DOCTOR OF SCIENCES

Behavioural model-based testing of software product lines

Devroey, Xavier

Award date:
2017

Awarding institution:
University of Namur

Behavioural model-based testing of software product lines

Xavier Devroey

Jury

Dr. Benoit Baudry
Inria, Rennes, France

Prof. Myra B. Cohen
University of Nebraska-Lincoln, USA

Prof. Vincent Englebert
University of Namur, Belgium

Prof. Patrick Heymans
University of Namur, Belgium

Dr. Axel Legay
University of Namur, Belgium

Dr. Gilles Perrouin
University of Namur, Belgium

Prof. Pierre-Yves Schobbens
University of Namur, Belgium

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Supervised by Prof. Pierre-Yves Schobbens and Prof. Patrick Heymans

University of Namur

PReCISE Research Center

Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique
ISBN : 978-2-87037-997-4
Dépôt légal: D/2017/1881/35

"It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to."

— J.R.R. Tolkien, The Fellowship of the Ring

ABSTRACT

Software Product Line (SPL) engineering is a sub-discipline of software engineering based on the idea that products of the same family can be built by systematically reusing assets, some of them being common to all members whereas others are only shared by a subset of the family. Since the inception of SPL engineering, concerns about testing SPLs emerged. The large number of possible products that may be derived from a SPL induces an even larger set of test cases, which makes SPL testing a very challenging activity. Past research focused on how to reuse testing assets from one product to another. In order to find as much bugs as possible without testing all the products, sampling techniques (like Combinatorial Interaction Testing (CIT) for SPLs) produce a representative subset of products to test in priority. They work in a family-based fashion by reasoning on a variability model of the product line. However they do not take other aspects, like behaviour of the products, into account.

In this thesis, we present a testing framework to perform family-based SPL behavioural model-based testing. We rely on Featured Transition Systems (FTSs), a compact formalism to represent the behaviour of a SPL, to perform various testing activities. Test case selection and prioritization are driven by the behavioural aspect of the SPL and may be done using three kinds of selection criteria: structural coverage criteria are based on the structure of the FTS; dissimilarity criteria seek to increase diversity amongst the selected test cases; and statistical criteria consider the usage of the system to drive the selection. Mutation analysis is performed using our Featured Mutants Model (FMM) and includes equivalent mutants detection at the model level. The result is a set of relevant test cases selected at the family level that may be used to define products to test in priority.

These approaches have been implemented in an open-source Variability Intensive Behavioural teSting (VIBeS) framework and evaluated on various models of different sizes, representing embedded systems and web-applications. Results demonstrate the applicability of FTSs to select and prioritize test cases and to perform mutation analysis, and confirm the relevance of combining variability models and behavioural models to enhance SPL model-based and mutation testing.

Keywords: model-based testing, software product line engineering, software testing

RÉSUMÉ

Le génie des Lignes de Produits Logiciels (LPL) est une sous-discipline du génie logiciel basée sur l'idée que les produits d'une même famille peuvent être construits de manière systématique en réutilisant des briques de base. Certaines sont communes à tous les produits de la famille et certaines sont spécifiques à un sous-ensemble de ces produits. Depuis la création de la discipline, le génie des LPLs s'intéresse à la question du test d'une LPL. Le grand nombre de produits possibles pouvant être dérivés d'une LPL amène un nombre encore plus important de cas de test. Les recherches précédentes se sont focalisées sur la réutilisation (d'une partie de) ces cas de test d'un produit à l'autre. Afin de ne pas devoir tester tous les produits possibles, des techniques d'échantillonnage produisent un sous-ensemble représentatif de produits à tester en priorité. Ces techniques raisonnent au niveau de la famille de produits en se basant sur le modèle de variabilité de la LPL. Cependant, elles ne prennent pas en compte d'autres aspects. Par exemple, le comportement des produits.

Dans cette thèse, nous présentons une infrastructure pour effectuer du test de LPL dirigé par les modèles, en raisonnant au niveau de la famille de produits. Nous utilisons des *Featured Transition Systems (FTSs)*, un formalisme compact pour représenter le comportement d'une LPL dans son ensemble, afin d'effectuer les différentes activités de test. La sélection et la priorisation de cas de test se font sur base du comportement de la LPL et sont dirigées par trois types de critères de couverture : les critères basés sur la structure du FTS, les critères basés sur la dissimilarité des cas de test et les critères statistiques basés sur l'utilisation effective des produits. Nous effectuons également une analyse des cas en test en utilisant la mutation et notre *Featured Mutants Model (FMM)*. Cette analyse comprend une détection des mutants équivalents. Le résultat d'un processus de sélection est un ensemble de cas de test, définis pour la famille de produits et pouvant servir à définir les produits à tester en priorité.

L'approche a été implémentée dans un *Variability Intensive Behavioural teSting (VIBeS) framework* open-source et évaluée sur différents cas d'étude de différentes tailles, représentant des systèmes embarqués et des applications Web. Les résultats démontrent l'applicabilité de l'approche pour sélectionner des cas de test et effectuer une analyse basée sur la mutation. Ils confirment la pertinence de combiner modèles de comportement et de variabilité pour améliorer le test de LPL dirigé par les modèles et la mutation.

Mots clés : test dirigé par les modèles, lignes de produits logiciels, test logiciel

ACKNOWLEDGEMENTS

The contributions presented in this thesis would not have been possible without the assistance and support of many people. First, I would like to thank my supervisors, Prof. Pierre-Yves Schobbens and Prof. Patrick Heymans for their advice and help during the past six years. They have very different backgrounds and work in different research fields, which was very enriching for a young Ph.D. student. Being part of their research group was an incredible opportunity that gave me the chance to do lot of *interesting things* (*sic*) and meet a lot of different people. I would like to offer my special thanks to Dr. Gilles Perrouin for his invaluable help all the way long. He is an incredible researcher and an endless source of ideas and advice on research and academic life as a Ph.D. student. Thank you Gilles for the guidance and numerous scientific (and non scientific) discussions!

I would also like to thank collaborators whose meeting contributed to develop this thesis: Dr. Mike Papadakis who introduced us to mutation testing and to whom, after this last months spent writing, I will be able to answer *yes* to the question *everything good?*; Dr. Axel Legay for his help on the formal aspects of FTSs, statistical testing, and automata language equivalence; Dr. Maxime Cordy for his expertise on FTSs and model checking; and Dr. Benoit Baudry for accepting being part of the thesis support committee and for the valuable comments after the mid-term test. All my thanks to the members of the jury for the time spent reading this manuscript and for the feedbacks: Dr. Benoit Baudry, Prof. Myra B. Cohen, Prof. Vincent Englebert, Dr. Axel Legay, and Dr. Gilles Perrouin.

Special thanks to my colleagues Benoît V. for the (numerous) chats and debates contributing to make every coffee break unique[1], Aude N. for the everlasting good mood and being a geek amongst geeks, Catherine for the pep talks and the enthusiasm each time we come up with a new idea, Julian *a.k.a.* best sysadmin ever, Anthony S. for the (long-overdue) bunker sheltering, Moussa who accepted to step into the *compiler project renewal*, Hajer and Nesrine, Cédric and Pierre-Antoine, Saria, Abdel, Fabian and Nicolas, Maxime and Loup, Julie, Tony, Adrien, James, Benoît F., Eleonora, and all the other players of the Friday lunch break, for the all good times spent together and contribution to make the Computer Science Faculty a great place to work.

Thank you Mathieu, Jeremy, Thomas, Axel, and Alexandre for accepting to collaborate with us and the good work that contributed to the research presented in

[1] … and for the logo!

this thesis. Thank you Maxime, Geoffroy, Guillaume, and all the students I have had the pleasure to work with, for the fun and not complaining too much about the (sometimes crazy) teaching assistant I may have been during the past 6 years.

To my longtime friends: thank you Steven, Aude, Marie, François, Laurent, Stéfanie, Aurore, Ariane, and Vincent for your friendship and support during all those years, and the many years to come.

Finally, I would like to express my utmost gratitude and love to my parents who always encouraged me to follow my dreams and supported me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Since the first computer program written by Ada Lovelace in the middle of the 19th century, variability allows one to reuse pieces of code in different contexts. For instance, variations in the input of a program triggers different behaviours and produces different outputs. In the same way, procedural abstraction allows to reuse procedures and functions in different contexts. This goes even further with the development of object oriented and other programming paradigms. In 1968, during the first NATO software engineering conference [222], Malcolm Douglas McIlroy, one of the pioneer of component-based software engineering, gave a talk entitled "Mass Produced Software Components" [209] where he advocates the development of component families: "*Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and timespace performance.*" Almost fifty years later, lot of systems are **variability intensive**. They are configurable or use a plugin-based architecture to be customizable in order to adapt to specific needs without requiring further development.

Those ideas are not new. In the early 20th century, Henry Ford achieved mass production of the *Model T* car by standardizing its components (to make then interchangeable) and reorganising the manufacturing process around an assembly line with dedicated tools and equipments, to allow unskilled workers to contribute to the building process. For years, the manufacturing industry achieved economies of scope based on this idea that a product of a certain family (*e.g.*, cars) may be built by systematically reusing assets, with some of them common to all family members (*e.g.*, wheels or bodywork) and others only shared by a subset of the family (*e.g.*, automatic transmission, manual transmission, or leather seats). The **Software Product Line (SPL)** paradigm [252] applies this idea to software products. In SPL engineering, we usually associate assets with so-called **features** and we regard a product as a combination of features. Features can be designed and specified using various modelling languages, while the set of legal combinations of features (that is, the set of valid products) is captured by a Feature Model (FM) [159].

As in single-system development, the engineer has to improve confidence in the different products of an SPL by using appropriate quality assurance techniques. Two popular approaches are **model checking** and **testing**. Model checking [59] performs systematic analyses on behavioural models in order to assess the satisfaction of the intended temporal and qualitative requirements and properties. As a complement to model-checking, testing [206] determines whether or not actual executions of the

system behave as expected.

Context and problem statement

In this SPL context, the large number of possible combinations of features makes product-based analysis (*i.e.,* testing or model checking every possible software product) intractable. For instance, the Linux kernel for x86 architectures (v.2.6.28.6) has 5,426 features (of which 4,744 may be selected by the end users) [16,284], which gives billions of possible products. As a point of comparison, an SPL with 33 independent and optional features is enough to build a unique product for every human on earth. An SPL with 320 independent and optional features has more products than the estimated number of atoms in the universe. Even small product lines requires a lot of effort to achieve a complete product-based analysis: in their recent work, Halin *et al.* [126, 127] report their effort to perform a complete product-based testing of JHipster, an open-source generator for Web applications with 48 features. It took 8 person/month to set up the testing infrastructure, 5.2 Terrabytes diskspace, and 4,376 hours (around 182 days) computation time to test all 26,256 products. Considering development practices, like continuous integration and delivery, fast release, *etc.*, this brute force approach cannot be applied in many cases.

To cope with the large number of products in a SPL, the model checking community devised, over the years, several efficient **family-based** analysis [301]: *i.e.,* analysis performed on the reusable assets of the SPL (called domain artefacts) instead of products, and using the feature model to consider valid combinations of those assets during the analysis. For instance, model checking of SPL specifications, expressed using a transition system with variability information [63, 105], allows to ensure that a given property holds for every product of the product line. Many other formal approaches, meant to detect undesired feature interactions (*i.e.,* undesired behaviour emerging when two or more features are involved in the same product), have been developed [52, 225] and successfully used to validate abstract models of SPLs [142, 282, 320]. Scalable application of formal methods to source code remains an open problem [16].

During the last decade, the research community has showed a growing interest in SPL testing [138]. SPL testing aims at validating an SPL by executing a *good enough* finite set of test cases (on a set of products). As testing all the products of a product line is infeasible, the main challenge is to select a representative **subset of all the products** and execute test cases over this subset. First work on behavioural SPL testing are mainly focused on how to reuse test cases from one product to another (by deriving them automatically from domain artefacts for instance) [138, 234]. Despite those advances, the development of practical SPL testing techniques is still in an immature stage [92, 98]. In particular, one question remains:

> How to **select** a representative subset of **products** and with which **test cases**?

Recent work tackles this problem by using **sampling** over the feature model. Combinatorial Interaction Testing (CIT) techniques have been adapted to the SPL

context [69,140,156,198,249,250] to ensure that combinations of features are present in at least one product to test: *e.g.,* pairwise sampling ensures that all valid pairs of features are present in at least one product. Other approaches use a dissimilarity heuristic to sample, based on a time and testing budget, a set of products as dissimilar as possible in terms of features [8,135]. Or use other information from the feature model (*e.g.,* features cost) [99,137,272,274]. The large majority of CIT approaches are model-based and use the feature model as main artefact to perform a product sampling, answering only to the former part of the question.

Contributions

In this thesis, we consider the **behaviour** of the SPL in **addition** of the feature model as the main driver of the test selection process. We present a **model-based** testing framework to select test cases and products, based on a **behavioural** model of the SPL. We rely on the advances made by the model checking community to describe SPL behaviour in a Featured Transition System (FTS) [63], a compact formalism used to represent the behaviour as a transition system where transitions are tagged with feature expressions specifying which products may fire the transition. As for other SPL approaches [163,224,294], FTSs are executed in a family-based fashion: *i.e.,* executions of parts common to several products are factorized, thanks to a variability aware execution engine. We define the notion of **abstract test case** as a sequence of actions to perform on the system and show how to, based on a set of abstract test cases, we sample relevant products. We devise several abstract test case selection strategies (with the corresponding algorithms) and define a compact formalism to improve mutation analysis. The different selection strategies and mutation analysis are implemented in our Variability Intensive Behavioural teSting (VIBeS) framework, and evaluated on several case studies, some of them part of the research literature and some of them specific to this thesis. Finally, we show how the abstract test cases are concretized using an Abstract test case Concretizer (AbsCon), a plugin for the QSpin Tailored Automated System Test Environment (QTaste).

Abstract test case over an FTS: Contrary to existing sampling based approaches, we do not seek to cover combinations of features but rather the behaviour of the product line. In this thesis, we adopt a model-based approach to select test cases from a FTS, representing this behaviour. Our first contribution is the definition of **abstract test case** and how it can be used to **sample relevant products** to test.

Abstract test case selection based on the FTS structure: Our first abstract test case selection strategy is based on the structure of the FTS. We redefine the **states**, **actions**, **transitions**, **transition-pairs**, and **paths coverage** for FTSs and provide a first all-states **selection algorithm**. We also define the notions of **abstract test case** and **test suite minimality**. Finally, we provide a **prioritization** strategy to order products to test according to the test cases they can execute.

Abstract test case selection based on a dissimilarity heuristic: The second abstract test case selection strategy uses a dissimilarity heuristic, which aims to maximise the fault detection rate by increasing diversity among abstract test cases [56, 131]. In this thesis, we present a **configurable dissimilar abstract test case selection algorithm** that uses random abstract test case selection and a **distance** function to guide the selection. The distance is defined on the actions of the abstract test cases that may optionally be combined, using a binary operator, with a distance (the Jaccard index product dissimilarity) defined over the set of products able to execute those abstract test cases. To characterise the actions used in an abstract test case, we consider set-based distances (Hamming, Jaccard, dice, and anti-dice distances) and a sequence-based distance (Levenshtein or edit distance).

Abstract test case selection based on usages: The last abstract test case selection strategy described in this thesis is based on the usage of the product line. This work is inspired from statistical testing [316], which selects abstract test cases from a usage model represented by a Discrete-Time Markov Chain (DTMC). The idea is to **select** abstract test cases from the usage model (and the FTS, since the usage model is agnostic of the variability constraints of the SPL), based on their **probability to happen**. The set of selected abstract test cases is also used to **prioritize products** to test, based on their behavioural usages.

Compact mutants model: Mutation analysis is a popular technique to assess the adequacy of a test suite. The idea is to inject artificial faults (using mutation operators) in the system under test and to execute the test suite against each one of the faulty systems (*i.e.,* mutants). This analysis may take time for a large number of mutants. In this thesis, we propose a product line approach of model-based mutation testing. Since mutants are small variations of the system, mutants are seen as members (*i.e.,* products) of a **mutants family** (*i.e.,* a product line of mutants). We define a compact mutants model using variability mechanisms: the Featured Mutants Model (FMM). We provide **algorithms** and **mutation operators** to build a FMM and describe its mechanism allowing to **execute** a test case on all mutants in one single execution. Finally, we show how FMM is used as a **compact representation** for first and higher orders mutants.

Equivalent mutant detection using automata language equivalence: Equivalent mutants are mutants whose behaviour is identical to the original system. As they cannot be distinguished by any test case, they do not bring new value to the analysis. This thesis enhance the model-based mutation testing research field by addressing one of its main challenge: the Equivalent Mutants Problem (EMP). We express EMP as a classical problem in automata theory, Automata Language Equivalence (ALE), and see how **language equivalence** may be used to **detect equivalent mutants** in strong and weak mutation scenarios. As baseline, we also provide two **randomized simulation** techniques to detect equivalent mutants: Random Simulation (RS) and Biased Simulation (BS).

VIBeS implementation: VIBeS is implemented in Java as an open-source multi-module Maven project. It allows one to **define** FTSs and perform the **various testing activities** described in this thesis using a front end Application Programming Interface (API). The source code is publicly available on GitHub (https://github.com/xdevroey/vibes) and the Maven artefacts have been deployed in the Maven central repository, making them available to other Maven users.

SPL case studies: We manually defined one new case study: the **card payment terminal SPL**, based on standard documentation [97]. We also semi-automatically reverse engineer five models of two Web applications, based on several months of log entries: one model for **Claroline**, a course management system, and four models for **WordPress**, an open-source Content Management System (CMS). The feature models, FTSs, and usage models are publicly available online (https://projects.info.unamur.be/vibes/) and may be used or adapted by the research community.

Structure of the thesis

The remainder of this manuscript is divided as follows: Part I gives the background used in the following parts. Chapter 1 introduces software product lines, feature models, and featured transition systems. Chapter2 presents the state of the art in product line testing. Part II contains the main contributions of this thesis. It presents our model-driven behavioural testing framework in Chapter 3 and the case studies used to illustrate and assess the different approaches in Chapter 4. Chapter 5 describes the different abstract test case selection techniques: structural coverage driven, dissimilarity driven, and usage-base driven. Chapter 6 presents mutation analysis using FMMs and how to detect equivalent mutants using automata language equivalence and random simulation. Finally, chapter 7 presents the empirical assessments of the elements from Chapters 5 and 6. Implementations are described in Part III. Chapter 8 presents VIBeS and Chapter 9 shows how abstract test cases may be concretized using AbsCon. Finally, Part IV and Chapter 10 conclude this thesis and present research perspectives.

Publications

The content of this thesis is based upon, reuses, and extends the following peer-reviewed publications of the author:

Journal

[84] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Statistical prioritization for software product line testing: an experience report. **Software & Systems Modeling**, 16(1):153–171, feb 2017

Conferences

[80] Xavier Devroey, Maxime Cordy, Gilles Perrouin, Eun-Young Kang, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Benoit Baudry. A Vision for Behavioural Model-Driven Validation of Software Product Lines. In Margaria T., Steffen B., and Merten M., editors, **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I**, volume 7609 of **LNCS**, pages 208–222, Heraklion, Crete, Greece, 2012. Springer

[83] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, and Pierre-Yves Schobbens. A variability perspective of mutation analysis. In **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014**, pages 841–844, Hong Kong, 2014. ACM Press

[86] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-yves Schobbens, and Patrick Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In Tiziana Margaria and Bernhard Steffen, editors, **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Proceedings, Part I**, volume 8802 of **LNCS**, pages 336–350, Corfu, Greece, 2014. Springer

[89] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In **Proceedings of the 38th International Conference on Software Engineering - ICSE '16**, pages 655–666, Austin, Texas, USA, may 2016. ACM Press

[182] Axel Legay, Gilles Perrouin, Xavier Devroey, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. On Featured Transition Systems. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, **SOFSEM 2017: Theory and Practice of Computer Science**, volume 10139 of **LNCS**, pages 453–463, Limerick, Ireland, 2017. Springer (invited paper)

[90] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation. In **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**, pages 424–429, Tokyo, Japan, 2017. IEEE

Workshops

[85] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards statistical prioritization for software product lines testing. In Andrzej Wasowski and Thorsten Weyer, editors, **Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems - VaMoS '14**, pages 1–7, Nice, France, 2013. ACM Press

[91] Xavier Devroey, Gilles Perrouin, and Pierre-Yves Schobbens. Abstract test case generation for behavioural testing of software product lines. In **Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14**, volume 2, pages 86–93, Florence, Italy, 2014. ACM Press

[87] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations. In **Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '15**, pages 59–66, Hildesheim, Germany, 2015. ACM Press

[81] Xavier Devroey, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. State machine flattening, a mapping study and tools assessment. In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 1–8, Graz, Austria, apr 2015. IEEE

[88] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based Similarity-driven Behavioural SPL Testing. In **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16**, pages 89–96, Salvador, Brazil, jan 2016. ACM Press

[127] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. Yo variability! JHipster: A Playground for Web-Apps Analyses. In **Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems - VAMOS '17**, pages 44–51, Eindhoven, Netherlands, feb 2017. ACM Press

Part I

Background

CHAPTER 1

SOFTWARE PRODUCT LINES

Product line engineering is a very common practice in the manufacturing industry. It allows the *large-scale production of goods and services tailored to individual customers' needs*, called **mass customisation** [252]. Success stories come from the automotive industry where product line have been invented in the first half of the 20th century. Under the market pressure, the development of product line engineering allows nowadays a customer to configure his car and have it delivered within three months. Product line and mass customisation are also used in other domains to personalise services for instance: with the latest developments of medical research (*e.g.,* imaging, DNA analysis, *etc.*) treatments plans are now customised to target the exact disease, reduce side effects, and accelerate recovery. To be tractable, a product line relies on **platforms**, *i.e., any base of technologies on which other technologies or processes are built* [252].

Software Product Line (SPL) engineering is the application of those ideas to software development: it is *the paradigm to develop software applications using platforms and mass customisation* [252]. One software, *i.e.,* a **product**, is built by combining **commonalities**, common to all the products of the product line, and **variabilities**, specific to only some products.

This chapter presents the main ideas and concepts of SPL engineering used in this thesis. Section 1.1 presents the development of SPLs and section 1.2 presents how variability is captured and managed in this process. Section 1.3 discusses different modelisation approaches to describe SPL behaviour, including featured transition systems, the formalism we use in the remainder of this thesis.

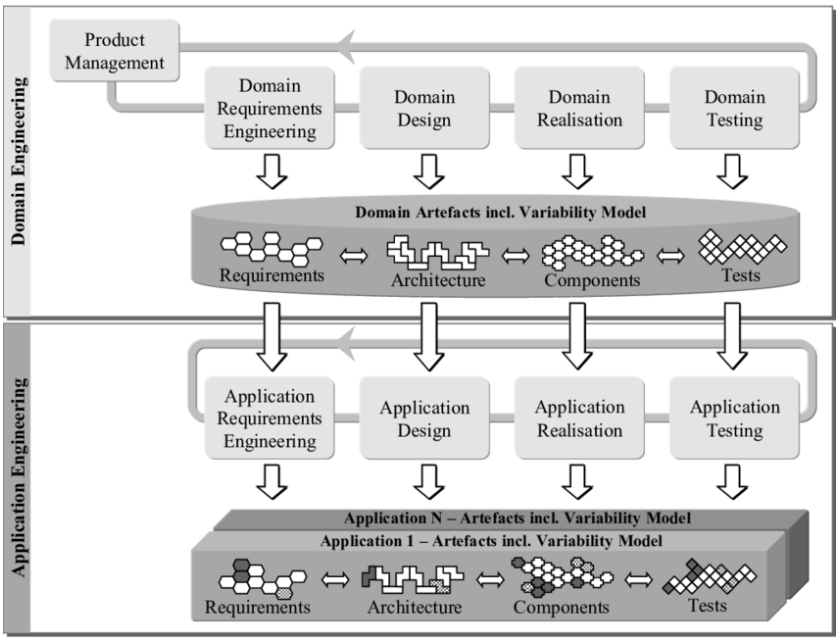


Figure 1.1: The SPL development processes [252]

1.1 Software product line engineering

SPL engineering consists in two development processes presented in Figure 1.1 [252]: **domain engineering**, where *commonalities and variabilities of the product line are defined and realised*, and **application engineering**, where *applications of the product line are built by reusing domain artefacts and exploiting the product line variability*. Each one of those two processes consists of several activities (including testing) and produces different artefacts. Domain artefacts are reused during application engineering in order to built one particular product.

This thesis focuses on behavioural testing at the domain level. We seek to select **relevant behaviours**, as well as **relevant products** able to execute those behaviours in order to ensure that the SPL fulfils its specification. To do so, we use a **model-based** approach, with a model abstracting the behaviour of all the SPL (including product-specific and SPL-common behaviours).

Commonalities and variabilities are captured trough the notion of **feature**. A feature is a *characteristic or end-user-visible behaviour of a software system* [16]. It is a first-class abstraction that helps to reason about the SPL [64]. In practice, features are mapped to domain artefacts (or parts of domain artefacts) that are combined to form products. The derivation of one product consists in selecting a valid set of features, called **configuration**, with the intended characteristics of the product, and combining the domain artefacts linked to those features. All the valid combinations of features are described in a **feature model**.

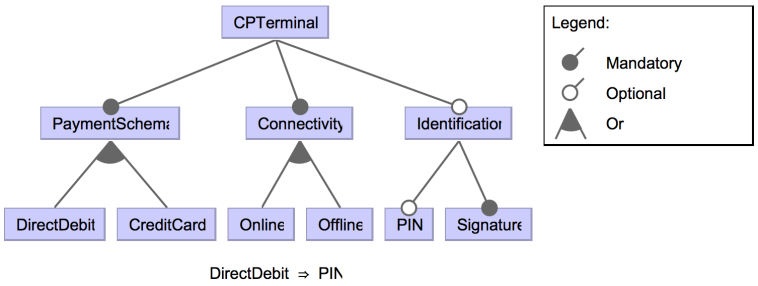


Figure 1.2: Card payment terminal simplified feature model

1.2 Feature models

Feature Models (FMs) describe all the valid combinations of features for SPLs. They have been introduced by Kang *et al.* in the Feature Oriented Domain Analysis (FODA) method [159] using a graphical representation. Lot of other graphical [14,40,185,305] and textual [62,211] representations have been proposed and formalised [32,276] since. In this thesis, we focus on graphical boolean FMs and their equivalence in Text-based Variability Language (TVL) [62].

A feature model is organised in a tree-like structure (formally a directed acyclic graph). The root feature, always present in all products, is decomposed in sub-features using *and* (all sub-features are selected if the parent feature is selected), *or* (at-least one sub-feature is selected if the parent feature is selected), and *xor* (exactly one sub-feature is selected if the parent feature is selected) relations. For instance, Figure 1.2 presents a (simplified) feature model for a card payment terminal product line. Root feature (*CPTerminal*) is decomposed into two mandatory features (*PaymentSchema* and *Connectivity*) and one optional feature (*Identification*). *PaymentSchema* (specifying if the terminal supports credit or debit cards) and *Connectivity* (specifying if the terminal is connected or not) features are decomposed into sub-features using a *or* relation. Additionally, feature models may have cross-tree constraints expressed as boolean expressions over the features, *e.g., DirectDebit* ⇒ *PIN*, specifying that debit card payment requires PIN authentication.

Listing 1.1 presents the TVL version of Figure 1.2. In this case, features are decomposed in sub-features using `allOf` (*and*), `someOf` (*or*), or `oneOf` (*xor*) relations. Additional cross-tree constraints (on line 14) may be specified and optional features are indicated using `opt` keyword.

The semantics of a feature model d, denoted $[\![d]\!]$, corresponds to all the valid products allowed by the feature model. In the remainder of this thesis, we consider only **boolean feature models**, *i.e.,* feature models where features have boolean values: *true* if the feature is selected and *false* otherwise. Since a boolean feature model may be transformed into a Conjunctive Normal Form (CNF) formula [32], denoted $CNF(d)$, its semantics corresponds to all the feature assignments that satisfies this formula. This may be computed using Boolean Satisfiability Problem (SAT) or Binary Decision Diagram (BDD) solvers [193,212].

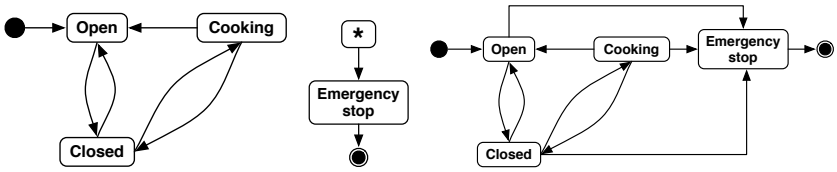
Listing 1.1: Card payment terminal simplified feature model in TVL format

```
1  root CPTerminal {
2    group allOf {
3      PaymentSchema,
4      Connectivity,
5      opt Identification
6    }
7  }
8
9  PaymentSchema {
10   group someOf {
11     DirectDebit,
12     CreditCard
13   }
14   DirectDebit -> PIN;
15 }
16
17 Connectivity group someOf {
18   Online,
19   Offline
20 }
21
22 Identification group allof {
23   opt PIN,
24   Signature
25 }
```

1.3 Behavioural modellisations of software product lines

Modern software systems are complex to build and maintain, counting thousands of lines of codes to achieve various purposes under different kinds of constraints (*e.g.*, time response, security, availability, *etc.*). To manage this complexity, software engineers use modeling [147]. This allows to abstract the system by focusing on some of its aspects. For instance, feature models focus on the features of a product line and their constraints. In SPL engineering, complexity worsen due to the variability inherent to product line. Models have to take features into account to represent this variability.

This section presents different modelling approaches to represent the behaviour of a software product line. Those approaches may be decomposed into two categories [16]: **annotation**-based approaches and **composition**-based approaches. Annotation-based approaches annotate a common model to indicate which parts of the model belongs to which feature(s). During product derivation (*i.e.*, selection of the features of a product), parts of the model marked with unselected features are removed to give a model of the product. Composition-based approaches models features as a set of composable model parts. During product derivation, those parts are combined to form the model of the product.



(a) Base model (b) Feature aspect (c) Product model

Figure 1.3: SPL behavioural composition-based modelling example [259]

1.3.1 Composition-based modelling approaches

Most composition-based modelling approaches are based on **aspect oriented modelling** [121, 259]. A base model representing the behaviour common to al the products of the product line is modified by weaving aspects specific to one or more features. For instance, Figure 1.3(a) presents the state machine base model of a oven in a smart home [259]. When a product with the feature *emergency stop* is derived, the aspect from Figure 1.3(b) is weaved in the base model to give the state machine of the product in Figure 1.3(c). To decide where to be applied, the aspect must specify one or more pointcuts using a pointcut expression: '*' in Figure 1.3(b).

Various formalisms based on transition systems [23, 73, 106, 170, 187–189] and Input-Output automata [179, 180] exist. Other composition-based modelling approaches are extensions of existing modelling languages where feature aspects may be weaved at some specific points of the system [10, 17, 33, 53, 116, 216, 223, 254, 275, 290]. Finally, UML models like state machines [281, 283] and sequence diagrams [119] have been extended to support composition-based modelling.

1.3.2 Annotation-based modelling approaches

There exists several annotation-based modelling approaches to represent the behaviour of a software product line. Most of them consider a based model annotated with variability information indicating which products of the product line it belongs to. State-based models include for instance Petri nets [141, 221, 256], modal I/O automata [178], Modal Transition Systems (MTSs) [22, 24, 25, 103, 105, 297, 299], FTSs [61, 63, 65, 66], and finite state machines [264, 264]. Other formalisms include PL-CSS (a process algebra) [122] and higher-level formalisms like UML activity diagrams [141].

Researches on annotation-based models for software product line verification have been conducted for years and are still developed by the model-checking community. Amongst all the existing notations, in this thesis, we focus on those derived from Labelled Transition System (LTS), a simple and yet expressive formalism to model the behaviour of a system:

Definition 1 (Labelled Transition System (LTS) [27]) *A LTS is a tuple* $(S, Act, trans, i)$*, where:*

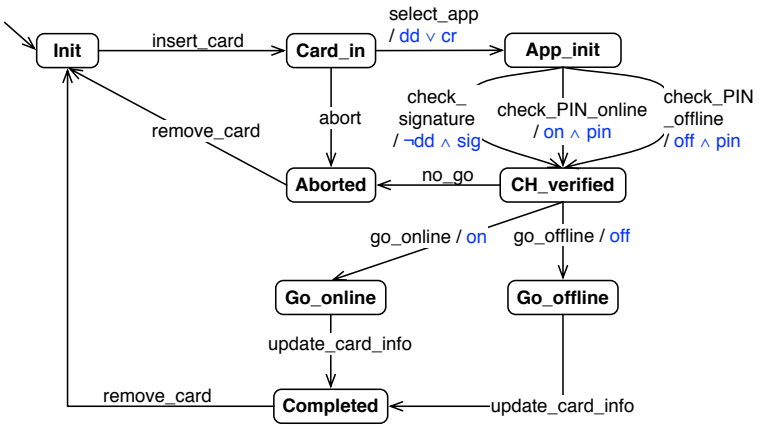


Figure 1.4: Card payment terminal simplified featured transition system

- *S is a set of* ***states****;*
- *Act is a set of* ***actions****;*
- *trans* $\subseteq S \times Act \times S$ *is a* ***transition*** *relation (with* $(s_1, \alpha, s_2) \in trans$*, denoted* $s_1 \xrightarrow{\alpha} s_2$*);*
- *and* $i \in S$ *is the* ***initial state****.*

This choice is motivated by the existence of powerful modelling languages [25, 103, 297], algorithms [63, 66], and tools [22, 72] developed by the model-checking community. Also, lot of other higher-level formalisms may be translated to LTS [81] allowing to make our results easily applicable to other modelling languages.

1.3.3 Featured transition system

Classen *et al.* [63] define Featured Transition System (FTS) to represent the behaviour of a product line. An FTS is an LTS where transitions have been annotated with feature expressions defining which products of the feature model is able to execute the transition. For instance, the FTS in Figure 1.4 presents the behaviour of all the products of the card payment terminal product line. The feature expressions on the transitions references the features of the feature model in Figure 1.2. First, the terminal is initialised (*Init*) and ready to proceed card payments. When a card is inserted, the terminal selects an appropriate payment method and launches the corresponding application (*App_init*). This can only be done if the terminal processes debit or credit cards, denoted by the feature expression $dd \vee cr$. Next step is to identify the card holder, either by using a Personal Identification Number (PIN) code, which can be done online ($on \wedge pin$) of offline ($off \wedge pin$), or a signature if the terminal does not process debit cards ($\neg dd \wedge sig$). If the identification succeeds (*CH_Verified*), the terminal process the transaction online (*Go_online*) or offline (*Go_offline*), updates the information on the card, and completes the transaction (*Completed*). Formally, FTSs are defined as follows:

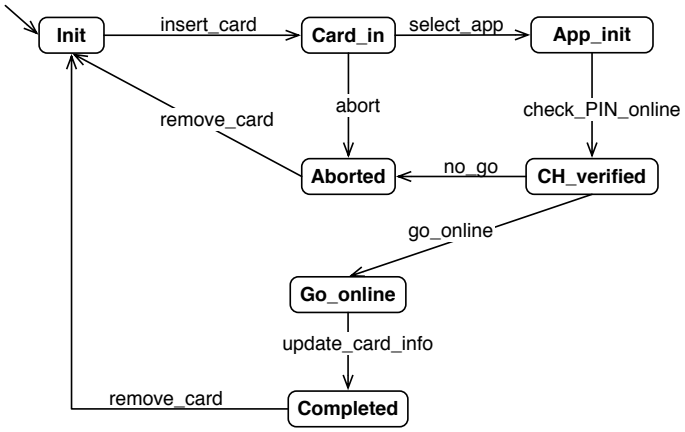


Figure 1.5: Card payment terminal product LTS

Definition 2 (Featured Transition System (FTS) [63]) *A FTS is a tuple* $(S, Act, trans, i, d, \gamma)$*, where:*

- S*,* Act*,* $trans$*,* i *are defined according to definition 1;*
- d *is a* ***feature model****;*
- $\gamma : trans \mapsto [\![d]\!] \mapsto \mathbb{B}$ *is a labelling function specifying for each transition which valid products may execute it; this function is represented as a boolean expression over the features of* d*, called* ***feature expression****.*

FTS projection: To derive the LTS of one particular card payment terminal product, the FTS is **projected** [63,66] on this product by pruning the transitions whose feature expressions are not satisfied and by removing the feature expressions. For instance, Figure 1.5 is the projection of the FTS in Figure 1.4 on the card payment terminal product supporting direct debit (*dd*) cards, with a PIN identification (*pin*) and an online connection (*online*). Formally, the projection operator is defined as follows:

Definition 3 (Projection operator [63]) *Let* $fts = (S, Act, trans, i, d, \gamma)$ *be an FTS, and* $p \in [\![d]\!]$ *be a product of the feature model* d*. The projection of fts onto* p*, denoted* $fts_{|p}$*, is the LTS* $(S, Act, trans', i)$ *where*

$$trans' = \{tr \in trans \mid SAT(p \wedge \gamma(tr))\}$$

Where SAT checks the satisfiability of the feature expression labelling the transition giving the product p*.*

Deterministic FTS: As for LTSs, an FTS is deterministic if, for all sequence of actions, there is at most one possible path (sequence of transitions) for those actions. Since FTS models the behaviour of all the products of a product line, it moreover requires to have satisfiable feature expressions on this path.

Property 1 (Deterministic FTS) *An FTS* $(S, Act, trans, i, d, \gamma)$ *is deterministic if:*

$$\forall (\alpha_1, \ldots, \alpha_n) \in (Act \cup \{\epsilon\})^*, \exists \text{ at most one } seq = i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_m} s_k \text{ such that } SAT\left(CNF(d) \wedge \bigwedge_{tr \in seq} \gamma(tr)\right)$$

Checking if an FTS is deterministic is computationally heavy. In the worst case, it requires a complete exploration of the model with multiple SAT calls.

Connected FTS: A connected FTS is an FTS that has no isolated state. All states may be reached from the initial state and reach back the initial state, *i.e.,* for each state, there exists a path starting from the initial state and going back to the initial state.

Property 2 (Connected FTS) *An FTS* $(S, Act, trans, i, d, \gamma)$ *is connected if:*

$$\forall s \in S, \exists seq = i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_m} s \xrightarrow{\alpha_n} \ldots \xrightarrow{\alpha_o} i \text{ such that } SAT\left(CNF(d) \wedge \bigwedge_{tr \in seq} \gamma(tr)\right)$$

One may use (for instance) an accessibility matrix (see Algorithm 2 from Section 5.2.1) to check that there is no isolated state. In the remainder of this thesis, most of our algorithms assume connected FTSs.

1.3.4 Related work

We choose FTS as formalism to model the behaviour of SPLs over MTS [105] and PL-CSS [122]. MTS is an extension of LTS where the set of transitions is partitioned into *may* and *must* transitions. *Must* transitions are transitions fired by all the products of the product line, while *may* transitions are fired by only some (undetermined) products of the product line. To relate a transition to the exact set of products able to execute it, Asirelli *et al.* [22, 23, 25] associates MTS to a branching-time temporal logic named Modal Hennessy-Milner Logic (MHML) representing the constraints between the features and the actions. The MHML formula may be derived from a feature model (representing those constraints) and the associated MTS but makes the relation between products and transitions unclear.

Product Line CCS (PL-CCS) [122] is a process calculus extending Milners's Calculus of Communicating Systems (CCS) [217] by adding a *binary variant* operator to represent alternatives features in a SPL. As for MTSs, PL-CSS does not include constraints between features and relies on an external *mu*-calculus [286] to express them.

In their work, Beohar *et al.* [39] analyse the **expressiveness** of FTS, MTS, and PL-CCS by comparing the set of products they can specify. Products specifications are represented using LTSs. They demonstrate that FTS is the most expressive formalism, followed by PL-CCS and MTS. Meaning that MTS models may be expressed using PL-CCS, and PL-CCS models may be expressed using FTS but not (always) the other way around.

1.4 Wrap up

In this chapter, we presented the standard software product line engineering process and focuses on domain level to perform behavioural model-based testing, *i.e.*, selecting relevant products and behaviour to test in the product line. SPL variability is encoded using a boolean feature model and behaviour is described using an annotation-based formalism: a connected FTS. We choose FTSs for their expressiveness and the simplicity of their encoding allowing powerful algorithms [63, 72] while preserving a readable relation between transitions and products able to execute them.

CHAPTER 2

SOFTWARE AND SOFTWARE PRODUCT LINES TESTING

Software testing is a process present in a majority of software developments. According to Mathur's definitions [206], it aims at evaluating if a software behaves as expected. When running a software system, one may face a **failure**, *i.e.,* an unexpected behaviour of the system. This failure is the propagation to the output of the system of one or more **bugs** (also called **faults**), coming from **errors** made during the writing of the source code of the software system or resulting from earlier issues in specifications. The goal of a software testing process is to find as many bugs as possible in a given software system, called System Under Test (SUT), in order to prevent failures to happen during the operation of the software system.

When it comes to product lines, testing becomes more complex. As the system under test is the set of products of this product line, using a standard testing process would require to derive all those products and, for each one of them, design and execute a test suite. This approach, called **product-based** [301], is intractable for the large majority of product lines. SPL testing requires to adapt standard testing process to minimize the effort by reusing testing assets from one product to another and prioritizing the products to test. To achieve this, most techniques adopt a model-based approach: *e.g.,* sampling a set of products to test from a feature model.

This chapter presents the state-of-the-art of SPL testing. Sections 2.1 and 2.2 presents software testing and model-based testing, Section 2.3 gives a view of SPL testing, and Section 2.4 focuses on existing model-based approach to SPL testing.

2.1 Software testing

The Software Engineering Body of Knowledge from the IEEE Computer Society [147] defines software testing as follows:

> Software testing consists of the **dynamic** verification that a program provides **expected** behaviours on a **finite** set of test cases, suitably **selected** from the usually infinite execution domain.

This definition, although not specific on how to perform software testing, includes different important aspects. First, the SUT has to be **executed** on a set of input values[1] in order to observe its behaviour. Second, to decide if the SUT behaves as expected, it must be possible, based on the outcomes of the SUT for a given input, to decide if the outcomes are acceptable or not. This is also referred to as **the oracle problem** [147]. Finally, the number of observed behaviours exercised by the test cases is **finite**: a software testing process is the result of a trade-off between limited resources, schedules, and unlimited test requirements. Therefore, the **test suite** (*i.e.,* the set of test cases) has to be properly selected in order to satisfy this trade-off using a **criterion**.

The software testing process itself may be implemented in various ways. Tretmans [302, 307] defines a typology of software testing processes based on three dimensions: the characteristic being tested, the scale of the SUT, and the information used to select test cases.

Characteristic being tested: The main characteristic being tested is the functionality (**functional testing**), which aims at checking that a SUT produces a correct output for a given input. Other characteristics includes (but are not limited to) robustness (**robustness testing**), which aims at checking that the SUT can resist to invalid conditions in its environment (*e.g.,* wrong inputs, hardware failures, network failures, other systems failures, *etc.*); performance (**performance testing**), which aims at checking that the SUT can resist heavy loads; usability (**usability testing**), which focuses on user interfaces problems; security (**security testing**), which aims at checking that the system is not vulnerable to malicious users; *etc.* In this thesis, we focus on functional testing.

Scale of the SUT: It indicates which parts of the system are considered during the execution of each test case: **unit testing** focuses on single units at a time (*e.g.,* a single method, a single function, a single class, *etc.*); **component testing** tests each part of the system separately, while **integration testing** checks that the different components work together correctly; finally, **system testing** considers the whole system to perform testing.

[1] In this case, input values may refer to input data or, more generally, to a specific input state of the SUT.

Information used to select test cases: Information may be either **white box** or **black box**. White box testing processes use the source code as input. They allow one to define selection criteria on the source code of the application: *e.g.,* statement coverage requires that each statement is executed at least once by one test case of the test suite. Black box testing processes will use the requirements of the SUT as input. In this case, the source code is not accessible and selection criteria are specified over the requirements: *e.g.,* input domain coverage requires to split the input domain in equivalence classes and to design test cases that will use at least one element of each class.

Most of the time, the selection of a test suite is done manually. For instance, it is very common for developers to write functional unit tests for their code before submitting it to a version control system. In most cases and with the right tool support, this may be enough. However, manual testing becomes expensive for larger systems, especially during integration and system testing [307].

2.2 Model-based testing

Automating test suite selection is not easy though. It requires an **input generator** that, for each test case, generates a sequence of input operations for the system (*e.g.,* a sequence of method calls); an **oracle** which decides, based on the input sequence and the outputs of the system if the test case is pass or fail; and a **selection criterion** (with a mechanism to measure it) in order to decide when to stop the selection. For instance, EvoSuite [107] is a white box Java test cases generator that uses an evolutionary algorithm to generate JUnits by analysing methods paths. Since the tool only uses the source code as input, the oracle is very limited and can only tell if a test case execution should throw an exception or not. To improve this oracle, it requires to analyse the specification of the methods to foretell the expected output.

