	$AllTransitions - t44$
M8_D1_D2	D-MBTDD	2	$t44$	$t13..t16, t19..t22,$ $t25..t28, t31..t34,$ $t37..t42, t49..t84,$ $t85..t132$
S04	-	4	$t11$	$AllTransitions - t11$
S04_D1	Regenerate-All	3	$t11$	$AllTransitions - t11$
S04_D1	D-MBTDD	3	$t11$	$t20..t23$
S04_D1_D2	Regenerate-All	3	$t11$	$AllTransitions - t11$
S04_D1_D2	D-MBTDD	3	$t11$	$t24..t36$
S04_D1_D2_D3	Regenerate-All	1	$t11$	$AllTransitions - t11$
S04_D1_D2_D3	D-MBTDD	1	$t11$	$t37..t42$
S04_D1_D2_D3_D4	Regenerate-All	1	$t11$	$AllTransitions - t11$
S04_D1_D2_D3_D4	D-MBTDD	1	$t11$	$t43..t50$
S04_D1_D2_D3_D4_D5	Regenerate-All	4	$t11$	$AllTransitions - t11$
S04_D1_D2_D3_D4_D5	D-MBTDD	4	$t11$	$t51..t54$

Table B.3: Quantity of generated test cases for tau values

Model ID	Tau 1	Tau 2	Tau 3	Tau 4	Tau 5
M1	7	5	8	6	7
M1_D1	8	10	9	8	10
M2	6	6	7	6	4
M2_D1	7	6	10	7	7
M3	1	2	2	2	2

Continued on next page

Table B.3 – continued from previous page

Model ID	Tau 1	Tau 2	Tau 3	Tau 4	Tau 5
M3_D2	2	3	1	2	1
M3_D2_D1	2	2	2	3	2
M3_D2_D1_D3	7	7	5	7	5
M4	1	1	1	1	1
M4_D1	4	2	2	1	1
M4_D1_D2	5	7	6	6	5
M5	2	8	4	3	2
M5_D1	6	5	2	4	2
M5_D1_D2	6	3	5	3	4
M5_D1_D2_D3	8	9	4	5	3
M6	1	2	1	1	1
M6_D1	7	7	6	10	9
M7	7	7	5	7	7
M7_D1	6	10	10	8	8
M7_D1_D2	9	9	9	9	8
M8	4	5	1	2	2
M8_D1	9	10	4	6	4
M8_D1_D2	4	10	6	7	5
S04	3	2	1	5	2
S04_D1	4	5	8	6	5
S04_D1_D2	3	4	7	3	2
S04_D1_D2_D3	10	8	5	2	5
S04_D1_D2_D3_D4	9	6	7	6	6
S04_D1_D2_D3_D4_D5	7	5	6	8	4

Table B.4: Condado examples information

Model ID	Approach	Final State
M1	-	*pwUp*
M1_D1	Regenerate-All	*pwUp*
M1_D1	D-MBTDD	*pwUp*
M2	-	*pwUp*
M2_D1	Regenerate-All	*pwUp*
M2_D1	D-MBTDD	*pwUp*
M3	-	*rckIdle*
M3_D2	Regenerate-All	*rckIdle*
M3_D2	D-MBTDD	*rckIdle*
M3_D2_D1	Regenerate-All	*rckIdle*

Continued on next page

Table B.4 – continued from previous page

Model ID	Approach	Final State
M3_D2_D1	D-MBTDD	*rckIdle*
M3_D2_D1_D3	Regenerate-All	*rckIdle*
M3_D2_D1_D3	D-MBTDD	*rckIdle*
M4	-	*clsUnlock*
M4_D1	Regenerate-All	*clsUnlock*
M4_D1	D-MBTDD	*clsUnlock*
M4_D1_D2	Regenerate-All	*clsUnlock*
M4_D1_D2	D-MBTDD	*clsUnlock*
M5	-	*controller*
M5_D1	Regenerate-All	*controller*
M5_D1	D-MBTDD	*controller*
M5_D1_D2	Regenerate-All	*controller*
M5_D1_D2	D-MBTDD	*controller*
M5_D1_D2_D3	Regenerate-All	*controller*
M5_D1_D2_D3	D-MBTDD	*controller*
M6	-	*ledAutoPwOff*
M6_D1	Regenerate-All	*ledAutoPwOff*
M6_D1	D-MBTDD	*ledAutoPwOff*
M7	-	*asActivatedOff*
M7_D1	Regenerate-All	*asActivatedOff*
M7_D1	D-MBTDD	*asActivatedOff*
M7_D1_D2	Regenerate-All	*asActivatedOff*
M7_D1_D2	D-MBTDD	*asActivatedOff*
M8	-	*emHorPending*
S04	-	*E0*
S04_D1	Regenerate-All	*E0*
S04_D1	D-MBTDD	*E0*
S04_D1_D2	Regenerate-All	*E0*
S04_D1_D2	D-MBTDD	*E0*