An alternative is to use a semi-automated approach: **model-based testing** [307]. It requires to define a model of the expected behaviour of the SUT (*i.e.,* a specification) that serves as input to an automated test suite selection tool. The model should be small enough to be cheaper than the analysis of the actual system, but accurate enough to describe the characteristics to test. The tool uses this model to generate a sequence of input and as oracle for each sequence. Model-based testing is the automation of black box test suite selection [307].

Figure 2.1 presents a generic model-based testing process. First the test engineer defines a **model** of the SUT. This model is provided to the test suite selection tool that, based on the **selection criterion** specified by the test engineer, selects an **abstract test suite**. This abstract test suite contains test cases expressed at the same abstraction level than the input model. They cannot be executed on the SUT as-is and require to be **concretized** to test scripts using mapping information to link abstract actions and inputs in the test cases to concrete SUT operations. The test scripts may then be **executed** by a dedicated test execution environment that operates the SUT (optionally using an adaptor layer to abstract complex SUT's operations) and produces a report with the test results. We present our implementation of this generic process in Chapter 3.

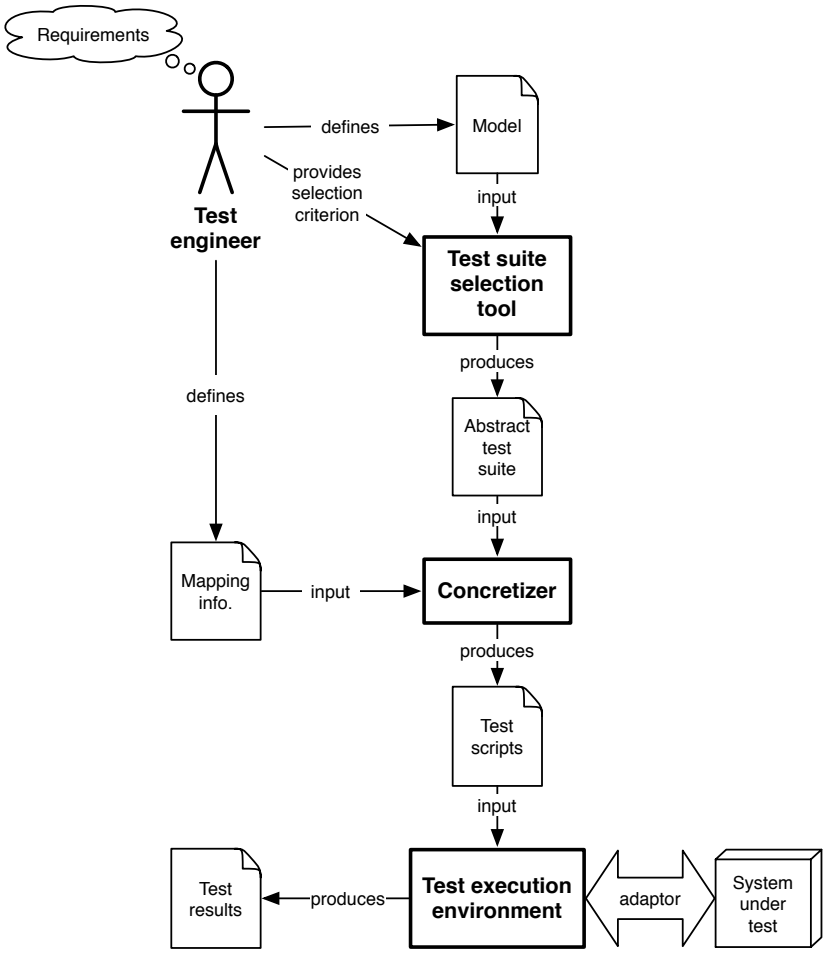


Figure 2.1: Model-based testing process

2.3 Software product line testing

As presented in Figure 1.1, SPL testing is performed on two different levels: domain testing and application testing [252]. During **domain testing**, reusable test artefacts are defined and validated for the SPL. Those artefacts are combined during **application testing** in order to test one particular product.

2.3.1 Software product line testing strategies

Domain testing processes include the definition of a test plan corresponding to the strategy used to test the SPL. Pohl *et al.* [252] define four kinds of strategies to validate a SPL:

Brute force: Brute force strategy consists in performing all the testing activities for all the possible products during domain engineering. Considering an FTS and a feature model, this would consist in deriving all the valid products from the feature model, and for each product, project the FTS on the product to derive test cases from the resulting LTS using a classical model-based testing approach [307]. As empirically shown by Halin *et al.* [126, 127], this strategy is expensive, not applicable in most cases, and is no more discussed here.

Pure application: Pure application strategy consists in performing testing only during application engineering. Each derived product is tested individually using a standard (non-SPL) testing procedure and no domain test artefacts are reused. Contrary to brute force, pure application strategy does not build all products, it only tests one when it is derived for a customer. As for the previous one, an FTS may be projected on the considered products and the resulting LTS used as input for a model-based testing approach. This strategy is no more discussed here. Interested reader can refer to single-system software testing literature [206, 307].

Sample application: Sample application strategy consists in selecting one or a few sample products (*i.e.,* to do a **product prioritization**) to test domain artefacts, but still requires to test other derived products during application engineering. Again, the FTS may be projected on the sampled products and the resulting LTS used as input for a model-based testing approach. The product sampling itself may be done using various methods: in this thesis, we use the FTS to drive it.

Commonality and reuse: Commonality and reuse strategy consists in testing parts common to all the products and preparing test artefacts for variable parts during domain engineering, and reusing test artefacts specific to a product during application engineering. Applied to an FTS and a feature model, this strategy is close to what we propose in this thesis. The mapping information used during abstract test case conretization (in Figure 2.1) have to be defined for all the actions of the FTS during domain engineering, in order to be reused across different products. To test the parts common to all the products, one has to project the FTS on the product containing only the features common to all the products of the SPL (if it exists) and use the resulting LTS to derive test cases using model-based testing.

On the one hand, *sample application* strategy does not produce variable test artefacts (and thus does not reuse domain test artefacts during application engineering). This may cause an overhead if the sampled products do not correspond to the products derived afterwards. On the other hand, *commonality and reuse* strategy allows to reuse domain test artefacts during application engineering, but validates only parts common to all products during domain testing, lacking at detecting problems in variable parts in an early development stage. A fifth strategy consists in **combining** *sample application* and *commonality and reuse* strategies: domain test artefacts contains variability information and may be reused and a sample of products is used to test common and variable parts of the product line. This allows to have a faster feedback on variable parts validation while still allowing

to reuse domain test artefacts during application testing. This last strategy is the one recommended for the framework developed in this thesis.

2.3.2 Software product line analysis classification

To support domain testing, a lot of existing approaches use analysis techniques to select test cases and prioritize products. Depending on the considered artefacts and the modality, Thüm *et al.* [301] classify SPL analysis in three categories:

Product-based analysis: In product-based analysis, products are built and analysed individually. This allows to use existing methods designed for single systems that will work on application artefacts only, but becomes expensive if the set of products to analyse is large. To overcome this, one may first prioritize products to perform the analysis only on a subset of all the products of the product line. In this case, the product-based analysis is combined with a family-based sampling technique.

Family-based analysis: Family-based analysis uses domain artefacts in combination with a feature model to perform a variability aware analysis of all the products of the product line at once. For instance, FTS model checking [63] is a family-based analysis. The ProVeLines [72] model checker uses the feature model and the feature expressions on the transitions to check that all the products satisfies the given property in one exploration of the FTS. If this is not the case, it prints the products that violates the property.

Feature-based analysis: In feature-based analysis, domain artefacts implementing a given feature are analysed in isolation without considering other features. Contrary to family-based analysis, feature-based does not consider the feature model during the analysis and are not able to detect undesired features interactions [320] (*i.e.,* undesired behaviours appearing only if a certain combination of features is present in the product).

As for SPL testing strategies, analyses may be combined. For instance, performing a feature-based analysis before a product-based or a family-based analysis may reduce the effort needed by the latter as feature-based analysis allows to factorise the analyse of elements that do not depend on other features. In our model-based testing framework, test cases and products are selected from domain artefacts (*i.e.,* a FTS and a feature model) and concretized into executable test scripts in a family-based approach, while the execution of the test scripts on each product is product-based.

2.4 Model-based software product line testing

There exists several testing approaches to software product lines and lot of them are model-based [92, 98, 234]. This may be explained by the complexity induced by the product lines variability. This variability is usually captured in a feature model reused during testing in combination with other domain artefacts (that may be models or

not). A large part of those approaches are focused on **sampling** a representative set of products to test in order to find as many undesired feature interactions as possible [92].

2.4.1 Feature interaction

Apel *et al.* [16] define a **feature interaction** between at least two features as *an emergent behaviour that cannot be easily deduced from the behaviours associated with the individual features involved.* A lot of those interactions are intended: for instance, a security feature may interact with other features by encrypting all connections. Problem occurs when an **unintended feature interaction** happens, usually resulting from a bug, and causes failures at runtime in the products containing the features involved. Test cases have to check that desired feature interactions are achieved, by checking that connections are encrypted for instance, but also that undesired interactions do not happen.

One of the specificities of software product line testing is thus to seek and find undesired feature interactions, usually occurring when a small number of features are involved [171]. For instance, amongst the 6 faults discovered in JHipster, Halin *et al.* [126] found that 5 are caused by interactions of size two and one by an interaction of size four. To find those undesired feature interactions, sampling techniques have been developed to select a representative set of products involving as many different feature combinations as possible. The most common sampling techniques are based on **Combinatorial Interaction Testing (CIT)** and known as t-wise product sampling.

2.4.2 *T*-wise sampling

T-wise sampling techniques sample a set of products in which all the t-uples of features allowed by the feature model are represented at least once. The parameter t is called the **strength** of the sampling. Since lot unintended feature interactions appear between two features [171], pairwise (2-wise) product sampling is the most studied approach [92, 199].

Many CIT approaches have been adapted to take constraints from the feature model into account in order to perform ***t*-wise sampling** [69, 199]. They are implemented in different tools like CASA [112, 113], SPLCAT [155], NIST-ACTS [183], MoSo-PoLiTe [233], PACOGEN [140], *etc.* Depending on the approach, a large variety of underlying techniques may be used [55, 199]. The main ones [69, 175] are **algebraic methods** [54, 129, 285], **greedy algorithms** [46–48, 67, 77, 112, 113, 154–156, 175, 184, 200, 285], **heuristic search** [68, 111], and **constraint programming** [140, 204, 233, 244, 245, 250].

Despite advances being made, introducing constraints during t-wise sampling yields scalability issues for large feature models [21, 120, 135, 226] and higher interaction strengths [135, 172, 251, 260]. To overcome those limitations, Henard *et al.* [135] developed a dissimilarity-driven sampling to mimic the t-wise criteria. This sampling tries to maximise the dissimilarity between the selected products.

2.4.3 Dissimilarity-driven sampling

Dissimilarity-driven product sampling is based on the following heuristic: dissimilar products are more likely to detect bugs than similar ones. This is empirically demonstrated for single systems (model-based) testing by Hemmati *et al.* [130]. The idea is to sample a set of products as dissimilar as possible, based on a distance measure. In their work, Henard *et al.* [135] consider the **distance** between products in terms of selected features: two products are dissimilar if the features that compose them are different.

The sampling is performed using an **evolutionary algorithm** that takes a fixed size for the sample and a duration as parameters. The Jaccard distance [149] between two products is used to compute the fitness value of the sample at each iteration. During the execution, products are ranked inside the sample according to their dissimilarity. Eventually, the algorithm produces a ranked list of products where most dissimilar ones are in the first positions. Contrary to CIT algorithm, the size of the sample is fixed from the beginning. This may require some prior decision from the test engineer, but is also very convenient when the testing budget is limited to a given number of products [126].

The algorithm is implemented in PLEDGE[2] and Henard *et al.* [135] empirically demonstrate the relevance of dissimilarity-driven sampling for software product lines for large feature models and higher strengths. Those results have been independently confirmed for smaller SPLs by Al-Hajjaji *et al.* [8]. Parejo *et al.* [242] extended this idea, using a genetic algorithm, by considering both functional and non-functional properties during the selection process. In this thesis, we extend this idea in Chapter 5 by considering both features and behavioural dissimilarity during the sampling. More details on distance measure, fitness value, and selection algorithm are provided in this chapter.

2.4.4 Related work

Other strategies to perform SPL mode-based testing at the domain level have been proposed. Lochau *et al.* [195, 197] define a model-based approach that shifts from one product to another by applying deltas to state-machine models. The approach allows to reuse test cases from previous products and derive of (re)test obligations for each new considered product. These deltas, are built incrementally when switching from one product to another and are part of the reusable domain test artefacts. In this thesis, we want to select relevant test cases and products before performing the effective tests. Cichos *et al.* [58] use the notions of 150% test model, *i.e.,* a test model of the behaviour of a product line, and of test goal to select test cases for a product line but do not define criteria at the SPL level. Beohar *et al.* [37] adapt *ioco* [303] to FTSs. The Input-Output Conformance (*ioco*) testing is a model-based testing approach that aims at building a conformance relation between a specification and a model of the running system, based on the inputs and outputs of this system [303, 304]. Contrary to this approach, we do not seek exhaustive testing

[2] See http://research.henard.net/SPL/PLEDGE/.

of an implementation but rather to select relevant abstract test cases based on the criteria provided by the test engineer.

2.5 Wrap up

This chapter presents the necessary background on software testing, a generic model-based testing procedure, and the most studied model-based software product line testing approaches. A lot of those approaches are family-based: they use domain artefacts and reason at the product line level. They seek to sample a subset of products to test in priority, based on the feature model and a sampling criterion. In this chapter, we present t-wise and dissimilarity-driven sampling. T-wise ensures that every t combination of features is present in at least one sampled valid product and dissimilarity-driven try to maximise the dissimilarity (in terms of features) between the sampled products.

Complementary to those approaches, the selection criteria developed in Chapter 5 take not only variability, but also behaviour into account. We define in Chapter 3 a **family-based** model-based testing framework that may be used in **combined *sample application*** and ***commonality and reuse*** testing strategies.

Part II

Testing Framework

CHAPTER 3

FRAMEWORK OVERVIEW

This chapter gives the general view of our behavioural model-based testing framework for software product lines [80] by instantiating the generic process described in Section 2.2. The main part of the framework concerns the **abstract test suite selection** at the product line (domain) level. Based on this selection, the test engineer may **prioritize** products to test and/or assess the test suite using **mutation analysis**. We focus on **behavioural** aspect: the selection is done from a FTS, defined by the test engineer.

3.1 Overview

Figure 3.1 gives a general view of our framework. The test engineer, based on requirements, defines a FTS (and a feature model). This FTS serves as input to our tool: the Variability Intensive Behavioural teSting (VIBeS) framework. The tool is operated by the engineer to select a test suite, based on a selection criterion coming from the requirements. For instance, one may want to select a test suite that covers the states of the FTS or that contains test cases as dissimilar as possible. Based on this test suite, the engineer may prioritize the products to test, *e.g.*, by selecting the product that allows to execute as much test cases as possible, and assess the quality of the test suite using mutation analysis.

Eventually, VIBeS produces an **abstract test suite**, with test cases representing sequences of actions from the FTS. This test suite is defined for the product line, which means that the abstract test cases, once concretized, may be executed by one or more products. The **concretization** is handled by the Abstract test case Concretizer (AbsCon) plugin, based on a **mapping** provided by the test engineer. AbsCon produces executable test scripts for the QSpin Tailored Automated System Test Environment (QTaste), a test case management and execution system that uses

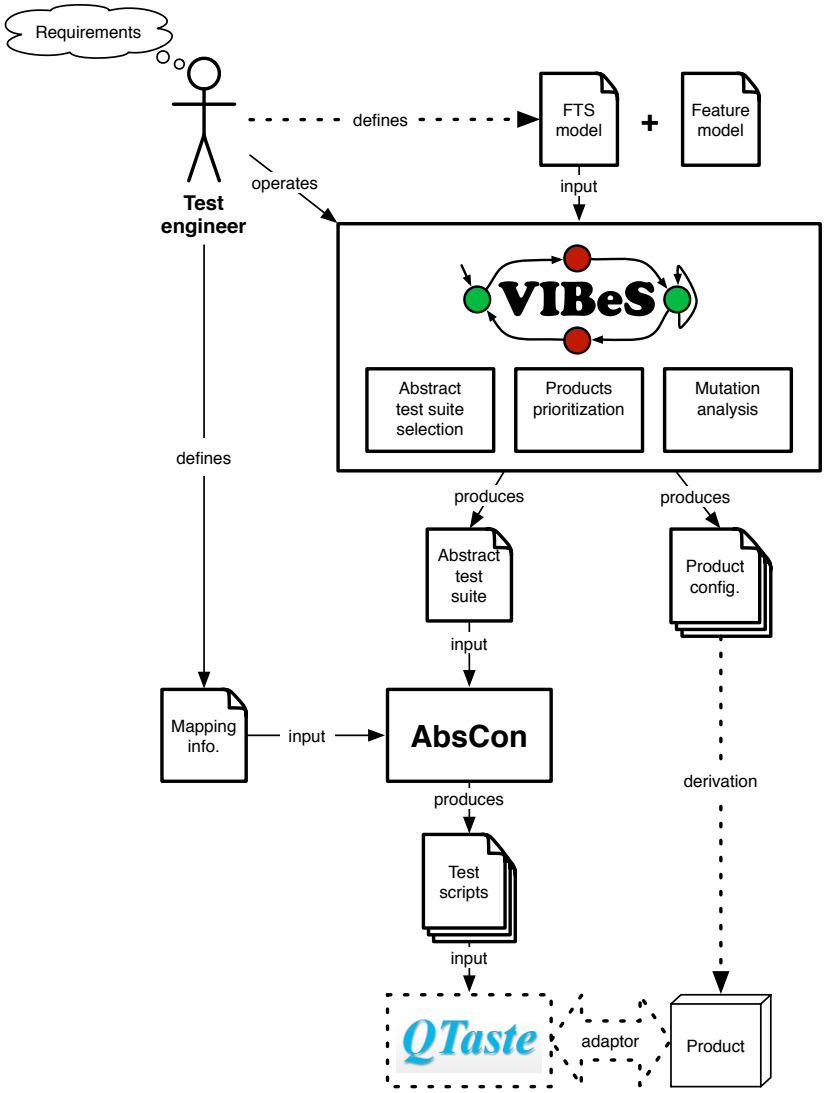


Figure 3.1: Behavioural SPL testing framework overview

an adaptor to handle communication with the product under test. The selection of the test cases to execute on the given product amongst all the test cases of the test suite is done in QTaste, based on the list of abstract test cases executable by the selected product.

The different elements of Figure 3.1 are detailed, discussed, and evaluated in the next chapters: abstract test suite selection and product prioritization are presented in Chapter 5; Chapter 6 discuss mutation analysis; Chapter 8 presents the implementation and usage of VIBeS; and Chapter 9 details how concretization may be performed with AbsCon to get test scripts executable by QTaste. Elements in dashed

lines in Figure 3.1 are out of the scope of this thesis and discussed in the next section.

3.2 Uncovered aspects

In this thesis, our main goal is to provide a framework to support the testing process. Software product line model-based testing is a complex task that involves many aspects, we identify the followings as out of the scope of this work.

3.2.1 Process management

VIBeS is a toolbox that allows to perform various testing activities based on the requirements of the test engineer. In a software engineering context, those activities are organised in process [147] that has to be managed and monitored. The definition of such product line model-based testing process falls out of the scope of this work.

3.2.2 Requirements definition

We assume that requirements are used both to define the FTS and the feature model, and the test suite selection criterion. The formalism used to define those requirements may take several forms: structured or semi-structured sentences [201], goal modelling [310], LTL formulas that the product line must satisfy [63], *etc.* How those requirements are elicited and formalised [26,213,227] is out of the scope of this work. We assume that the test engineer knows the requirements when operating VIBeS.

3.2.3 FTS model definition

In its implementation, VIBeS allows to define FTSs using a Java DSL or with an XML file. The process used to define this FTS model is out of the scope of this work.

Model the detailed behaviour of the SPL using an FTS is intractable. Instead, the test engineer abstracts the behaviour to be able to run analysis on the model. This abstraction is done by defining abstract actions (*i.e.,* actions that represent one or more effective functions or methods calls) or by focusing on the subset of behaviour under test. As FTSs may be used to express the semantic of other variability-aware behavioural languages [74], one may also use an abstract modelling language: *e.g.,* FORML [281, 283]. In such a case, we assume that a bidirectional mapping can be defined between the abstract modelling language and FTS.

3.2.4 Feature model definition

We assume that the feature model used with the FTS is a boolean feature model that may be translated to a CNF formula [32, 276]. As for FTSs, the definition and the formalism used to define this feature model are out of the scope of this work. Considering non-boolean feature models requires to use constraint solvers other than SAT or BDD solvers, and to adapt FTSs' feature expressions to take non-boolean types into account.

3.2.5 Product derivation

VIBeS produces a set of product configurations to test. This set is represented as a set of boolean assignments of the feature model. The derivation process of the actual product (in dashed lines in Figure 3.1) is out of the scope of this work. This derivation may take several forms and may be automated or not [252].

3.2.6 Test scripts execution

Test scripts management, covering executions of the scripts and reporting, is handled by QTaste. QTaste is an industrial test environment that provides functionalities to execute test scripts using an adaptor to manipulate the product under test. More details about QTaste are provided in Chapter 9.

3.3 Wrap up

This chapter presents an overview as well as the limitations of our behavioural model-based product line testing framework. We assume connected FTSs and boolean feature models as inputs. The different elements of the framework are detailed in the next chapters.

CHAPTER 4

CASE STUDIES

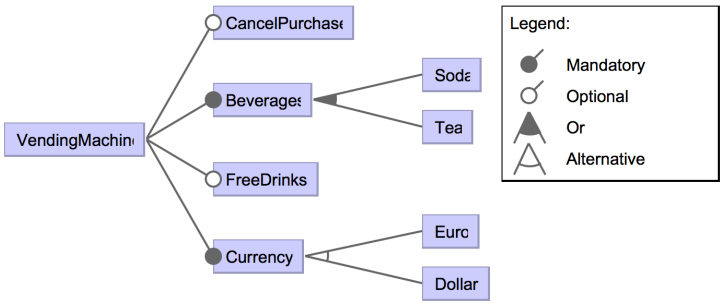
We present in this chapter the different case studies used along the thesis. We consider models from different sources with varying size. Our models are: the soda vending machine model (*S. V. Mach.*) in Section 4.1; the mine pump (*Minepump*) in Section 4.3; the WordPress models (*AGE-RR*, *AGE-RRN*, *Elsa-RR*, and *Elsa-RRN*) in Section 4.5; the Claroline website (*Claroline*) in Section 4.6; and the random models (*Random 1-4*) in Section 4.8. We compare the models characteristics in Section 4.7 and define the threats to validity linked to the selection of our case studies in Section 4.9.

4.1 Soda vending machine

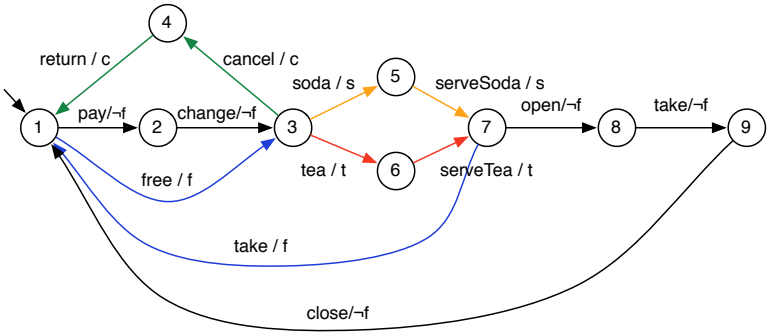
The soda vending machine SPL is a classical beverage vending machine described by Classen *et al.* [63]. The feature model is presented in Figure 4.1(a): it sells soda (s) or tea (t) in euro or dollar, may offer free drinks (f), and optionally allows to cancel a purchase (c). The FTS describing the behaviour of the SPL is presented in Figure 4.1(b) (for readability, the feature names have been shortened): the user either pays or chooses a free drink (if feature f has been selected); he may cancel its purchase (if feature c has been selected); he chooses a beverage, allowed by the selected features; and gets his beverage directly (if feature f has been selected) of after opening the machine (otherwise).

4.2 Card payment terminal

Figure 4.2(a) presents the feature model of a card payment terminal manually defined using Eurocard-Mastercard-Visa (EMV) specification document [97]. This machine accepts card payment with a given payment schema (direct debit and/or



(a) Feature model

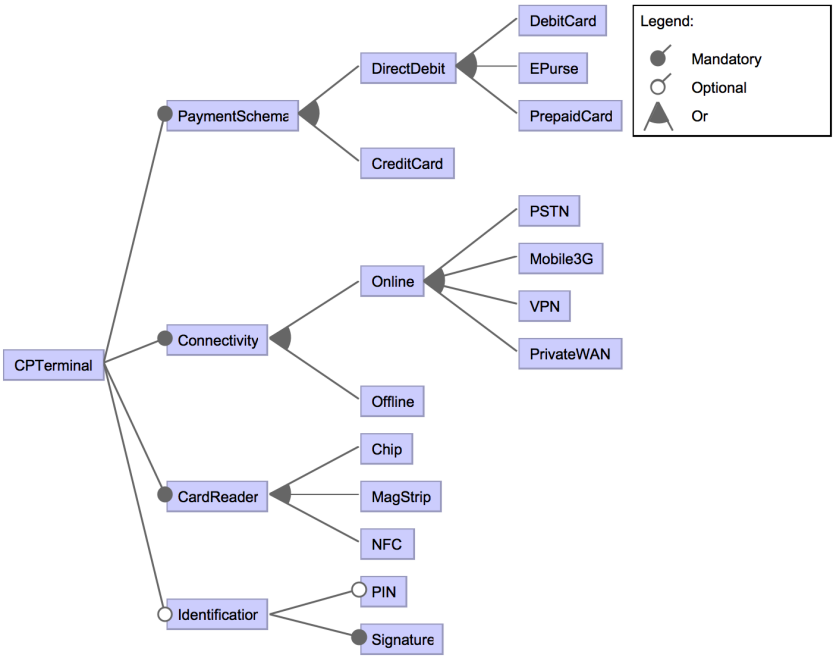


(b) Featured transition system

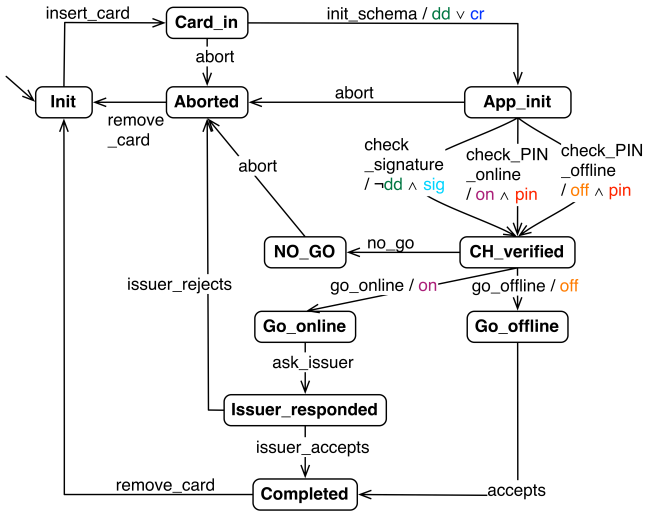
Figure 4.1: Soda vending machine

credit card). It works online, using one or more connectivity option, or offline with the card payment service. Cards are read using a chip, the magnetic strip, or near field contact (NFC); and card owner may be authenticated using signature and optionally PIN code.

The FTS in Figure 4.2(b) presents the behaviour (we want to test) of the card payment terminal (minus one technical intermediate state) to processes a payment. First, the Card Holder (CH), *i.e.*, the user, inserts his card in the terminal. If the card is a direct debit (dd) or a credit card (cr), the terminal will proceed the initialisation of the transaction. In this version of the product line, the card holder must always identify himself using either: his signature if the card is not a direct debit card and if the terminal supports the signature identification (sig); or his secret PIN if the terminal supports PIN identification (pin), which may be checked online (on) or offline (off) with the transaction processor company. If the identification succeeds, the terminal will proceed online or offline to the payment, otherwise, the transaction is aborted. Whether the transaction succeeds or not, the card holder removes his card from the terminal at the end of the process.



(a) Feature model



(b) Featured transition system

Figure 4.2: Card payment terminal

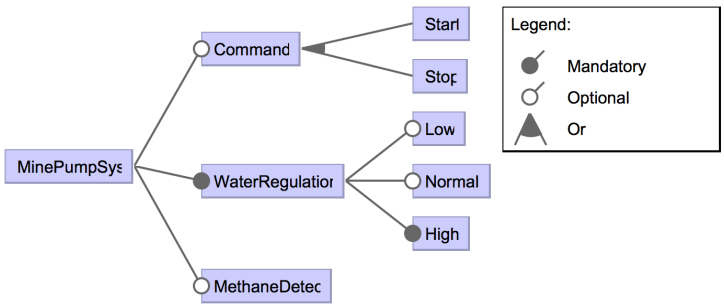


Figure 4.3: Minepump feature model

4.3 Minepump

The Minepump model [60] represent a product line of pumps designed to keep a mine shaft clear of water and (optionally) avoid the danger of a methane explosion. Figure 4.3 presents the feature model of the SPL. A pump has a water regulator that can detect the level of water in the shaft. It may be equipped with a methane detector and a command interface allowing to manually start and stop the pump. The FTS describing the behaviour of the pumps has 25 states and 41 transitions.

4.4 Sferion™landing symbology function

Sferion™ is an industrial situational awareness suite for helicopters flying in degraded visual environments [6, 84, 267]. The landing symbology function supports the pilot during the landing approach by marking the intended landing position on ground using a head-tracked Helmet Mounted Sight and Display (HMS/D) and Hands On Collective And Stick (HOCAS). Depending on the selected feature, the landing may be marked by the handling pilot only or by bots pilots. The spatial awareness is enhanced during the final landing approach by displaying 3D conformal visual cues on the helmet with optionally real reference of the object. The ground (and optionally obstacles) in the landing zone are detected and classified using a real-time Obstacle Warning System (OWS). Depending on the customer and the helicopter platform, the landing symbolog1y function may have different features (Figure 4.4) selected: *ELOP* or *HELLAS* sensors for the *OWS*; *SI_sensor_based* or *SI_from_DB* as slope indication provider for landing position; the Thales or Elbit HMS/D; the HOCAS from Honeywell or from Aviation Systems inc.; a database provided by the helicopter platform or a Cassidian database.

The models have been designed by engineers using MaTeLo [9] tool, OVM and Matelo Product Line Management (MPLM) [268]. They have originally been presented by Samih et al. [267]. MaTeLo supports the description of statistical usage models by using hierarchical Markov chains. MaTeLo's usage model is a DTMC, where the nodes represent the major states of the system and the transitions are labelled with the actions or operations of the SUT with their probability to be fired.

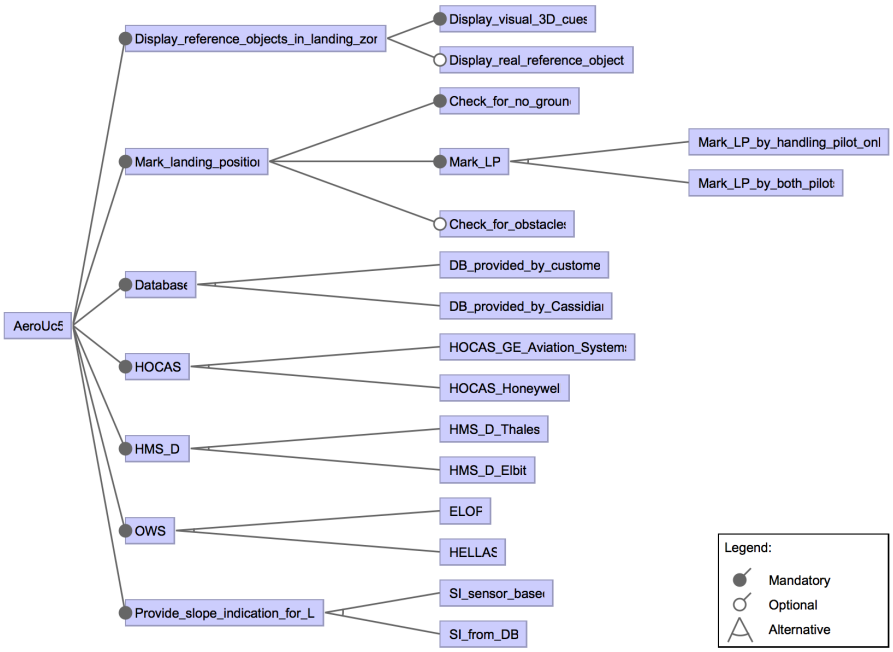


Figure 4.4: Sferion™landing symbology function feature model

In a DTMC, the transitions are tagged with a probability representing the likelihood, when we are in the starting state, to execute the transition, and the action performed when the transition is executed. Each action is associated with zero, one or more requirements. The variability is described using OVM (Orthogonal Variability Model), each variation point is associated to zero, one or more requirement(s). The mapping, encoded in MPLM, between the variation points and the usage model transitions is made through the requirements. MPLM and MaTeLo tools support the product-based test derivation approach.

We encoded the Sferion™ landing symbology function models using our formalisms: the usage model has been flattened to remove hierarchy (by hand in 1/2 day); the OVM model has been translated to a feature model (by hand in 1/2 day); and the mapping between features and behaviour has been encoded using an FTS, generated from the MaTeLo usage model, the OVM model and the MPLM mapping model (in 1 day).

4.5 WordPress, an open-source CMS

WordPress [319] is a popular open-source Content Management System (CMS) used by more than 60 million websites [71]. It includes a plugin architecture and a template system, allowing one to modify its behaviour by adding new functionalities (*i.e.,* plugins), or the rendering of the website (*i.e.,* themes), respectively. In February

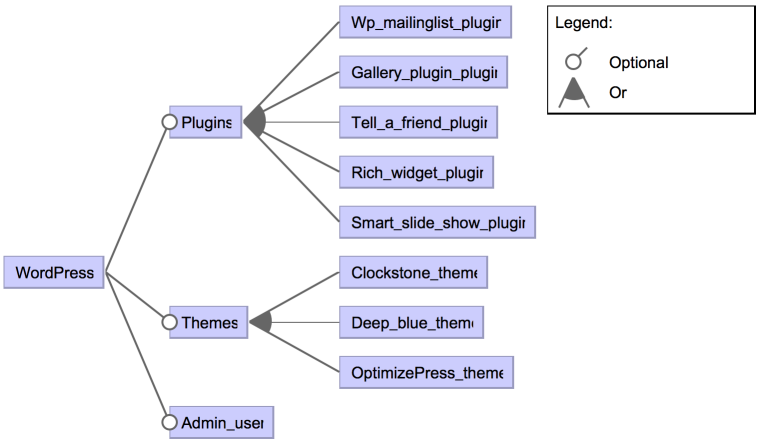


Figure 4.5: Simplified WordPress feature model

2017, the WordPress database (https://wordpress.org/) counted 48,898 plugins and 4,462 (latest) themes.

We reverse-engineered the feature models and the FTSs of two WordPress instances, AGE and Elsa portals, based on their Apache webserver log file. The *Assemblée Générale des Étudiants* (AGE) website (http://www.age-namur.be) is the portal of the general student assembly of the University of Namur. It uses a dedicated WordPress instance and provided us a log file with 1,285,592 entries from May 2013 to March 2014. The European Law Students' Association (Elsa) of Louvain-la-Neuve (http://elsa-lln.be) also uses a dedicated WordPress instance and provided us a log with 48,823 entries from February 2014 to the end of April 2014.

4.5.1 Feature model

Our WordPress feature model has 3 optional core features: *Plugins*, *Themes*, and *Admin_user*. One configuration (*i.e.*, product) of the feature models represents the minimal instance needed to play a test suite. The *Admin_user* feature will be selected only if a test case requires to access the administration pages.

To identify the plugins and themes used in the instances, we analysed the Apache webserver log entries. Each time an HTTP request is addressed to the server, one entry is created in the log file with the following format [15]: the IP address of the visitor sending the request; a login if the visitor is identified on the system; the date and time of the request; the HTTP request itself, beginning with GET, POST, *etc.*, followed by a URL and the protocol version; the status code sent back to the client; the size of the object returned to the client; the website the client reports having been referred from; and finally the information on the client's browser. For instance, a request to the `index.php` page of the WordPress instance from a Mozilla Firefox navigator will add the following entry in the log file:

```
1  66.155.40.250 - - [08/Nov/2013:11:38:11 +0100] "GET /index.php?p=potins&
       action=aimepas&id=135 HTTP/1.1" 200 708388 "-" "Mozilla/5.0 (Windows
       NT 6.1; WOW64; rv:25.0) Gecko/20100101 Firefox/25.0"
```

Plugins and themes resources are placed in specific folders on the webserver. This allows us to filter the URL of the log entries using the following regular expressions:

- `.*/wp-content/plugins/([^/]+)/.*` to detect access to a plugin resource;
- `.*/wp-content/themes/([^/]+)/.*` to detect access to themes resources;
- `.*/wp-admin/.*` to detect that an administration page has been used.

The names of the plugins and themes appears right after the `plugins/` and `themes/` part of the URL (matched by the `([^/]+)` part of the regular expression). Since the logs have been anonymized, we do not have the identification of the users and rely on the last regular expression to detect administrator accesses to the WordPress instance.

We ended up with two feature models: one with 45 features for the AGE WordPress instance, and one with 70 features for the Elsa WordPress instance. Figure 4.5 presents a simplified version of the Elsa WordPress instance (AGE instance follows the same pattern). The root feature *WordPress* is decomposed in three optional sub-features: *Admin_user*, *Themes*, and *Plugins*. Each plugin or theme (resp.) appearing in a log entry will be a sub-feature of the *Plugins* or *Themes* (resp.) feature.

Another approach to build the feature model would have been to mine the plugins and themes repository of WordPress. This would give a much larger feature model: more than 50,000 features. Since we are in behavioural model-based context, we seek to find incorrect behaviour in the product line. This incorrect behaviour may be caused by feature interaction or may have another root cause (*e.g.,* an error in the source code introducing faults in all the products). In order to select our test cases, we need a behavioural model of the SPL, which is derived from the Apache Web server log. In this context, considering all the WordPress plugins and themes in the feature model has little meaning since only the behaviour of the plugins and themes activated in the running WordPress instances (appearing in the log) will appear in the behavioural model. This is why we consider only those plugins and themes as relevant for testing and add them in the feature models.

Similarly, Sánchez *et al.* [270, 271] used a white box testing approach to test the Drupal CMS. They rely on documentation, source code, issue tracking system, and Git versioning repository to reverse engineer the system's feature model. Stress is put on product selection, whereas we focus on behaviour selection (at the product line level) in a black box approach.

4.5.2 Featured transition system

We reverse engineered the FTSs of the AGE and Elsa WordPress instances using a 2-gram (bigram) inference method [291, 292]. This method uses a set of **user sessions** (*i.e.,* sequences of HTTP requests) to generate a navigational model of a website. In the generated FTS, the states represents the last user request and the transitions represents the sequence between two requests. Transitions leading to a request identified as part of a plugin or a theme, or as accessible only by an

administrator (using the regular expressions from the previous section) are labelled with the corresponding feature expression.

The n-gram inference method has been proposed by Sprenkle *et al.* [291, 292] and used to test website in a black box fashion. They experiment the inference with different configurations and values of n greater than 2 and found out that a small n allows better diversity in the behaviours (ending up in more diverse test cases), and requires less sessions to reach growth stability of the model. Small n also simplifies the generation and results in a more compact model. This motivates our choice to select 2-gram for the inference.

User sessions: One user session corresponds to a sequence of HTTP requests, representing the sequence of pages consulted by the user. User sessions can be extracted from the Apache webserver log by grouping entries with the same IP address (assuming that one IP corresponds to one user) and logged within a same time frame. This means that two entries in a user session may not be distant by more than a maximal timeout (we arbitrarily choose to set a timeout of 3 minutes). If the timeout is reached, the next entry will be the first of a new user session.

To build the user sessions, we only consider some relevant elements of the HTTP request [292]. This allows to group behaviour of various users to identify common usage scenarios. Amongst the possible group of elements, we considered:

- **Request type** and **Resource** (RR), which uses the type of HTTP request (*e.g.,* POST or GET) and the resource name (*e.g.,* `/index.php`) for a user session element;
- **Request type**, **Resource**, and **parameters Names** (RRN), which also uses the HTTP request type and the name of the resource, but also the name of the parameters in the resource (*e.g.,* `?p=&action=&id=`).

Bigram inference: Using a bigram inference, the next state only depends on the current state of the system. As a consequence, user session entries are considered two by two: the last user request, which is the current state, and the next request of the user, which is the next state of the system.

Algorithm 1 presents the bigram inference of the FTS for a WordPress instance, based on a set of user sessions. The elements of the FTS are initialised at line 2. For each session (line 3), the sequence of requests enriches the model: sessions starts from a virtual state s_0 and the first sequence adds a transition from this state to a new state corresponding to the first element of the sequence (lines 4 to 6); transitions are labelled with actions representing the request of another page (lines 5 and 11); and each new transition is labelled with a feature expression (lines 8 and 14). This feature expression corresponds to the conjunction of the previous feature expression if there is one or true otherwise, and the name of a plugin, a theme, or the administrator user if the requested resource matches one of the corresponding regular expression from section 4.5.1. Function *fLabels* enriches γ definition, based on its previous definition and the given transition. This process iterates for each request in the sequence (line 9), the starting state corresponding to the target state of the previous iteration. Finally, each sequence terminates by a special action

Input: *sessions*: the set of non empty user sessions
Output: *fts*: an FTS representing a navigational model

1 **begin**
2 $S = \{s_0\}$; $Act = \emptyset$; $trans = \emptyset$; $i = s_0$; $\gamma = \rightarrow (\rightarrow \perp)$;
3 **for** ***sess*** $\in$ ***sessions*** **do**
4 *S.add(sess[0])*;
5 *Act.add(req(sess[0]))* ;
6 $tr = s_0 \xrightarrow{req(sess[0])} sess[0]$;
7 *trans.add(tr)*;
8 $\gamma = fLabels(\gamma, tr)$;
9 **for** $i \in [1; sess.size[$ **do**
10 *S.add(sess[i])*;
11 *Act.add(req(sess[i]))*;
12 $tr = sess[i\text{-}1] \xrightarrow{req(sess[i])} sess[i]$;
13 *trans.add(tr)*;
14 $\gamma = fLabels(\gamma, tr, sess[i])$;
15 **end**
16 $Act.add(req(s_0))$;
17 $trans.add(sess[sess.size - 1] \xrightarrow{req(s_0)} s_0)$;
18 **end**
19 $fts = (S, Act, trans, i, fm(sessions), \gamma)$;
20 **return** fts;
21 **end**

Algorithm 1: WordPress bigram FTS building

$req(s_0)$, resetting the system, and ends in the virtual state s_0 (line 17). The algorithm returns an FTS with the inferred navigational model and a feature diagram build using the method described in section 4.5.1 (line 19). We implemented Algorithm 1 in an open source tool: Yet Another Model Inference tool (YAMI), available at https://github.com/xdevroey/yami.

We built four FTSs: two using Request type and Resource (RR) parts of the URLs as sequence elements in the user sessions (**AGE-RR** with 772 states and 6,639 transitions, and **Elsa-RR** with 384 states and 1,214 transitions), and two using Request type, Resource, and parameter Names (RRN) parts of the URLs (**AGE-RRN** with 1,101 states and 10,960 transitions, and **Elsa-RRN** with 615 states and 1,771 transitions). The process took 3 seconds to process the 3,964 sessions of the Elsa models (average session size=9.57, σ=46.73) and 54 second to build the 147,173 sessions of the AGE models (average session size=5.10, σ=61.04) on a Ubuntu Linux machine (Linux version 3.13.0-65-generic, Ubuntu 4.8.2-19ubuntu1) with an Intel Core i3 (3.10GHz) processor and 4GB of memory.

4.6 Claroline, a course management system

The instance of Claroline at University of Namur[1] is the main communication channel between students and lecturers and is used by approximately 7000 users. Students may register to courses and download documents, receive announcements, submit their assignments, perform online exercises, *etc.* Claroline is a configurable system [69]. Contrary to classical SPL, the selection of the features does not occur during the development of the software (at design time in a SPL lifecycle) [252], but during its execution (at runtime). Thus, a product can dynamically evolve while the system is running: this requires the system architecture to be able to accommodate evolutions, by following plugin-based or component-based architectural styles. Thanks to the versatility of the feature concept [64], it is possible to represent design time and runtime product using the same formalism (FM), as product semantics is ultimately given through the mapping with the FTS. In the Claroline case, features represent installation parameters. A product represents a running Claroline instance with a minimal set of data.

4.6.1 Feature model

We manually built the FM from the Claroline documentation and from inspection of a locally installed Claroline instance (Claroline 1.11.7) in approximately 3 days (by one person). The FM in Fig. 4.6 (additional constraints have been omitted) describes Claroline with three main features: *User*, *Course* and *Subscription*. *Subscription* may be open to everyone (*opt OpenSubscription*) and may have a password recovery mechanism (*opt LostPassword*).

User corresponds to the different possible user types provided by default with a basic Claroline installation: unregistered users (*UnregisteredUser*) who may access courses open to everyone and registered users (*RegisteredUser*) who may access different functionalities of the courses according to their privilege level (*Student*, *Teacher* or *Admin*). The last main feature, *Course*, corresponds to the page dedicated to a course where students and teacher may interact.

A course has a status (*Available*, *AvailableFromTo* if the course is available only during a specific period, or *Unavailable*), may be publicly visible (*PublicVisibility*) or not (*MembersVisibility*), may authorize registration to identified users (*AllowedRegistration*) or not (*RegistrationDenied*) and may be accessed by everyone (*FreeAccess*), identified users (*IdentifiedAccess*) or members of the course only (*MembersAccess*). Moreover, a course may have a list of tools (*Tools*) used for different teaching purposes, *e.g.,* an agenda (*opt CourseAgenda*), an announcement panel (*opt CourseAnnoucements*), a document download section where lecturers may post documents and students may download them (*opt CourseDocument*), an online exercise section (*opt CourseExercise*).

Since we are in a testing context, one product of the FM does not represent a complete Claroline instance, but the minimal instance needed to play a test suite. Basically, it maps to a Claroline instance with one particular user and one particular

[1] http://webcampus.unamur.be

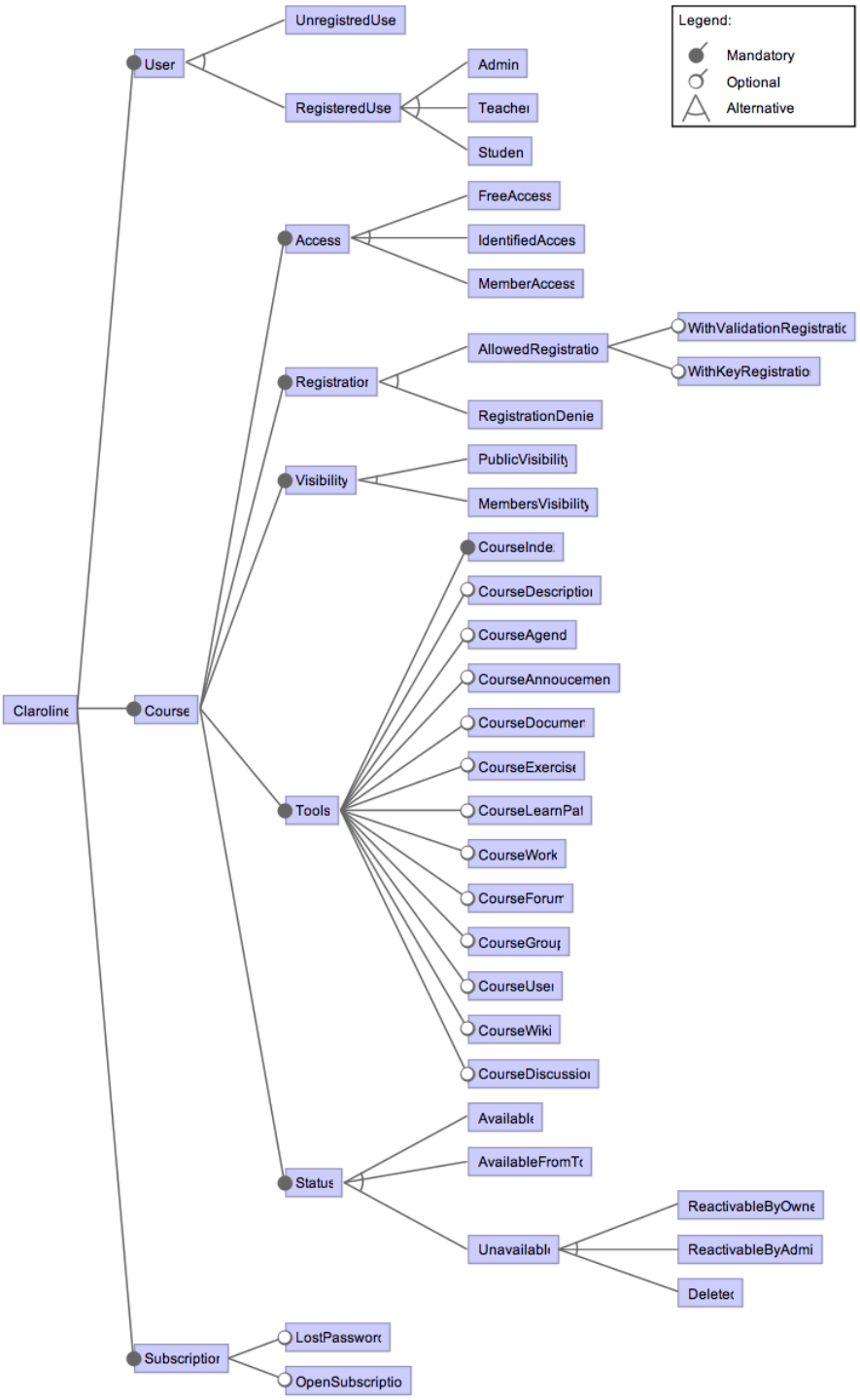


Figure 4.6: Claroline feature model

Table 4.1: Characteristics of the FTSs of the different case studies

Model	States	Trans.	Act.	Avg. deg.	BFS height	Back lvl tr.
S. V. Mach.	9	13	14	1.44	5	3
C. P. Term.	11	17	16	1.54	7	4
Minepump	25	41	23	4.64	15	9
Sferion™	25	46	12	1.84	16	14
AGE-RR	772	6,639	772	8.60	328	408
AGE-RRN	1,101	10,960	1,101	9.96	426	662
Elsa-RR	384	1,214	384	3.16	194	174
Elsa-RRN	615	1,771	615	2.88	369	289
Claroline	106	2,055	106	19.37	1	105

course. This is similar to the technique presented by Segura et al. [278] used to represent the testing entry domain of a e-commerce web site. In order to represent a complete Claroline instance (with all its users and courses), we need to introduce cardinalities [214] on the *User* and *Course* features in order to have multiple users and multiple courses. Eventually we obtained a FM with 44 features.

4.6.2 Featured transition system

Regarding the FTS, we also used a bigram inference method (see section 4.5.2) on a 5.26 Go Apache webserver log with 45,210,987 entries from January 2013 to September 2013. Contrary to the AGE and Elsa models, we only consider the resource names in the user sessions. This simplification is coherent with our testing context where the FM is used to map a Claroline instance with one particular user and one particular course.

Finally, lines 8 and 14 have been omitted in algorithm 1 for the Claroline FTS and transitions have been subsequently tagged manually (in the produced model) with feature expressions based on the knowledge of the system (via the documentation and the local Claroline instance). As for the WordPress models, we added an initial state s_0, but made all states in the FTS accessible from s_0. This allows to simulate a web browser access both to the root page or directly to a sub-page of the website (*e.g.*, from a direct link sent in an email), which is a very common way to access Webcampus. The final FTS consists of 106 states and 2053 transitions and has been built in approximately 4 days (by the author).

4.7 Models characteristics

Table 4.1 details the employed FTS models. For each model, we measure: the number of states (**States**); the number of transitions (**Trans.**); the number of actions (**Act.**); the average degree of the different states that correspond to the average number of incoming/outgoing transitions per state (**Avg. deg.**), computed as the number of transitions divided by the number of states; the maximal number of states between the initial state and another state when traversing the FTS in breadth-first search

Table 4.2: Characteristics of the FMs of the different case studies

Model	Feat.	Common feat.	Mand. feat.	Opt. feat.	Prod.
S. V. Mach.	9	3	2	2	24,000
C. P. Term.	21	4	4	2	4,774
Minepump	9	3	2	4	32,000
Sferion™	25	13	10	2	64,000
AGE-RR	45	1	0	3	4.3980e+12
AGE-RRN	45	1	0	3	4.3980e+12
Elsa-RR	70	1	0	3	1.4757e+20
Elsa-RRN	70	1	0	3	1.4757e+20
Claroline	44	10	9	16	5.4067e+06

(**BFS height**); the number of transitions starting from a state and ending in another state with a lower level when traversing the FTS in breadth-first search (**Back lvl tr.**).

Table 4.2 presents the main characteristics of the FMs of the different cases studies. Those characteristics have been computed using SPLAR [210], the library used by the Software Product Lines Online Tools (SPLOT) [211] to perform its analyses. For each FM, it gives the number of features (**Feat.**), the number of features common to all products (**Common feat.**), the number of mandatory (**Mand. feat.**) and optional (**Opt. feat.**) features in the model, and the number of possible products (**Prod.**) for this FM.

4.8 Additional random LTS models

Additionally to the previously presented case studies, we generated four random LTS models (*Random 1-4*). Those models are used during the assessment of our mutation analysis approaches in Sections 7.5 and 7.6.

In his work, Pelánek [246–248] measured different properties (like the ones from Table 4.2) of real world software system LTSs. The idea behind the procedure we used to generate those LTSs is to mimic those properties:

(i) we generate a set of random graphs (basically directed arcs and nodes) and compute the different measures (as the ones defined in Table 4.1) on them;
(ii) we select those graphs that are likely to represent a real system, *i.e.,* those having a small average degree, a large BFS height and a small number of back level edges (in this order);
(iii) we apply a random labelling multiple times and computed the occurrence probability, *i.e.,* the probability of the labels to obtain a set of randomly generated LTSs;
(iv) we select the LTS that had the following properties: the probability of the most occurring label in the LTS was less than or equal to 6%, and the cumulated probability of the 5 most frequently occurring labels was less than or equal to 20%;
(v) we arbitrarily set the initial state to the first state in the graph.

Table 4.3: Characteristics of the random LTSs

Model	States	Trans.	Act.	Avg. deg.	BFS height	Back lvl tr.
Random 1	10,000	13,652	120	1.37	7,924	3,303
Random 2	15,000	20,488	300	1.37	11,865	4,899
Random 3	15,000	20,488	210	1.37	11,865	4,899
Random 4	15,000	20,488	150	1.37	11,865	4,899

The characteristics of the four generated random models are presented in Table 4.3.

4.9 Threats to validity

The case studies presented in this chapter are used (in Chapter 7) to assess test case selection and mutation analysis. We cannot guarantee that those case studies are representative of all the existing systems. In order to mitigate the validity threats, we choose different kinds of systems, coming from different sources: embedded systems designed by an engineer and Web-based applications where the model has been reverse-engineered from a running instance. The random models were built from a set of generated LTSs in order to match the real system state-space measures.

4.10 Wrap up

In the remainder of this thesis, we consider different case studies, representing different kinds of systems, and coming form different sources. Our largest FTSs represent the usage of **Web applications**: WordPress (AGE-RR, AGE-RRN, Elsa-RR, Elsa-RRN) and Claroline. The other models represent **embedded systems** with a more constrained behaviour. Four additional randomly generated LTS models, that mimic properties of real world system, are used during mutation analysis assessments.

CHAPTER 5

BEHAVIOURAL TEST CASE SELECTION

In Model-Based Testing (MBT), test cases are selected automatically from a partial representation of expected behaviour of the System Under Test (SUT) (*i.e.,* the model). For most systems, it is intractable to select all the possible test cases from the model. The test engineer relies on selection algorithms that maximize a given criterion, a metric one the adequacy of a test suite [206]. In this chapter, we define the notion of **abstract test case** (in Section 5.1), a test case selected from a Featured Transition System (FTS), a mathematical structure to compactly represent the behaviour of a SPL (see Definition 2). Based on this definition, we describe three families of criteria:

(i) **structural selection criteria** (in Section 5.2), adapted from the classical labelled transition systems to FTSs [86];

(ii) **dissimilarity selection criteria** (in Section 5.3), which aims at selecting abstract test cases as dissimilar as possible, both in terms of behaviour and in terms of products covered by the test cases [88];

(iii) and **usage selection criteria** (in Section 5.4), which selects abstract test cases based on the previous usage of the SPL [84, 85].

For each criterion, we give its definition, an abstract test case selection algorithm, and prioritisation methods to prioritize the products of the SPL required to execute the selected abstract test cases.

5.1 Abstract test case over an FTS

In a Model-Based Testing (MBT) approach, test cases are automatically selected from a model of the system under test. This derivation is done in several steps: first, **abstract test cases** are selected from the model, an FTS in our case, using a given criterion; those abstract test cases are then refined, using additional information

in order to be executable by the System Under Test (SUT). This section, and the remainder of this chapter cover the first step: abstract test case selection. Chapter 9 covers the second step: abstract test case concretization.

First, let us define the notion of abstract test case for FTSs. We define an abstract test case over an FTS as a sequence of **actions** from this FTS, such that there exists a sequence of **transitions** in this FTS with the given actions.

Definition 4 (Abstract test case) *Let fts* $= (S, Act, trans, i, d, \gamma)$ *be an FTS. An abstract test case* t *is a finite sequence* $(\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in Act$ *and there exists a sequence of transitions in trans such that*

$$i \xrightarrow{\alpha_1} s_k \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_l$$

For instance, an abstract test case for the soda vending machine SPL described in Section 4.1 may be *(pay, change, soda, serveSoda)*. This abstract test case is a sequence of actions in the FTS (see Figure 4.1(b)) representing a behaviour of this SPL.

5.1.1 Positive and negative abstract test cases

We distinguish two kinds of test cases: **positive test cases** trigger a desired/expected behaviour of the system under test; and **negative test cases** trigger an undesired behaviour of the system under test [153,307]. In our case, a positive abstract test case is defined as a sequence of actions executable by the *fts*, while a negative abstract test case is a sequence of actions not executable by the *fts*. Once concretized, negative abstract test cases typically represent sequences of actions that every product of the product line should forbid.

In a LTS (*lts*), an abstract test case $t = (\alpha_1, \ldots, \alpha_n)$ is executable, denoted $lts \overset{t}{\Longrightarrow}$, if there exists a sequence of transitions starting from the initial state and labelled with $\alpha_1, \ldots, \alpha_n$ [303,304]. For an FTS (*fts*), to be executable, the sequence of transition must moreover have compatible feature expressions. In other words, a sequence of actions is executable by *fts* if there exists at least one product (p) which, when *fts* is projected onto p (denoted $fts_{|p}$), is able to execute it:

$$\left(fts \overset{\alpha_1,\ldots,\alpha_n}{\Longrightarrow}\right) \Leftrightarrow \left(\exists p \in [[d]] : fts_{|p} \overset{\alpha_1,\ldots,\alpha_n}{\Longrightarrow}\right)$$

Example: For instance, the abstract test case *(pay, change, soda, serveSoda, take)* is not executable on the soda vending machine FTS since it mixes both vending machines that offers free drinks and vending machines that do not. Practically, this can be detected in the FTS (and the FM) by detecting incompatible feature expressions on the transitions: $\neg f$ on transitions $s_1 \xrightarrow{pay/\neg f} s_2$ and $s_2 \xrightarrow{change/\neg f} s_3$, and f on transition $s_7 \xrightarrow{take/f} s_1$.

In testing, unlike model checking [27], we only consider finite sequences of actions. Since FTSs (as LTSs) do not have final/accepting state *per se*, in order to decide if a sequence of actions represents a desired behaviour of the system, we

chose to consider the initial state of an FTS as a final state. Positive abstract test cases have to end their execution in the initial state (*e.g.*, state s_1 in the soda vending machine FTS).

Definition 5 (Positive abstract test case) *Let $fts = (S, Act, trans, i, d, \gamma)$ be an FTS. A positive abstract test case $t = (\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n \in Act$, is a finite sequence of actions such as there is at least one product from d able to execute t, and this execution ends in the initial state:*

$$\exists p \in [\![d]\!] : fts_{|p} \stackrel{t}{\Rightarrow} i$$

Definition 6 (Negative abstract test case) *Let $fts = (S, Act, trans, i, d, \gamma)$ be an FTS. A negative abstract test case $t = (\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n \in Act$, is a finite sequence of actions such as for every product from d, the product is not able to execute t or this execution does not end in the initial state:*

$$\forall p \in [\![d]\!] : fts_{|p} \stackrel{t}{\not\Rightarrow} i$$

When derived from the soda vending machine FTS, a positive abstract test case has to start from s_1 and end in s_1 and only fire transitions with compatible feature expressions. For instance, abstract test case *(free, soda, serveSoda, take)* is a positive abstract test case, while *(free, soda, serveSoda, open, take, close)* and *(free, soda, serveSoda)* are negative abstract test cases (the first one fires transitions with incompatible feature expressions and the second one does not end in the initial state when it is executed on the FTS).

In the remainder, we mainly focus on positive abstract test cases and simply write **test case**. A **test suite**, defined for a SUT, is a set of test cases.

5.1.2 Test suite product selection

When abstract test cases are concretized, the result (*i.e.*, concrete test cases, represented as a sequence of operations on the system) has to be executed on one or more products of the SPL. The set of products able to execute a test case may be calculated from the FTS (and the FM). It corresponds to all the products (*i.e.*, set of features) of the FM that satisfy all the feature expressions associated to the transitions fired by the abstract test case when it is executed on the FTS:

Definition 7 (Test case product selection) *Given an FTS $fts = (S, Act, trans, i, d, \gamma)$ and a positive abstract test case $t = (\alpha_1, \ldots, \alpha_n)$ with $(\alpha_1, \ldots, \alpha_n) \in Act$, the set of products able to execute t is defined as:*

$$prod(fts, t) = \{p \in [\![d]\!] \mid fts_{|p} \stackrel{t}{\Longrightarrow} i\}$$

It corresponds to all the products able to execute the sequence of actions in t. From a practical point of view, the set of products contains all the products satisfying the conjunction of the feature expressions $\gamma(s_k \xrightarrow{\alpha_i} s_{k+1})$ on the path(s) of t and the FM d. When d is boolean, it may be transformed to a boolean formula in CNF

where features become variables [76]. The existence of a product for a test case is equivalent to the satisfiability of the following formula, that can be checked by a SAT solver:

$$\bigvee_{pt \in paths} \left(\bigwedge_{i=1}^{n_{pt}} \left(\gamma(s_k \xrightarrow{\alpha_i} s_l) \right) \right) \wedge CNF(d)$$

For instance, the set of products for the test case *(free, soda, serveSoda, take)*, derived from the vending machine FTS, contains all the products of the SPL that offer free soda. Similarly, for a test suite, we have:

Definition 8 (Test suite product selection) *Given an FTS fts* $= (S, Act, trans, i, d, \gamma)$ *and a test suite* $s = \{t_1, \ldots, t_n\}$*, where* $t_1, \ldots, t_n$ *are positive abstract test cases, the set of products able to execute the test suite:*

$$prod(fts, s) = \bigcup_{t_i \in s} prod(fts, t_i)$$

If we have a test suite (s) with two test cases *(free, soda, serveSoda, take)* and *(free, tea, serveTea, take)*, the set of products contains all the products of the SPL that offers free soda or free tea.

We will consider that for a given test suite (s), a set of products (M) is adequate, if M contains enough products to execute the test cases in s:

Definition 9 (s-adequate set of products) *Let fts be an FTS and* $s = \{t_1, \ldots, t_n\}$ *be an abstract test suite where* $t_1, \ldots, t_n$ *are positive abstract test cases. The set of products* M *is s-adequate, denoted* $M \overset{s}{\Longrightarrow}$*, if each test case in s may be executed by at least one product in M:*

$$\forall t \in s : \exists p \in M, fts_{|p} \overset{t}{\Longrightarrow} i$$

Since one of the main concern in SPL testing is to reduce the number of products needed to execute the test, we also define the selection of the minimal s-adequate set of products required to execute a test suite:

Definition 10 (P-Minimal test suite product selection) *Let fts be an FTS and* $s = \{t_1, \ldots, t_n\}$ *be an abstract test suite where* $t_1, \ldots, t_n$ *are positive abstract test cases. A minimal s-adequate set of products needed to execute the test suite, denoted* $mprod(fts, s) = M$*, is a subset of* $prod(fts, s)$ *such that M is s-adequate and there is no subset of M that is s-adequate:*

$$\left(M \overset{s}{\Longrightarrow} \right) \wedge \left(\forall M' \subset M, \left(M' \overset{s}{\not\Longrightarrow} \right) \right)$$

Example (continued): There are two products able to execute all the test cases in the test suite s: one that allows to cancel purchase and one that doesn't. The p-Minimal set of products for s is a set with only one of those two products. The decision of the products to include (or not) should be taken by the test engineer, depending for instance on the cost linked to the derivation of each product.

5.2 Structural selection criteria

In order to efficiently select test cases, the test engineer has to provide **selection criteria** [206, 307]. We redefine hereafter classical structural coverage selection criteria for **connected** FTSs as a function, returning for a given FTS and a test suite, a value between 0 and 1 specifying the coverage degree of the executable abstract test suite over the FTS: 0 meaning no coverage and 1 the maximal coverage.

Definition 11 (Coverage criterion) *A coverage criterion is a function cov that associates an FTS and a test suite over this FTS to a real value in* $[0,1]$.

Classical structural coverage criteria are defined as follow, we illustrate each coverage criteria with test suites satisfying the criteria for the soda vending machine FTS defined in figure 4.1(b):

Definition 12 (State/All-states coverage) *The state coverage criterion is related to the ratio between the states visited by the test cases pertaining to the test suite and all the states of the FTS. When the value of the function equals to* 1*, the test suite satisfies the **all-states coverage**.*

Example (continued): The all-states selection criterion is the weakest structural selection criterion [307], it specifies that when executing the test suite, each state has to be visited at least once. On the soda vending machine, an all-states covering test suite may be:

{ (pay, change, soda, serveSoda, open, take, close);
(free, tea, serveTea, take);
(free, cancel, return) }

Definition 13 (Action/All-actions coverage) *The action coverage criterion is related to the ratio between the actions triggered by the test cases pertaining to the test suite and all the actions of the FTS defined. When the value of the function equals* 1*, the test suite satisfies **all-actions coverage**.*

In this case, a satisfying test suite for a coverage of 1 on the soda vending machine may be the same as the one defined for the all-states coverage.

Definition 14 (Transition/All-transitions coverage) *Transition coverage is related to the ratio between transitions covered when running test cases on the FTS and the total number of transitions of the FTS. When this ratio equals to* 1*, then the test suite satisfies **all-transitions** coverage.*

The all-transitions coverage specifies that, ideally, each transition is fired at least once when executing the abstract test suite on the FTS. Again, the test suite defined using the the all-states coverage criteria satisfies the all-transitions coverage.

Definition 15 (Transition-pair/All-transition-pairs coverage) *The transition-pairs coverage considers adjacent transitions successively entering and leaving a given state. When the coverage function reaches the value of* 1*, then the test suite covers* ***all-transition-pairs****.*

Example (continued): The all-transition-pairs coverage specifies that for each state, each couple of entering/leaving transitions has to be fired at least once. On the soda vending machine, a test suite that covers all-transitions-pairs may be:

{ (pay, change, soda, serveSoda, open, take, close);
(pay, change, cancel, return);
(pay, change, tea, serveTea, open, take, close);
(free, soda, serveSoda, take);
(free, tea, serveTea, take);
(free, cancel, return) }

Definition 16 (Path/All-paths coverage) *Path coverage takes into account simple executable paths (i.e., paths that does not fire the same transition twice), that is sequences of transitions starting from and ending in the initial state. If the coverage function value computing the ratio between the number of simple executable paths covered by the test cases and total number of simple executable paths in the FTS is* 1*,* ***all-paths*** *coverage has been reached.*

The all-path coverage specifies that each simple executable path in the FTS should be followed at least once when executing the test suite. On the soda vending machine, it gives a test suite equal to the one defined for all-transitions-pair coverage.

5.2.1 All-states selection algorithm

We present hereafter a (simple) algorithm to select a test suite satisfying the all-states coverage criteria. This algorithm builds abstract test cases iteratively, using a heuristic based on an accessibility matrix for the FTS. The idea is to start from the initial state with an empty abstract test case and a $true$ feature expression. At each iteration, the algorithm branches out the current partial abstract test case into multiple partial abstract test cases, one per outgoing transition. For each transition, if the feature expression of the transition is compatible, the corresponding action is added to one of the partial abstract test cases. The algorithm then selects the partial abstract test case with the highest score: *i.e.,* the one where the target state of the selected transition may lead to the highest number of states that has not yet been visited by a previously computed abstract test case. The target state becomes then the new current state for the next iteration, and the feature expression associated to the transition is conjoined with the current feature expression.

We present hereafter three algorithms involved in the computation of a all-states covering test suite: the computation of the **accessibility matrix** for a FTS using a variant of the Warshall algorithm [263], the **heuristic** which computes for a given state its score, and finally the **selection algorithm** which computes the test suite satisfying the all-states coverage criterion.

Accessibility matrix computation: The accessibility matrix A gives for two states (s_1, s_2) the products able to execute a non-empty paths from s_1 to s_2. This matrix corresponds to the transitive closure of the FTS and is computed using the Warshall algorithm [263]. Contrary to an accessibility matrix computed for a classical LTS, the entry for s_1 and s_2 (noted $A[1,2]$) is not $true$ or $false$ (*i.e.,* there exists a path from s_1 to s_2 or not), but rather the products for which there exists a path from s_1 to s_2.

In our implementation of the algorithm, as for in ProVeLines [72], the products able to execute a transition tr (noted $\gamma\ tr$) are represented using feature expressions (*i.e.,* boolean expressions over features). An entry of the accessibility matrix A is a feature expression. *E.g.,* for the soda vending machine in Figure 4.1(b), the simplified entry $A[1,4]$ is $(\neg f \vee f) \wedge c$, states that there exists a path in the FTS from s_1 to s_4 for all the products of the SPL having the $cancel$ feature and having or not $free$ drinks.

Input: $fts = (S, Act, trans, i, d, \gamma)$: a connected FTS
Output: A: an accessibility matrix

```
1  begin
2      ∀ s_i, s_j ∈ S : A[i,j] = ⋁_{t=(s_i,α,s_j)∈trans} γ t;
3      for k ∈ [1,#S] do
4          for i ∈ [1,#S] do
5              for j ∈ [1,#S] do
6                  A[i,j] = A[i,j] ∨ (A[i,k] ∧ A[k,j]);
7              end
8          end
9      end
10     return A;
11 end
```

Algorithm 2: Warshall algorithm computing the accessibility matrix of an FTS

Algorithm 2 presents the adaptation of the Warshall algorithm for FTSs. The output of the algorithm is the accessibility matrix A for the given FTS. First A is initialised with the feature expressions conditioning the transition from one state to another (line 2). In the next steps, the matrix is updated by computing all the possible intermediate paths for each pair of states (line 6).

Score computation: Once we have the accessibility matrix, we use a branch and bound algorithm which explores the FTS according to our heuristic. We choose a simple heuristic: it computes a score equal for a given state to the number of states not yet covered by current test suite and accessible from this state.

Algorithm 3 presents the score computation for a given accessibility matrix A and a state s_k. This score is computed dynamically during the selection of an abstract test case by iterating over the k-th line of the accessibility matrix A (line 3). The score is incremented by 1 for every cell corresponding to a not yet reached state (line 5). Before the increment, we verify (at line 4) that the feature expression is compatible with the feature model and the actual partial abstract test case feature expression

```
   Input: A: an accessibility matrix;
   k: the index (in A) of the current state;
   e: a feature expression;
   d: the feature model;
   toVisit: the states not yet covered;
   Output: score: the score associated to s_k
1  begin
2      score = 0;
3      for j ∈ [1;#S] do
4          if s_j ∈ toVisit ∧ SAT(A[k, j] ∧ CNF(d) ∧ e) then
5              score = score + 1;
6          end
7      end
8  end
9  return score;
```

Algorithm 3: Partial abstract test case score computation

e (*i.e.,* there exist one product able to execute the partial abstract test case) using a SAT call and the CNF representation of the feature model d.

All-states covering abstract test suite selection: The all-states test suite selection algorithm is described in Algorithm 4. This algorithm produces an abstract test suite that satisfies the all-states coverage criterion. First the states to visit set (*toVisit*) is initialised to S (line 2), all the states of the FTS. The candidates to consider (*candidates*) are the paths with one transition going out from the initial state i (line 4). Each candidate is a couple with a path (*path*) and a score computed using Algorithm 3. At this stage, the test suite (*testsuite*) is empty (line 7).

The main loop of the algorithm (line 8) computes the abstract test cases and continues as long as states to visit in the FTS remain. In this loop, the best candidate c (with the highest score) is picked (line 9) and removed from the list of candidates to consider.

If the last state of this candidate is the initial sate i, we found an abstract test case (*i.e.,* the sequence of actions on the path) which is added to the test suite if the path contains states not yet visited (line 13). The states of the path are removed from the states to visit (line 14) and the algorithm picks the next candidate at the next iteration.

If the last state reached by the abstract test case is not the initial state, the exploration continues and new candidates are computed. Each outgoing transitions of the last state of the path is added to a new candidate if there exists a product able to execute the new partial abstract test case (checked using a SAT call at line 19) and for each one, a new score is computed (lines 20 and 21).

Simplification for large models: To scale to our largest models, a simplification has been implemented in the algorithm: we ignore the feature expressions and check the validity of a path (*i.e.,* the satisfiability of the feature expression) only before

Input: $fts = (S, Act, trans, i, d, \gamma)$: a connected FTS;
A: the accessibility matrix of fts;
Output: ts: the selected test suite

```
1  begin
2      toVisit = S;
3      candidates = ∅;
4      foreach tr = (i, α, s_k) ∈ trans do
5          candidates = candidates ∪ {((tr), score(A, k, γtr, toVisit))};
6      end
7      testsuite = ∅;
8      while toVisit ≠ ∅ do
9          c = (path, score) ∈ candidates such as score is maximal in candidates;
10         candidates = candidates \{c};
11         if last state of c.path is i then
12             if c.path contains states from toVisit then
13                 add sequence of actions on c.path to testsuite;
14                 remove states on c.path from toVisit;
15             end
16         else
17             foreach transition tr starting from last state of c.path do
18                 fexpr = ⋀_{tr_i ∈ c.path}(γtr_i) ∧ (γtr) ∧ CNF(d);
19                 if SAT(fexpr) then
20                     c' = (c.path++tr, score(A, k, fexpr, toVisit));
21                     candidates = candidates ∪ {c'};
22                 end
23             end
24         end
25     end
26     return testset;
27 end
```

Algorithm 4: All-states test suite selection

adding an abstract test case to the test suite (line 13). Before adding the abstract test case to the test suite and removing the visited states from the $toVisit$ set, we perform a satisfiability call (SAT) on the conjunction of the feature model ($CNF(d)$) and the feature expression of the abstract test case ($fexpr$). This simplification, implemented after our first run of the algorithm, reduces the number of SAT calls which are very costly, but might increase the number of invalid partial abstract test cases, depending on the feature models and the feature expressions in the FTS.

To avoid a explosion of the *candidates* set, the paths and their scores may be represented using a tree structure, where states are nodes and transitions are branches between two nodes. A walk in the tree represents a path in the FTS.

5.2.2 Test case and test suite minimality

Usually, when performing test case selection, one wants to have a test suite as small as possible while ensuring the best coverage. Contrary to single systems where only the size of the test suite is taken into account, when performing SPL testing, we also have to consider the number of products needed to execute the test suite. We define the **size** of a test suite as the number of transitions triggered by its test cases.

Definition 17 (Test suite size) *The size of a test suite s corresponds to the number of transitions triggered in a FTS fts when executing the test cases of s on fts, denoted*

$$fts \overset{s}{\Longrightarrow}$$

This allows to differentiate a test suite s_1 with test cases only triggering a minimal set of transitions to satisfy a coverage criterion from a test suite s_2 also satisfying this coverage criterion, but with longer test cases triggering transitions that do not contribute to the coverage. For a given FTS *fts*, we denote $s_1 < s_2$ if

$$\left(fts \overset{s_1}{\Longrightarrow}\right) < \left(fts \overset{s_2}{\Longrightarrow}\right)$$

As opposed to current practice, the size of the test suite does not take the number of test cases into account. Two test suites with the same size may have different number of test case. This metric is more representative of the behaviour of the SPL covered by a test suite. As for test suites, we define the size of a test case as the number of transitions triggered by this test case.

Definition 18 (Test case size) *The size of a test case t corresponds to the number of transitions triggered in a FTS fts when executing t on fts, denoted*

$$fts \overset{t}{\Longrightarrow}$$

Depending on the product line under test, the test engineer decides if the test suite has to contain lots of small test cases, to ease the debugging process when a test case fails for instance, or few longer test cases, if the setup required to execute each test is expensive for instance.

For such a distribution of test cases sizes in a test suite, the selection process compromises between the size of the test suite and the number of products needed to execute this test suite. We define the former as the **minimal** test suite property, and the latter as the **P-minimal** test suite property.

Property 3 (Minimal test suite) *A test suite s over a given FTS fts = (S, Act, trans, i, d, γ) is minimal w.r.t. a selection criteria cov iff* $\nexists s'$ *such that* $s' < s$ *and* $cov(fts, s') \geq cov(fts, s)$.

Property 4 (P-minimal test suite) *A test suite s over a given FTS fts = (S, Act, trans, i, d, γ) is product-minimal (p-minimal) regarding a selection criteria cov iff* $\nexists s'$ *such that* $(cov(fts, s') \geq cov(fts, s)) \wedge (\#mprod(fts, s') < \#mprod(fts, s))$.

In other words, a test suite is minimal if there exists no smaller test suite with a better coverage, and a p-minimal test suite represents the minimal set of test cases (with the best coverage) such that the number of products needed to execute all of them is minimal.

Example (continued): For instance, the abstract test suite *{(pay, change, soda, serveSoda, open, take, close); (free, tea, serveTea, take);(free, cancel, return) }* is minimal for the all-states-coverage criterion but not p-minimal since it needs at least two different products (*i.e.,* free and not free machines) to be executed on the soda vending machine. A p-minimal abstract test suite satisfying the all-states coverage could be: *{(pay, change, soda, serveSoda, open, take, close); (pay, change, tea, serveTea, open, take, close); (pay, change, cancel, return) }*, which only needs one product to execute the abstract test suite.

5.2.3 Product prioritisation

When designing a test suite using a selection criterion, one of the most interesting products to configure in order to execute the tests is the one that allows to satisfy this criterion as much as possible (for the given test suite). For the structural criteria, the satisfaction level is defined as the structural coverage of the test suite on the FTS.

For a given product, we define the **p-coverage** as the coverage reached by the execution of a test suite on this product and the **p-coverage upper bound** as the product which is able to execute the subset of a test suite with the best coverage.

Definition 19 (P-coverage) *Let s be an abstract test suite over fts, a given FTS, and a covering criterion cov. Given a product $p \in prod(fts, s)$ and $s_p \subseteq s$ the set of all abstract test cases of s executable by p, the p-coverage is the coverage reached when executing s_p :*

$$pcov(fts, s_p) = cov(fts, s_p)$$

The most interesting product to configure first is the one providing the best coverage for the test cases that it can execute:

Property 5 (P-coverage upper bound) *Given a test suite s over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ and a covering criterion cov. Given a product $p \in prod(fts, s)$ and $s_p \subseteq s$ the abstract test suite executable by p, the product p is the p-coverage upper bound iff $\nexists\, s'_p \subset s$ executable by $p' \in prod(fts, s) : p' \neq p$ such that $cov(fts, s'_p) > cov(fts, s_p)$.*

Test suite based product prioritisation uses the p-coverage upper bound to order the products: for a given test suite, the product that can be used to reach the best coverage (*i.e.,* that can execute as many test cases as possible and/or those with the best coverage) are ranked first.

Example (continued): For the soda vending machine SPL, the two p-coverage upper bound products for the all-states p-minimal test suite *{(pay, change, soda, serveSoda, open, take, close); (pay, change, tea, serveTea, open, take, close); (pay, change, cancel, return) }* are the one with all the optional features selected and selling beverage in Euro or Dollar (which are mutually exclusive features).

5.2.4 Related work

Coverage testing for SPL targets both **variability** and on **behavioural** models. Many approaches targeting variability models (mostly feature models) exploit ideas of **CIT** to sample products, yielding product sets even for very large feature models. A more complete description of those approaches may be found in Section 2.4). To compare with our approach, we assess the the benefits of such techniques in terms of **behavioural coverage** in Section 7.3.

At the behavioural level, several techniques have also been proposed. One of those considers incremental testing in the SPL context [167, 197, 233, 309]. For example, Lochau *et al.* [195, 197] proposed a model-based approach that shifts from one product to another by applying *deltas* to statemachine models. These deltas enable automatic reuse and adaptation of the test model and derivation of retest obligations. Oster *et al.* [233] extend combinatorial interaction testing with the possibility to specify a predefined subset of products in the set of products to test. There are also approaches focused on the SPL code by building variability-aware interpreters for various languages [161]. Based on symbolic execution techniques such interpreters are able to run a very large set of products with respect to one given test case [224]. Cichos *et al.* [58] use the notion of 150% test model (*i.e.,* a test model of the behaviour of a product line) and test goal to derive test cases for a product line but do not redefine coverage criteria at the SPL level. At the code level, Li *et al.* [192] focuses on test specification and values reuse from one product to another by using a genetic algorithm that integrates software fault localization techniques and structural coverage of the program.

Finally, Beohar *et al.* [37–39] propose to adapt the $ioco$ framework proposed by Tretmans [303] to FTSs. Contrary to this approach, we do not seek exhaustive testing of an implementation but rather to select relevant abstract test cases based on the criteria provided by the test engineer.

5.3 Dissimilarity selection criteria

Dissimilarity testing is a technique used to select a test suite amongst all possible test cases, which aims to maximise the fault detection rate by increasing diversity among test cases [56, 131]. This diversity is characterised by a **dissimilarity heuristic** defined over the different test cases. For instance, in behavioural model-based testing, one may define a distance between two test cases in a LTS (or an FTS) as the number of actions that differ from one test case to the other. Hemmati *et al.* [131] empirically demonstrate that in single system model-based testing, dissimilar test suites find more faults than similar ones. Mondal *et al.* [218] explored how code

coverage and test case diversity interact in fault-finding abilities of test suites. Results are better for diversity-based test suites, though results are overlapping. The authors conclude that coverage and diversity may complement each other nicely in a multi-objective search-based scenario.

Henard *et al.* [135] applied dissimilarity testing to SPL in order to sample and prioritize products to test. The idea was to mimic the combinatorial interaction testing (CIT) sampling for SPLs [196, 250], in which valid combinations of features are covered at least once. CIT-based sampling for large SPLs raises a computational challenge because of the number of features and constraints involved, forming a large and complex search space. This approach shown good results for large feature models (up to 7000 features) and its relevance has been independently confirmed by Al-Hajjaji *et al.* [8].

Considering this body of knowledge, we combine dissimilarity for SPLs at the product level, that maximises product coverage, with test case dissimilarity, that maximises behaviour coverage, to proceed to a bi-objective test case selection.

5.3.1 Product dissimilarity

Considering a product as a set of feature, dissimilarity between products may be computed using **set-based** distances. To compute the product dissimilarity distance, we choose to use the Jaccard index [149] which shows good results in Henard *et al.* work [135].

Jaccard index product dissimilarity: Giving two sets of products s_1 and s_2, the Jaccard index dissimilarity ($jaccard_p$) is the ratio between the number of products common to s_1 and s_2, and the total number of products in s_1 and s_2:

$$jaccard_p(s_1, s_2) = 1 - \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$$

5.3.2 Actions dissimilarity

The dissimilarity distance between two sequences of actions may be computed using **sequence-based** distances or **set-based** distances (like the Jaccard index) if we assimilate sequences of actions to sets of actions [130, 131]. To select dissimilar test cases, we consider both **set-based** distances (Hamming, Jaccard, dice, and anti-dice distances) and a **sequence-based** distance (Levenshtein or edit distance) [125]. We give hereafter the definitions and the algorithms used to compute the different dissimilarity measures, based on the considered distances.

Hamming actions dissimilarity: Hamming distance [273] is used as a basic **sequence-based** edit-distance between two sequences with the same length. It is defined as the minimum number of edit operations needed to transform the first sequence into the second. In most cases, the sizes of the considered sequences differ [130, 131]. In such case, the Hamming distance may be used as a **set-based** distance by building two binary vectors (one per sequence) indicating for each

sequence which elements amongst all the possible elements are in the sequences and which are not.

For instance, for two sequences of actions $seq_1 = (\alpha_1, \alpha_2, \alpha_4)$ and $seq_1 = (\alpha_1, \alpha_2, \alpha_3)$ and a set of possible actions $Act = \{\alpha_1, \ldots, \alpha_5\}$, the binary vectors is $v_1 = [11010]$ for seq_1 and $v_2 = [11100]$ for seq_2, and the Hamming distance equals to $2/5$. To transform the similarity measure to a dissimilarity measure, we subtract the Hamming distance from 1 and define Hamming dissimilarity as:

$$hamming_a(seq_1, seq_2, Act) = 1 - \frac{|seq_1 \cap seq_2| + |Act \setminus seq_1 \setminus seq_2|}{|Act|}$$

Jaccard actions dissimilarity: As for product dissimilarity, the Jaccard actions dissimilarity between two sequences of actions seq_1 and seq_2 is the ratio between the number of different actions common to the two sequences and the total number of different actions in the two sequences:

$$jaccard_a(seq_1, seq_2) = 1 - \frac{|seq_1 \cap seq_2|}{|seq_1 \cup seq_2|}$$

Dice and anti-dice actions dissimilarity: Dice (Gower-Legendre) and anti-dice (Sokal-Sneath) are **set-based** distances [130, 131], using a generalisation of the Jaccard index formula. For two sequences seq_1 and seq_2, considered here again as sets of actions, the general formula is:

$$jaccard_a(seq_1, seq_2, w) = 1 - \frac{|seq_1 \cap seq_2|}{|seq_1 \cup seq_2| + w * (|seq_1 \cup seq_2| - |seq_1 \cap seq_2|)}$$

The dice dissimilarity is obtains by fixing the w parameter to 0.5 and the anti-dice dissimilarity is obtained by fixing w to 2.0:

$$dice_a(seq_1, seq_2) = jaccard_a(seq_1, seq_2, 0.5)$$

$$antidice_a(seq_1, seq_2) = jaccard_a(seq_1, seq_2, 2.0)$$

Levenshtein actions dissimilarity: The Levenshtein distance [186], also called edit distance, between two sequences of actions indicates the number of insertion, deletion and replacement operations to perform on the first sequence to obtain the second one. This number, divided by the maximal length between the two sequences, gives us the Levenshtein actions dissimilarity measure ($levenshtein_a$).

Algorithm 5 presents the Levenshtein dissimilarity computation using a classical dynamic programming approach (we consider that the insertion, deletion, and replacement costs equal 1). If the two sequences are the same (line 2), the dissimilarity is equal to 0. If only one of the two sequences is empty (line 5), the dissimilarity is 1. In the general case, the algorithm uses two tables (line 8) to compute the current edit distance and memoize the previous iteration of the algorithm (initialised at line 9). At each iteration i for $i \in [0; seq_1.size[$ (line 12), table *current* is updated based on *previous* such as $\forall j \in [0; seq_2.size[$, value in $current[j+1]$ represents

```
   Input: seq1, seq2: two sequences
   Output: Levenshtein dissimilarity measure
1  begin
2      if seq1 == seq2 then
3          return 0 ;
4      end
5      if seq1 or seq2 are empty then
6          return 1 ;
7      end
8      int[seq2.size + 1] previous; int[seq2.size + 1] current;
9      for i ∈ [0; seq2.size + 1[ do
10         previous[i] = i;
11     end
12     for i ∈ [0; seq1.size[ do
13         current[0] = (i + 1);
14         for j ∈ [0; seq2.size[ do
15             int cost = seq1[i] == seq2[j] ? 1 : 0;
16             current[j + 1] = min(current[j], previous[j + 1], previous[j] ∗ cost);
17         end
18         current = previous.copy() ;
19     end
20     return current[seq2.size]/max(seq1.size, seq2.size);
21 end
```

Algorithm 5: Levenshtein dissimilarity computation

the edit distance between the subsequences $seq_1[0..i]$ and $seq_j[0..j]$. Value in *current* is updated (line 16) by taking the minimal cost between deleting, inserting, or substituting (if needed) a letter at position j in seq_2 to align it on seq_1.

5.3.3 Random test case selection

In our previous work [91] we presented a first implementation of a random test case selection algorithm. This algorithm was not optimal as it performs validation *a posteriori*: it first generates a sequence of actions using the FTS without considering feature expressions and verifies afterwards that this sequence may be executed by a valid product of the product line using a SAT solver. We improved this algorithm to directly consider sequences of actions executable by a valid product.

Algorithm 6, takes as input a FTS (without deadlock) and produces a random test case executable by at least one product of the SPL. The algorithm loops while we are not back to the initial state (line 5). At each iteration, a transition is selected such as if its associated action is added to the partial test case, at least one product of the product line may execute it (line 8), the action of this transition is then added to the partial test case (line 9) and the algorithm moves to the next state (line 10).

Input: $fts = (S, Act, trans, i, d, \gamma)$: a connected FTS
Output: A random positive abstract test case
Data: $t, candidate$: test case; tr: transition; $next$: state

```
1  begin
2      t = ();
3      tr = null;
4      next = i;
5      while (tr = null) ∨ (tr.s_j ≠ i) do
6          candidate = t.copy();
7          tr = random({next --α--> s_j ∈ fts.trans |
8              prod(fts, candidate.add(α)) ≠ ∅});
9          t = t.add(tr.α);
10         next = tr.s_j;
11     end
12     return t;
13 end
```

Algorithm 6: Random positive abstract test case selection

5.3.4 Bi-objective test suite selection

Our bi-objective test case selection [88] tries to maximise both the product and the behaviour coverage of a test suite. To do so, it computes a **dissimilarity distance** between test cases based on the products able to execute each test case (using the FTS) and the actions appearing in those test cases. Two test cases are dissimilar (distance equals 1) if they do not share the same actions and they may be executed on dissimilar products. Formally, we define the dissimilarity between two test cases as follows:

Definition 20 (Test cases dissimilarity) *Given an FTS fts and two test cases* $t_1 = (\alpha_1, \ldots, \alpha_n)$ *and* $t_2 = (\beta_1, \ldots, \beta_n)$ *derived from fts, the dissimilarity between* t_1 *and* t_2 *is defined as:*

$$diss(fts, t_1, t_2) = diss_p(prod(fts, t_1), prod(fts, t_2)) \otimes diss_a((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$$

Where $diss_p : [[d]] \times [[d]] \rightarrow [0, 1.0]$ *computes a dissimilarity distance between the products,* $diss_a : Act^+ \times Act^+ \rightarrow [0, 1.0]$ *computes a dissimilarity distance between the actions of the test cases, and* $\otimes : [0, 1.0] \times [0, 1.0] \rightarrow [0, 1.0]$ *is an operator combining the products and actions distances to return a global dissimilarity distance between the two test cases.*

In our evaluation in Section 7.2, we use the multiplication ($\times$) and average (avg) operators for $\otimes$. The multiplication operator considers that two test cases are highly dissimilar only if both their actions and products are dissimilar. The average operator soften this constraint: two test cases with a high dissimilarity value only for their products or only for their actions are also considered as dissimilar (although less dissimilar than two test cases with highly dissimilar actions and products).

Input: fts: a connected FTS;
k: the number of test cases;
d: the duration
Output: A test suite s maximizing the fitness value return by $fit()$

```
1  begin
2      s =< >;
3      for i ∈ [0; k[ do
4          s.append(random(fts));
5      end
6      start = time();
7      while time() < start + d do
8          sort(s);
9          candidate = s.copy() ;
10         candidate.removeLast() ;
11         candidate.append(random(fts));
12         if fit(fts, candidate) > fit(fts, s) then
13             s = candidate;
14         end
15     end
16     return s;
17 end
```

Algorithm 7: Search-based dissimilarity selection

Bi-objective test suite selection may be formulated as a search-based problem: given an FTS, amongst all the possible test cases, select a test suite such that the dissimilarity between the test cases in this test suite is maximal. To efficiently explore the search space, we use a meta-heuristic: a (1+1) evolutionary algorithm without mutation nor crossover [95, 135]. This algorithm selects, for a given FTS, a time budget (d), and a number of test cases (k), a test suite with k test cases with the objective to maximise the dissimilarity. Test suites are characterized using a **fitness function** which associates a score to a test suite, denoting the dissimilarity degree between the test cases. This fitness function is defined based on a dissimilarity distance:

Definition 21 (Fitness function) *Given a dissimilarity diss and a test suite* $(t_1, \ldots, t_k)$ *selected from a FTS fts, the fitness function* $fit : FTS \times Act^+ \times \ldots \times Act^+ \rightarrow \mathbb{R}_+$ *is defined as the sum of the distances between the different test cases:*

$$fit : (fts, t_1, \ldots, t_k) \mapsto \sum_{j>i}^{k} diss(fts, t_i, t_j)$$

Selection algorithm: Algorithm 7 presents the bi-objective (1+1) without mutation nor crossover evolutionary algorithm we use to select a dissimilar test suite. First, the algorithm initialises a random sequence of test cases (line 3) with k elements; this set is then improved in order to maximise the fitness value during a given time d (line 7).

At each iteration of the main loop, the current set is sorted using the dissimilarity distance between test cases (line 8). Sorting is performed by considering either **local distances** or **global distances**. For local distances, it computes the distances between every pair of test cases: $\forall i, j \in [0; k[, dist[i, j] = diss(fts, s[i], s[j])$. It then selects the pair of test cases such that $dist[i, j]$ is maximal and put them at the beginning of the list. This process loops until all elements of the set have been processed to give a list of test cases $(t_a, t_b, t_c, t_d, t_e, t_f, \ldots)$ where $dist[a, b] \geq dist[c, d] \geq dist[e, f] \geq \ldots$

The global distance works in the same way: first a pair of test cases as dissimilar as possible is selected; then test cases are added in such a way that the dissimilarity with the previously selected test cases is maximal. This gives us a list of test cases $(t_a, t_b, t_c, \ldots)$ where, $\forall x \neq b$, $dist[a, b] \geq dist[a, x]$; and $\forall x \neq b, c$, $(dist[a, c] + dist[b, c]) \geq (dist[a, x] + dist[b, x])$; *etc.*[1]

At the end of the sorting algorithm, the most dissimilar (global or local) elements are at the beginning of the list s. The next step is to replace the last element of this list by a new random test case (line 10). If the fitness value of this new set (*candidate*) is better than the previous one, this candidate becomes the new set of test cases (line 13). Hemmati *et al.* evaluated 320 similarity scenarios, including those where a new test case is not randomly selected and found this (1+1) strategy to be cost-effective [131].

5.3.5 Product prioritisation

As for Henard *et al.* [135], our search-based dissimilarity selection algorithm (locally or globally) ranks the test cases, depending on the configuration of the algorithm. For a test suite selected using this algorithm, we have $s = (t_1, t_2, \ldots, t_n)$ with t_1 more dissimilar than t_2 (for selections made with the local distance) or more dissimilar that $(t_2, \ldots, t_n)$ (for selections made with the global distance). Like structural based product selection (see Section 5.2.3), the most interesting product to configure to execute the product able to execute the longest sequence of $(t_1, \ldots, t_k)$, with $k \leq n$, such that the dissimilarity criterion is satisfied as much as possible.

5.3.6 Related work

To the best of our knowledge, our bi-objective approach is the first to consider dissimilarity for behaviour and product in SPL testing, since previous research on the topic has solely focused on sampling dissimilar products of the feature model: Henard *et al.* [135] defined a product sampling approach based on selected features dissimilarity to mimic the t-wise coverage for large systems and high values of t. This approach has been shown effective also on smaller systems (with fewer products) by Al-Hajjaji *et al.* [8]. Parejo *et al.* [242] extended this idea, using a genetic algorithm, by considering bot functional and non-functional properties during the selection process. All these approaches are based on the pioneering works on dissimilarity testing at the code level (*e.g.,* [130, 218]).

[1] For more information about local and global sorting, the interested reader may refer to [135].

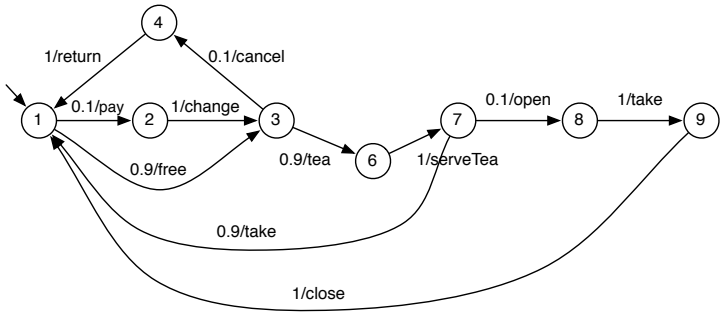


Figure 5.1: Soda vending machine usage model

5.4 Usage selection criteria

Contrary to structural and dissimilarity based methods that use the structure of the FTS model in order to select a test suite satisfying respectively a coverage or dissimilarity criterion, we propose to use the actual behaviour of the running products of the SPL to select (and subsequently prioritize) test cases [84, 85]. Our work is based on statistical testing [316], which eventually selects test cases from a **usage model** represented by a Discrete-Time Markov Chain (DTMC).

Definition 22 (Usage Model (UM) [316]) *A usage model is equivalent to a DTMC where transitions are additionally decorated with actions, i.e., a tuple* $(S, Act, trans, P, i)$ *where :*

- $(S, Act, trans)$ *are respectively a set of states, a set of actions, and a set of transitions* $(trans \subseteq S \times Act \times S)$*;*
- $P : trans \rightarrow [0,1]$ *is the probability function that associates each transition* (s_i, α, s_j) *the probability for the system in state* s_i *to execute action* α *and reach state* s_j*;*
- $i \in S$ *is the initial state;*
- $\forall s_i \in S : \sum_{\alpha \in Act, s_j \in S \bullet s_i \xrightarrow{\alpha} s_j} P(s_i \xrightarrow{\alpha} s_j) = 1$*, that is, the total of the probabilities of the transitions leaving a state must be equal to* 1.

Note that in their original definition [316], DTMCs have no actions. We need them here to relate transitions in a DTMC with their counterpart in an FTS. Also, we consider that there is a single initial state i.

For instance, Figure 5.1 presents a usage model of the soda vending machine SPL described in section 4.1: each transition has been tagged with a probability which represents, amongst all the possible products of the SPL, the probability of the transition to be fired. As we base our usage model on the actual usage of the product line, some states and transitions may be missing in the usage model if the behaviour linked to those states and transitions has not been observed in any product of the SPL. This corresponds to transitions with a probability equal to 0 and states with only input transitions with a probability equal to 0. In our example, we have no

vending machine serving soda. Transitions with actions *soda* and *serveSoda* and state 5 have been removed from the usage model presented in Figure 5.1.

The usage model is agnostic to the variability inherent to the SPL. It only represents the usage scenarios of the SPL under test as well as their respective probability and serves as basis to select test cases. There are two ways to associate usage scenarios to SPL products:

- the **family-based** approach [84, 85] consists in exploiting logs as a source of user information and infer the usage model using machine learning techniques. We can then extract behaviour according to a probability range and relate them to the FTS and feature models of the SPL. As they are extracted from the usage model, behaviours can be run on the FTS to determine which products/features are involved. Some behaviours may also correspond to no product and thus no be executable. Those behaviours are errors coming from the usage model inference or indicators of errors in the FTS. Either case, they can be reported to the test engineer to correct the models.
- the **product-based** approach developed by Samih *et al.* [266, 268] consists in creating usage models from the onset, taking into account requirements and feature models as well as translating expert knowledge to probabilities in the usage model. Concretely, requirements are related to features via a product matrix, while the usage model directly relates its transitions to probabilities and requirements of the SPL. Then, testers have to manually specify the products they are interested in and, with the help of the MaTeLo tool [9, 267], derive a pruned usage model corresponding to the behaviour of these products and perform automated test case selection.

5.4.1 Product-based test selection

Product-based test selection is straightforward: the test engineer selects one product to test by selecting features in the feature model, the tool then automatically use the projection operator on FTS to extract a LTS corresponding to the product, and prunes the usage model accordingly. The probabilities of the removed transitions are proportionally distributed on adjacent transitions, so that the probability axiom $\forall s_i \in S : \sum_{s_j \in S, \alpha_k \in Act} P(s_i, \alpha_k, s_j) = 1$ holds and balance between the probabilities of the transitions from a same source state are kept [268]. Finally, the tool selects test cases using statistical testing algorithms on the usage model [104, 316].

This scenario is proposed by Samih *et al.* [266, 268] in the MaTeLo Product Line Manager (MPLM) tool [267]. Product selection is made on an orthogonal variability model (OVM) and mapping between the OVM and the usage model (build by a system expert using MaTeLo [9]) is provided via explicit traceability links to functional requirements. This process requires to perform the selection of the product of interest on the variability model and does not exploit the usage model during this selection.

5.4.2 Family-based test selection

Contrary to product-based test selection, family-based test selection supports partial coverage of the SPL by the usage model (like the soda vending machine usage model presented in Figure 5.1). The key idea is to select abstract test cases (*i.e.,* sequences of actions, not necessarily executable by one product of the SPL) from the usage model according to their probability to happen using an interval given by the engineer. Only abstract test cases from the model with a probability in this interval are considered. *E.g.,* one may be interested in analysing highly probable behaviours (interval $[0.5, 1]$). Only one abstract test case has a probability in this range in the soda vending machine usage model: $Pr(free, tea, serveTea, take) = 0.729$, which corresponds to the behaviour "serving tea for free". The selected abstract test cases are filtered using the FTS in order to keep only positive abstract test cases (executable by at least one product of the SPL) and a pruned FTS (FTS'). The FTS' represents the minimal behaviour of the FTS needed to execute the positive abstract test cases, it is used latter to prioritize the products to test (see section 5.4.3).

Abstract test case selection from the usage model: The first step is to extract abstract test cases from the usage model according to the desired parameters. To perform abstract test case selection in a usage model $dtmc$, we apply an all-paths algorithm parametrized with a maximum length l_{max} for finite abstract test cases and an interval $[Pr_{min}, Pr_{max}]$ specifying the minimal and maximal values for the probabilities of selected abstract test cases. Formally:

$$\begin{aligned} allpaths(l_{max}, Pr_{min}, Pr_{max}, dtmc) = \{(i, \alpha_1, ..., \alpha_n, i) \mid \\ n < lmax \wedge (\nexists k : 0 < k < n, i = s_k) \\ \wedge (Pr_{min} \leq Pr(i, \alpha_1, ..., \alpha_n, i) \leq Pr_{max})\} \end{aligned}$$

where

$$Pr(i, \alpha_0, \ldots, s_n) = \prod_{j=0}^{n-1} P_{dtmc}(s_j, \alpha_j, s_{j+1}).$$

We initially consider only (finite) abstract test cases starting from and ending in the initial state i (assimilated to an accepting state) without passing by i in between. These abstract test cases correspond to a coherent execution scenario in the usage model. The l_{max} bound allows the algorithm to scale to large usage models [85].

The interval $[Pr_{min}, Pr_{max}]$ is provided by the engineer based on his knowledge of the SPL and the selection purpose: an interval with high values (*e.g.,* $[0.5, 1]$) gives highly probable behaviours of the SPL. This is often desired for a non-regression testing scenario where the engineer wants to ensure that the main functionalities of a SPL are still reliable after an update [206]. The engineer may also be interested in testing behaviours with a low probability as they may find rare bugs not discovered by the users of the products. Such strategies can be used, *e.g.,* for intrusion detection [110].

The interval, its relevance, and selected test cases depend on the usage model source (*e.g.,* built from running products, manually built by an engineer, *etc.*) and

the usage model shape (*i.e.,* number of states, transitions, average states degree, *etc.*). To have an idea of the interval to choose and the number of abstract test cases that are selected, the engineer may use random walks in the usage model to select random abstract test cases with their probability and see how they are distributed.

Practically, this algorithm builds an exploration tree where each node represents the exploration of a state. The exploration of a branch of the tree is stopped when the depth is higher than l_{max}. This parameter is provided to the algorithm by the test engineer and is used to avoid infinite loops during the exploration of the usage model.

For instance, the execution of the algorithm on the soda vending machine example (um_{svm}) presented in Figure 5.1, with a l_{max} value of 7 (the size of the maximal simple path) and an interval $[0, 0.1]$ to capture the least probable abstract test cases, gives 5 finite abstract test cases:

$$
\begin{aligned}
&allpaths(7;0;0.1;um_{svm}) = \{\\
&(pay, change, cancel, return); (free, cancel, return);\\
&(pay, change, tea, serveTea, open, take, close);\\
&(pay, change, tea, serveTea, take);\\
&(free, tea, serveTea, open, take, close)\}
\end{aligned}
$$

During the execution of the algorithm, the abstract test case *(free, tea, serveTea, take)* has been rejected since its probability (0.729) is not between 0 and 0.1.

The downside is that the algorithm may possibly enumerate all the paths in the usage model depending on the l_{max} value. This can be problematic and we plan in our future work to use symbolic executions techniques inspired by work in the probabilistic model checking area, especially automata-based representations [65] in order to avoid exploring all paths.

FTS-based abstract test case filtering and FTS pruning: We do not make any assumptions about the source of the usage model. Therefore, this step serves as a sanity check to ensure that selected traces correspond to behaviour that may be executed by at least one valid product of the SPL. If this is not the case, there is an error in the usage model or in the FTS. Depending on how the model has been built, the error may come from the engineer (*e.g.,* missing transition/state, extra transition/state, wrong feature expression on a transition of the FTS, *etc.* during the modelling) or from the model inference method used to generate the model from a set of execution traces. Such errors have to be detected and reported to the engineer who has to decide what to do: either correct the usage model or the FTS in order to avoid illegal behaviours; or ignore the error if it is not significant.

The idea to filter abstract test cases and keep only positive abstract test cases is to use the FTS to detect negative abstract test cases by running them on it. Practically, we build a second **FTS** which represents only the behaviour of the SPL appearing in the positive abstract test cases selected from the usage model. This FTS' represents a prioritized subset of the original FTS [86].

Input: $fts = (S, Act, trans, i, d, \gamma)$: a connected FTS;
s: the set of abstract test cases to filter
Output: fts', an FTS representing fts restricted to the behaviour in s, and s', the set of positive abstract test cases from s

```
1  begin
2      S' = {i}; Act' = ∅; trans' = ∅; i' = i; d' = d; γ' = ∅;
3      s' = s;
4      for t ∈ traces do
5          if ∃ s_k ∈ S, fts =t⇒ s_k then
6              Act' = Act' ∪ t;
7              S' = S' ∪ states(fts, t);
8              trans' = trans' ∪ transitions(fts, t);
9              γ' = fLabels(fts, t)γ';
10         else
11             s' = s' \ {t};
12         end
13     end
14     fts' = (S', Act', trans', i', d', γ');
15     return (fts', s');
16 end
```

Algorithm 8: FTS' building and positive abstract test cases filtering

Algorithm 8 presents how to build the FTS' (fts') from a set of abstract test cases (s), with positive abstract test cases filtered during the algorithm (into s'), and a given FTS (fts). The initial state of fts' corresponds to the initial state of the fts (line 2) and d in fts' is the same as for fts (line 2). For each abstract test case, if it is executable on fts (line 5), then the states ($states(fts, t)$), actions (t) and transitions ($transitions(fts, t)$) visited in fts when executing the trace t are added to fts' (line 6 to 8). On line 9, the $fLabels(fts, t)$ function is used to enrich the γ' function with the feature expressions of the transitions visited when executing t on the fts. It has the following signature:

$$fLabels : (FTS, Act^*) \rightarrow (trans \mapsto [\![d]\!] \mapsto \mathbb{B}) \rightarrow (trans \mapsto [\![d]\!] \mapsto \mathbb{B})$$

On line 9, $fLabels(fts, t)\gamma_{fts'}$ returns a new function $\gamma'_{fts'}$ which, for a given transition $tr = (s_i \xrightarrow{\alpha_k} s_j)$, returns $\gamma_{fts}(tr)$ if $\alpha_k \in t$ or $\gamma_{fts'}(tr)$ otherwise.

In our example, the set of finite traces with a probability between 0 and 0.1 selected in step 1 contains two negative abstract test cases: *(pay, change, tea, serveTea, take)* and *(free, tea, serveTea, open, take, close)*, which both lead to an execution condition containing the $free \wedge \neg free$ feature expression. Those 2 negative abstract test cases (mixing free and not free vending machines) cannot be executed on the soda vending machine FTS (of Figure 4.1(b)) and are rejected by Algorithm 8. The generated FTS' is presented in Figure 5.2.

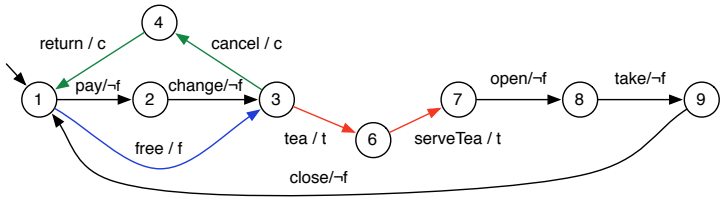


Figure 5.2: Soda vending machine FTS'

5.4.3 Product prioritisation

Product-based test selection assumes that relevant products from which test cases are selected have already been prioritized. Relevant products are used to prune the usage model (in order to keep only the behaviour of the selected product) and test cases are derived using a statistical testing algorithm.

When performing a family-based test selection from a usage model, relevant abstract test cases are directly selected from the usage model representing the behaviour of the SPL. Those abstract test cases are then filtered to keep only positive abstract test cases. At the end of algorithm 8, we have an FTS' and a set of (positive abstract) test cases. This set of test cases covers all states and transitions of the FTS'. Since they come from the usage model, it is possible to order them according to their probability to happen. This probability corresponds to the the cumulated individual probabilities of the transitions fired when executing the finite trace in the usage model. A test case $t = (i, \alpha_1, \ldots, \alpha_n, i)$ corresponding to a path $(i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} i)$ in the usage model has a probability $Pr(i, \alpha_1, \ldots, \alpha_n, i)$ to be executed.

At this step, each test case t is associated to the set of products $prod(t, fts')$ that can actually execute t. Product prioritisation may be done by sorting the test cases according to their probability to be executed, giving a set of products for each test case t.

For instance, for the positive abstract test case t = *(pay, change, tea, serveTea, open, take, close)*, selected for our example, the products have to satisfy:

$$\neg f \wedge t \wedge CNF(d)$$

Where d is the feature model of the soda vending machine (presented in Figure 4.1(a)), transformed into a boolean formula using the *CNF* function. This gives us a set of 8 products (amongst 32 possible):

$$\{(v, b, cur, t, eur); (v, b, cur, t, usd); (v, b, cur, t, c, eur);$$
$$(v, b, cur, t, c, usd); (v, b, cur, t, s, eur); (v, b, cur, t, s, usd);$$
$$(v, b, cur, t, s, c, eur); (v, b, cur, t, s, c, usd)\}$$

Each of them executing t, which is the behaviour of the soda vending machine product line with the lowest probability ($Pr(t) = 0.009$ in the usage model).

5.4.4 Related work

To the best of our knowledge, there is no approach prioritizing behaviours statistically for testing SPLs in a family-based manner. There have been SPL test efforts to sample products for testing such as t-wise approaches [69, 70, 155, 250]. More recently sampling was combined with prioritisation thanks to the addition of weights on feature models and the definition of multiple objectives [135, 156]. However, these approaches do not consider SPL behaviour in their analyses.

Efforts to combine sampling techniques with modelling ones (*e.g.,* [196]) exist. These approaches are product-based, meaning that they may miss opportunities to reuse tests amongst sampled products [315]. We believe that benefiting from the recent advances in behavioural modelling provided by the model checking community [24, 25, 63, 65, 105, 180, 262, 298], sound MBT approaches for SPL can be derived and interesting scenarios combining verification and testing can be devised.

To consider behaviour in an abstract way, a full-fledged MBT approach [307] is required. Although behavioural MBT is well established for single-system testing [303], a survey [234] shows insufficient support for SPL-based MBT. Metzger and Pohl further emphasizes the need for inter-model consistency and minimizing test redundancy across the lifecycle (domain and application engineering) [213]. We believe that the FTS formalism, natively equipped with features as a first-class concept, is pivotal to inter-model verification support and supports combination of quality assurance techniques both at the domain and application engineering levels as our integration between family-based and product-based statistical test selection illustrates.

Our will is to apply ideas stemming from statistical testing [292] and adapt them in an SPL context. For example, combining structural criteria with statistical testing has been discussed by Gouraud *et al.* [117] and Thévenod-Fosse and Waeselynck [300]. We do not make any assumption on the way the usage model is obtained: via an operational profile [220], by analysing the source code or the specification [300], or from the running application logs [114, 292]. In the absence of a source of information for the usage model, one could think of a uniform distribution of probabilities over the usage model. As noted by Whittaker [316], in such case, only the structure of abstract test cases would be considered and therefore basing their selection on their probabilities would just be a means to limit their number in a mainly random testing approach. In such cases, it is better to use structural test selection [104].

We use MaTeLo [9] to select test cases from a product model. Other tools like JUMBL [255] would have qualified. Both are model-based statistical testing tools, supporting the development of statistical usage models using Markov chains, the analysis of models, and the selection of test cases [308]. However, none of them are able to natively handle SPL models. We use the MaTeLo Product Line Manager (MPLM) tool [266, 268] to generate models for a product of the SPL, which are then used to select test cases.

5.5 Wrap up

In this chapter, we described three abstract test case selection and prioritisation approaches: structural, dissimilarity, and usage based test cases selection criteria.

Structural selection criteria: Structural selection and prioritisation criteria are based on the structure of the FTS representing the product line to test. Those criteria consider the number of states, actions, transitions, transitions-pairs, or paths covered in the FTS to guide the abstract test case selection. Once selected, the prioritisation of the products to test is done according to the abstract test cases executable by those products: products that can reach the best coverage by executing as many test cases as possible and/or those with the best coverage are ranked first.

Dissimilarity selection criteria: Dissimilarity selection criteria aim at selecting abstract test cases as diverse as possible. The dissimilarity is defined by the test engineer by combining basic distance functions taking the actions and products covered by the test cases as input. We used a (1+1) without mutation nor crossover evolutionary algorithm to select the test cases, based on a given time budget and a given size of the test suite.

Usage selection criteria: Usage-based selection criteria may be used in two ways: family-based and product-based test selection and prioritisation. Family-based selection and prioritisation extracts products of interest according to the probability of their execution traces gathered in a usage model. We thus select a subset of the full SPL behaviour given as a FTS. This allows us to construct a new FTS, FTS', representing only the executions of relevant products. This FTS' can be analysed all at once to enable test reuse amongst products to scale during testing activities. Product-based selection and prioritisation requires the testers to select a product of interest before the usage model is pruned, leaving only its executions associated to it [266–268]. Though these approaches may seem antagonistic, family-based prioritisation can gracefully complement the product-based one by suggesting products of interest.

CHAPTER 6

MUTATION ANALYSIS

Mutation analysis is an established technique to either evaluate test suite effectiveness [13, 115, 230] or support test case selection [108, 230, 239]. It works by injecting artificial defects, called **mutations**, into the code or the model under test, yielding **mutants**, and measures test effectiveness based on the number of detected mutants.

Researchers have provided evidence that detecting mutants results in finding real faults [13, 158] and that test cases designed to detect mutants reveal more faults than other test case selection criteria [29, 230]. This has been shown to be the case for model-based mutation too [35]: Aichernig *et al.* [2] report that model mutants lead to test cases that are able to reveal implementation faults that were neither found by manually defined test cases, nor by the actual operation, of an industrial system. In addition, model-based mutation's premise is to identify defects related to missing functionality and misinterpreted specifications [49]. This is desirable since code-based testing fails to identify these kinds of defects [146, 314].

Despite its advantages and the advances made by the research community, mutation analysis still faces important issues [152]:

(i) due to the large number of mutants that needs to be generated and **executed** by the test cases, mutation analysis may be expensive. While this problem has been investigated for code-based mutation [157, 240], it remains open in the model-based context. Since typical real-word models involve thousands of mutants and test suites involve thousands of test cases, millions of executions are needed. Addressing this problem is therefore vital for the scalability of mutation analysis [152, 230];

(ii) in order to generate mutants that denote subtle faults, Jia and Harman [151] propose to use **higher order** mutants. A higher order mutant is the results of a mutated mutant, *i.e.,* a 2nd-order mutant is generated by applying a mutation to a 1st-order mutant, a 3th-order mutant is generated from a 2nd-

order mutant, *etc.* Empirical evidences show that higher order mutants may be used to subsume first order mutants, reducing the number of mutants to execute [253], and are harder to kill, which may be useful to select better test cases [177]. As for mutants execution, higher order mutation remains an open challenge in model-based context;

(iii) the **equivalent mutants problem** concerns the mutants whose behaviour is identical to the original artefact (code or model). As they cannot be distinguished by any test case, those mutants skew the analysis and cannot be used to select new test cases.

We address those issues for model-based mutation using the software product line framework. Artificial defects are injected using **mutation operators** on the model under test, producing a mutant. A mutant is thus a small variation of the model under test and the set of generated mutants may be grouped to form a **mutants family**. Based on this idea, we use variability aware mechanisms (FTSs and feature models) to represent a mutant family [83, 89].

Mutation analysis, as performed in this chapter, is done following a **product-based** approach. It assumes that a product has been selected, that the FTS has been projected on this product to get the corresponding LTS, and that only the subset of test cases executable on this product have been kept in the test suite. An FTS and a feature model represent here the mutant family for one product (*i.e.,* LTS). Discussion on extensions of this work to perform mutation analysis in a family-based approach (*i.e.*, on the FTS and feature model of an SPL) is presented in Section 10.2.2, along with other perspectives of this thesis.

In the remainder of this chapter, we present model-based mutation analysis and give an example in Section 6.1. Section 6.2 presents the Featured Mutants Model (FMM) and address the first ans second issues [89]. Section 6.3 applied automata language equivalence to detect equivalent mutants and compares it with two random simulations approaches [90]. Finally, we wrap up and present perspectives for our work.

6.1 Model-based mutation analysis

To address the problems of mutation analysis at the model level, we take our inspiration from our research on software product lines. The idea is to consider mutants as **members** of a **mutants family**[1]. Considering mutants as part of a family rather than in isolation yields considerable advantages: shared execution at the model level [63] and compact representation of a set of mutants. This contrasts with existing product line approaches of mutation analysis [162, 163, 224] which require code and hence do not apply to model mutants.

In the following sections, we use as example the specification of one product from the card payment terminal product line described in Section 4.2. This product allows both direct debit and credit payments, may perform it online or offline, and

[1] *Member* is a synonym for *product* or *variant*, and **family** is a synonym for *product line*. For the sake of clarity, we will refer to mutants as *members* of a *mutants family*.

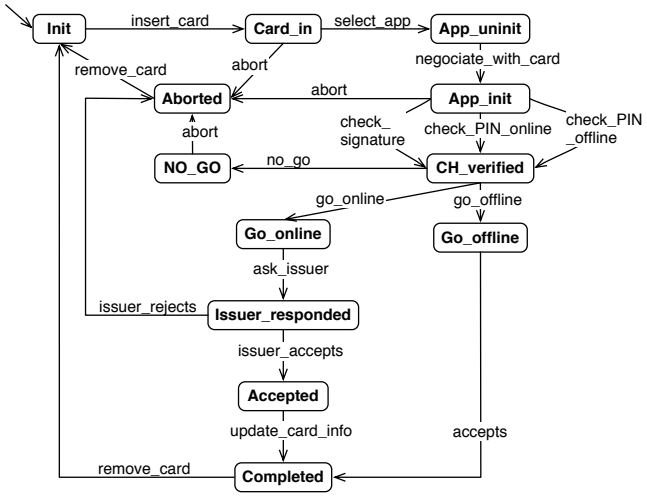


Figure 6.1: Card payment terminal product original LTS

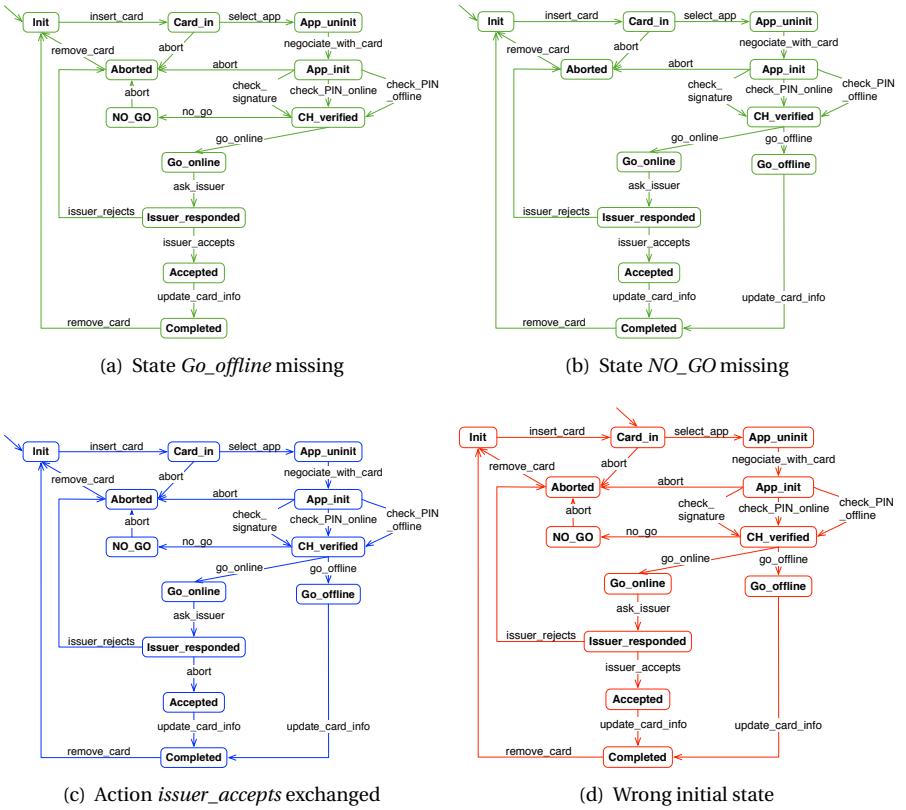
allows to identify the card holder using both signature or PIN code. The features from the feature model in Figure 4.2(a) selected to form this product are:

> { *CPTerminal, PaymentSchema, DirectDebit, DebitCard, CreditCard, Connectivity, Online, VPN, Offline, CardReader, Chip, MagStrip, Identificator, PIN, Signature* }

The LTS describing the behaviour, obtained using the projection operator on the FTS defined in Figure 4.2(b), is presented in Figure 6.1.

In model-based mutation analysis, mutants are introduced based on model transformation rules, called **mutation operators**, that alter the system specification. Model based mutation is thus a black box testing technique, unlike code-based mutation that requires access to the code and so is white box. An example of mutant obtained from the *state missing operator* applied on the *Go_offline* state of the card payment terminal system, is presented in Figure 6.2(a). There are two kinds of mutants, *first-order* mutants when the original and the mutant models differ by a single model transformation, and *higher order* mutants, derived from the original model after multiple transformations.

When a mutant is detected by a test case, it is called **killed**. In the opposite situation, it is called **live**. In our case, a mutant is killed if a positive abstract test case cannot be executed. For instance, the test case t_1 = (*insert_card*, *select_app*, *negociate_with_card*, *check_PIN_online*, *go_offline*, *update _card_info*, *remove_card)* will kill the mutant of Figure 6.2(a) since it fails to execute completely. A test case that can be completely executed on a mutant will not detect (kill) it, *e.g.*, the test case t_2 = (*insert_card, select_app, negociate_with_card, check_PIN_online, no_go, abort, remove_card*) will leave the mutant of Figure 6.2(a) live because it can be executed completely.



(a) State *Go_offline* missing

(b) State *NO_GO* missing

(c) Action *issuer_accepts* exchanged

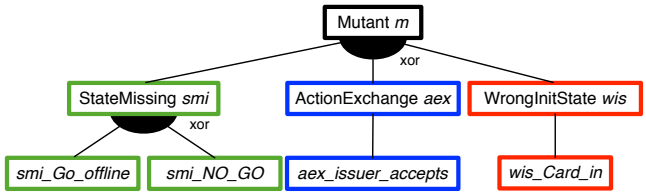
(d) Wrong initial state

Figure 6.2: Card payment terminal product mutants

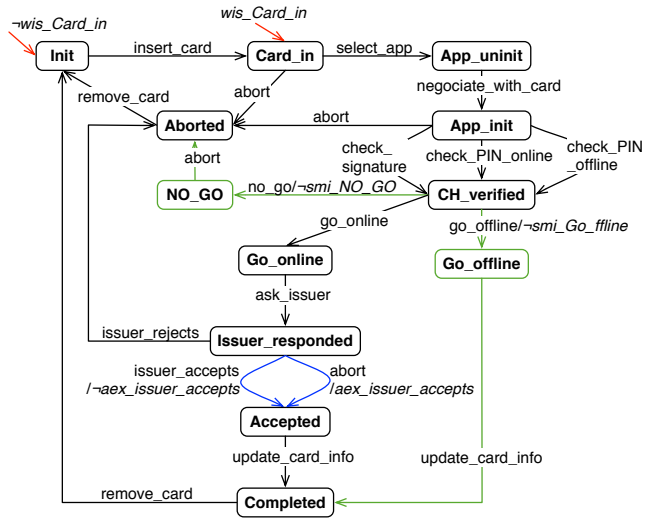
To measure the adequacy of testing, a standard metric called the **mutation score** is used. It is defined as the ratio of mutants killed by the test suite under assessment to the total number of considered mutants. To calculate the mutation score, one has to execute the whole test suite against every selected mutant. In our case, we consider deterministic LTSs and stop the execution of a test case as soon as the LTS is unable to fire the next action. For the test case t_1 on the mutant in Figure 6.2(a), the execution is stopped when it reaches the *CH_Verified* state as it may not execute the next action (*go_offline*) in t_1 and the mutant LTS is considered killed by t_1. In the following, we call this approach (*i.e.*, executing each test against each mutant model separately), the **enumerative approach**.

6.2 Featured mutants model

To represent the variations introduced by the mutation operator (*i.e.*, the result of the application of the mutation operator) and the behaviour of the mutants, we use a feature model and an FTS. Each feature in the feature model represents a mutation of the LTS representing the behaviour of the system under test. When a member



(a) Feature model



(b) Featured transition system

Figure 6.3: Card payment terminal product FMM

with one mutation (here represented as a feature) is selected and used with the projection operator on FTS, we obtain the behaviour (represented as a LTS) of the corresponding mutant. This allows us to embed all mutants in a single model, called the Featured Mutants Model (FMM):

Definition 23 (Featured Mutants Model (FMM)) *A FMM is a couple (fts,fm), where fts is an FTS representing the behaviour of the original system and all the mutants of the family, and fm is its associated feature model where each feature represents a mutation of the original system.*

For example, the FMM of Figure 6.3 has a feature model in Figure 6.3(a) with 3 instances of mutation operators: the state missing (*SMI*) operator, which produces a mutant where one state is missing; the action exchange (*AEX*) operator, which produces a mutant where one transition has its action changed (to another action); and the wrong initial state (*WIS*) operator, which produces a mutant where the initial state has been set to another state. In this instance of the feature model, the SMI operator has been applied twice (*smi_go_offline* mutant presented in Figure

6.2(a) and *smi_NO_GO* mutant presented in Figure 6.2(b)), and the AEX and WIS operators have been applied one time each (*aex_issuer_accepts* mutant presented in Figure 6.2(c) and *wis_Card_in* mutant presented in Figure 6.2(d)). This feature model represents four first-order mutants, where at most one leaf feature is selected. The FTS in Figure 6.3(b) represents all the possible variations, corresponding to the four mutation operators, of the original TS.

In order to derive one particular mutant from the FMM, one may use the FTS projection operator. Practically, this operator will first need a valid member of the mutant family, representing the desired mutant, *e.g.*, $p = \{m, smi, smi\_Go\_offline\}$; then, each feature expression of the FTS is evaluated with features belonging to the member replaced by true, and other features replaced by false; finally, transitions with a feature expression evaluated to false (*i.e.*, where $\gamma\, p = false$) and states that become unreachable are removed from the FTS. For instance, the projection of the FMM of Figure 6.3 on p will produce the mutant LTS of Figure 6.2(a).

To represent the effect of the WIS operator, we modify the FTS definition (Definition 2) to replace the initial state i by a total function $init$ that indicates if a state of the FTS is the initial state of the system:

Definition 24 (FTS for FMM) *A FTS is a tuple* $(S, Act, trans, init, d, \gamma)$*, where:*

- $S, Act, trans, d, \gamma$ *are defined according to definition 2;*
- $init : S \mapsto ([\![d]\!] \mapsto \mathbb{B})$ *is a total function indicating, for each state, for which product this state is the initial state and defined such that for every product there is exactly one initial state.*

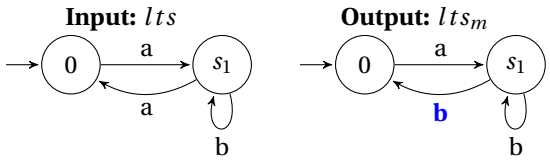
To be compliant with existing tools, this modification is implemented using an artificial initial state s_i, such that for every product, there is an outgoing transition from s_i, with no label and a feature expression indicating that this transition may only be fired by this product, and going to the initial state of the product.

6.2.1 Building the featured mutants model

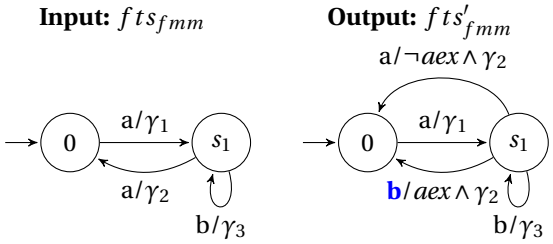
We rely on the state-of-the-art operators proposed by Fabbri *et al.* [102] to generate mutants from a LTS:

SMI State Missing operator: removes a state (other than the initial state) and all its incoming/outgoing transitions;
WIS Wrong Initial State operator: changes the initial state;
AEX Action Exchange operator: replaces the action linked to a given transition by another action;
AMI Action Missing operator: removes an action from a transition, leaving an ϵ-transition without action;
TMI Transition Missing operator: removes a transition;
TAD Transition Add operator: adds a transition between two states;
TDE Transition Destination Exchange operator: modifies the destination of a transition.

Each operator can be used to generate mutants using the enumerative approach, where each mutant is formed as a new variation of the original LTS (possibly in-



(a) Enumerative approach



(b) FMM approach

Figure 6.4: An example of mutation, the AEX operator

troducing non determinism in the case of AEX and TAD operators), or using the FMM approach, where each mutant is an addition to the feature model and the FTS. We detail hereafter the mutant generation procedures (the list of operators and the complete description of their effects is available in Appendix A).

Input: *lts*: the original LTS to mutate;
Ops: the set of mutation operators to use;
times : $Op \rightarrow \mathbb{N}$: a function specifying for each operator the number of applications
Output: *muts*, the set of produced mutants

```
1 begin
2     muts = ∅ ;
3     foreach op ∈ Ops do
4         foreach i ∈ [1; times(op)] do
5             muts = muts ∪ op(random(ts)) ;
6         end
7     end
8     return muts;
9 end
```

Algorithm 9: Mutant generation, enumerative approach

Enumerative approach: In the enumerative approach, each operator instance (*op*) is defined as a model transformation with input a LTS (*lts*) representing the behaviour of the product. It produces another (mutant) LTS (lts_m) representing the result of this operator instance on *lts*. For instance, AEX(s_1, s_0, b), shown on Figure 6.4(a), replaces the action a on transition $s_1 \xrightarrow{a} s_0$ by b. Algorithm 9 details the

enumerative approach where the set of mutants (*muts*) is produced by applying each operator (in *Ops*) with random parameters a number of times (defined for each operator by the *times* function) on the original LTS (line 5).

FMM approach: In the FMM approach, an operator (Op_{fmm}) is defined as a model transformation of a FMM (representing existing mutants), that produces a FMM representing (the previously existing mutants and) the result of the Op_{fmm} mutation **on the original TS** (obtained in the FMM's FTS by replacing the features by false in the feature expressions).

For instance, in Figure 6.4(b), the AEX_{fmm} operator instance replaces the action a on transition $s_1 \xrightarrow{a} s_0$ of the base model by b as follows:

(i) adding the feature expression $\neg aex$ on transition $s_1 \xrightarrow{a/\gamma_2} s_0$, stating that $s_1 \xrightarrow{a/\neg aex \wedge \gamma_2} s_0$ may be fired only if the *aex* mutation is inactive (and if γ_2 is true);

(ii) adding a transition $s_1 \xrightarrow{b/aex \wedge \gamma_2} s_0$, stating that the transition is fired with a b action only if the *aex* mutation is active (and if γ_2 is true);

(iii) adding an *aex* feature to fd_{fmm} representing the mutation done by Op_{fmm} (not shown in Figure 6.4(b)).

Input: *lts*: the original LTS to mutate;
Ops: the set of mutation operators to use;
$times : Op \rightarrow \mathbb{N}$: a function specifying for each operator the number of applications
Output: $fmm = (fts_{fmm}, fm_{fmm})$, the FMM

```
1  begin
2      γ = (λt → true) ;
3      fts_fmm = (S, Act, trans, i, fm_fmm, γ) ;
4      fm_fmm = (m) ;
5      foreach op ∈ Ops do
6          foreach i ∈ [1; times(op)] do
7              fmm = op_fmm(fmm) ;
8          end
9      end
10     return fmm;
11 end
```

Algorithm 10: Mutant generation, FMM approach

Algorithm 10 details the automated FMM building approach. We start with the original LTS and a γ function that labels each transition with a true feature expression (line 2). The feature model of the FMM is initialised to a root element m (line 4). We then apply mutation operators (Ops_{fmm}) a specified number of times ($times(op)$ line 6). Contrary to the enumerative approach, the mutation operators are applied on the FMM under construction, which is reused in the next iteration (line 7). This is mandatory as the FMM contains all the previous mutations that are taken into account in the model transformations (*e.g.*, the γ_i expressions in Figure

6.4(b)). As we choose to only perform mutations on the original LTS, this forbids operator composition on (previously) mutated elements. Doing so ensures that first-order mutation maps to only one edit of the original LTS and that higher order mutants do not edit the same elements of the original LTS more than once. Further details about the operators and specificities of the transformations can be found in Appendix A.

6.2.2 Featured mutants model execution

A test case for a product (*i.e.,* a LTS) is defined as a sequence of actions in this LTS (*lts*), such that one execution form a path starting from and ending at the initial state (i): $t = (\alpha_1, ..., \alpha_n)$ such that $(i \xrightarrow{\alpha_1} s_1, ..., s_{n-1} \xrightarrow{\alpha_n} i)$. Recall that in the enumerative approach, if a test case cannot be executed by the mutant (denoted $m \overset{t}{\not\Rightarrow}$) or does not end in the initial state of the original LTS (considered as the accepting state), it is considered killed. Otherwise, the mutant is considered live. The set of live mutants, for t in the set of mutants *muts*, is defined as:

$$liveEnum(muts, t) = \{m \in muts \mid m \overset{t}{\Longrightarrow} i\}$$

In the FMM approach, a test case can be executed on an FMM *fmm* (denoted $fts_{fmm} \overset{t}{\Longrightarrow}$), if there exists at least one mutant or the product is able to execute it. The enumerative approach executes each test case on each mutant separately. In contrast, one execution of a test case on the FMM explores all the reachable mutants (identified by the collected feature expression γ). The set of live mutants in the FMM approach is defined as:

$$liveFMM(fmm, t) = \{p \in [\![fm_{fmm}]\!] \mid fts_{fmm|p} \overset{t}{\Longrightarrow} i\}$$

Concretely, all possible paths in fts_{fmm} starting from i and ending in i will be considered, which allows to deal with possible non-determinism introduced by a mutation. The live mutants are those able to execute at least one of those paths, *i.e.,* those for which the product p satisfies all the feature expressions on the transitions of the considered path.

For instance, the test case:

$$t = (insert\_card, select\_app, negociate\_with\_card, check\_PIN\_offline, go\_offline, update\_card\_info, remove\_card)$$

Executing the FMM of Figure 6.3, it will fire the following transitions:

$$(\xrightarrow{/\neg wis\_Card\_in} Init,$$
$$Init \xrightarrow{insert\_card} Card\_in,$$
$$Card\_in \xrightarrow{select\_app} App\_uninit,$$
$$App\_uninit \xrightarrow{negociate\_with\_card} App\_init,$$
$$App\_init \xrightarrow{check\_PIN\_offline} CH\_verified,$$
$$CH\_verified \xrightarrow{go\_offline/\neg smi\_go\_offline} Go\_offline,$$
$$Go\_offline \xrightarrow{update\_card\_info} Completed,$$
$$Completed \xrightarrow{remove\_card} Init)$$

These transitions may only be fired by mutants for which all the features expressions are $true$. Here, mutants need to respect the following constraint:

$$\neg wis\_Card\_in \wedge \neg smi\_go\_offline$$

All mutants in the feature model of Figure 6.3(a) that satisfy this feature expression remain live after the execution of t. The set of mutants killed by the test case is computed using the conjunction of fm_{fmm} and the negation of this feature expression: $fm_{fmm} \wedge (wis\_Card\_in \vee smi\_go\_offline)$, which corresponds to the set of mutants:

$$\{(m, wis, wis\_Card\_in), (m, smi, smi\_go\_offline)\}$$

In practice, $liveFMM(fmm, t)$ produces a feature expression representing all the live mutants as detailed in Algorithm 11. Initially, the algorithm computes all the paths in fts_{fmm} corresponding to the sequence of actions in t (line 3). For one path, the conjunction of the feature expressions gives the mutants able to execute this path (line 5). Effort is saved this way by ignoring unreachable mutants and by sharing the execution of the common transitions. This conjunction disjuncts with the conjunctions of the others paths to get the feature expression representing all the live mutants (line 5). This step results in savings due to merging of the considered executions. For performance reasons, the *paths* variable uses a tree representation to merge common prefixes of different paths.

We implemented the different mutant operators described in Appendix A in order to perform classical mutation testing (enumerative approach) as well as FMM generation and execution in VIBeS, our Variability Intensive Behavioural teSting Java framework.

6.2.3 FMMs as higher order mutants model

Higher order mutants can be valuable since some of them tend to be hard to kill [128]. However, the number of mutants grows exponentially according to the order n and

Input: $fmm = (fts_{fmm}, fm_{fmm})$, the FMM;
$t = (\alpha_1, ..., \alpha_n)$: a test case defined over the original LTS
Output: *live*, the feature expression representing the mutants live after executing t on fmm

```
1 begin
2     live = false;
3     paths = {(i --α1/γ1--> ... --αn/γn--> i)};
4     foreach p ∈ paths do
5         live = live ∨ (⋀_{γi∈p} γi)
6     end
7     return live;
8 end
```

Algorithm 11: FMM mutant execution

explodes the involved cost. This is obvious in Algorithm 9, for the enumerative approach, which generates all the $n-1$ mutants to generate the nth-order ones.

Using the FMM approach, modelling higher order mutation comes at (nearly) no cost. In a FMM (fts_{fmm}, fm_{fmm}), the set of allowed mutants (*i.e.*, variations in fts_{fmm}) is represented by the feature model (fm_{fmm}). For instance, the constraints in the fm_{fmm} of Figure 6.3(a) allows to have exactly one mutant at a time. Meaning that all valid mutants (members) of this FMM will have at most one variation from the original LTS made by a mutation operator, *e.g.*, Figure 6.2(a) has (only) *smi_go_offline* feature active.

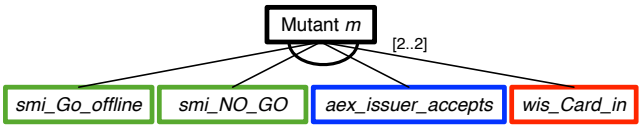


Figure 6.5: The order 2 FMM of the card payment terminal example

The nth order mutants are represented by modifying the constraints on the fm_{fmm} so that they have exactly n mutations at a time. It means that generating the FMM using Algorithm 10 will also generate the FTS (which will be the same) for order 1 to n FMMs. For instance, the card payment terminal product has the same FTS, for all orders as shown in Figure 6.3(b), but differs on the feature model that is described by Figure 6.5 by the group cardinality stating that exactly 2 subfeatures have to be selected. The FMM will compactly represent all the $C_2^4 = 6$ 2nd-order mutants.

All-order mutants: Using the same argument, we generalize to mutants of any order by setting the group cardinalities of the feature model in Figure 6.5 to [1..4]. In this case, the FMM represents a single model with all possible orders of mutants (with n between 1 and the number of leaf features in the FMM's feature model). A valid member (mutant) of the feature model will contain at least one application

of a mutation operator,*e.g.,* a mutant $m = \{m, smi\_go\_offline\}$, but also $m' = \{m, smi\_go\_offline, smi\_NO\_GO\}$, or $m'' = \{m, smi\_go\_offline, wis\_Card\_in\}$, *etc.* In this case, the FMM compactly represent all the $2^4 - 1 = 15$ n-order mutants.

The number of live mutants after the execution of a test case (t) on a FMM (fmm) can be obtained by counting the number of SAT solutions (*i.e.,* the number of possible assignments for each feature) to $fm_{fmm} \wedge liveFMM(fmm, t)$, where fm_{fmm} is the FMM feature model encoded as a boolean formula, *i.e.,* the disjunction of the mutations (Ops): $fm_{fmm} = \bigvee_{o \in Ops} o$. For a test set ($s$), the number of live mutants is computed by counting the number of SAT solutions to

$$\left(\bigvee_{o \in Ops} o \right) \wedge \left(\bigwedge_{t \in s} liveFMM(fmm, t) \right).$$

6.3 Equivalent mutants problem

Despite its potential, mutation analysis faces a number of challenges that currently prevent wider adoption [152, 237]. One of them is the *Equivalent Mutants Problem (EMP)*. It concerns the mutants whose behaviour is identical to the original artefact (code or model). Such mutants cannot be distinguished by any test case, a situation that raises two issues: (i) they hamper the use of the criterion as a stopping rule by skewing the mutation score measurement (the number of killed mutants divided by the total number of mutants); and (ii) they do not bring any new value to the test selection techniques as they attempt to kill mutants that have no chance to be killed.

Mutant equivalence can take two forms [237]: (i) equivalence between mutants and the original system; (ii) equivalence between two mutants (not with the original system). Mutants of case (i) are called **equivalent** while mutants of case (ii) are called **duplicate**. We focus here on mutants that are behaviourally equivalent to the original system, *i.e.,* mutants of case (i).

6.3.1 EMP and automata theory

The model-based formulation of the EMP can be expressed as a classical problem in automata theory: **Automata Language Equivalence (ALE)**. The accepted language of an automaton is formed by all the sequences of actions (words) that can be accepted *i.e.,* starting in the initial state and ending in a final state. Therefore, if a mutant m accepts the same language as the original o (*i.e.,* is language-equivalent to the original), then there is no test sequence s that can distinguish the mutant from the original:

$$\forall s, s \in \mathscr{L}(o) \Leftrightarrow s \in \mathscr{L}(m)$$

There are various relations defined between two automata that we can compute to determine whether they are language-equivalent. Among them, we can cite bisimulations or trace equivalence [27]. In the last years, the verification community came up with dedicated relations concepts such as bisimulations up to congruence [42] or antichains [94] to address language equivalence. In model-based mutation

testing, Aichernig *et al.* investigated language inclusion (but not equivalence) using refinement checking [5] in order to select mutant-killing test cases.

Although tackling the language equivalence and inclusion problems from different angles and heuristics, all these techniques may face exponential blow-up since both language inclusion and equivalence were demonstrated to be PSPACE complete [173]. While worst-case complexity can seem discouraging, various heuristics have been proposed to limit the effects of this complexity in practice. One of our goals is to determine the applicability of an exact language equivalence algorithm to address the EMP [42]. The algorithm selected due to its availability, reported performance over the state of the art and ability to handle non-determinism that mutations may incur. In the next section, we also present two baseline algorithms that run generated traces to distinguish original and mutants' behaviours.

6.3.2 Strong and weak mutation

Jöbstl [153] discussed the conditions, identified by DeMillo and Offutt [79], that must be fulfilled to kill a mutant:

(i) "the **necessity condition** says that the state of the mutated program after some execution of the mutated statement must be incorrect with respect to the original program. This implies that the mutated statement must be reached. This is necessary, but not sufficient";
(ii) "the **sufficiency condition** says that the final state of the mutant must differ from the final state of the original program, *i.e.,* the necessary incorrect intermediate state must propagate to an incorrect final state."

Satisfying the necessity condition alone is referred to as **weak mutation**, while satisfying both is **strong mutation** [145, 206].

At the model level, our simulations detect an incorrect state if an abstract test case that is valid with respect to the original LTS, is invalid on the mutant LTS, or vice-versa. Indeed, when executed, an abstract test case induces one or more *runs* (alternating sequences of states and actions), depending on the presence of non-determinism. If one run does not contain all the actions of the abstract test case (*i.e.,* the run is **incomplete**), it is because of the presence of an incorrect state preventing the subsequent actions to be fired. If all runs are complete, the original and the mutant are assumed equivalent for this test case. Necessity and sufficiency conditions affect the final states of these runs. For Weak Mutation (WM), these states can map to any state of the LTS (like for abstract test cases). For Strong Mutation (SM), we need to account for the fact that LTSs have no final states: in this case, we assimilate the initial state to a final state and consider only positive (or negative) abstract test cases for strong mutation.

The ALE approach uses automata that have explicit initial and final states. For weak mutation, we generate automata in which all states are final, and for strong mutation the initial state is the only final state.

6.3.3 Mutant equivalence analysis

As explained in the previous sections, equivalent mutant detection may be done using automata language equivalence. Since this is a PSPACE complete problem, we also propose two randomized approaches: **Random Simulation (RS)** and **Biased Simulation (BS)**. Those approaches are straightforward: we generate random (either fully random or biased random) traces from the original (resp. mutant) model and run them on the mutant (resp. original) model. If a trace fails to execute on one of the models, it serves as a counterexample and disproves equivalence. Otherwise, the mutant is considered **probably equivalent** and testers have to decide whether they want to perform more simulations or switch to an exact method.

Automata language equivalence: The ALE approach we selected for comparison is developed by Bonchi and Pous [42]. It can be thought of as an extension to non-deterministic LTSs of the Hopcroft-Karp algorithm. In particular, they introduce a bisimulation relation called **up to congruence** that requires to explore less states than the original algorithm. This approach also avoids to build the complete deterministic finite TS and performs determinisation on-the-fly. This makes such an approach particularly relevant: non-determinism may be introduced locally by mutations (our original models are deterministic), thereby limiting determinisation scope[2].

Randomized simulations: Algorithm 12 presents our generic randomized simulation approach: N abstract test cases are selected (respectively) from the original model (line 2) and the mutant model (line 8), and executed (respectively) on the mutant model (line 4) and the original model (line 10). In case of non deterministic behaviour, all the possible paths are considered for the execution of the abstract test case. If one equivalence test fails, the algorithm stops and returns an abstract test case, either valid for the original LTS (line 5), such as

$$(o \overset{t}{\Longrightarrow} i) \wedge \neg(m \overset{t}{\Longrightarrow} i)$$

or an abstract test case, invalid for the original LTS (line 11), such as

$$\neg(o \overset{t}{\Longrightarrow} i) \wedge (m \overset{t}{\Longrightarrow} i)$$

This generic simulation algorithm is instantiated through two strategies for trace generation (lines 2 and 8): Random Simulation (RS) and Biased Simulation (BS). The parameter N is computed using the Chernoff-Hoeffding bound.

Random simulation: Random simulation assumes a uniform distribution of on the transitions enabled in each state, that is, such abstract test cases are selected randomly (*select* call on lines 2 and 8 in Algorithm 12) by accumulating the actions

[2]This is indeed the case in our evaluation: between 0% and 15.5% of the mutants are non-deterministic (see Section 7.6).

Input: o, the original LTS;
m, the mutant to compare to o;
N, the total number of traces to generate;
k, the length of the abstract test cases;
Output: t, an abstract test case differentiating m from o or *none* if m is potentially equivalent to o.

```
1  begin
2      traceset = select(o, N/2, k) ;
3      foreach t ∈ traceset do
4          if ¬(m ⟹^t) then
               // mutant fails to execute t, returns t
5              return valid(t) ;
6          end
7      end
8      traceset = select(m, N/2, k) ;
9      foreach t ∈ traceset do
10         if ¬(o ⟹^t) then
               // original LTS fails to execute t, returns t
11             return invalid(t) ;
12         end
13     end
14     return none;
15 end
```

Algorithm 12: Mutant equivalence analysis: generic randomized simulation

α_i triggered by a random walk of a given length k in the LTS. For weak mutation (WM RS), the only constraint is to start the random walk from the initial state i. Strong mutation (SM RS) requires a random walk starting from and ending in i[3].

Biased simulation: The Biased Simulation (BS) approach exploits the basic characteristics of mutation testing: mutations are localised and they create (most of the time) behavioural differences. It assumes that those differences are detected by an abstract test case t which, when executed on the original LTS o or on its mutant m, goes through one of the states affected by the mutation. For instance, the *transition missing operator* produces a mutant by removing a transition $a \xrightarrow{\alpha_i} b$ from the original LTS. The BS approach (that does not know which mutation has been performed) selects abstract test cases in o and m, such that their executions $m \overset{t}{\Longrightarrow}$ or $o \overset{t}{\Longrightarrow}$ reach a or b (where a syntactic difference between the models is detected). Such states, called **infected states**, have been shown to help identifying equivalent

[3] After few tries, this method (*i.e.*, using a random walk until the initial state i is reached) showed very poor results on our largest models (we set a timeout of one hour for one equivalence detection) and is therefore not further discussed. See Section 7.6 for more details.

mutants at the code level [30, 229] and to speed up mutation analysis at the model level [168]. This motivates us to adopt this strategy in our biased simulation.

In practice, the set of infected states S_{infect} is computed by checking syntactic differences between the original and mutant LTSs. It will include:

(i) connected states (*i.e.*, states accessible from the initial state) from one model which are not present in the other, and

(ii) states with differences in their input/output transitions: in number of transitions or in action names, considering any pair of states $< s_o, s_m >$ where s_o is a state in the original TS, s_m a state in the mutant TS, such that their names are identical.

An alternative is to instrument the mutant generator to keep track of the list of infected states while generating the mutants. Our goal is to be able to apply this strategy without any information on how the mutants are generated (*e.g.*, generated by other frameworks than ours) and to fairly compare with an exact approach that makes no assumption on the locality of differences. Once the set of infected states S_{infect} is obtained (by any means), the second step is to generate traces that cover such infected states.

For weak mutation (WM BS), an abstract test case t is selected (*select* call on lines 2 and 8 in algorithm 12) by concatenating the actions of the shortest walk from the initial state i to a randomly chosen state $a \in S_{infect}$ and a random walk starting from a. To proceed, the first step during abstract test case selection is to compute the shortest distance (*i.e.*, the number of transitions) between each state of the original LTS o (or its mutant m respectively) and the initial state i of o (or m respectively) using a standard breadth-first search [75].

For strong mutation (SM BS), instead of a random walk starting from a, the algorithm will consider the actions of a path starting from a and returning to i using the computed shortest distance: the distance from a to i will (not strictly) decrease each time a transition is taken in the path.

Estimating the number of required runs: An important parameter for simulation is the number of abstract test cases selected from the original (resp. mutant) model and run on the mutant (resp. original) model: $N/2$. Under the hypothesis that abstract test cases are uniformly distributed we can bound the equivalence probability and estimating the number of runs needed achieve these bounds. Herault *et al.* [139] suggested to use the Chernoff-Hoeffding bound to estimate the number $N/2$ of required runs to limit the equivalence probability depending on the approximation parameter $\epsilon > 0$ and a confidence parameter $\delta < 1$. If $N/2 \geq \frac{4\log(2/\delta)}{\epsilon^2}$ then we have:

$$Pr\left[equiv(m,o)\right] = Pr\left[\left|\frac{A}{N/2} - p\right| \leq \epsilon\right] \geq 1-\delta$$

Where A is the number of successful runs that is either $m \stackrel{t}{\Longrightarrow}$ or $o \stackrel{t}{\Longrightarrow}$ for a given abstract test case t. In practice, we compute $2A/N$ only when the algorithm has exhausted all the runs and set $N = \frac{8\log(2/\delta)}{\epsilon^2}$ for the number of runs as we have to account for two-way simulation (*i.e.*, two simulations): the number of runs is thus doubled.

It has to be observed that regarding biased simulations, the distribution of abstract test cases will not be uniform as the infected states force abstract test cases to explore only given portions of the model, *viz.* where the mutations are. Although this inequality may not hold in this case, we alleviate this threat by not trying to interpret the δ and ϵ values for biased simulations: they are for us a convenient means to compute N. Furthermore, keeping the same number of runs for random and biased simulations allows comparing their execution times and recalls (See Section 7.6).

6.4 Related work

Program mutation was proposed as a rigorous testing technique [50]. The idea was then applied to test specification models [230] and recently to resolve software engineering problems such as the improvement of non-functional properties [176], locating [238] and fixing software defects [181]. Here we briefly discuss works related to model-based mutation and code-based mutation.

6.4.1 Model-based mutation

The idea of model-based mutation has been elicited by Gopal and Budd [49] who called it *Specification Mutation*. Specification mutation promises to identify defects related to missing functionality and misinterpreted specifications [49]. This is desirable since these kinds of defects cannot be identified by any code-based testing technique [146, 314], including code-based mutation.

Gopal and Budd [49] studied mutation for specifications expressed in logic. Similarly, Woodward [318] mutated and experimented with algebraic specifications. Mutating models like finite state machines and Statecharts has also been done by Fabbri *et al.* [101]. Hierons and Merayo [144] used Probabilistic Finite State Machines. All these studies suggested a set of operators and report some exploratory results. Amman *et al.* [11] suggested comparing the original and the mutated specification models using a model checker in order to generate counterexamples. These can then be used as test cases for the system under test. Black *et al.* [41] defined a set of operators based on empirical and theoretical analysis. They also defined a process of using them based on the SMV model checker. Contrary to our approach, none of these methods considers the mutation efficiency.

Recent research focuses on mutating behavioural models. Aichernig *et al.* [2, 3, 5] defined UML state machines mutant operators and used them to insert faults in the models of an industrial system. These were used to design tests. The approach has a formal ground but neither considers optimising the test execution, nor higher order mutation. Belli and Beyazit [34, 36] compare event-based and state-based model mutation testing. Both approaches were found to have similar fault detection capabilities. The authors also report that it seems more promising to perform higher order mutation than first-order mutation but did not provide evidence in support of this argument. Krenn *et al.* [169] made available their MoMuT tool, but it is dedicated to test selection and not mutant execution or equivalent mutant detection as our approach. In their most recent work [168], they use an idea similar to FMM by

triggering mutations during exploration of the model, avoiding execution of similar prefixes in different mutants. Additionally, MoMuT does not support higher order mutation. Recently, Granda *et al.* [118] defined mutation operators for UML class diagrams.

Other applications of model-based mutation are to test model transformations and test configurations. Mottu *et al.* [219] defined a fault model relevant to the model transformation process based on which they propose a set of mutant operators. Finally, Papadakis *et al.* [235] demonstrated that model-based mutation of the combinatorial interaction testing models has a higher correlation with the actual fault detection than the use of combinatorial interaction testing. Thereby, they provide ground to the argument that model-based mutation might be more effective than the other model-based testing methods.

6.4.2 Code-based mutation

In the context of code-based mutation, executable mutants are needed. This introduces a compilation overhead which is proportional to the number of mutants. To reduce this cost, Untch *et al.* [306] proposed mutant schemata, an approach that replaces the program operators with schematic functions. These functions introduce the mutants at runtime and thus, only one compilation is needed. Ma *et al.* [202] suggested using bytecode translation, a technique that introduces the mutants directly at the bytecode level and thus avoid multiple compilations.

To reduce the test execution overhead, several optimizations have been proposed. Delamaro and Maldonado [78] suggested recording the execution trace of the original program and consider only the mutants that are reachable by each of the employed tests. Along the same lines, Mateo and Polo [205] suggested stopping mutant executions when they cause infinite loops. Jackson and Woodward [150] suggested parallelizing the mutant execution process. Kapoor and Bowen [160] proposed ordering the mutants in such a way that the test execution is minimized. Papadakis and Malevris [240] used mutant schemata to identify mutants that are reached and infected by the considered tests. They then reduce test execution by considering only the mutants that cause infection. This technique was later evaluated by Just *et al.* [157] who found that it reduces test execution by 40%.

6.4.3 SPL and other variability models mutation

Henard *et al.* [133, 136] and Arcaini *et al.* [19, 20] define mutant operators for feature models and use them to assess the ability of a set of products (*i.e.,* a test suite in their case) to find faults. Along the same lines, Lackner and Schmidt [174] define mutant operators for the mappings of features with other model artefacts, and Al-Hajjaji *et al.* [7] define mutation operators for preprocessor-based variability. Arcaini *et al.* [18] also mutated feature models in order to detect anomalies software artefacts [18]. Finally, Henard *et al.* [133], Reuling *et al.* [261], Matnei Filho and Vergilio [207] mutate the feature model to select products to test.

6.4.4 Mutant equivalence analysis

Previous work demonstrated that equivalent mutants skew the mutation score measurements and thus hinder the effectiveness of the method [203]. Unfortunately, it has been proven that judging whether a code mutant is equivalent to the original code is an undecidable problem [51]. This means that there is no solution to the general case of this problem. Luckily, since mutations are small syntactic changes, heuristics can identify several classes of them [237]. Two types of such heuristics exist in the literature: those that operate in a static manner and those that are dynamic.

Static techniques include the use of compiler optimizations [228], constraint solving [229], program slicing [143], data-flow patterns [164], and formal verification [30]. All these techniques are effective at detecting certain types of equivalent mutants, *i.e.*, trivial equivalences [237], but unfortunately, they are not applicable to model mutants.

Dynamic techniques measure the differences between the test executions of the original and mutant programs and identify likely non-equivalent mutants. Schuler and Zeller [277] and Papadakis *et al.* [236] measure the impact on coverage, while Kintis *et al.* [165] measure the impact on other mutants (second-order mutants). Our technique shares the same notion of equivalence because we check the model trace in order to judge it. However, we do not consider executable code as we only deal with model mutants. We also sample execution in order to increase the efficiency of the process. It is to be noted that we have a different notion of equivalence since we deal with behavioural models. Therefore, differences in traces imply different behaviours, which is not the case for executable code.

Non-determinism complicates equivalence detection both at the code [243] and model levels [4]. Patel and Hierons [243] associate predictions from pairs of inputs and outputs of the mutant program and check whether these predictions can be discarded by the original program, hence showing non-equivalence. This is not applicable to our case since our models do not have outputs. Aichernig and Jöbstl [4] also encode the semantics of the action models in terms of constraints and use refinement to check conformance in the context of non-determinism. In our case, RS/BS manage non determinism in the TSs by considering all the possible runs.

Perhaps the closest work is that of Papadakis and Malevris [241] who sample execution paths according to their length (select the k-shortest paths), symbolically execute them and judge mutant equivalence based on the selected paths. The main differences with our approach are that we additionally sample paths that cover infected states and we operate on behavioural models instead of actual code representation.

6.5 Wrap up

In this chapter we discuss our contributions to model-based mutation analysis to: (i) generate, represent, and effectively execute mutants using a family-based approach to model-based mutation testing, named Featured Mutants Model (FMM); and (ii) tackle the equivalent mutant problem at the model level using an exact lan-

guage equivalence (called Automata Language Equivalence (ALE)) and two random simulations approaches (called Random Simulation (RS) and Biased Simulation (BS)).

Compact mutant model: **FMM** takes advantage of the variability formalisms (*i.e.,* FTS and feature model) to compactly represent all possible mutations on a single model. To do so, we rely on modelling the behaviour of the product under test and view mutant operators as model transformations that can be activated on demand by selecting features in the feature model of the FTS. It allows to generate mutants of any order and assess test effectiveness via an optimised execution scheme. Testing behavioural models with FMM is a completely automated process that involves no extra manual or computational effort over previous approaches.

In short, the use of FMM has the following **benefits**: first, it can easily reason about and **generate behavioural mutants**; second, it can significantly **speed up** the evaluation of test suites against mutants (up to 1,000 times, as showed in Section 7.5); and finally, it can efficiently perform **higher order mutation**.

Equivalent mutant problem: To tackle the **Equivalent Mutants Problem (EMP)** at the model level, we offer two baseline algorithms based on random simulation, and compare them to language equivalence under **weak** and **strong** mutation scenarios. Our experiments in Section 7.6 demonstrates the efficiency of the exact approach for the weak mutation scenario. For strong mutation, our biased simulations, that pre-process the models to detect states that are infected by mutations, are efficient (up to 1,000 times faster) on models that contain more than 300 states, limiting detection errors to 8%. This suggests using **simulations first** to quickly discard many non-equivalent mutants, and then employing **exact approaches** only on a small amount of probably equivalent mutants to speed up equivalence analysis.

There is room for improvement. First, we will extend our experiments to other forms of equivalence and tools: for instance, the usage of a random based simulation directly in the FMM to quickly filter out non equivalent mutants. We would also like to switch from the pure equivalence analysis to test selection concerns by analysing counter-examples. Our long-term goal is to draw attention on the applications of language equivalence for mutation testing and develop further EMP-dedicated solutions.

CHAPTER 7

EMPIRICAL ASSESSMENT

This chapter presents the empirical assessment performed to validate the test suite selection criteria described in Chapter 5 and the mutation analysis described in Chapter 6. Each assessment is explained in a separate section. Each section starts with the research question(s) driving the assessment, describes the setup and results, and discuss those results and their validity.

Section 7.1 presents the assessment of the all-states selection described in Section 5.2.1; Section 7.2 presents the assessment of the dissimilarity-based selection described in Section 5.3; Section 7.3 assesses how classical feature model coverage criteria, like t-wise, cover the behaviour of a product line using two state-of-the-art tools; and Section 7.4 presents the assessment of the usage selection and prioritization criteria described in Section 5.4.

For mutation analysis, Section 7.5 presents the assessment of the mutants execution using FMMs described in Section 6.2 in a product-based mutation analysis scenario. Finally, Section 7.6 presents the assessment of the mutant equivalence detection for products using automata language equivalence and simulation (described in Section 6.3).

7.1 All-states selection criteria

To assess the all-states selection algorithm described in Section 5.2.1, we use coverage measures to answer to the following research question [91]:

- ***Coverage ability*** How does the coverage the selected test suite compare to the coverage of the randomly selected test suites?

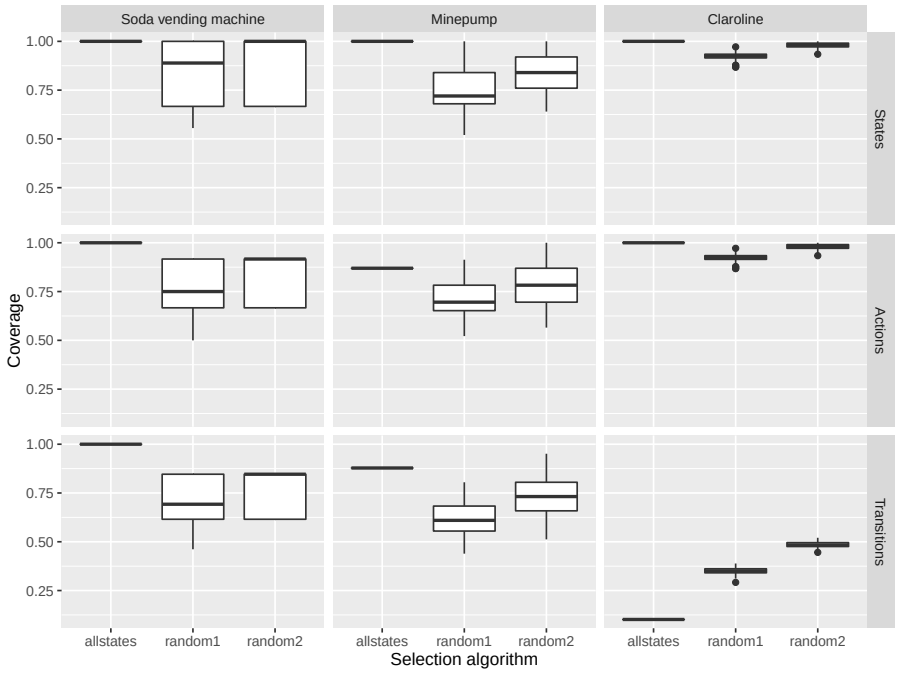


Figure 7.1: Structural coverages of the *allstates*, *random1*, and *random2* test suites

7.1.1 Setup

Our evaluation has been performed on three case studies: the Soda vending machine (see section 4.1), the Minepump (see section 4.3), and Claroline (see section 4.6). For each of them, we select random test suites and a test suite satisfying the all-states coverage criterion using Algorithm 3.

We select random test suites with the same number of test cases (*random1*) as the ones selected by all-states algorithm (*allstates*) to enable direct comparison of coverage. We also randomly select larger test suites (*random2*) to figure out coverage gains. In total, for each model, we select one *allstates* test suite and 100 *random1* and *random2* test suites. For the soda vending machine, the *allstates* and *random1* test suite contains 5 test cases, and the *random2* test suite contains 20 test cases. The *allstates* and *random1* test suite of the Minepump contains 12 test cases and the *random2* test suite contains 20 test cases. And for Claroline, the *allstates* and *random1* test suite contains 105 test case, and the *random2* test suite contains 200 test cases.

This assessment was performed on a Windows 7 machine with an Intel Core i3 (3.10GHz) processor and 4GB of memory.

7.1.2 Results

Figure 7.1 presents the states, actions, and transitions coverage of the different test suites. By construction, abstract test suites selected using our all-states algorithm (*allstates* in Figure 7.1) covers all the states for the three FTSs. On average, the random algorithm does not perform well to cover all the states of the different models. On the contrary, the random algorithm performs better at covering transition on the largest model (Claroline): an average of 48.45% for the 200 randomly selected test suites (*random2*) against 10.17% for the all-states selection algorithm.

7.1.3 Discussion

We investigated the difference between the *allstates* and *random(1,2)* transitions coverage for the Claroline model and found that the all-states test suite contains only short test cases (2 actions). This is due to our heuristic. Since it prefers states that have a path to uncovered states, the initial state has the highest score in the Claroline model (because nearly all the states in the models have transitions coming from and going to the initial state, due to the web nature of the application). Changing the heuristic to avoid direct return to the initial state may improve the results for this kind of models (where each state is strongly connected to the initial state) but may increase the complexity of the algorithm and select inadequate abstract test cases for other kinds of models.

These results are in line with the fact that all-states coverage criterion is poor to cover transitions in single systems [307], we assume that this is also the case for FTSs. Of courses an all-transitions algorithm would have given much better results (and all-states coverage). However our preliminary evaluation of such an algorithm resulted in huge scalability problems for the Claroline case study; after more than 3 days of selection and a text file describing the test suite of more 250 GB, our systems ran out of memory (and also of hard drive space!). Exhaustive computation of all-transitions for moderate size FTS is therefore not an option in most cases.

Finally, we observe that the test suite selected to cover all-states in the Claroline FTS also covers all the actions. This is due to the nature of the model, since each state represents a page and each action represents a link followed from a page (*i.e.,* a state in the FTS) to another page (*i.e.,* another state), there are as many actions as there are states. Each action is thus covered. This is not always the case, *e.g.,* the soda vending machine in Figure 7.1.

7.1.4 Threats to validity

Internal validity: The all-states selection algorithm has been simplified to reduce the number of SAT calls which are very costly. This simplification gives good results on our largest model (Claroline) due to the few constraints on the FTS. On other models with more constraints on the different transitions, this simplification may give poor results since it can potentially select a lot of invalid test cases. We intend

to compare the actual implementation which uses a SAT solver with binary decision diagrams (BDDs) which have performed better when processing FTSs [65].

The random selection of a test suite does not check whether there are duplicates abstract test cases or not. Since the size of the test suites considered for *random2* test suites is larger than the size of the all-states covering test suite, this thread is limited. To avoid this, one may implement a filter to check that newly selected test suites are not duplicated.

Construct validity: The Claroline FTS and its FM contain few constraints ending in a SPL with lots of products. We believe this is a typical characteristic of web applications which are a particular class of system. This has influenced the implementation of the random algorithm in order to minimize the number of SAT calls (which are costly in CPU time). It has also influenced the heuristic during the selection of the test suite covering all-states and gives very short test cases in regard to the size of the system. We plan to apply our algorithms on large industrial systems with more constrained FM and FTSs in order to validate our conclusions.

External validity: To avoid too many SAT calls, we verify that an abstract test case is executable *a posteriori* by calling the SAT solver once with the conjunction of the FM (represented as a boolean formula) and the feature expression of the transitions of the test case. We repeat the building of an abstract test case while it is not executable. Since the largest FTS model we considered does not have a lot of behaviours exclusive to subsets of the product line, this implementation of the random algorithm works fast. This may be not the case for other models with a lot of constrained behaviour.

As discussed by Inozemtseva et al. [148], a test suite with a good coverage does not guarantee the effectiveness of this test suite. However, in a first attempt to compare our selection algorithms, coverage seems to be a reasonable metric.

7.2 Dissimilarity selection criteria

We report hereafter on our evaluation of dissimilarity driven test suite selection [88]. In order to compare the different dissimilarity selections, we define the following research questions:

- ***Dissimilarity relevance*** **(RQ.1)** How does the similarity-driven search based approach compare to all-actions and random test selection with respect to fault finding and product coverage?
- ***Distance impact*** **(RQ.2)** How does the choice of a given distance influences the results?

7.2.1 Setup

We consider 4 models from different sources with different sizes as input to different test case selection processes. The four model are: the Soda vending machine (see section 4.1), the Minepump (see section 4.3), the card payment terminal (see section

Table 7.1: Number of test cases and selection time of the all-actions test suites

Model	**Time** ($d_{allactions}$)		**Test cases** ($k_{allactions}$)	
	$\overline{time}$	σ	$\overline{count}$	σ
S. V. Mach.	1.03 sec.	0.093	3.86	0.35
Minepump	1.18 sec.	0.189	11.14	0.99
C. P. Term.	1.24 sec.	0.263	5.0	0.76
Claroline	3.42 sec.	1.814	52.86	2.95

Table 7.2: Number of faulty states, transitions, and actions seeded in the models

Model	**Faulty States**		**Faulty Transitions**		**Faulty Actions**	
	$\overline{faults}$	σ	$\overline{faults}$	σ	$\overline{faults}$	σ
S. V. Mach.	4.6	0.8	5.9	1.0	5.7	1.0
Minepump	12.4	1.4	19.3	1.7	9.6	1.5
C. P. Term.	5.3	0.9	7.9	1.1	6.8	1.1
Claroline	52.1	2.9	896.8	13.3	32.3	2.9

4.2), and Claroline (see section 4.6). In order to avoid bias using random selection, we run the evaluation 6 times for each model and each configuration of the algorithm (6 configurations overall) presented in this section.

Test suites selection: For each model, we select a test suite which satisfies the **all-actions** coverage criteria (*i.e.,* when executing all the test cases, all the actions of the FTS are executed at least once). We measure the selection time ($d_{allactions}$) and the number of test cases ($k_{allactions}$) and report in Table 7.1. The **number of test cases** and the **selection time** are used as input for the dissimilar test suite selections. To assess time impact (d in Algorithm 7) on the results, we consider the following values: $1 \times d_{allactions}$, $2 \times d_{allactions}$, $10 \times d_{allactions}$, and $100 \times d_{allactions}$ to parametrize the time during which the evolutionary algorithm runs. We configure the algorithm using different **distances** to compute the fitness function: the Jaccard index for product dissimilarity; and the Hamming distance, Jaccard index, dice and anti-dice, and Levenshtein distances for actions dissimilarity. We combine product dissimilarity and actions dissimilarity using the multiplication and average operator ($\otimes$), and also consider action dissimilarity alone. For each configuration, we run the algorithm using local and global distances for the sort. For each model, A **random** suite of $k_{allactions}$ test cases is also selected (see Algorithm 6). In total, we selected 122 test suites for each model.

Fault injection and test suites execution: Fault seeding is a popular technique to assess and compare test suites coverage [12, 13, 206]. The idea is to inject faults in SUT and measure the number of faults detected by the test suite. In this evaluation, we choose to artificially inject faults into the FTS by tagging state, transitions and actions as faulty. We randomly select states, transitions, and actions to assume them as faulty (*i.e.,* containing a fault), if a state/transition/action is selected more than once, it is only counted as 1 during the fault detection. We then execute the test suites on the FTS and consider that a fault is revealed as soon as the faulty states is

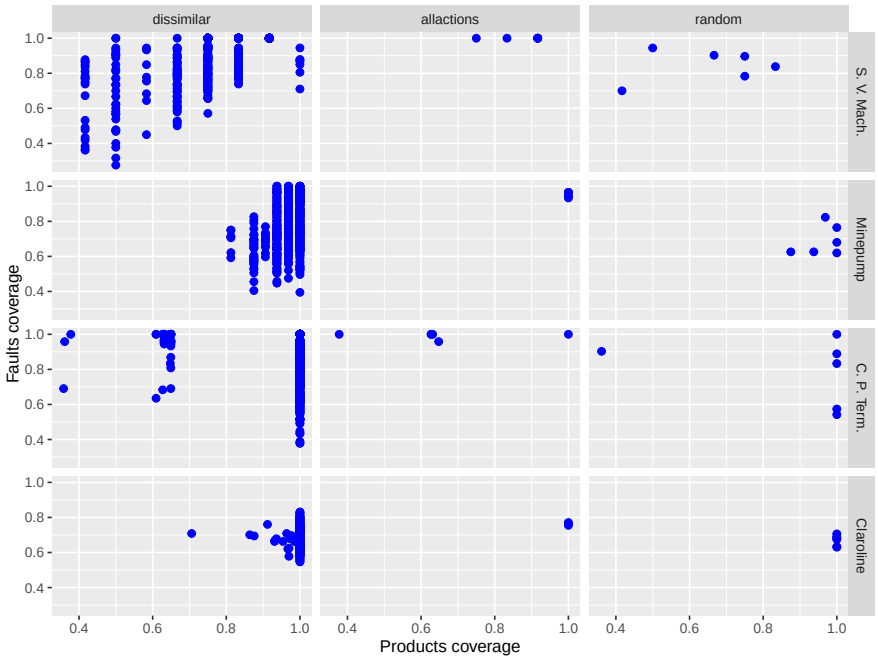


Figure 7.2: Faults coverage of the all-actions, random, and dissimilar test suites

reached, the faulty transitions are fired, and the faulty actions are executed. Using information coming from previous versions of the system (*i.e.,* from a bug tracker), this would allow one to tag elements of the model that are more likely to contain faults. In our case, we do not have access to such information and use a random selection with an upper bound of 66% of faults of the states, actions, and transitions of the FTS. This gives a measure is close to states, actions, and transitions coverage but still allows to finely compare the different approaches. Table 7.2 presents the average number of faults seeded in the different models during the evaluation.

7.2.2 Results

Figure 7.2 presents the coverage distribution of the different test suites selected using dissimilarity, all-actions coverage, and random algorithm. The x-axis is the percentage of products covered by the test suite: for a suite s and a feature model d, it corresponds to $\frac{\#prod(d,s)}{\#[[d]]}$. The y-axis is the percentage of faults (states, transitions, or actions) discovered when executing the test suite.

To characterize the test suites (*i.e.,* , the solution space of our bi-objective selection), we compute a **reference front**, by taking the Pareto front of **all** the points in Figure 7.2. This reference front contains all the sets of test cases maximising the fault and products coverages (*i.e.,* , the best solutions). We give hereafter for each model a podium with the 3 (or more if they have the same frequency) optimal configurations

Table 7.3: Hypervolumes values for the Claroline case-study

		Hamming			Jaccard			Dice		
	t	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*
	1	0.688	0.662	0.690	0.673	0.705	0.690	0.693	0.709	0.690
Loc.	*2*	0.674	0.699	0.705	0.679	0.673	0.684	0.667	0.689	0.670
sort	*10*	0.702	0.681	0.661	0.721	0.685	0.700	0.720	0.669	0.679
	100	0.667	0.693	0.672	0.672	0.720	0.713	0.691	0.703	0.678
	1	0.736	0.710	0.724	0.694	0.697	0.727	0.695	0.683	0.710
Glob.	*2*	0.711	0.708	0.755	0.722	0.718	0.715	0.723	0.670	0.729
sort	*10*	0.740	0.733	0.804	0.723	0.696	0.730	0.690	0.683	0.738
	100	0.794	0.800	0.831	0.747	0.729	0.756	0.750	0.714	0.755

		Antidice			Levenshtein		
	t	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*
	1	0.711	0.692	0.668	0.691	0.722	0.691
Loc.	*2*	0.688	0.690	0.659	0.666	0.662	0.677
sort	*10*	0.667	0.674	0.651	0.691	0.681	0.696
	100	0.696	0.685	0.655	0.655	0.678	0.687
	1	0.666	0.672	0.699	0.684	0.693	0.717
Glob.	*2*	0.677	0.705	0.728	0.708	0.687	0.733
sort	*10*	0.701	0.692	0.746	0.701	0.714	0.771
	100	0.747	0.740	0.758	0.747	0.730	0.781

All-action	**Random**
0.771	0.706

of the dissimilarity-based selection providing solutions that are on the reference front:

- **Soda vending machine** (47 optimal solutions)
  - Hamming avg., global, $t = 10$ ($freq. = 0.056$)
  - Hamming avg., global, $t = 1$ ($freq. = 0.056$)
  - Hamming avg., global, $t = 2$ ($freq. = 0.056$)
  - Hamming avg., global, $t = 100$ ($freq. = 0.056$)
- **Minepump** (6 optimal solutions)
  - Jaccard sing., global, $t = 2$ ($freq. = 0.222$)
  - Jaccard avg., global, $t = 10$ ($freq. = 0.222$)
  - Antidice sing., global, $t = 1$ ($freq. = 0.222$)
- **Card payment terminal** (64 optimal solutions)
  - Levenshtein sing., global, $t = 2$ ($freq. = 0.029$)
  - Antidice sing., global, $t = 10$ ($freq. = 0.029$)
  - Levenshtein sing., global, $t = 2$ ($freq. = 0.029$)
  - Antidice mul., global, $t = 2$ ($freq. = 0.029$)
  - Antidice avg., global, $t = 2$ ($freq. = 0.029$)
- **Claroline** (1 optimal solution)
  - Hamming sing., global, $t = 100$ ($freq. = 1.0$)

Finally, Table 7.3 presents hypervolume for the Claroline model for the different test suites. The hypervolume corresponds, for a test suite s, to the volume of the solution space dominated by s [45, 132]. A high value of hypervolume correspond to

a set of test cases with a better fault and product coverage. Rows and columns show the parameters values used for dissimilarity-based selection: the top rows indicate the actions dissimilarity distance ($diss_a$) and the operator ($\otimes$) used to combine it with the product distance ($diss_p$) or if the actions dissimilarity distance is used alone, *i.e.,* in a single-objective configuration of the algorithm (denoted by *Sing.* in the table); the leftmost columns indicate which sorting method is used (global or local) and the time considered for the algorithm ($1 \times d_{allactions}$, $2 \times d_{allactions}$, $10 \times d_{allactions}$, or $100 \times d_{allactions}$).

The raw results for the 6 executions of the different test case selection algorithms and their fault finding evaluation may be downloaded at http://projects.info.unamur.be/vibes

7.2.3 Discussion

Dissimilarity relevance: Regarding **RQ.1**, dissimilarity-based approaches are always able to obtain the optimal results in terms of fault finding ability and coverage. On the three small models, these results are sometimes matched by the all-actions and random approaches (Figure 7.2). However, the latter appear less frequently: neither random nor all-actions are on the podium of optimal solutions in terms of frequency. Additionally on the Claroline case study, the only optimal solution found is a search-based one. We therefore confirm the good results of similarity-driven testing for single product testing [218] and product selection at the feature model level [135] for behavioural test case selection in an SPL context.

The most important finding is that being fully bi-objective is not necessarily an advantage: on the 13 approaches present in the frequency podiums, only 6 are bi-objective. Additionally, a single objective approach dominates alone the Claroline case. This may be due to the nature of the case study, which is not heavily constrained: it is easy to obtain by chance dissimilar products. All-actions performance may benefit of this situation as well. Time may be involved in the explanation: for a given amount of time, a bi-objective configuration will necessarily iterate less than a single-objective one.

When bi-objective is optimal, the average (*Avg.*) composition operator gives the best results as only one *Mul.* approach appears on our podiums. Time given to the search-based algorithm has an (expected) influence on the quality of the results. This is apparent on the Claroline case where the best hypervolumes are obtained by approaches that are given $t = 100$.

Distance impact: If we consider all the podiums, Hamming and Jaccard-based distances (Dice, Antidice, Jaccard) clearly win over Levenshtein. This may seem surprising since Levenshtein is the only one that is sequence-based taking into account the order of the actions. Levenshtein is more computationally expensive than Hamming and Jaccard-based distances, implying less iterations of the algorithm for a given amount of time. When employed alone (single-objective) on actions, it appears to be the second most performing distance on the Claroline case.

7.2.4 Threats to validity

To keep the comparison fair between the different test suite selection algorithms, we use the same number of test cases and duration time of the all-actions test suite selection to parametrize random and dissimilar test suites selections. As the dissimilar test suite selection is based on a (1+1) evolutionary algorithm [95], it is very sensitive to the maximal execution time (d). The all-actions test suite selection may be very fast (for the soda vending machine for instance). In order to assess the time influence in the quality of selection for the evolutionary approach, we chose to repeat the test case selection with different d values.

We chose to use a (1+1) evolutionary algorithm [95] to maximize the dissimilarity of the selected test suite. This algorithm is simple and to parametrize, and it showed good results to select products to test [135]. Many other algorithms, like adaptive random testing [57], used to select dissimilar test cases exist [131]. A comparison between those different algorithms is left for future work.

The complete process described in Section 7.2.1 has been repeated 6 times for each model on a Ubuntu Linux machine (Linux version 3.13.0-65-generic, Ubuntu 4.8.2-19ubuntu1) with an Intel Core i3 (3.10GHz) processor and 4GB of memory. The complete experiment took approximately 4 days.

7.3 Behavioural coverage of products sampling techniques

Products selection approaches solely based on the feature model, such as t-wise testing, have gained momentum as they are able to scale to large SPLs. However, these methods are agnostic with respect to behaviour: the sampled products have no reason to satisfy any given structural behavioural criterion. In this section, we report on our investigation [87] on the behavioural coverage of two products selection approaches: t-wise selection and dissimilarity-based selection. To do so, we describe hereafter our **initial assessment** in order to answer the following research question:

- ***Behavioural coverage*** Which behavioural coverage do dissimilarity and t-wise sampling achieve?

To address this question, we use four SPLs: each is modelled by a FTS and its related feature model. We then apply t-wise and dissimilarity techniques to sample a set of products. By projecting the FTS for each sampled product, we get a labelled transition system from which it is possible to compute the coverage of the FTS representing the whole SPL.

Preliminary results indicate that full coverage of states, transitions and actions can indeed be achieved with few products (no more than 3) and that 3-wise sampling worked best in these cases. Dissimilarity works better than t-wise for $t = \{1,2\}$, although a detailed comparison is beyond the scope of this assessment. All these samplings obtain full coverage with more products than needed, indicating an interesting potential for mixing products coverage and behavioural coverage rather than systematically considering them in isolation.

Table 7.4: SPLCAT and PLEDGE parameters

Model	**SPLCAT**	**PLEDGE**		**Config.**
	t	x	d	
Soda V. M.	1	3	30 sec.	3
	2	6	30 sec.	6
	3	14	30 sec.	14
Minepump	1	2	30 sec.	2
	2	7	30 sec.	7
	3	13	30 sec.	13
Aero UC5	1	2	60 sec.	2
	2	8	60 sec.	8
	3	15	60 sec.	15
Claroline	1	6	60 sec.	6
	2	21	60 sec.	21
	3	71	60 sec.	71

7.3.1 Setup

This initial assessment is performed on the soda vending machine, the Minepump, the Sferion™landing symbology function, and Claroline. Regarding t-wise sampling, we elicit the SPLCAT tool [155] for its performance [133] and PLEDGE [135] for similarity testing. Behaviour of the different models is represented using FTSs. To measure the behavioural coverage we use state, transition and action coverage for each product selected using the different structural criteria.

To perform our assessment, we carry out the following steps for each model and each tool (SPLCAT and PLEDGE):

(i) select a set of products from the feature model using each tool;

(ii) project the FTS on each product, to get the behavioural model (*i.e.,* , the LTS) corresponding to this product (we use the projection operator defined by Classen et al. [63]): it creates a new labelled transition system by removing all transitions that may not be executed by the product, unreachable states, and unused actions (feature expressions are dropped during the process);

(iii) for each product, compute the coverage of its behavioural model: divide the number of states, transitions, and actions in the projection by the number of states, transitions, and actions in the FTS. The cumulated coverage is calculated by dividing the number of different states, transitions, and actions in the projections by the number of different states, transitions, and actions in the FTS. The states, transitions, and actions appearing more than once are thus counted only once.

The feature model of each case study is used as input to the SPLCAT and PLEDGE tools to select sets of products. The SPLCAT tool can sample, for a given model and a given t between 1 and 3, a set of valid products satisfying the 1-wise, 2-wise, or 3-wise coverage criteria over the feature model. The PLEDGE tool can sample, for a given feature model, a given number of products x, and a certain time d, a set containing x products, using an evolutionary algorithm maximising the dissimilarity (in terms of features selected) amongst products in this set. We use as x the number of products

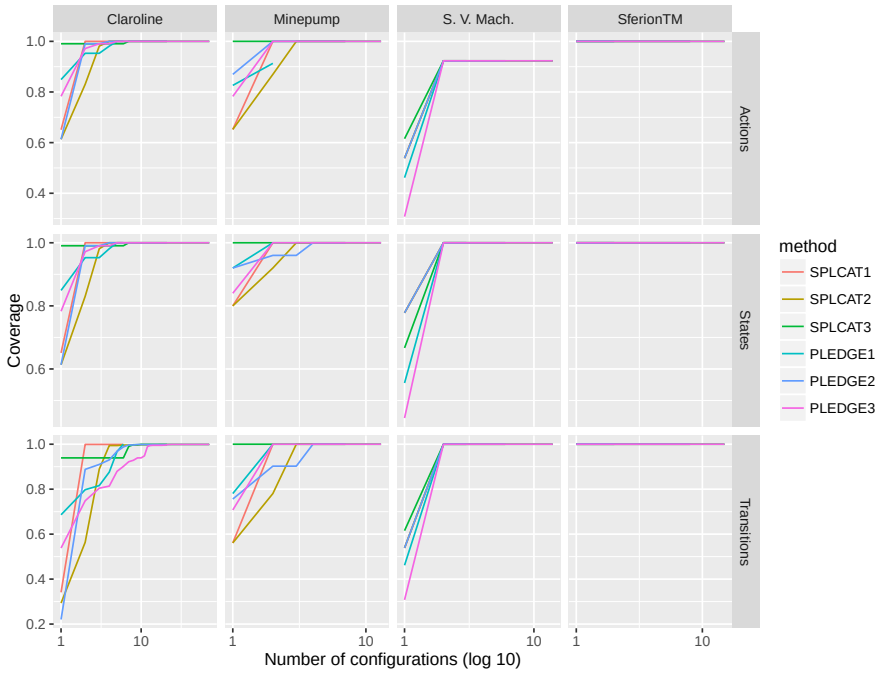


Figure 7.3: Behavioural coverage of products selected using SPLCAT and PLEDGE tools

sampled by SPLCAT for each model. The d parameter has the default value 60 seconds, except for smaller models where it had, after few trials, to be reduced to 30 seconds to avoid memory errors during execution. Table 7.4 presents the different parameters used for each model and the number of sampled products. We ran the tools on a Ubuntu Linux machine with an Intel Core i3 (3.10GHz) processor and 4GB of memory.

7.3.2 Results

Figure 7.3 presents the accumulate behavioural coverage in terms of actions, states, and transitions of the products selected using SPLCAT and PLEDGE. Results for the soda vending machine exhibit the same tendencies as those of the Minepump. The Sferion™landing symbology function has a coverage of 100% for states, transitions, and actions for every product sampled using SPLCAT and PLEDGE, because its FTS has only 4 transitions specific to 2 different features, 2 transitions for each feature, present in each product. Although the number of selected products is higher (as shown in Table 7.4), in general, the cumulated coverage value did not increase further after 7 products for states and actions coverage and 14 products for transitions coverage.

7.3.3 Discussion

Regarding our research question, for the considered case studies and settings, we rapidly obtain a complete coverage: relatively few products are needed to fully cover states, transitions, and actions for the two tools reported here. This is to be expected for our small case studies, but on a larger model (Claroline), this tendency tends to be confirmed. Of course, this assessment need to be replicated on a larger sample of behavioural models, but this seems encouraging for the usage of structural coverage criteria at the feature model level beyond the scope of detecting behavioural feature interactions [58].

On the small feature models, exact t-wise coverage (SPLCAT) yields better coverage on all our behavioural criteria for $t = 3$. This further indicates that higher values than the usual 2-wise are relevant [293] and therefore should be used when the number of products is reasonable. PLEDGE tends to outperform SPLCAT on 1-wise (each feature is covered at least once) and 2-wise with a smaller number of products. Since for a given execution time, a smaller number of products means more time to evolve the population (set of products) and less time spent computing distances amongst them, maybe the poor performance of PLEDGE for $t = 3$, the largest number of products, can be explained in such a way. We also use the local maximum distance (called *greedy* in the tool), which is outperformed in terms of coverage by the global maximum distance (called *NearOptimal* in the tool) [133]. It also seems that some of the memory errors we run into (related to thread creation) can be accounted by this default choice of the algorithm. Indeed, threads are associated to evolutions of the population and local distance algorithm is fast: we therefore have a thread explosion problem on these small feature models. Therefore, additional settings and trade-offs need to be investigated to be able to compare the tools. Detailed tool comparison in this context is beyond the scope of our research question and is therefore left for future work.

Finally, for both approaches, this initial assessment shows that there is also a need for prioritisation and optimal behavioural coverage. For example, amongst the 71 products selected by SPLCAT ($t = 3$), only one is sufficient to cover all states on both the Minepump and Claroline models. This product can be found directly using the all-states coverage test case selection algorithm and the p-coverage upper bound property. If dissimilarity and t-wise coverage are shown to consistently sample products that achieve good behavioural coverage, as this assessment suggests, then they can be used as first filters on very large feature models (assuming an intractable FTS for a test case selection algorithm) to prune the FTS and then perform a behavioural coverage driven test case selection. Prioritisation may be initiated at the FTS level and combined with behavioural/structural criteria. There is no such one-criteria-fits-all approach in this endeavour: an all-states criterion may poorly cover transitions (*e.g.*, on the Claroline case). Exploring synergies between these criteria, both at the structural and behavioural models, therefore seems the best option.

7.3.4 Threats to validity

The PLEDGE input parameters are arbitrarily chosen. To keep a fair comparison between the results of the PLEDGE and SPLCAT tools, we keep the same number of products x as sampled by SPLCAT. Estimating the time d, however is more tricky. In their comparison, Henard *et al.* [133] use the same sampling time as SPLCAT. Unfortunately in our case, some t-wise computations take less than 1 second in SPLCAT and PLEDGE does not allow to enter such values. Thus we initially went for the default values provided by the tool. As mentioned above, playing with a wider range of parameter values and with different similarity algorithms will mitigate this threat.

FTS models relate variability to behaviour using feature expressions on transitions, other modelling languages may relate variability to behaviour in other ways (*e.g.,* associate variability to states instead of transitions), which will give different results for the state, transitions and actions coverage. FTS is a basic formalism to which we can easily transform other modelling languages and mappings. Thus, we can investigate the influence of the mapping between features and behavioural models.

7.4 Usage selection and prioritization criteria

In this section, we report on the **feasibility** of using family-based test selection [84, 85] by applying it on two systems: the first one is Claroline (see section 4.6), and the second one is the landing symbology function, part of Sferion™(see section 4.4). Validation of the product-based test selection has been done by Samih *et al.* [266]. We assess the feasibility of our approach using the following research questions:

- ***FTS pruning*** **(RQ.1)** What are the reductions gains (model pruning) achieved by applying statistical prioritization?
- ***Modelling*** **(RQ.2)** What is the modelling effort induced by our approach and what are the consequences of modelling choices?
- ***Scalability*** **(RQ.3)** How does prioritization scale to increasing probability ranges and what are the implications for testing?

It is difficult to provide precise thresholds for these criteria. Testing should fit a given budget, which is a complex trade-off involving testing time, human and infrastructure resources, level of system coverage desired, *etc.* Statistical approaches covered in this thesis are flexible to meet such a tradeoff. We argue that fixing meaningful thresholds values requires additional experience, especially in industrial settings where they both can be set and assessed. We therefore leave this issue for future work, giving both quantitative and qualitative information stemming from our experience applying our techniques on Claroline and Sferion™.

7.4.1 Claroline, an online course management system

Setup: We derived the usage model, from the same anonymized Apache web-server log as for Claroline FTS (see section 4.6.2), using a classical bigram inference

Input: $sessions$: the set of non empty user sessions
Output: um: a usage model representing a navigational model for the given user sessions

1 **begin**
2 $S = \{s_0\}$; $Act = \emptyset$; $trans = \emptyset$; $\tau(s_0) = 1$;
3 **for** **sess** $\in$ **sessions** **do**
4 $S.add(sess[0])$;
5 $Act.add(req(sess[0]))$;
6 $tr = s_0 \xrightarrow{req(sess[0])} sess[0]$;
7 $trans.add(tr)$;
8 $count(tr) = count(tr) + 1$;
9 **for** $i \in [1; sess.size[$ **do**
10 $S.add(sess[i])$;
11 $Act.add(req(sess[i]))$;
12 $tr = sess[i-1] \xrightarrow{req(sess[i])} sess[i]$;
13 $trans.add(tr)$;
14 $count(tr) = count(tr) + 1$;
15 **end**
16 $Act.add(req(s_0))$;
17 $tr = sess[sess.size - 1] \xrightarrow{req(s_0)} s_0$;
18 $trans.add(tr)$;
19 $count(tr) = count(tr) + 1$;
20 **end**
21 **for** $(s_k \xrightarrow{\alpha_i} s_l) \in trans$ **do**
22 $$P(s_k \xrightarrow{\alpha_i} s_l) = \frac{count(s_k \xrightarrow{\alpha_i} s_l)}{\sum_{(s_k \xrightarrow{\alpha_j} s_m) \in trans} count(s_k \xrightarrow{\alpha_j} s_n)};$$
23 **end**
24 $um = (S, Act, trans, P, \tau)$;
25 **return** um;
26 **end**

Algorithm 13: Bigram usage model building

technique [114, 269, 292]: algorithm 1 has been adapted to produce a usage model giving algorithm 13. As previously, we consider resource names in the user sessions as states (lines 4 and 10) and add transitions between those names if they appear successively in the user sessions (lines 7 and 13). To compute the probability of each transition, we count the number of occurrences of the transitions (lines 8, 14, and 19) and for each transition, we divide this count by the total number of occurrences having the same source state (lines 21 and 22).

The $allpaths$ algorithm has been applied four times to the Claroline usage model with a maximal length (l_{max}) of 98 (the maximal path length without any loop in the usage model), a maximal probability (Pr_{max}) of 1, and four different minimal probabilities (Pr_{min}), 1e-4, 1e-5, 1e-6, and 1e-7, to observe patterns. Additionally,

Table 7.5: Claroline family-based test selection results

	run 1	**run 2**	**run 3**	**run 4**
l_{max}	98	98	98	98
Pr_{min}	1e-4	1e-5	1e-6	1e-7
Pr_{max}	1	1	1	1
a.t.c.	211	1,389	9,287	62,112
p.a.t.c.	211	1,389	9,287	62,112
$\overline{size}$	4.82	5.51	6.35	7.17
σ	1.54	1.54	1.62	1.66
$\overline{proba.}$	2.06e-3	3.36e-4	5.26e-5	8.10e-6
σ	1.39e-2	5.46e-3	2.12e-3	8.18e-4
FTS' st.	16	36	50	69
FTS' tr.	66	224	442	844

the algorithm has been parametrized to consider each transition only once (*i.e.,* a transition does not appear more than once in a selected trace). This modification has been made since we discovered after a few runs that the algorithm produced a lot of traces with repeated actions, which is of little interest for product prioritization. Repetitions were due to the huge number of loops in the Claroline usage model.

Results: Results for the four different minimal probabilities are presented in Table 7.5. Execution times range from less than a minute for the first run with 211 abstract test cases (a.t.c) to ±8 hours for run 4 with 62,112 abstract test cases on a Ubuntu Linux machine (Linux version 3.13.0-65-generic, Ubuntu 4.8.2-19ubuntu1) with an Intel Core i3 (3.10GHz) processor and 4GB of memory.

All abstract test cases selected in the usage model are positive (p.a.t.c.), this is caused by the nature of the Claroline FM: most of the features are independent from each other and few of them have exclusive constraints. The bigram solution used to generate the usage model fits well in this case, as there is no abstract test case selected in the usage model that has been rejected. Sprenkle et al. [292] experimentally demonstrate that increasing the n in n-gram generation of the usage model does increase the size of the generated model in a non linear way (as long as n is between 2 and 10). Increasing the n value in our case would just result in an unnecessary increase of the model complexity.

As expected, the average size of the abstract test cases ($\overline{size}$) increases as the Pr_{min} decrease (a lower probability allows longer traces to be selected). The average size of the user sessions used to generate the usage model is 9.88.

As explained in Algorithm 8, it is possible to prune the original FTS using the positive abstract test cases in order to consider only the valid products capable of executing those test cases. In this case, it eventually reduces the number of states (FTS' st.) and transitions (FTS' tr.) from 106 and 2,055 (resp.) to 16 and 66 (resp.) in run 1 and to 69 and 844 (resp.) in run 4. As expected, by controlling the interval size we can reduce the number of traces to be considered and yield easily analysable FTS'.

Table 7.6: Sferion™landing symbology function family-based test selection results

	run 1	**run 2**	**run 3**	**run 4**	**run 5**
Pr_{min}	1e-1	1e-2	1e-3	1e-4	1e-5
Pr_{max}	1	1	1	1	1
a.t.c.	0	0	8	50	306
p.a.t.c.	0	0	8	50	306
$\overline{size}$	0	0	15	16.68	18.58
σ	0	0	0.76	1.17	1.39
$\overline{proba.}$	0	0	2.19e-3	5.67e-4	1.19e-4
σ	0	0	1.51e-3	9.30e-4	4.23e-4
FTS' st.	0	0	18	23	23
FTS' tr.	0	0	12	12	12
FTS' act.	0	0	27	40	42

	run 6	**run 7**	**run 8**	**run 9**	**run 10**
Pr_{min}	1e-6	1e-7	1e-8	1e-9	1e-10
Pr_{max}	1	1	1	1	1
a.t.c.	1870	8622	36582	123534	err
p.a.t.c.	1870	8622	36582	123534	err
$\overline{size}$	20.85	22.99	25.15	27.17	err
σ	1.62	1.77	1.88	1.97	err
$\overline{proba.}$	2.20e-5	5.02e-6	1.21e-6	3.59e-7	err
σ	1.76e-4	8.25e-5	4.01e-5	2.18e-5	err
FTS' st.	23	23	23	23	err
FTS' tr.	12	12	12	12	err
FTS' act.	42	42	42	42	err

7.4.2 Sferion™landing symbology function

Setup: Engineers will probably have to run the algorithm several times using different minimal and maximal probabilities intervals in order to refine the selection. In our first attempt, we applied our trace selection algorithm 10 times with a maximal probability value of 1 and a minimal probability value ranging from 10^{-1} to 10^{-10} and a maximal length of 50. As explained hereafter, some of those runs did not return any relevant results.

Results: The results of the execution are showed in table 7.6. The run 10 did not return any results due to the too wide range of considered probabilities, giving too many possible paths in the usage model. This is not a problem for our approach as a wide range of probabilities is not very useful for prioritization. According to those results, the interval with the most probable abstract test case is between 1e-3 and 1e-2. We re-run the algorithm with the minimal probabilities 5e-3 and 2.5e-3: the execution with an interval between [0;5e-3] returned no test case; the execution with an interval between [0;2.5e-3] returned 2 test cases (*test case 1* and (*test case 2*)) with an average probability of 4.62e-3 and a length of 14. Those two abstract test cases are the most probable behaviours of the landing symbology function and may be executed by all the products of the product line.

In order to get a more concrete product, we select longer abstract test cases from the usage model by using the classical state-coverage criterion [206]. This criteria specifies that, when executing a test suite on the system, all the states of the system have to be visited at least once. Generating an abstract test case from the usage model using this criteria gives us one abstract test case visiting all states (*test case 3*). As for previously selected abstract test cases, we execute it on the FTS to ensure that there exists at least one product able to exercise this behaviour: this gives us a set of 64 products.

7.4.3 Discussion

We organise our discussion on the final results regarding feasibility for statistical prioritization SPL testing according to the criteria mentioned above: (i) FTS pruning; (ii) modelling; and (iii) scalability.

FTS pruning (RQ.1): In both cases, it was possible to substantially prune the FTS models according to frequent behaviours: from 28% to 85% reduction *w.r.t.* the number of states (for Sferion™and Claroline) and up to 99,994% reduction *w.r.t.* to transitions (for Claroline). These important reduction factors are interesting in the sense that it is possible to use statistical selection to deal with additional computationally expensive coverage criteria that would not be directly applicable on the original model (*e.g.,* all-paths coverage on the Claroline FTS [91]).

Regarding the number of products associated with the selected test cases, the situation is less favourable. In the Claroline case, the least probable test case in run 1 is already associated to 260 products. The main reason is that test cases are small in size yielding short associated feature expressions. Most Claroline users therefore seems to visit few pages after the login one. Because the source Apache Log is anonymised, it is impossible to investigate further in this direction. To note that the set of 260 products is reduced to 20 products if we consider only courses available to student with an id, which is the most classical scenario in the University of Namur Claroline instance. The tester will have to use his knowledge of the application domain in order to reduce the number of products to test.

The Symbology function exhibits more complex behaviours as witnessed by test cases' sizes. For Sferion™, there are test cases that can be executed by all the products of the SPL. While from pure product selection perspective this is a bad result, two additional observations need to be made. First, the usage model was provided by experts to focus on the most relevant behaviours: it seems they did perform correctly this task as most part of the described behaviour concern all the products of the SPL. Second, there are opportunities to reuse abstract test cases amongst products: these two abstract test cases can be used to derive a small number of concrete test cases covering all products. This strategy can be used to explore interaction problems [224]. Finally, feature models of our considered systems have very few constraints (*e.g.,* $Mark\_landing\_position \Rightarrow HOCAS$, $Check\_for\_obstacles \Rightarrow OWS$, *etc.*) amongst features, which clearly influence

product reduction ability. While such an open feature model is not surprising for a web based system, this is more unusual for an embedded SPL.

Modelling (RQ.2): Using statistical prioritisation in both case studies involved some modelling: the family-based scenario allowed us to extract automatically the usage model using a machine learning technique, while the Sferion™product one relied on SPL and testing experts to explicitly provide the required usage model. However, what is common to both scenarios is the necessity to provide variability models (in OVM or TVL) and mapping from features to behaviours either by means of FTS or mapping matrices [266, 268].

Both approaches try to keep requirements from test models separated. Such a separation of concerns does not guarantee that these models are correct (learning behaviours from anonymous logs entails approximations and hand-made usage model are not free from biases either) but helps finding discrepancies as they are generally provided by stakeholders having different perspectives and skills. Keeping these models separated was also useful to integrate our approach with tools like MaTeLo that do not take into account natively features in their usage models but provide additional facilities such as risk management or customer satisfaction during test case selection.

Separation of the usage model from the FTS also allows to use different usage models, depending on the objective of the test engineer. For instance, Claroline has a fine grained access control system. In its basic setup, it comes with three user roles: Administrator, Teacher and User. According to its role, a user may or may not perform different actions and access different functionalities. Claroline also allows public access to some pages and functionalities (*e.g.,* a course description) to anonymous users. We think that since those four user roles have very different usage profiles, a better approach would be to create four different usage models, one for each user role. Unfortunately, it was not possible in our case with the provided Apache access log since the user roles have been erased in the anonymisation process.

Keeping the usage model and the FTS separated may be detrimental to the analysis as some invalid abstract test cases may be first extracted from the usage model and then removed. Even if this was not the case on the considered SPLs, this may happen in more constrained SPLs. One strategy could be to start with separated models and to merge them in a feature-aware usage model (*e.g.,* using feature-aware discrete-time Markov chain [262, 298]) once enough confidence is gained on both models. This is left for future work.

The effort spent in modelling activities depends on the case study: for the Claroline case study, the usage model has been automatically generated, the FTS has been semi-automatically generated and the variability model has been hand crafted. Given the size of Claroline (442.399 LOC), the total effort spent in modelling activities is deemed reasonable (around 7 days). The Sferion™case study models a critical system, the modelling and testing efforts are important but have to be supported by the company in order to guaranty a safe and sound product. The additional effort required to derive the FTS from the Sferion™models is small (around 2 days).

Scalability (RQ.3): Final results show that the scalability of our implementation mainly depends on the $[Pr_{min}, Pr_{max}]$ interval and the shape of the usage model. We notice that if the model is large (Claroline) and/or the probability interval very large computation time obviously increases and may even lead to no result at all (*e.g.*, run 10 in Table 7.6). In this case we encountered memory overflows. The **allpaths** algorithm used in our implementation seems to perform well as long as the $[Pr_{min}, Pr_{max}]$ interval is not too wide, even on large usage model (like the Claroline case). Thus, we rely upon the tester to choose a relatively small probability interval in order to extract behaviours that results in the desired amount of test cases and products. So far, we explored these intervals manually to find tradeoffs. This exploration can be automated if additional criteria (such as the maximum number of products desired) are specified. Other state space exploration techniques will have to be investigated to improve the algorithm (*e.g.*, limit the length of the selected test cases is amongst the simplest, or use simulation techniques [27]). It should be noted that, for more specialized explorations, such as finding the most probable path, dedicated algorithms like the one proposed by Viterbi [313] may be used.

7.4.4 Threats to validity

To implement our approach, we choose to use a **allpaths** algorithm with some restrictions (maximal length of the selected test cases) in order to avoid infinite executions. This choice may be not optimal but the **allpaths** exploration ensure (in worst case) a complete exploration of the usage model. The input models (DTMC as a usage model, FTS and TVL) are not the only possibilities to represent usages, behaviour, and variability of the SPL. In our second SPL, we showed how we translate other input models in order to apply our approach.

7.5 Mutants execution

This section reports on our comparison [89] between the FMM approach, that uses a compact representation to factorize the mutants execution against each test case, and the enumerative approach, where each mutant is executed individually against each test case, in terms of execution time. And evaluates the usage of FMMs to perform higher-order mutation analysis. As in Section 6.2, we adopt a product-based strategy: the systems under test, used as original systems to perform the mutation analysis, are products derived from our case studies described in Chapter 4. In order to conduct this assessments, we formulate our research questions as follows:

- ***Execution time*** **(RQ.1)** How does the FMM scheme compare with the enumerative approach in terms of execution time?
- ***Higher-order mutation*** **(RQ.2)** Is higher-order mutation under the FMM scheme tractable?

Table 7.7: Test suites characteristics

Model	Random test suite		All-actions test suite		
	$\overline{size}$	σ	*count*	$\overline{size}$	σ
Soda V. Mach.	4.78	1.34	3	5.33	2.08
Minepump	5.65	1.23	9	6.11	1.45
Claroline	17.11	16.97	11	13.18	9.20
AGE-RR	21.13	24.58	274	27.11	33.62
Elsa-RR	10.57	13.12	109	21.10	33.45
Elsa-RRN	10.45	14.05	148	23.20	43.49
Random 1	469.62	279.34	2	468.50	118.09

Table 7.8: Mutants count per operator

Model	WIS	TMI	AEX	TDE	TAD	AMI	SMI	Total
Soda V. Mach.	1	1	1	1	1	1	1	7
Minepump	2	4	4	4	4	3	2	23
Claroline	9	188	205	204	205	189	9	1,009
AGE-RR	73	525	663	663	663	516	75	3,178
Elsa-RR	36	102	121	121	121	106	38	645
Elsa-RRN	57	153	177	177	177	155	57	953
Random 1	942	1,276	1,365	1,365	1,365	1,295	954	8,562

7.5.1 Setup

To perform the assessment, we project the FTSs of the case studies described in Chapter 4 to get the original LTSs that will serve as basis for our mutation analysis. We consider products from different sources with varying size. Our models are:

- a soda vending machine product (*Soda V. Mach.*) that includes all features;
- a Minepump product (*Minepump*) that includes all features;
- a Claroline product (*Claroline*) that includes all features and an *Admin* user;
- three WordPress products (*AGE-RR*, *Elsa-RR*, and *Elsa-RRN*) that include all features of their respective feature models
- one random model (Random 1).

Test cases: For each model, we select one test suite using random walks on the LTS and one test suite satisfying the all-actions criterion. The test suites are then executed with the enumerative and the FMM processes. Table 7.7 records the average size (and standard deviation) of the randomly selected test cases, the size of the selected all-actions coverage-driven test suite and the average size (and standard deviation) of its test cases. The size of the random test suite is arbitrarily fixed to 100 test cases.

Model mutants: We used the operators presented in Annex A. Operators modifying states (WIS and SMI) or transitions (TMI, AEX, TDE, TAD, and AMI), respectively, were applied arbitrarily for 1/10 of the number of states or transitions, respectively, in the model (with 1 as bottom value). Since the operands are randomly chosen, we forbid multiple applications of any operator on the same operands to avoid

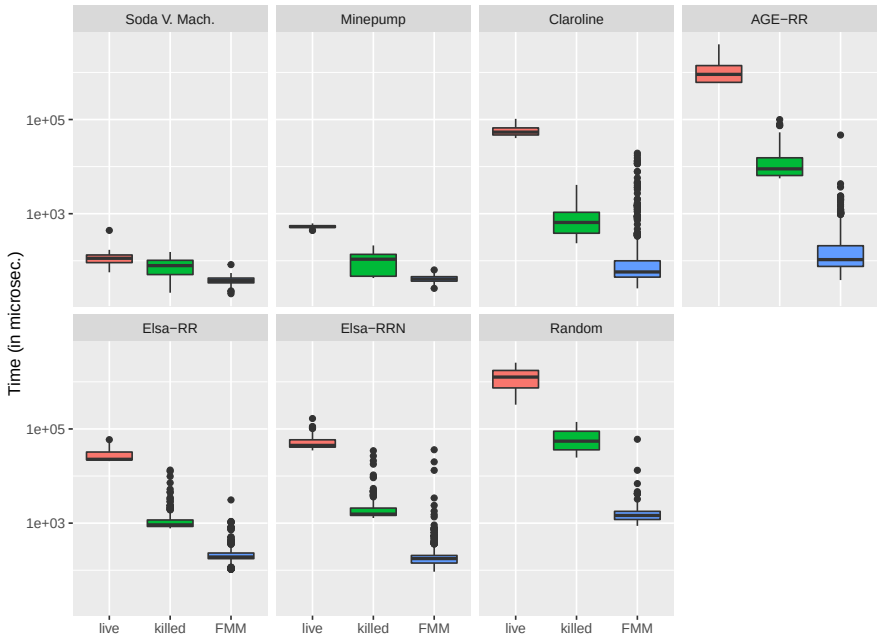


Figure 7.4: Execution time required by test cases to executed with live and killed mutants and the FMM mutants

duplicated mutants [237]. Table 7.8 presents the number of mutants generated per operator for the studied models.

Mutants execution: To avoid execution time bias from the underlying machines, we execute each test case 3 times with each considered mutant (for the enumerative version) and on the FMM. Experimentation was performed on an Ubuntu 14.04 LTS Linux (kernel 3.13) machine with Intel Core i3 (3.10GHz) processor and 4GB of memory. The complete experiment took approximately 2 weeks.

7.5.2 Results

Figure 7.4 presents the distribution of the test execution time (in logarithmic scale on the *y* axis) for each studied model with a box plot. The first two columns represent the total execution time taken by each test case when executed on the live mutants and on the killed mutants according to the enumerative approach. The third box presents the execution time of the FMM (FMM approach). Note that while the killed mutants do not require a complete execution in the enumerative approach, it is required for the FMM mutants. This might provide an advantage to the enumerative approach. To assess this, we consider the killed and the live mutants separately. In all cases, we measure only the execution of the models, avoiding time bias due to I/O operations. As the execution time of a test case partially depends on its size, the

Table 7.9: All-order mutation scores

Model	# mutants	All-actions			Random		
		#Lv.	*MS*	*Time*	*#Lv.*	*MS*	*Time*
Soda V. Mach.	127	1	0.99	1.10	1	0.99	17.67
Minepump	8,388,607	1	>0.99	1.84	1	>0.99	15.72
Claroline	5.49e+303	Timeout			Timeout		
AGE-RR	4.71e+956	Timeout			Timeout		
Elsa-RR	1.46e+194	2916	>0.99	37.78	144	>0.99	10.19
Elsa-RRN	7.61e+286	36	>0.99	150.32	16	>0.99	83.04
Random 1	2.62e+2577	Memory overflow			Memory overflow		

high number of outliers in Figure 7.4 is explained by the variation of the test cases sizes.

For the enumerative approach, executing a test case on mutants that will remain live takes more time than executing the same test cases on mutants that are killed. This was expected since killed mutants do not require a complete execution of the test case. In both cases, the FMM execution runs faster, *i.e.*, running a test case on all the mutants at once is very fast, despite the more complex (needed) exploration of the FMM's FTS. Detailed statistics over the execution time of the models and mutation scores are presented in Annex B.

7.5.3 Discussion

Execution time: Regarding **RQ1**, the box plots of Figure 7.4 (and the values in Annex B) confirm that the execution time required by the FMM approach is considerably lower than the time required by the enumerative approach. The difference escalates to several orders of magnitude when considering live mutants. The difference between family-based and enumerative approaches increases with the size of the model, indicating the improved scalability of our approach.

To evaluate the statistical significance, we use a Wilcoxon rank-sum test for the different models we considered: we obtain a p-value of $1.343e-09$ for the random model and p-values smaller than $2.2e-16$ for the other models, confirming the hypothesis that FMM significantly outperforms the enumerative approach, when considering 0.001 significance level.

Higher-order mutation: Table 7.9 presents the number of all-order mutants for our models, the number of mutants live after executing the random and all-actions test sets (computed using SAT4J 2.3.5), and their mutation score. For each test set and model, the table records the number of possible mutants (*# mutants*), the number of live mutants after the test set execution (*#Lv.*), the mutations score (*MS*) and the SAT computation time (*Time*) in seconds. *Memory Overflow* denotes an overflow during SAT solving, improving this step by, for instance, reducing the boolean formula to process is part of our future work. Columns 5 and 8 give the SAT-solving computation time (we set a timeout of 12 hours).

Overall, our results suggest that higher-order mutation under the FMM scheme is tractable, answering positively to **RQ2**. In particular, all-order mutation achieves very good mutation scores ($MS \geq 0.99$) when compared to first-order mutation when this score can be computed. In our future work, we intend to: improve the scalability of mutation score computation; and assess the practical relevance of higher-order in test sets comparison.

Only one mutant is live for the soda vending machine and the Minepump products. This mutant is a first order mutant resulting from the TAD operator. Indeed, the TAD operator adds new transition which cannot be detected by test cases solely selected from the original LTS, since this transition does not exist in this model. All-order mutation enables to quickly kill mutants of any order an to focus on the interesting ones from a selective mutation perspective. For example, the 2,916 remaining live mutants resulting from the execution of the all-action test suite are relevant to study the mutation operators involved. Of course, they can also be used to select test cases killing them in order to augment the test suite. Exploring all-order mutation score in selective mutation or test case selection scenarios are part of our future work.

7.5.4 Threats to validity

We chose to apply mutants for 1/10 of the states and/or transitions of the mutated model. This might result in more (or less) mutants than needed for the larger models. However, this is expected when using mutation. Additionally, since model-based mutation is applied to the system's abstraction, abstract actions represent many concrete actions. It is therefore important to ensure a good coverage of most of the model actions.

TS and FTS executions are different, and do not use the same algorithms. In order to decrease the bias in measuring execution time, both executions of the models have been done using VIBeS, our **Variability Intensive Behavioural teSting framework** Java implementation. The two execution classes are different but use a variant of the same algorithm described in Section 6.2. Moreover, we used the `Stopwatch` Java class to measure the call to the execute method (*i.e.,* model loading and result writing time have been omitted). Finally, we ran each test case 3 times on each mutant model (LTSs and FMMs) to avoid bias due to process concurrency.

7.6 Mutant equivalence analysis

This section presents our empirical assessment of the Automata Language Equivalence (ALE), Random Simulation (RS), and Biased Simulation (BS) approaches to detect equivalent mutants [90]. As in Section 6.3, we adopt a product-based strategy: mutants are generated from products, derived from the case studies defined in Chapter 4 (and from 4 additional randomly generated models). To conduct this assessment, we define the following research questions:

- ***Random/biased simulations and ALE*** **(RQ.1)** What is the impact of weak and strong mutation on BS/RS *vs.* ALE performance?

- ***Non-equivalent mutant detection*** **(RQ.2)** How many non-equivalent mutants are effectively detected by the RS and BS approaches?
- ***Worst case scenario*** **(RQ.3)** What are the worst case execution times for the ALE and BS/RS approaches?

7.6.1 Setup

To answer these RQs, we consider several models of different kinds of systems and apply the following procedure to each of them:

(i) we generate a set of mutants from the model using the operators presented in Annex A for orders 1, 2, 5, and 10;
(ii) for each order, we sample 100 non-equivalent mutants (using the ALE algorithm to guarantee non-equivalence) to form the mutant set M;
(iii) for each mutant in M, we measure the execution time and result of: 3 executions of weak mutation random and biased search (WM RS/BS), and 3 executions of strong mutation-biased search (SM BS) algorithms[1] with 4 different values of δ and ϵ; and the executions of the ALE algorithm.

In the following we detail the different steps of the procedure. The assessment has been performed on a Debian 3.16.7 x86_64 GNU/Linux running on a 16 cores, 2.2 GHz, 16Gb RAM virtual machine.

Models: We carry out the assessment on 12 different models coming from different sources and with varying size. The models are:

- a soda vending machine product (*S. V. Mach.*) that includes all features;
- a card payment terminal product (*C. P. Term.*) that includes all features;
- a Minepump product (*Minepump*) that includes all features;
- a Claroline product (*Claroline*) that includes all features and an *Admin* user;
- four WordPress products (*AGE-RR*, *AGE-RRN*, *Elsa-RR*, and *Elsa-RRN*) that include all features of their respective feature models;
- four random models (Random 1-4).

Mutant generation and sampling: First-order mutants are generated using the operators presented in Annex A. Each operator is applied (arbitrarily) 10 times on the *S.V.Mach.*, *C.P.Term.*, and *Minepump* products. Due to the small size of the models, applying the same mutation operator more than 10 times is not relevant. Operators are also applied (arbitrarily) 500 times on the other models. In the same way, N-order mutants (with N equal to 2, 5, or 10 in our case) are generated by applying the same operators 10 or 500 times (depending on the model) on $(N-1)$-order mutants. After the generation, we perform a random sampling of 100 mutants (when available) for orders 1, 2, 5, and 10, giving us a set M with 370 mutants for the *S.V.Mach.*, *C.P.Term.*, and *Minepump* models, and 400 mutants for the other models. To ease mutant generation, we use the compact representation provided by FMMs.

[1] As explained in Section 6.3, SM RS is not considered for the assessment due to the poor results during our initial attempts.

Non-determinism: We checked all the 4710 mutants and found that only 3.54% of them are non-deterministic (*i.e.,* there exists a sequence of actions for which there is at least two possible paths in the mutant). Nevertheless, there is a great disparity amongst models as the non-determinism rate varies from 0% for *Elsa-RRN* to 15.5% for *Claroline*. Higher-order mutation greatly influenced non-determinism rates: the sole order 10 is responsible for 53% of all non-deterministic mutants. In terms of mutation operators, TAD accounts for a large majority of non-deterministic first-order mutants (78%) and AEX for the remaining 22%. At higher orders, these two operators are largely involved. They are absent only in the *Minepump* model where TDE and AMI appear for two non-deterministic mutants.

Algorithm execution: To run the language equivalence algorithms (for WM and SM), we use the HKC library [43], an OCaml implementation of the ALE algorithm [42] compiled using OCamlbuild. This tool handles non-deterministic TSs using different strategies: the automata may be processed either forward of backwards, and the exploration strategy may be breadth-first or depth-first. For each mutant, we execute the HKC library using each of the 4 possible configurations. The input models and their mutants have been transformed from our XML format to the Timbuk input format supported by HKC.

The random and biased simulation algorithms are implemented in Java using multi-threading to parallelize trace selection and execution as described in Algorithm 12 (lines 2, 4, 8, and 10). In our experiments, we set up the algorithm with 4 threads and run 4 instances in parallel on our virtual machine with 16 cores. We run the simulation algorithms with 4 different values of δ and ϵ determining the number of traces selected and executed (N in Algorithm 12):

- RS1/BS1: ($\delta = 1e-10$, $\epsilon = 0.01$, $N = 1,897,519$);
- RS2/BS2: ($\delta = 1e-10$, $\epsilon = 0.1$, $N = 18,975$);
- RS3/BS3: ($\delta = 1e-5$, $\epsilon = 0.1$, $N = 9,764$);
- RS4/BS4: ($\delta = 1e-1$, $\epsilon = 0.1$, $N = 2,396$).

For all the simulation configurations and all models, we fixed the trace length k to 3,000, which was our compromise between performance and non-equivalence detection: setting k to BFS height led to crashes in some cases. In order to answer RQ3, we also run each algorithm (RS1/BS1 to RS4/BS4, plus the 4 possible ALE configurations) with the model itself as the mutant. Those (unrealistic) equivalent detection runs between the model and itself are only used to approximate the worst computation time of the different algorithms.

7.6.2 Results and discussion

Random/biased simulations and ALE: Figure 7.5 presents the execution time per mutant of the studied algorithms, which is detailed in the Appendix. Regarding weak mutation scenarios, the ALE approach is the fastest in all cases in eleven of our models. On the *AGE-RN* model, biased simulations are faster for the largest numbers of runs. However, the results are at the limit of non-significance (see Table 7.10), so that the only clearly significant result is for **BS1** on this model. For *AGE-RNN*,

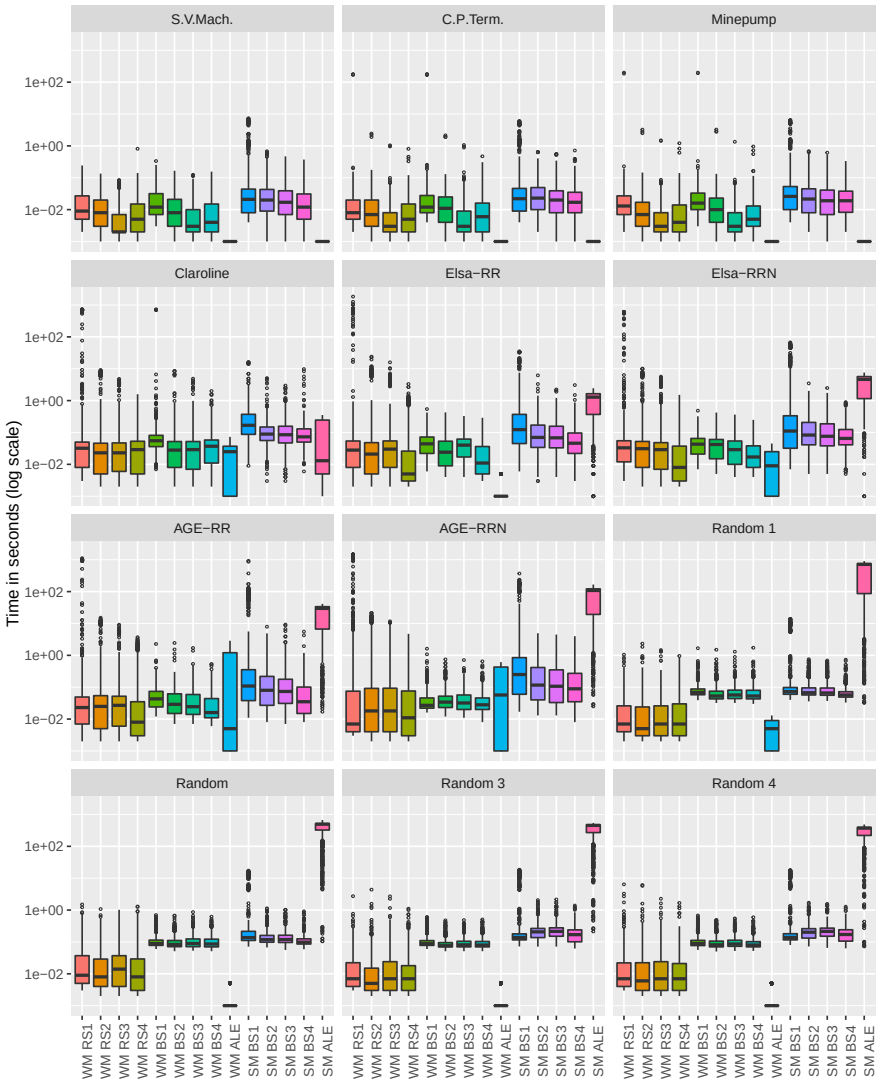


Figure 7.5: Execution time of the equivalent mutant detection approaches

execution times for biased simulations are non-significant. Random simulations are also faster than ALE on *AGE-RRN* but only certain settings are significant. We thus conclude that the ALE approach is more interesting in terms of execution time. When we compare the two forms of simulations, for the smallest models, biased simulations are either on par for the smallest models or slightly better. Additional computations such as the breath-first search used for biased simulation do not cause significant overhead. For the largest random models, random simulations are faster. In these cases, the overhead of computing infected states and paths that cover

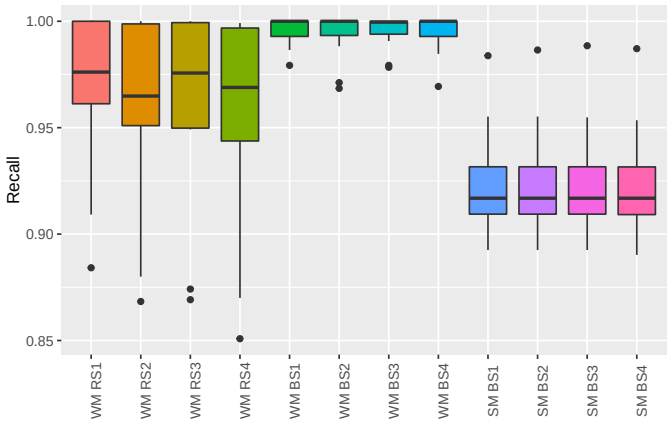


Figure 7.6: Non-equivalent mutant classification recall

these states is greater and random simulation is faster. However, lower standard deviations for biased simulation execution times over random ones make the BS approach easier to use.

Regarding strong mutation, several observations can be made. First, random simulations provide very high execution times compared to biased simulations or the ALE algorithm (the analysis of one model is stopped after one hour). This may be due to the difficulty to reach the initial state again when performing random walks in the TSs. Second, biased simulations are faster than ALE executions for models larger than 300 states. On the largest models, biased simulations can be up to 1,000 times faster. We thus conclude that these are the most interesting situations in which to use BS, for mutation analysis. On smaller models, the ALE algorithm's performance is quite impressive and therefore should be privileged.

Non-equivalent mutant detection: To answer RQ2, we compute the non-equivalent mutant classification recall of the BS/RS algorithms (in Figure 7.6), *i.e.,* the percentage of non-equivalent mutants detected by the BS/RS amongst the selected mutants. By construction, the ALE algorithm has a recall of 100%, it is therefore not shown here. It is also noted that the precision is 100% since all the non-equivalent mutants detected are indeed killable, by construction of our mutant set.

All our simulations obtain a recall higher than 85%, with a clear advantage for biased simulations which never achieve worse than 95% for the weak mutation scenario. As for time, deviation in the recall is smaller for biased simulations thus making the approach more predictable in addition of being more reliable. We also observe that the random simulations are more sensitive to the number of runs: we need more of them to discover discrepancies by luck. This effect cannot be observed for biased simulations. A possible explanation is that the number of runs required to cover infected states with traces is lower than the number we provided.

For strong mutation, the BS approach's recall decreases to around 92% ($\overline{recall}$ = 92%, σ = 3%): amongst the 5113 non-equivalent mutant non-detections (over a total of 64529 non-equivalent mutant evaluations), 1905 (37%) were TAD mutants, 1755 (34%) were WIS mutants, 545 (11%) were TDE mutants, and 459 (9%) were 2nd-order TAD mutants (*i.e.*, TAD-TAD mutants); the rest of non-equivalent mutants not detected is distributed amongst different operators with less than 2% for each. This decrease may be due to the difficulty to find a path to the initial state: for strong mutation, the BS trace selection algorithm will consider traces starting from, and ending in, the initial state. This means that mutations creating (TAD) or modifying (TDE) a back-level transition will not be detected using SM BS. Concerning WIS mutants, we believe that, as the WIS operator only changes the initial state of the TS, the set of infected states (S_{infect}) is empty, which is equivalent in our implementation of SM BS to considering all the states infected.

Worst case scenario: Figure 7.7 presents a compact view of the worst execution time of the different algorithms (RQ3). We grouped the different results by the kind of model: embedded system, web-application, or randomly generated model. As expected, the RS/BS execution time is directly correlated to the δ and ϵ values: a lower number of traces selected and executed (N) takes less time. Overall, the time of the ALE executions grows with the size of the model, reaching 5660 seconds (more than one and a half hour) for the worst WM ALE execution time on the Random 2 model.

7.6.3 Threats to validity

Construct validity: The RS/BS δ and ϵ values have been arbitrarily chosen. The first values (RS1/BS1: $\delta = 1e-10$, $\epsilon = 0.01$) are the same as in Hérault *et al.* [139]. As the number of traces selected and executed N equals to $\frac{8\log(2/\delta)}{\epsilon^2}$, we chose to run the algorithm with 3 higher parameters values in order to reduce N. We cannot guarantee that our parameter values are relevant for any model. They will rather depend on the model size, the desired approximation (ϵ) and confidence (δ), and the time budget allowed for the equivalence analysis.

To the best of our knowledge, the HKC library [43] was the only publicly available tool able to perform ALE checking on non-deterministic TSs. We cannot guarantee that there are no other other tools providing the same features with lower execution time. To avoid bias in the random selections in the RS/BS algorithms, we execute each configuration of the different algorithms 3 times.

Conclusion validity: To confirm our observations on the recall of the RS/BS algorithms, we test the null hypothesis between the outputs of our algorithm (the mutant is equivalent/non-equivalent) and a random equivalent/non-equivalent assignment using a Wilcoxon rank sum test. The p-value lower than 2.2e-16[2] discredits

[2] Due to floating point precision, value $2.2e-16$ corresponds to the smallest possible p-value computable with R.

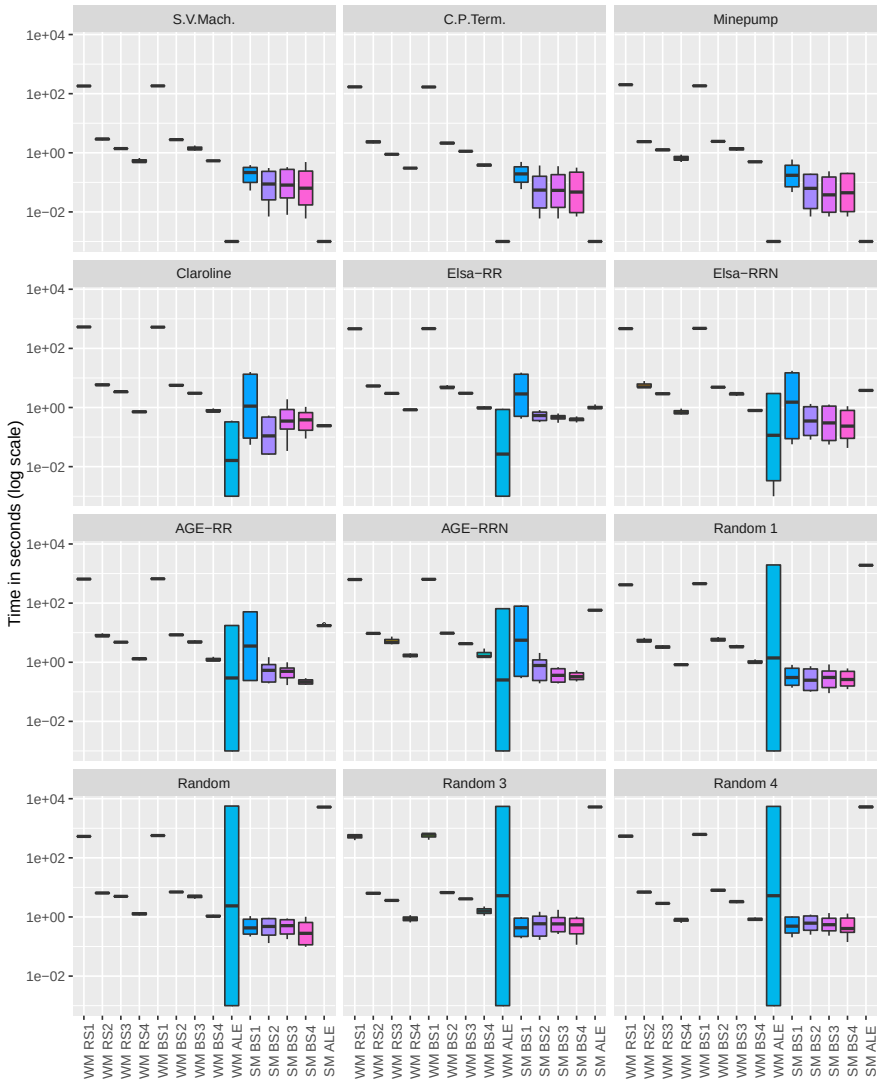


Figure 7.7: Worst execution time of the equivalent mutant detection using the model itself as mutant

the null hypothesis showing that the equivalent/non-equivalent detection recall is significant.

To confirm the statistical difference between the execution times of the RS/BS and ALE algorithms, we test the null hypothesis between RS/BS execution time and ALE execution time for weak and strong mutation for each of our input models using a Wilcoxon rank sum test. For weak mutation, the results of this statistical test are shown in Table 7.10: for every model except *AGE-RR*/*AGE-RRN* models, the p-value

Table 7.10: P-values of the Wilcoxon rank sum test between the WM RS/BS execution times and the WM ALE execution times.

Model	WM RS1	WM RS2	WM RS3	WM RS4
S.V.Mach.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
C.P.Term.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Minepump	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Claroline	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RR	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RRN	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
AGE-RR	$2.866e-03$	$9.676e-03$	$2.021e-02$	$\mathbf{3.249e-01}$
AGE-RRN	$\mathbf{8.143e-02}$	$8.379e-04$	$6.981e-04$	$2.162e-02$
Random 1	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 2	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 3	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 4	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$

Model	WM BS1	WM BS2	WM BS3	WM BS4
S.V.Mach.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
C.P.Term.	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Minepump	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Claroline	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RR	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Elsa-RRN	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
AGE-RR	$9.107e-03$	$4.744e-02$	$\mathbf{6.405e-02}$	$\mathbf{1.382e-01}$
AGE-RRN	$\mathbf{5.991e-01}$	$\mathbf{7.076e-01}$	$\mathbf{5.674e-01}$	$\mathbf{5.168e-01}$
Random 1	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 2	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 3	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$
Random 4	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$	$\leq 2.2e-16$

is lower than 2.2e-16, discrediting the null hypothesis and showing a significant difference in the execution times. The execution times of *AGE-RR*/*AGE-RRN* model are only significant for RS1 to RS3, BS1, and BS3 (for *AGE-RR*); and RS2 to RS4 (for *AGE-RRN*). For strong mutation, all the p-values were lower than 2.2e-16, showing a significant difference in execution time between the BS algorithm and the ALE algorithm in a strong mutation scenario.

7.6.4 Lessons learned

From our experiment we draw the following lessons: (i) regarding weak mutation and independently of the size or nature of the models, the ALE approach provides faster and exact answers. This indicates that state-of-the-art language equivalence algorithms can be used successfully for such a task. (ii) Regarding strong mutation, biased random simulations are of interest for the web and the random models, and gains increase with the size (from one to three orders of magnitude). Recalls of 90% and above allow to use such simulations as reasonably reliable fast filters to discard non-equivalent mutants, leaving to ALE algorithms “difficult" cases so as to accelerate the analysis of large mutants bases. (iii) Biased simulations are

more predictable in terms of execution time and recall. Additionally, drastically increasing the number of runs does not affect their performance as opposed to random simulations. (iv) The configuration of the ALE algorithm (forward/backward processing, or breadth-first or depth-first exploration) has very little influence on the total execution time (regarding equivalent mutant detection). This may be explained by the fact that mutations occur randomly and therefore do not privilege any graph traversal strategy.

7.7 Wrap up

This chapter presents the empirical assessment and their results validating the the selection criteria and mutation analysis described in Chapters 5 and 6. Future work include generalisation of the results by using all the case studies from Chapter 4 for each assessment and comparison of the different test suite selection criteria using mutation analysis.

Part III

Implementation

CHAPTER 8

VARIABILITY INTENSIVE BEHAVIOURAL TESTING FRAMEWORK

Variability Intensive Behavioural teSting (VIBeS) is the framework we developed to support the testing activities described in this thesis. It is designed as a Maven project, decomposed in several Maven modules, to allow flexibility and rapid prototyping. In total, VIBeS has around 16,000 lines of code distributed amongst 307 Java classes. It is released (since its inception) under the MIT license and publicly available on GitHub (https://github.com/xdevroey/vibes). Each assessment from Chapter 7 has been performed using one particular version of VIBeS and each one of those versions is available in the Maven Central Repository[1]. This allows one to reproduce the assessments using the same version of the tool enforcing reproducibility of our results.

This chapter presents the architecture (in Section 8.1) and the usages (in Section 8.2) of the last version (v.1.1.6) of VIBeS. Section 8.3 concludes this chapter and presents future developments.

8.1 Architecture

VIBeS is built as a set of **Maven modules**. Maven is a industrial build management tool build upon the *convention over configuration* philosophy [289]:

> "Convention over configuration is a simple concept: systems, libraries, and frameworks should assume reasonable defaults. Without requiring unnecessary configuration, systems should "just work"."

[1] See https://search.maven.org.

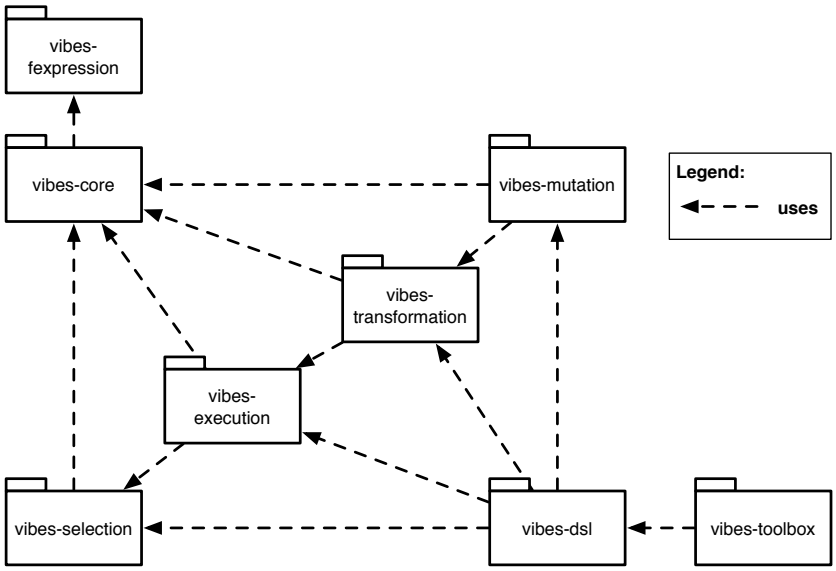


Figure 8.1: VIBeS modules dependency graph

This philosophy allows to have a flexible and extensible (using Maven plugins) build process, while ensuring a minimal configuration effort for the developer. Maven is able to build Java application and libraries (called **artefacts** in the Maven world), including compilation, JUnit test execution, and Jar packaging. Each artefact is described by a **Project Object Model (POM)**, containing a unique identifier and other information about the artefact, dependencies to other artefacts, *etc.* The artefact unique identifier is a triplet: a **group** identifier; an **artefact** identifier; and a **version** number. For instance, the version of VIBeS described is this chapter is the artefact `be.unamur.info:vibes:1.1.6`, where the group identifier, artefact identifier, and version number are separated by ':'.

Maven allows to organise a project hierarchically. The root project is a **Maven project**, while the sub-projects are **Maven modules**. VIBeS's Maven project produces the artefact `be.unamur.info:vibes:1.1.6`, which is only a POM without any associated Jar. VIBeS's project has several modules (with the same group identifier and version number as the parent project) regrouping different aspects of the framework. For instance, the modules `vibes-core` contains the core classes used to model FTSs, module `vibes-selection` contains Java classes to perform test case selection from a FTS model, *etc.* Figure 8.1 presents the different modules from VIBeS and the dependencies between them: module `vibes-core` uses the `vibes-fexpression` modules that allows to represent and manipulate feature expressions; module `vibes-selection` presents an API to select test cases from a transition system (FTS, LTS, or usage model); module `vibes-execution` contains the API to execute abstract test cases on the transition systems; `vibes-transformation` allows to transform, serialize, or deserialize the transition systems using different formalisms

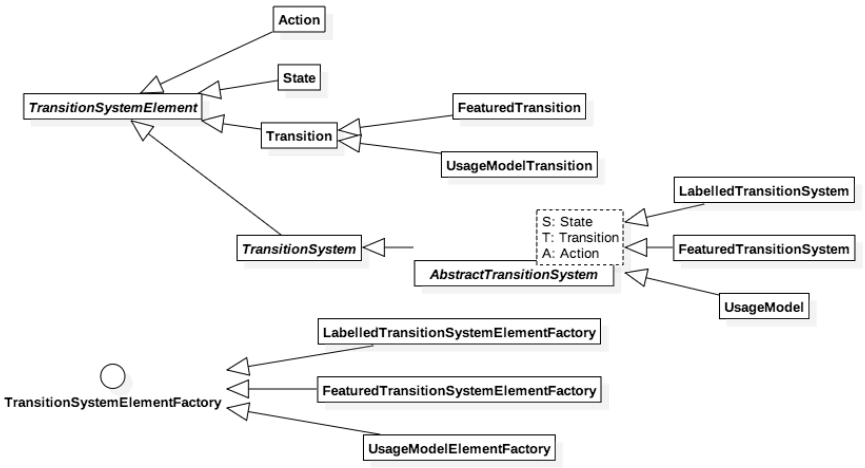


Figure 8.2: VIBeS type hierarchy class diagram

(*e.g.*, XML, DOT file, timbuk automata, *etc.*); module `vibes-mutation` contains the API to perform mutation analysis; `vibes-dsl` encapsulates the different API to present a unified Java DSL to perform the different testing activities using VIBeS; and `vibes-toolbox` contains all the sub-modules that uses the Java DSL to implement toolboxes to perform particular tasks (*e.g.*, generate mutants).

Classes from the `vibes-core` modules represent the transition systems used by VIBeS. Figure 8.2 presents the class hierarchy of the different elements. Each element of a transitions system (actions, states, transitions, and the transition system itself) extends the `TransitionSystemElement` class. Transitions are either simple transitions (`Transition` class) or transitions labelled by feature expression (`FeaturedTransition` class) or a probability (`UsageModelTransition` class). Transition systems (`LabelledTransitionSystem`, `FeaturedTransitionSystem`, and `UsageModel`) extends the `AbstractTransitionSystem` class, which is parametrized with the type of states, transitions, and actions used for this transition system (featured transitions for the FTSs for instance). To manage the creation of the different elements of a transition system, a dedicated factory, extending `TransitionSystemElementFactory`, is used.

Figure 8.3 presents how transition systems are represented. A transition system (class `TransitionSystem`) is a collection of actions and states, with an initial state. Each state has incoming and outgoing transitions, and each transition is labelled with an action belonging to the same transition system. The default transition system implementation (class `AbstractTransitionSystem`) uses a factory to build the different states, actions, and transitions. This insures to respect the representation invariant of the class.

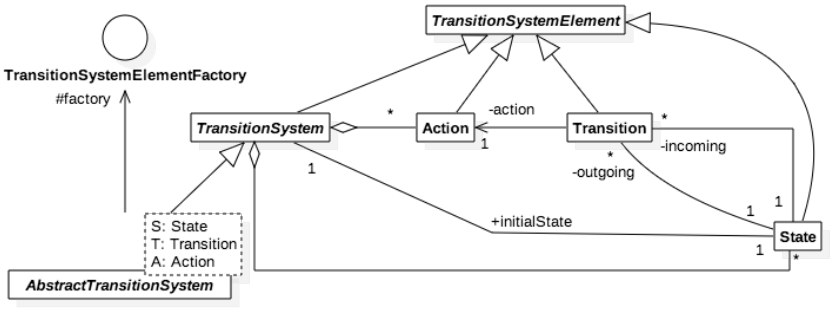


Figure 8.3: VIBeS transition systems class diagram

8.2 API usage

The simplest way to use VIBeS is to create a new Maven project and add a dependency to `be.unamur.info:vibes-dsl:1.1.6`:

```
<dependency>
    <groupId>be.unamur.info</groupId>
    <artifactId>vibes-dsl</artifactId>
    <version>1.1.6</version>
</dependency>
```

This artefact contains an API that encapsulates most of VIBeS usages. The `vibes-dsl` module has been built following the same philosophy as the Apache Camel Java DSL[2] to chain method calls in order to facilitate the model definition, test case selection, mutation analysis, *etc.*

8.2.1 Model definition

Definition of a new FTS is done by extending the `FeaturedTransitionSystemDefinition` class and implementing the abstract `define` method of that class. This method calls other inherited methods to define the initial state, states, actions, and transitions. For instance, the soda vending machine FTS from Section 4.1 is defined as:

```
public class SodaVendingMachineModel extends
     FeaturedTransitionSystemDefinition{

    private static final String[] S = new String[]{"s1", "s2", "s3", "s4",
         "s5", "s6", "s7", "s8", "s9"};

    @Override
    protected void define() {
        initial(S[0]); // Define the initial state
```

[2] See http://camel.apache.org.

```
            from(S[0]).action("pay").fexpr("!f").to(S[1]);
            from(S[0]).action("free").fexpr("f").to(S[2]);

            from(S[1]).action("change").fexpr("!f").to(S[2]);
            from(S[2]).action("cancel").fexpr("c").to(S[3]);

            from(S[3]).action("return").fexpr("c").to(S[0]);
            from(S[3]).action("soda").fexpr("s").to(S[4]);
            from(S[3]).action("tea").fexpr("t").to(S[5]);

            from(S[4]).action("serveSoda").fexpr("s").to(S[6]);
            from(S[5]).action("serveTea").fexpr("t").to(S[6]);

            from(S[6]).action("take").fexpr("f").to(S[0]);
            from(S[6]).action("open").fexpr("!f").to(S[7]);

            from(S[7]).action("take").fexpr("!f").to(S[8]);
            from(S[8]).action("close").fexpr("!f").to(S[0]);
        }
    }
```

The model can be instantiated by calling the `getTransitionSystem` method:

```
FeaturedTransitionSystem svm = new SodaVendingMachineModel().
    getTransitionSystem();
```

Transitions may be tagged with an action or a feature expression. Feature expressions must respect the following grammar:

```
fexpression: 'true'
      | 'false'
      | featureName
      | '(' fexpression ')'
      | '!' fexpression
      | fexpression ('&&' | '||') fexpression ;

featureName: LETTER (LETTER | DIGIT)* ;

LETTER: 'A'..'Z' | 'a'..'z' | '_';
DIGIT: '0'..'9';
```

Feature names must begin by a letter, and may contain letters or digits. Operators have the following priority rules: negation (’!’) takes precedence over conjunction (’&&’) and disjunction (’||’).

8.2.2 Test suite selection

Test suite selection may be done manually, or using one of the selection criterion defined in Chapter 5.

Manual test suite selection: Manual selection is done by extending the `TestSetDefinition` class. As for the model definition, the `define` method has to be implemented and call inherited methods to define the test cases:

```
public class ManualTestSuite extends TestSuiteDefinition{

    @Override
    protected void define() {
        id("testRefund").action("pay").action("change").action("cancel").
            action("return").end();
        id("testFreeTea").action("free", "tea", "serveTea", "take");
        id("testNoFreeSoda").action("pay", "change").action("soda", "
            serveSoda").action("open", "take", "close");
    }
}
```

Actions may be added one by one or by groups in the same `action` method call. The test suite is instantiated using the `getTestSet` method:

```
TestSet testSuite = new ManualTestSuite().getTestSet();
```

Random selection: Random test suite selection selects a defined number of test cases by random walks in FTS. This requires the feature model of the FTS from which test cases are selected to ensure that selected test cases are positive abstract test cases. This feature model is encapsulated in a constraint solver (Sat4j[3] for instance), accessed trough a facade that implements the `SolverFacade` interface. To load the feature model, one may use a feature expression (a CNF boolean expression representing the feature model) or load it from a DIMACS CNF file. In the following example, we load the feature model from a DIMACS CNF file:

```
// Feature model loading
DimacsModel featureModel = DimacsModel.createFromDimacsFile("svm.dimacs");
SolverFacade solver = new Sat4JSolverFacade(featureModel);

// Random test suite selection
TestSet randomSuite = randomSelection(svm, solver);
```

All-states selection: Selection of a test suite satisfying the all-state criterion is done by importing the `allStatesSelection` static method from the `AllStates` class:

```
// Feature model loading
DimacsModel featureModel = DimacsModel.createFromDimacsFile("svm.splot.
    dimacs");
SolverFacade solver = new Sat4JSolverFacade(featureModel);

// All-states test suite selection
TestSet allStatesSuite = allStatesSelection(svm, solver);
```

[3] See http://www.sat4j.org.

Dissimilarity-driven selection: Test suite selection using a dissimilarity heuristic is done using the `Dissimilar` class. The heuristic may be configured using the following methods:

```
// Feature model loading
DimacsModel featureModel = DimacsModel.createFromDimacsFile("svm.splot.
    dimacs");
SolverFacade solver = new Sat4JSolverFacade(featureModel);

// Dissimilar selection
from(svm, solver)
  .during(30000) // specify duration
  // specify local or global distance and how to compute dissimilarity
  .withLocalMaxDistance(ftsDissimilarity(solver, levenshtein(), avg()))
  // Specify the number of test cases
  .generate(5);
```

Duration specifies how long the algorithm will run, using a local (`withLocalMaxDistance`) or global (`withGlobalMaxDistance`) distance computation. The distance itself is defined using one of the static methods that works on the actions alone (`hamming`, `jaccard`, `dice`, `antidice`, or `levenshtein`), or in combination with product distance (`ftsDissimilarity`), using a binary operator (`avg`, `mul`, or any other `BinaryOperator` object).

Usage-based selection: Usage based selection is more complex. First, it requires to select a set of test cases in the usage model using `BoundedProbabilityGenerator` class. Those test cases are then executed on the FTS that is pruned to keep only transitions activated by at least one test case. This is done using the `Prunning.prune` method. Complete usage-based test suite selection is not yet encapsulated in `vibes-dsl`. This is part of our future work.

8.2.3 Saving and loading models

Models may be saved in XML format using the `TransitionSystemXmlPrinter` class. To load a model from an XML file, one may use the `TransitionSystemXmlLoaders` class:

```
// Load model
FeaturedTransitionSystem svm = loadFeaturedTransitionSystem("svm.xml");

// Save model
print(svm, "svm.xml");
```

The XML model of the soda vending machine is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<fts xmlns="http://www.unamur.be/xml/fts/" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance">
  <start>state1</start>
  <states>
    <state id="state1">
      <transition action="pay" fexpression="!FreeDrinks" target="state2"/>
```

```
      <transition action="free" fexpression="FreeDrinks" target="state3"/>
    </state>
    <state id="state2">
      <transition action="change" fexpression="!FreeDrinks" target="state3"
          />
    </state>
    ...
  </states>
</fts>
```

8.2.4 Performing mutation analysis

Mutation analysis consist of mutants generation (in an enumerative or FMM approach) and mutants execution. Those analysis are done at the product level (on LTSs), mutation analysis for FTSs is part of our future works.

Mutation operators configuration: Operators may be configured using the `Mutagen` class. This is useful to define the selection strategy of the elements to mutate. For instance, one can create a state missing (SMI) operator that will only remove particular states (chosen randomly):

```
MutationOperator op = Mutagen.stateMissing(svm)
  .stateSelectionStrategy(svm.getState("s4"), svm.getState("s8"), svm.
      getState("s9"))
  .done();
```

One may also configure a set of operators using an XML configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <!-- Default mutant size (may be redefined) -->
  <mutantsSize>200</mutantsSize>
  <!-- Default selection strategies (may be redefined) -->
  <actionSelection>
    be.unamur.transitionsystem.test.mutation.RandomSelectionStrategy
  </actionSelection>
  <stateSelection>
    be.unamur.transitionsystem.test.mutation.RandomSelectionStrategy
  </stateSelection>
  <transitionSelection>
    be.unamur.transitionsystem.test.mutation.RandomSelectionStrategy
  </transitionSelection>
  <!-- Default uniqueness of each mutant (may be redefined) -->
  <unique>true</unique>
  <!-- Operators -->
  <operators>
    <operator>
      <class>be.unamur.transitionsystem.test.mutation.ActionExchange</class
          >
    </operator>
    ...
    <operator>
      <class>be.unamur.transitionsystem.test.mutation.WrongInitialState</
          class>
```

```
        <mutantsSize>100</mutantsSize>
        <actionSelection>be.dummy.MyActStrategy</actionSelection>
        <stateSelection>be.dummy.MyStStrategy</stateSelection>
        <transitionSelection>be.dummy.MyTrStrategy</transitionSelection>
        <unique>false</unique>
      </operator>
    </operators>
  </config>
```

Default configuration (selection strategy, number of mutants to generate, uniqueness on mutant based on the selected operands of the operator) may be redefined for each mutation operator.

Mutants generation (enumerative approach): Mutants may be generated enumeratively: each mutant is generated in a new XML file. For instance:

```
LabelledTransitionSystem lts = loadLabelledTransitionSystem("product.xml");
configure("operatorsConfig.xml")
  .outputDir("mutants/") // Generates mutants in the given folder
  .mutate(lts);
```

Mutants generation (FMM approach): One can also generate a FMM and save it in an XML and a TVL files :

```
LabelledTransitionSystem lts = loadLabelledTransitionSystem("product.xml");
FeaturedMutantsModel fmm = configure("operatorsConfig.xml")
  .ftsMutant("fmm.fts") // Generates FMM's FTS with given name
  .tvlMutant("fmm.tvl") // Generates FMM's FD in TVL format
  .mutate(lts);
```

Mutants execution: Finally, execution of a test suite on the FMM is done using the `getAliveMutants` static method of the `FeaturedMutantsModels` class:

```
FExpression alive = getAliveMutants(testSuite.get(0), fmm);
solver.addConstraint(alive);
Iterator<Configuration> solutions = solver.getSolutions();
while(solutions.hasNext()) {
  System.out.println(solutions.next());
}
```

8.2.5 VIBeS toolboxes

VIBeS architecture in Maven modules allows to do rapid prototyping. Module `vibes-dsl` encapsulates the common operations performed during the testing activities to reduce as much as possible custom developments. Beside `vibes-dsl` module, VIBeS also comes with a set of toolboxes. Each of those toolbox is a executable Jar file with a command line interface that may be used to perform standard tasks. The existing toolboxes are:

- `toolbox-model-statistics`: to print statistics about a transition system;

- `toolbox-testcase-generation`: to select test suites using various criteria;
- `toolbox-transformation`: to transform an XML transition system to other formats (*e.g.*, Graphviz DOT);
- `toolbox-products-analyze`: to print the number of products for a given feature model and a given feature expression;
- `toolbox-mutant-generation`: to generate mutants (enumeratively or using a FMM);
- `toolbox-fmm-execution`: to execute test cases on a FMM;
- `toolbox-mutation-equivalence`: to detect equivalent mutants using simulation or automata language equivalence;
- `toolbox-mutant-sampling`: to sample a set of mutants (from different orders) from a given FMM.

8.3 Wrap up

In this chapter, we present VIBeS, the implementation of our testing framework. We use VIBeS to perform various testing activities, including model definition, test suite selection, and mutation analysis.

VIBeS is built as a Maven project with several modules. Each module is dedicated to one particular aspect of the testing activities. Test engineers may use VIBeS by accessing the API of the different modules, or by using the Java DSL that encapsulates API calls and simplifies usages.

This modular architecture allows to perform rapid prototyping while allowing adhoc developments for specific needs. We choose to use Java as the interface language for the test engineer rather than a dedicated DSL or modelling language. This avoids to switch between syntaxes and, we assume, lowers the learning curve for new users. This idea stems from industrial practices (like Apache Camel[4]) and seems to be confirmed by other trending behavioural testing tools (like Cucumber [201] where, except for the behavioural description of the system, all elements are defined in Java).

[4] See http://camel.apache.org.

CHAPTER 9

TEST CASE CONCRETIZATION USING ABSCON

Test definition and execution is an essential but time-consuming task during system development. To speed up the process, model-based testing and other related approaches propose to select abstract test cases and to automatically concretize them, based on mapping information provided by the test engineer. This mapping may take one of the following forms [307]: (i) an adapter which interprets the actions and assertions of the abstract test case and execute them on the SUT; (ii) a transformation from the abstract test cases to code executable directly on the SUT; or (iii) a mixture of the above two. In this last case, an abstract test case is transformed into executable code which uses an intermediate adapter (like an intermediate library for instance) to bridge the gap between the test case and the SUT.

In this chapter, we describe the Abstract test case Concretizer (AbsCon) developed by Jeremy Vanhecke [311] during his master thesis, we co-supervised with Dr. Gilles Perrouin and Prof. Patrick Heymans. AbsCon is defined as a QTaste [257] plugin, an open-source industrial data-driven test case definition and execution environment, used to perform black-box testing on various kinds of systems. QTaste abstracts the SUT's interface by using an adapter called **test API**, test cases are written in Python where the operations on and the readings from the SUT's interface are encapsulated into calls to the test API dedicated to the kind of the SUT.

For instance, to test a Web-application, QTaste encapsulates the access to the elements of the Web page in a Web test API which is responsible to perform the effective Selenium (a popular Web browser automation tool [279]) calls. After considering different options, we chose to define AbsCon as a QTaste plugin for the following reasons: (i) QTaste is an open source industrial tool, used to test various kinds of systems, from Web-applications to mobile applications and even cyber-physical systems [93], thanks to its test API adaptation mechanism; (ii) plugin development is already included in QTaste and this architecture was suggested by a QTaste de-

veloper; (iii) the inital goal of AbsCon was to concretize abstract test cases selected by VIBeS using additional mapping information. To this end, abstract test cases are defined in AbsCon using an XML file, where each test case is a sequence of actions and assertions on the SUT. But this definition is not specific to VIBeS, it also allows QTaste test engineers to define test cases in a more abstract and systematic fashion (rather than directly Python scripts), as long as they follow the same pattern (*i.e.,* sequences of assertions and actions).

The remainder of this chapter is as follows: Section 9.1 gives a general description of the QTaste environment, Section 9.2 describes AbsCon's concretization process as well as the required mapping information, Section 9.3 presents AbsCon's implementation, advantages and limitations are discussed in Section 9.4, Section 9.5 discusses related work. Finally, Section 9.6 wraps up the chapter and presents some perspectives.

9.1 Test automation using QTaste

The QSpin Tailored Automated System Test Environment (QTaste) [257] is an open source functional and non-functional black-box test environment developed in Java and Python. It has been originally developed by Qspin Experts[1] in order to automate testing process of medical cyber-physical systems developed by IBA[2] and used for proton therapy. Since its inception, QTaste has been extended to support different kinds of SUTs, like Web-applications, mobile applications, or more classical desktop applications [93]. It is released as an open source project on GitHub under GNU GPL 3.0 license [257].

9.1.1 Overview of the QTaste environment

QTaste follows the data-driven testing philosophy [28]: data used by the tests are externalized in order to allow test cases parametrization. Each test case is written in Python and describes a sequence of steps, *i.e.,* **operations** executed by the SUT or **verifications** of the outputs produced by this SUT, using the given data as input. For instance, when testing a form which values are recorded in a database, one test case fills the form with the given data and check that the values are effectively recorded in the database. This test case is repeated with different values (*e.g.,* positive, null, and negative values for numeric fields) specified in a separate CSV file and automatically executed by QTaste on the SUT.

QTaste adapter mechanism: QTaste provides *test APIs* which communicate with the SUT and manage the operations executions and/or SUT's outputs reading. Each test API consists in a Java interface, defining the operations and information accessible by the test cases, and a Java implementation of this interface which manages communication with the SUT. This mechanism allows QTaste to test a large variety

[1] http://www.qspin.be
[2] https://iba-worldwide.com

Listing 9.1: Google search test case

```
1  from qtaste import *
2
3  api = testAPI.getSelenium(INSTANCE_ID='Google')
4
5  def init():
6    api.openBrowser(testData.getValue("BROWSER"))
7    api.windowMaximize()
8    api.open("https://www.google.be/")
9    api.waitForPageToLoad("15000")
10   if api.getTitle() != "Google":
11     testAPI.stopTest(Status.FAIL)
12
13 def searchAndClick():
14   api.type("id=lst-ib", testData.getValue("SEARCHVALUE"))
15   api.clickAt("name=btnK", "0.0" )
16   api.waitForPageToLoad("15000")
17   api.click("link=" + testData.getValue("LINKTOCLICK"))
18   if api.getTitle() != testData.getValue("LINKTITLE"):
19     testAPI.stopTest(Status.FAIL)
20
21 def exit():
22   api.stop()
23
24 doStep(init)
25 doStep(searchAndClick)
26 doStep(exit)
```

of systems: Web-applications using a Selenium-based test API, hardware components with dedicated API, or any other kind of system for which a test API may be developed. The test API, together with the configuration of the SUT instance is called a **Testbed**: this mechanism allows to write test cases independently from the execution environment, using only test (and standard Python) API(s). Once all the test cases have been executed, QTaste generates a summary report, with the number of success and fails, the Testbed used, for each test case, the CSV lines used, *etc.*

Example: Listing 9.1 presents a (simplified) test case for the Google search engine that is executed for each line of the external CSV file. It launches a Web browser and connects to the Google search website, fills the search field with a string, and click on a specified link. Line 3 creates a Selenium instance test API, which manages the connection to the browser; lines 5, 13 and 21 declare the steps of this test case, called at lines 24, 25, and 26; explanations about each step is given as a comment in Python format (not shown here) and is used during the generation of the test reports. At each step, the test API instance is used to manipulate the browser user interface (lines 6 to 10, 14 to 18, and 22) according to the data provided in the external CSV file (identified by column names at lines 6, 14, 17, and 18). Finally, each step may check assertions on the outputs of the browser to validate the execution (lines 10 and 18).

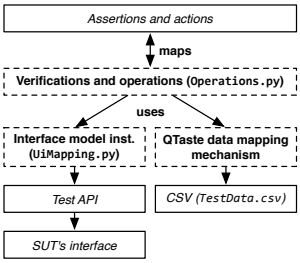


Figure 9.1: Mappings in AbsCon

9.1.2 Advantages and limitations

The main advantage of QTaste is the test API mechanism, allowing test cases to manipulate a large variety of SUTs using a general purpose programming language: Python. Expressing test cases using a general purpose and popular programming language like Python benefits from the large number of available Python libraries. This can be very handful when writing test cases in order to perform more complex operations or access external resources. The environment provides extensibility mechanisms to the test engineers in order to write dedicated adapters between QTaste and SUTs, and describes the usage of those adapters in a test API. Coupled to the externalisation of data and SUT's configuration, it improves test cases **reusability** and **automation** of the test process [307].

As it works as a black-box test environment, QTaste access the SUT through its interface, manipulated by the test cases trough the test API. This means that whenever the interface and/or the test API evolve, all the test cases using this interface and/or modified test API are impacted, increasing **maintenance cost** [307]. AbsCon provides an additional abstraction layer separating the different concerns thus reducing maintenance costs when combined with abstract test cases as presented in the next section.

9.2 Test cases concretization

AbsCon was originally developed to support abstract test case concretization [311]. In AbsCon, an abstract test case is a sequence of abstract **assertions** and **actions**, usually automatically derived by a model-based testing tool [307]: VIBeS in this case [82]. To bridge the gap between VIBeS and AbsCon, abstract test cases produced by VIBeS are enriched with the intermediate states (representing assertions on the system's state) visited when executing the abstract test case. This modification is coherent with our assumption that the behavioural model used to select abstract test cases is deterministic.

The concretization process translates the abstract test case into a (concrete) test case executable by QTaste: (resp.) **assertions** and **actions** are **mapped** to (resp.) **verifications** and **sequences of operations** manipulating the SUT through the test API. The most common way to perform this task is to give, for each assertion and

each action, the corresponding Python code. It allows to improve the reusability and automation, while decreasing the maintenance costs (each assertion or action is defined only once in the mapping).

However, access to the SUT's interface elements remains hardcoded in the different test cases (*e.g.,* lines 10 or 15 in Listing 9.1). This may raise one or more issues:

(i) element of the SUT's interface are accessed using test API methods, requiring to know and provide at each method call the **access method** (*e.g.,* using the element's `id` or `name` or at lines 14 and 15 in Listing 9.1) and the **access value** (*e.g.,* `lst-ib` at line 14 and `btnK` at line 15 in Listing 9.1);

(ii) besides being cumbersome when writing test cases, requiring access method and value in each method calls may also raise problems, as not all elements of test API may be called on all elements of the SUT's interface (*e.g.,* for a Web-application, it is only possible to type text in text fields or in text areas), which will only be checked when running the test case;

(iii) as previously, when an interface or test API element is updated, all the actions and/or assertions using this element are impacted, requiring to update the mapping in different places and thus increasing the maintenance cost (with a more limited magnitude).

To mitigate those issues, we divide the mapping in AbsCon in three elements, as illustrated by the dashed boxes in Figure 9.1: a SUT's interface elements mapping trough a **model instance** of this interface; a **data mapping**; and an **assertions and actions mapping**, giving for each (resp.) assertion/action the (resp.) verifications/operations to perform on the SUT. The verifications and operations on the SUT are defined as Python functions that will use the interface model instance, using the methods of the different elements, and the external data. The external data are recorded in a CSV file and managed using QTaste's dedicated mechanism. The interface model, *i.e.,* a set of Python classes, uses one or more test APIs in order to execute the operations and retrieve information on/from the SUT.

The model of the interface and the assertion/actions mapping is detailed in the following sections. To illustrate those different mappings, we will use Google instant search as SUT and consider the following test cases:

(i) open Google search website, enter a keyword, see that search results are printed, click on a link, and check that the website is loaded;

(ii) open Google search website, enter a keyword, see that search results are printed, deactivate the instant search in the parameters and validate, go back to the main page, and check that search results are not printed when a keyword is entered;

(iii) open Google search website, enter a keyword, see that search results are printed, deactivate the instant search in the parameters and cancel, go back to the main page, and check that search results are printed when a keyword is entered.

Test case (i) checks the common usage of Google instant search, test case (ii) checks that, when the instant search is deactivated in the parameters options, the instant search is not performed, and test case iii checks that when instant search is deacti-

Listing 9.2: Google instant search test cases in AbsCon XML format

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <realisation id="Google testing">
3    <uimodel>web</uimodel>
4    <uimapping>UiMappings.py</uimapping>
5    <operations>Operations.py</operations>
6    <datas>TestData.csv</datas>
7    <tests>
8      <test>
9        <action>start</action>
10       <action>goHomePage</action>
11       <assert>onHomePage</assert>
12       <action>inputSearchString</action>
13       <assert>searchResultsPrinted</assert>
14       <action>clickLink</action>
15       <assert>pageLoaded</assert>
16       <action>exit</action>
17     </test>
18   ...
19   </tests>
20 </realisation>
```

vated but the change is cancelled in the parameters options, instant search is still active. Test cases are defined in XML format as a sequence of assertions and actions: lines 8 to 17 in Listing 9.2 gives test case i definition (other test cases are omitted). Additional information are the SUT's interface model to use (line 3), the path to the Python file defining the instance of this model for the Google instant search interface (line 4), the path to the Python file defining the operations mapping (line 5), and the path to the external CSV data file (line 6).

9.2.1 SUT's interface model

A SUT's interface model describes for a particular family of SUTs the different elements accessible when performing black-box testing. For instance, for Web-applications, the `web` model at line 3 in Listing 9.2 is defined (here using a class diagram notation to ease the reading) in Figure 9.2. It described the different elements we can found on a Web page: each class has to access the QTaste Selenium test API (using inherited attribute `api`) and will extend `WebElement`; `WebBrowser` objects will `start` and `exit` the Web browser specified for the current test case execution using the data from the external CSV file (as on line 6 in Listing 9.1); a `WebPage` is available at a given URL `address`, may be opened (and expected to load before a given `timeout`), closed, and has a title; `WebElements` will appear on this page, each one is accessible using an `accessMethod` (*e.g.,* XPath) and an access value (*e.g.,* an XPath query to this element), may or may not exist on the page, has a value, and may be clicked; the different elements we identified (relevant for the examples of this paper) are `WebButton`, `WebLink`, `WebRadioButton`, `WebText`, `WebPicture`, and `WebEditBox` which may be filled using textual values.

In AbsCon, each interface model is defined in Python. For one particular SUT's

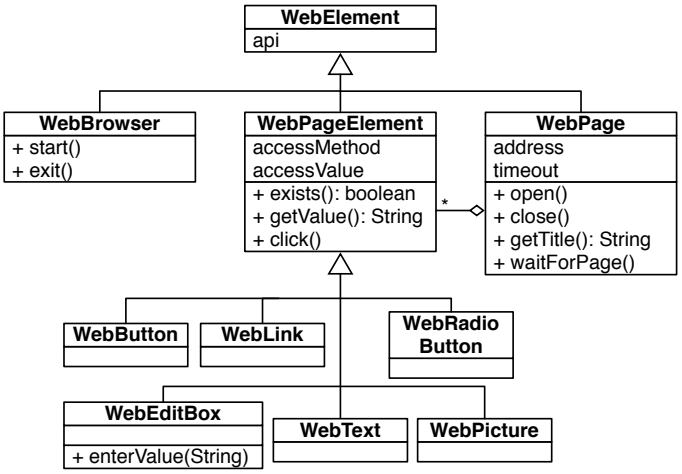


Figure 9.2: Web-applications SUT's interface class diagram (`web`)

Listing 9.3: Google instant search interface model instance (`UiMapping.py`)

```
1  from uimodel_web import *
2
3  #mapping definitions
4  googlePage = WebPage("https://www.google.be/", 5000)
5  searchBar = WebEditBox("id", "lst-ib")
6  searchButton = WebButton("name", "btnK")
7  disableInstSearch = WebRadioButton("xpath", "//div[@id='instant-radio']/div
       [3]/span")
8  enableInstSearch = WebRadioButton("xpath", "//div[@id='instant-radio']/div
       [2]/span")
9  ...
```

interface, this model is instantiated to represent the elements accessible to the test cases. For instance, for Google instant search, the `web` model instance is defined in `UiMapping.py` (Listing 9.3), as specified at line 4 in Listing 9.2. Each object is built using the dedicated constructor, which will require in most cases an access method and an access value: *e.g.,* search bar is accessed using its `id` in the page, which is `lst-ib` (line 5), or using an XPath expression (lines 7 and 8). As for test APIs, interface models may be reused across different SUTs, as long as they share the same kind of interface (Web pages in this case).

9.2.2 Assertions and actions mapping

AbsCon extracts assertions and actions from the abstract test cases (Listing 9.2). Each assertion is mapped to a verification (*i.e.,* a function returning true or false) over the SUT's interface; and each action is mapped to a sequence of operations over elements of the SUT's interface (*i.e.,* again, a function).

Listing 9.4: Verifications and operations mapping (`Operations.py`)

```
1  from qtaste import *
2  from UiMappings import *
3
4  #Actions definition
5  def goHomePage():
6    googlePage.open()
7
8  def inputSearchString():
9    searchBar.enterValue(testData.getValue("SEARCHVALUE"))
10 ...
11 #Asserts definition
12 def searchResultsPrinted():
13   googlePage.waitForPage()
14   if (not(navPicture.exists())):
15     time.sleep(3) # wait for loading and retry
16   return navPicture.exists()
17 ...
```

Verifications as well as operations are defined using the interface model instance defined in `UiMapping.py` (and will manipulate the different elements using the methods defined for those elements) and the QTaste data mapping mechanism in order to retrieve data from the external `TestData.csv` file. The mapping between the assertions and actions from the abstract test case is done by using the same name for the verifications and operations functions.

For instance, Listing 9.4 presents the verifications and operations mapping (`Operations.py`) for the Google instant search test cases from Listing 9.2. Function `inputSearchString` (line 8) corresponds to the action with the same name in the test cases and inputs a search string, coming from the `SEARCHVALUE` column of the external CSV file, in the Google search bar defined in `UiMappings`. Function `searchResultsPrinted` (line 12) corresponds to the assertion with the same name in the test case, and returns true if the navigation picture from the result page is loaded.

9.2.3 Test cases generation and execution

Once the mappings are defined, AbsCon generates concrete (*i.e.*, executable) test cases for QTaste: for each test case, it creates a Python script which imports the mappings and executes a sequence of `doStep` and `doAssert` calls using the verifications and operations functions. Those Python files, with the `TestData.csv` file, are used as input for QTaste to execute the test cases on the SUT and generate a summary test report.

Listing 9.5 presents the result of the generation for test case i: each `doStep` call (part of the standard QTaste API) corresponds to one action in the test case and will execute the given function. The `doAssert` function, defined by AbsCon, calls the given function (corresponding to an assertion in the test case) and prints the given error message if the call returns false.

Listing 9.5: Generation result for test case i

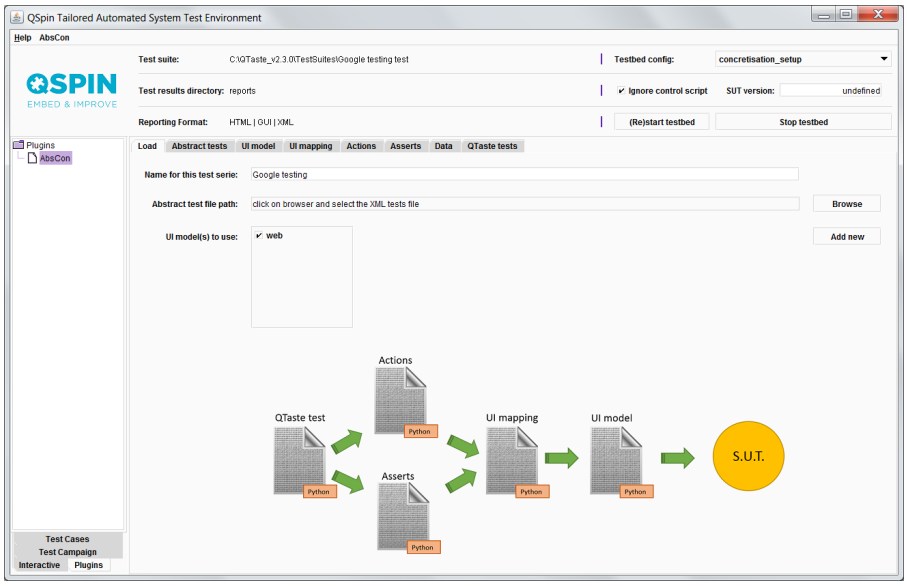
```
1  from qtaste import *
2  from Operations import *
3
4  #Assert
5  def doAssert(method, message):
6    res = method()
7    if res == 0:
8      raise QTasteTestFailException(message)
9
10 #Steps
11 doStep(start)
12 doStep(goHomePage)
13 doAssert(onHomePage, "assertion onHomePage has failed")
14 doStep(inputSearchString)
15 doAssert(searchResultsPrinted, "assertion searchResultsPrinted has failed")
16 ...
```

9.3 Implementation

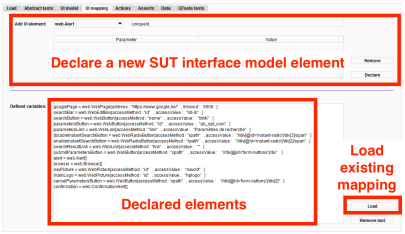
QTaste's plugin development functionality has been developed for specific requests made to QSpin. To the best of our knowledge, there is no plugin developer documentation available, but thanks to a QSpin developer guidance in the GitHub repository (where some examples of basic QTaste plugins are available), the development of AbsCon plugin was made possible. Basically, the plugin mechanism allows one to build his own user interface in a specific area inside the QTaste's user interface. Plugins have to be developed in Java (like QTaste) and must use the standard QTaste libraries.

9.3.1 Graphical user interface

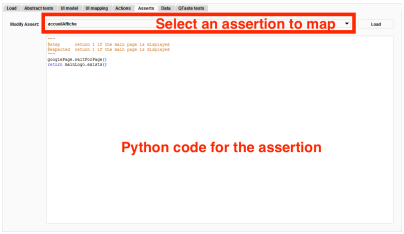
AbsCon provides a graphical user interface integrated into QTaste (as shown in Figure 9.3(a)) with different tabs, one for each mapping phase. When executing the plugin, the first step is to load the abstract test cases from an external XML file, AbsCon extracts the different actions and assertions for which a mapping has to be provided and presents them under the `Abstract tests` tab. The second step is to define or load a SUT's interface model (under tab `UI model`) and to instantiate this model under the `UI mapping` tab presented in Figure 9.3(b): for each element of the interface, one has to instantiate a class of the interface model (selected using a drop-down list) by providing the required parameters for the constructor, and add it to the mapping using the `Declare` button (or load an existing mapping using the `Load` button). Actions and assertions mappings are given using the two next tabs: the user select the action/assertion using a drop-down list and provides the Python code for this action/assertion as shown in Figure 9.3(c) (the assertion mapping tab in this case). In the `Data` tab, the user provides the data in an editable table, and finally generates the QTaste executable test cases in the `QTaste tests` tab.



(a) Main tab



(b) SUT interface mapping tab



(c) Assertions mapping tab

Figure 9.3: AbsCon plugin printscreens

9.3.2 AbsCon plugin architecture

Figure 9.4 gives an overview of the AbsCon packages diagram. The plugin architecture follows a classic model-view-controller pattern, divided in three Java packages: `type` one in charge of the model, `ui` package with all the classes in charge of the view, and `manager` one which contains the controller classes.

The plugin execution is orchestrated by the `AbsCon` class, which extends the `AddOn` QTaste class: it overrides the `loadAddon`, `unloadAddon` and `getConfigurationPane` methods. The two first contain all the operations to perform when QTaste loads or unload the plugin and its functionalities. The third one is the method that gives to QTaste environment the user interface of the AddOn (*i.e.*, it returns a `JPanel` that contains the plugin graphical user interface).

Model classes (package `type`) implement the different concepts presented in section 9.2: abstract test cases (`AbsractTest`) are represented as a sequence of

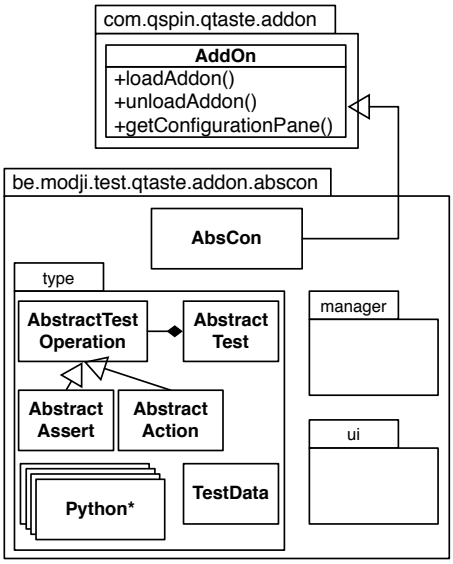


Figure 9.4: AbsCon packages diagram

assertions (`AbstractAssert`) and actions (`AbstractAction`). The SUT's interface model and the assertions and actions mapping are encoded using Python (classes `PythonClass`, `PythonConstructor`, `PythonMethod`, and `PythonParameter`, abbreviated `Python*` in Figure 9.4).

Finally, the data feeding the CSV file used by QTaste during test cases execution is represented by the `TestData` class. Manager classes (from the `manager` package) play a role of controller during the plugin execution and dialogue with the classes from the `ui` package, in charge of the AbsCon plugin graphical user interface.

9.3.3 Source code

AbsCon is released as an open source project on GitHub, under the GNU public license. It can be downloaded at the following address: https://github.com/modji-be/AbsCon. The project is written in Java and has 34 classes, 248 methods, and 4864 lines of code.

9.4 Discussion

In this section, we discuss AbsCon's abstraction and user interface modelling mechanisms, maintenance costs, and mappings, based on our experience with the tool. A complete and rigorous evaluation of the tool, using a controlled experiment [317] with test engineers, is left for future work.

Abstraction: AbsCon heavily relies on **abstraction**, of the SUT's interface on the one hand, and on the assertion and actions on the other hand. This abstraction

layer allows to define each mapping independently from the higher or lower levels: abstract test cases are defined using assertions and actions with meaningful names for the user/test engineer; each assertion and each action is mapped to a Python function representing a verification or an operation, and is defined as a manipulation of the SUT's interface meaningful from a user point of view (depending on the nature of the SUT, the user may be a human or another system), thanks to the SUT's interface model; finally, the SUT's interface model encapsulates the test API calls in charge of the effective communication with the SUT.

User interface modelling: Another option to the modelling of the SUT's interface described in this chapter would be to use User Interface Description Languages (UIDLs) [123] such as USIXML [194]. These languages provide generic constructs (organized in one or more metamodels that represent both platform independent and platform specific views, according to Model-Driven Architecture principles [166]) allowing to model any kind of user-interface (including non-conventional interfaces such as voice-enabled ones).

However, the use of such proposals in ours raises the following problem: the number of concepts they are offering being quite large, modelling a simple user interface can be cumbersome and complex, unless we tailor the language to specific needs. In our context, we do not to try to model the whole user interface but the subset concerned by the tests. We therefore adopted a lightweight approach that has the complementary advantage of not requiring any new modelling language to learn, by exploiting Python's object-orientation facilities. Furthermore, as initially mentioned, QTaste's spectrum is larger than testing graphical user interfaces.

Maintenance costs: The goal of our abstraction layer is to reduce the overall complexity of the test cases and to decrease the **maintenance costs**. Indeed, when the SUT's interface evolves, only the mapping to this interface has to be (potentially) changed, AbsCon can then re-generate concrete test cases for QTaste that will serve for non-regression. This process is much lighter than the update of QTaste test cases as it will (potentially) require to update all the test cases containing code that manipulate the SUT's interface (using directly the test API in this case). In the same way, when functionalities are added to the SUT, only the new interface elements, and verifications and operations mappings have to be added. New abstract test cases may then be written for those elements and AbsCon can re-generate a complete set of test cases for the whole SUT.

Mappings: The definition of the different mappings may represent an additional effort during test activities. However, different aspects have to be taken into account. First, the SUT's interface model depends solely on the nature of the SUT, *e.g.,* Web-application for the `web` model from Figure 9.2. Once defined, this model may be reused across different projects. As this model abstracts the test API by defining methods from the interface point of view (*e.g.,* `click`, `open`, `getTitle`, *etc.* in Figure 9.2), we believe that it will also soften QTaste's learning curve. Second, the definition of the mappings enables integrating existing model-based testing

techniques (*e.g.*, [82, 307]) rather than defining a new complete test development process.

In our opinion, the most time consuming task will be to **identify** and **map** the different SUT's **interface elements**. This cost may be reduced in some cases using existing tools: for instance, *Inspect* [215] or *SwingInspector* [295] are tools used to identify and access graphical user interface elements in classical desktop applications. In our `UiMapping.py` example in Listing 9.3, we used Firefox's inspection tool to identify the different elements on a Web page. Depending on the nature of the SUT, this mapping may also be partially or totally automated (this will be part of our future works), like for Web-applications for which each element on a Web page describes itself using HTML tags.

9.5 Related work

Test case concretization techniques are classified by Utting *et al.* [307] in 3 categories: **adaptation** approaches abstracts the SUT by using a wrapper (also called an adapter), test cases call this wrapper in order to execute operations on the SUT; **transformation** approaches transform abstract test cases into test cases directly executable by the SUT, possibly using additional information; and **mixed** approaches also transform abstract test cases in executable test cases, but using an adaptation layer in order to abstract the SUT. Using this classification, QTaste uses adaptation to abstract the SUT using its test APIs and requires to write test cases which will use those test APIs.

There exists other adapters, like **Selenium** and **Sahi** [265] to test Web-applications, or **AutoHotKey** [1] to test Windows applications. Tools like **Sikuli** [258] and **Squish** [109] provide adaptation mechanisms to perform graphical user interface testing using techniques like image recognition, or recording and playback. None of these tools natively support abstract test case concretization.

Other transformation and mixed tools like **TOTEM** [44], **SpecExplorer** [312], **MaTeLo** [96], **Smartesting** solutions [280], or **STALE** [191] implement full model-based testing approaches, including abstract test case generation and concretization from different modelling languages (*e.g.*, UML Testing Profile [28], etc).

Rather than having a complete transformation chain (from models to executable test cases), we developed AbsCon in order to plug it on an existing approach (VIBeS in this case), concretize abstract test cases, no mater their origin as long as they are described as sequences of actions and assertions, and get executable test cases on a generic and industrial test environment like QTaste.

As for VIBeS, other model-based testing approaches produce abstract test cases that are concretized using existing tools, this is the case for **Skyfire** [190] which uses a transformation approach to produce **Cucumber** [201] abstract test cases from UML diagrams. Cucumber is a popular behaviour-driven development tool that aims at producing typical examples of the behaviour of a system under development, described using a semi-structured language: **Gherkin**. Those examples are used as acceptance tests and concretized using a Java annotations based mechanism, mapping semi-structured sentences to Java methods using a defined string pattern. The

executable test cases are run in standard JUnit environment. Cucumber could have been another test execution environment target, but, to the best of our knowledge, it does not provide any SUT's interface abstraction mechanism (like QTaste's test APIs) and would have required more effort to define a programmer friendly abstraction mechanism of this interface.

9.6 Wrap up and perspectives

In this chapter, we presented AbsCon, a QTaste plugin developed to concretize abstract test cases represented as sequences of actions and assertions. The adaptation mechanism provided by QTaste's test API is enhanced by a programmer friendly way to encapsulate the calls to this API using a common model specific to the kind of the SUT's interface. This model, **reusable** for different SUTs as long as their interface are of the same kind, defines the possible interactions with the SUTs. An instance of this model, specific to a SUT, is used in operations and verifications corresponding to actions and assertions defined in the abstract test cases. Using the different mappings, AbsCon is able to generate test cases executable in QTaste.

Originally developed to bridge the gap between VIBeS and concrete test cases, AbsCon offers multiple advantages, even in a non model-based testing context. We chose to implement it over an existing industrial test case management and execution tool, which will, we believe, eases its broader adoption. As a standalone tool (*i.e.,* not used in an model-based testing chain), AbsCon enhances QTaste's **genericity** by **raising the abstraction level** of different elements: the SUT's interface and test APIs, thanks to the SUT's interface model mechanism; and the test cases themselves by allowing to provide definitions using abstract actions and assertions (which is to the user) instead of Python scripts.

So far, the plugin has only been used on small examples, a more complete validation is part of our future works. We will also explore automated SUT's interface mapping possibilities using existing inspection tools. Finally, another potentially interesting research direction is the definition of the test cases using a structured natural language (like Gherkin [201]) as an input to AbsCon instead of XML files. This could be used to automatically define, not only the actions and assertions, but also the data to use during the test cases execution. Ideally, the definition of the test cases in a structured language would be processed by AbsCon to populate both the list of assertions and actions to map, the elements of the SUT's interface to use (based on the text describing the test cases steps), and the CSV file used by QTaste.

More details on AbsCon can be found in Jeremy Vanhecke's master thesis [311].

Part IV

Postface

CHAPTER 10

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Software product line testing is a complex, yet essential, task to guarantee a **good enough** quality level of the products. The test engineer has to compromise between the large number of products to test and a limited testing budget. This requires a strong and usable framework to support the whole testing process, from test case selection at the domain level to test case concretization into a test script for one particular product.

In this thesis, we present a model-based **behavioural** SPL testing framework. Our approach relies on formal ground without sacrificing usability in a unified and flexible enough model-driven framework. We believe that this combination will foster the usage of efficient SPL testing techniques, thus improving the confidence in the SPL paradigm.

10.1 Summary of contributions

By working on domain artefacts with a FTS and a feature model, we developed a **behavioural family model-based testing framework**. The output of the whole chain is a test suite for, potentially, one product, a subset of products or even the whole product line according to the provided selection criteria. We consider three types of criteria: criteria based on the **structure** of the FTS; criteria based on a **dissimilarity heuristic**; and criteria based on **usages** of the SPL.

As a complement to selection criteria, mutation testing allows to improve a test suite by assessing its quality using mutation analysis and select test cases for the live mutants. To face the cost of such analysis for a large number of mutants, we propose to take advantage of the variability formalisms to compactly represent all

possible mutations in a single model: the **Featured Mutants Model (FMM)**. In a FMM, each feature represents a mutation: *i.e.,* a model transformation representing the application of a mutation operator on the model. It allows to generate mutants of any order and assess test effectiveness via an optimised execution scheme.

To detect equivalent mutants that may impair the analysis, we offer two baseline algorithms based on **random simulation**, and compare them to **language equivalence** under weak and strong mutation scenarios. Our evaluations suggest to use simulations first to quickly discard many non-equivalent mutants, and then employ exact approaches only on a small amount of probably equivalent mutants to speed up equivalence analysis.

Our framework, called **VIBeS**, is implemented in Java as an open-source modular Maven project. Each module is dedicated to one particular aspect of the testing activities. Each one has a dedicated API and is also encapsulated in a Java DSL to simplify usages. VIBeS is the reference implementation to assess the different elements presented in this thesis. Our empirical assessments are performed on several case studies, representing embedded systems, with manually defined models, and Web applications, with semi-automatically reverse engineered models.

10.2 Perspectives and future work

This section presents our perspectives and future potential research directions to improve test case selection and model-based product line mutation analysis.

10.2.1 Test case selection

Test case selection may be improved in different ways: we limit ourselves to two possible directions, presented hereafter.

Multi-objectives selection: Although we consider structural, dissimilar, and usage criteria separately, we may **combine** those different approaches in order to refine the test case selection. Typically, test case selection using structural criteria, when applied to an FTS, becomes a compromise between the coverage of the behavioural model, the number of test cases selected, and the number of products required to execute those test cases. **Evolutionary algorithms**, like the one used for dissimilarity selection, can handle this kind of optimisation problems.

Dissimilar test case selection can also be extended with different measures (*e.g.,* test cases structural coverage) as well as different ways to combine them to perform dissimilarity selection. For instance, the operator used to combine the different distances ($\otimes$) can be refined to take more measures into account and balance them, depending on their significance, to foster one or more particular distances.

Aspects other than usage or functional elements (like structural coverage or dissimilarity) of the product line may also play a role in the test case selection process. *E.g.,* the cost (in time and/or material) linked to the configuration of some products can be taken into account: algorithms can be modified to prefer test

cases requiring cheaper products for their execution. Recent developments in the SPL verification and validation community tend to consider more and more **non-functional properties** of SPLs, both at the feature model level [31, 100, 124, 232, 242, 287, 288] and in the behavioural model (like usage information) [231, 262, 296, 298]. All this information can be used to tune the evolutionary algorithm to refine the test case selection.

Mutation coverage driven selection: Since *mutants are a valid substitute for real faults* [158], we envision to develop test case selection techniques based on mutation coverage of a FMM. The idea is to select test cases designed to **detect live mutants**, using for instance counter-examples generated by a FTS model checker tool [72]. This would allow to select, not only positive abstract test cases (test cases that the products should be able to execute), but also negative abstract test cases (test cases that the product should not be able to execute) [11], enhancing the test case selections presented in Chapter 5.

10.2.2 Towards model-based product line mutation analysis

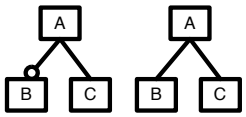
Model-based mutation analysis requires to use a set of mutation operators to produce mutants from the specification of the product line under test. Those operators are usually designed based on empirical studies build upon large error repositories, or on a fault model defined to list potential failure causes in a system [206]. McGregor [208] defines a fault model for software product lines and describes for each step of the product line development, the kind of fault that may appear.

In the remainder of this section, based on the contributions of Chapter 6, we sketch a vision of a complete **model-based product line mutation analysis**. We discuss how our contributions and existing research literature on mutation testing contribute to this vision and point out future research directions. We consider the following artefacts as possible candidates for mutation:

- the feature model, defining how features may be combined to form valid products;
- domain artefacts, reusable across several products (*i.e.,* the FTS);
- the mapping between the feature model and the domain artefacts (*i.e.,* the feature expression labels); and
- the derivation process used to bind variability in the domain artefacts to get one product (*i.e.,* the FTS projection operator).

The last artefacts are the products themselves (that can be mutate using standard mutation techniques [152, 206]).

Feature model mutation: Feature model mutation is already largely covered by literature to assess product sampling (in this case, test cases are valid products of the product line) and generate better samples [19, 133, 174, 261], or to detect errors and repair feature models [18, 20, 136]. This last line of research uses mutation on the feature model in order to automatically improve or repair it when an error or an inconsistency is found. We distinct two kinds of mutation operators: mutation



(a) Original (b) Mutant

Figure 10.1: Example of syntactic mutation of a feature model [19]

operators working on the **syntax** of the feature model and operators working on the **semantic** of the feature model. Operators working on the semantic are operators working on the syntax of the formalism used to express the semantic of the feature model (*i.e.*, a boolean formula). We use syntax and semantic only to clearly distinct the abstraction level at which the mutation is performed.

Syntactic mutation: Mutation operators working on the syntax of the feature model transform the graphical (or textual) representation of the feature model to produce a mutant. For instance, Arcaini *et al.* [19] define operators that will transform an alternative into a *Or* or a *And*, make an optional feature mandatory, change *requires* constraint to *exclude* constraint, *etc.* Figure 10.1 presents an example of syntactic change in a small feature model: the optional constraint in Figure 10.1(a) has been transformed to a mandatory feature in Figure 10.1(b).

Semantic mutation: Semantic mutation works on the semantic of the feature model: a boolean expression over the features of the feature model [276]. Application of those operators will first require a flattening of the feature model into a CNF formula [134] and apply mutations on this formula. Henard *et al.* [133, 136] use this strategy to select products to test in order to detect mutations of the feature model. The feature model is first transformed into a CNF formula which is used an input for two mutation operators modifying on clause of the CNF formula. For instance, the feature model of Figure 10.1(a) is transformed into the following CNF clause:

$$fm = A \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg A \vee C)$$

Which may then be mutated (using Henard *et al.*'s *second operator* [136]) to:

$$fm_m = A \wedge \neg B \wedge A \wedge (\neg C \vee A) \wedge (\neg A \vee C)$$

Since the CNF formula represents the feature model, it is possible to show an equivalence between syntactic and semantic mutation operators. However, one application of a syntactic mutation operator may require several applications of semantic mutation operators (and *vice versa*). For instance, the syntactic operator transforming an alternative to a *And* will require several transformations on the CNF formula representing the feature model.

Featured transition system mutation: FTSs describe the behaviour of all the products of a product line. Mutate a FTS comes to modify the behaviour of a whole

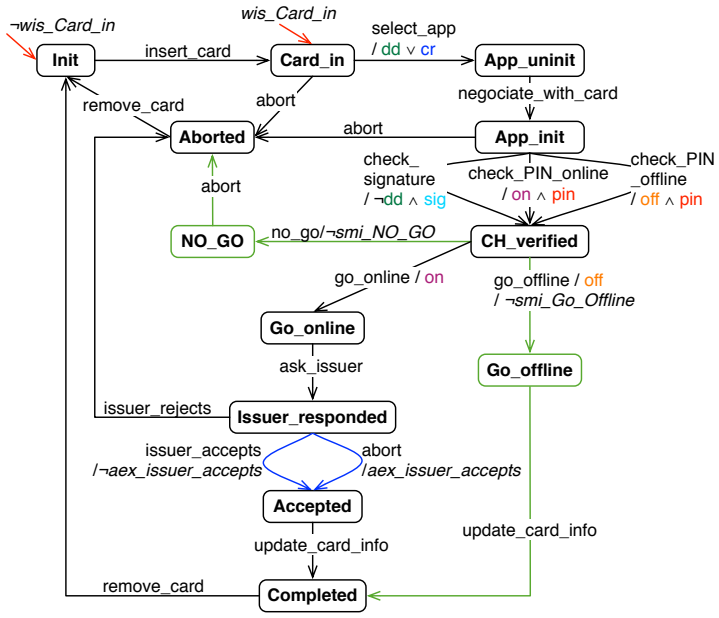
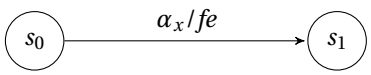


Figure 10.2: Card payment terminal product line FFTS

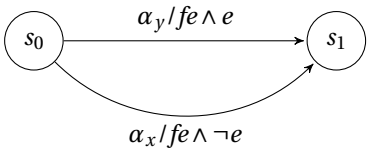
product line (or at least the products able to execute the modified part of the FTS). The mutation operators defined in Annex A may be used to produce mutants. In such case, the FMM has a FTS with two feature models and two γ labelling functions. The first feature model d_p and γ_p function are used to represent all the products of the product line, and the second feature model d_m and γ_m function are used to represent all the mutants of the mutants family. For instance, the result of the card payment terminal product line of Figures 4.2(b), mutated using the state missing, action exchange, and wrong initial state operators, produces the FTS in Figure 10.2. The FMM is composed of this FTS, the feature model of the product line (in Figure 4.2(a)), and the feature model of the mutants family (in Figure 6.3(a)). As for feature model mutation, modifying the FTS using model transformations are **syntactic mutations** and affects the product line as a whole. *E.g.*, removing one transition using transition missing operator removes it for all the products of the product line.

One may want to mutate only the behaviour of a certain subset of products. In this case, the operators defined in Annex A have to be modified in order to consider only a given subset of products, represented as a feature expression e. For instance, if we mutate the following transition in an **enumerative** way, with e representing a subset of the products defined by the feature expression fe:

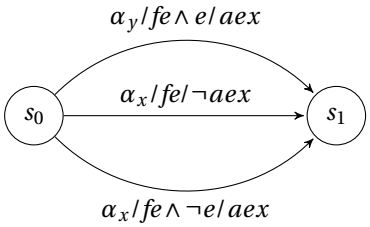


By applying the AEX operator to change action α_x to α_y, but only for the subset of

products designed by e, we have the mutant:



Where the original transition is restricted to $fe \wedge \neg e$ and the mutated transition is only activated for products respecting $fe \wedge e$ feature expression. Using the **FMM** approach, we need to duplicate transitions to take the feature expression representing the mutant into account:



Modifying only a subset of the product line requires to take the **semantic** of the FTS (*i.e.,* the FTS as a compact representation of all the products of the product line) into account.

Feature expression mutation: Feature expressions are boolean expressions over features. They are used to represent the set of products able to execute a given transition in a FTS: it **maps** the variability defined in the feature model to the behavioural description of the product line. Feature expression mutation may be done using classical boolean mutation operators [206] in order to mutate this mapping.

Projection operator mutation: The projection operator is used to bind the variability in a given FTS by resolving the feature expressions for a given product: *i.e.,* an assignment of the feature variables, *true* denoting a feature included in the product and *false* a feature not included in the product. Mutate the projection operator introduces faults in the **derivation process**, which results in a faulty specification of the product behaviour (*i.e.,* a wrong LTS). The mutation operators may include, for instance, switching feature assignment (f becoming $\neg f$), considering all features to true or false, returning a constant value (all features expressions are evaluated to true of false), *etc.*

Wrap up: In this section, we present possible solutions and research directions to apply a model-based product line mutation analysis. Mutation may be done using different **artefacts** as inputs of the mutation operators and works at different levels of abstraction. We distinct mutation performed at **syntactic** level from mutation at **semantic** level. Both syntactic and semantic mutations of the feature model

changes the set of valid products of the SPL. Existing approaches may be included in VIBeS. Those kinds of mutations may be detected by a test case, if one of the transitions fired by this test case on the original system may not be fired any more on the mutant. Concretely, the feature expression of this transition has to violate the constraints of the mutant feature model. FTS mutation for only a subset of the product line requires to modify the existing set of mutation operators (defined in Annex A). We believe that those mutations are more subtle as they allow to modify only a limited subset of the products, corresponding intuitively to undesired feature interactions (at the model level) preventing this subset of products to behave as expected. Other kinds of mutation includes mutation of the feature expressions in the FTS (using classical boolean mutation operators) and mutation of the projection operator (using new operators).

Future work: In our future work, we will refine the vision sketched here and enhance VIBeS's mutation analysis by following the aforementioned directions. We will also further investigate scalability issues regarding mutation analysis for any order mutants. This implies the optimisation of the boolean formulas or approximate computation heuristics. To address the equivalent mutant problem in a family-based fashion, we intent to investigate usage of automata language equivalence (or other equivalence approaches) for FTSs. For now, FMMs are used only to represent behavioural mutations. We intend to extend this set of mutation operators to mutate variability information of the input FTS and feature model. Finally, scalability of mutation analysis for large SPLs has to be evaluated in the long run.

10.3 Final remarks

This thesis is dedicated to SPL testing but we believe that the contributions may be extended to other kinds of systems. SPLs are variability intensive systems that have been developed in a structured process, divided into domain engineering and application engineering [252]. But variability is not limited to product lines. As suggested by the name of our implementation Variability Intensive Behavioural teSting (VIBeS) and the Web application case studies used in this thesis, our work may be used to test other sorts of systems: plugin-based system like WordPress for instance.

Relevant test suites and products selection becomes even more important considering the way software is developed nowadays. For instance, Agile methods, continuous integration and delivery, and fast releasing requires to execute test cases using a limited testing budget (usually overnight), making test cases selection critical to ensure a required quality level. This also raises further research directions: for now, VIBeS' inputs are a FTS and its feature model. Since FTS is an abstract formalism, it allows to be expressed using other modelling languages like fPromela [63], dedicated to FTS model checking. We believe that using lightweight modelling languages dedicated to testing (like Gherkin [201]) with variability information may foster the adoption of SPL testing techniques and help the community to master the variability more and more present in software systems.

With the emergence of the so-called *DevOps*, development and system operation teams inside an organisation are aligned and collaborate intensively to ensure rapid, frequent, and reliable software building, testing, and delivery. This includes a sharing of information between development environment and operating environment. In such case, the input model may even be enhanced (automatically or not) with other kinds of information automatically inferred from the running systems, the versioning engine, or the continuous integration server. This allows to tailor and refine the test case selection even further. It would allow to include variability intensive systems in a continuous test cycle, where test cases are selected and executed based on inputs from different sources contributing to a continuous quality improvement cycle.

APPENDIX

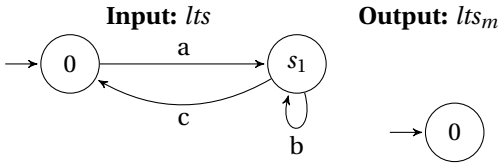
MUTATION OPERATORS

In the following sections, we define mutation operators and FMM mutation operators, inspired from Fabbri et al. [101].

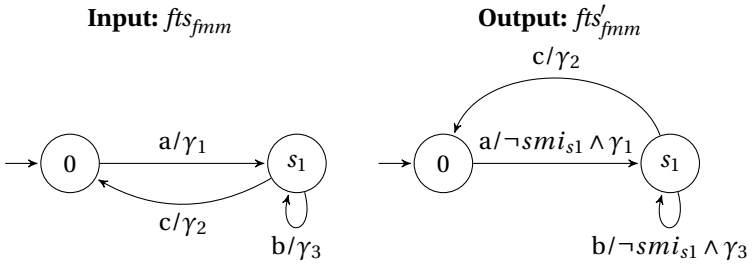
A.1 State missing (SMI)

Removes a random state $s_i \neq i$ (except initial state).

A.1.1 Enumerative approach



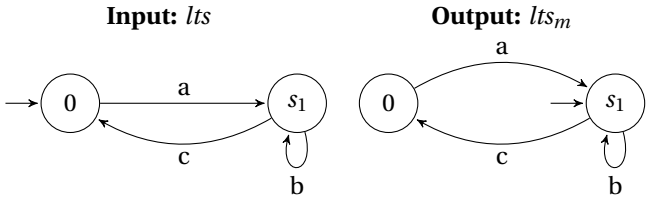
A.1.2 FMM approach



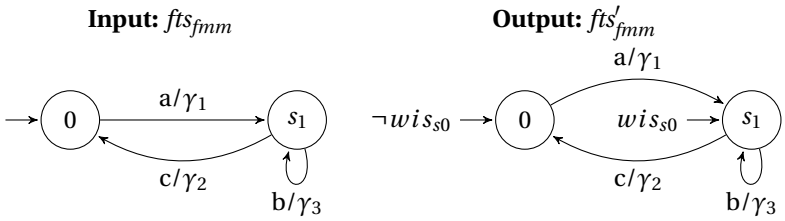
A.2 Wrong initial state (WIS)

Modifies the start state of the transition system to another random state.

A.2.1 Enumerative approach



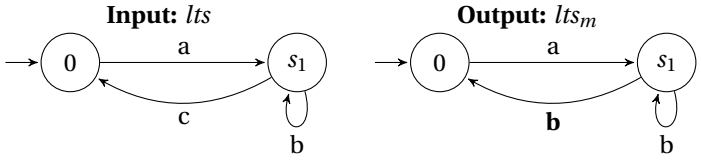
A.2.2 FMM approach



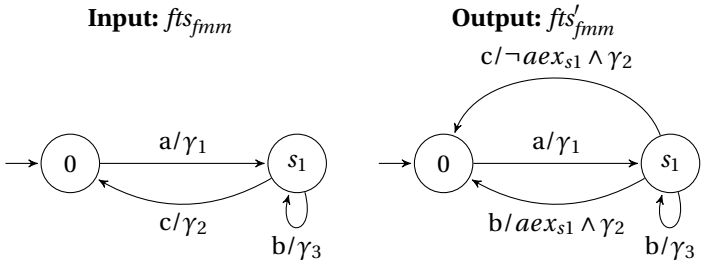
A.3 Action exchange (AEX)

Modifies an action on a transition and replace it by another action.

A.3.1 Enumerative approach



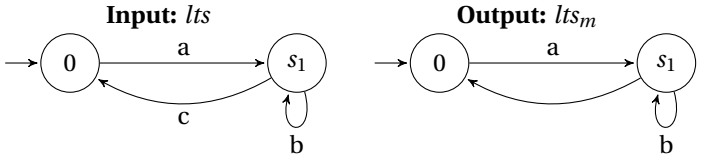
A.3.2 FMM approach



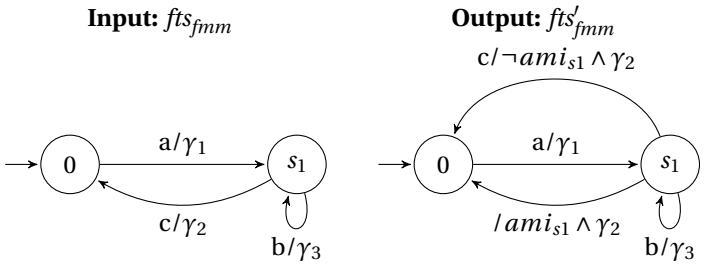
A.4 Action missing (AMI)

Removes the actions from a transition.

A.4.1 Enumerative approach



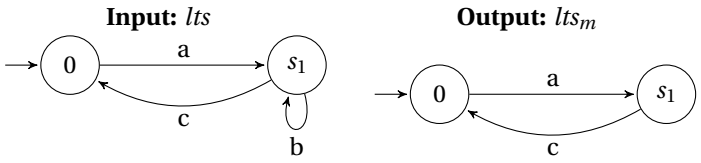
A.4.2 FMM approach



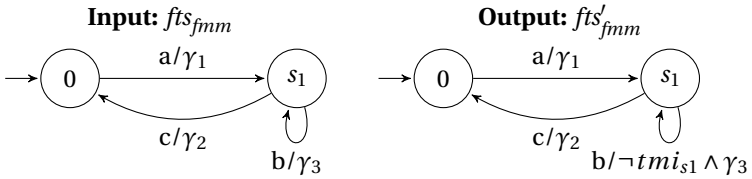
A.5 Transition missing (TMI)

Removes a transition from the system.

A.5.1 Enumerative approach



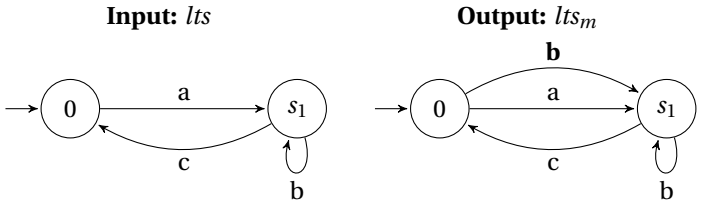
A.5.2 FMM approach



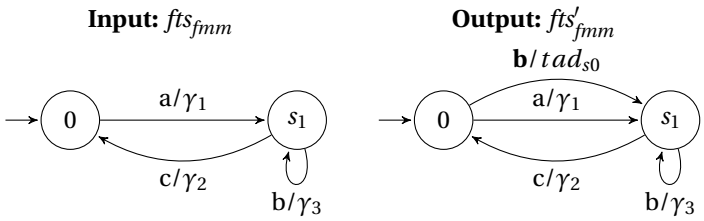
A.6 Transition add (TAD)

Adds a transition to the system by randomly picking up two states and an action. Note: this corresponds to the event extra operator in [101]. Adding an action without adding a transition with this action has no sense since it can not be detected without being fired.

A.6.1 Enumerative approach



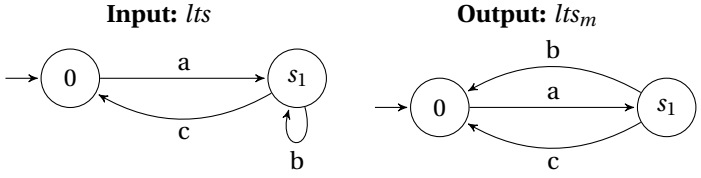
A.6.2 FMM approach



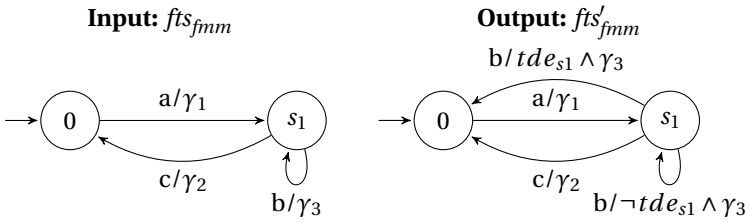
A.7 Transition destination exchange (TDE)

Changes the destination of a transition to the system by randomly picking up another state in the system.

A.7.1 Enumerative approach



A.7.2 FMM approach



APPENDIX B

MUTANTS EXECUTION TIME RESULTS

This annex presents the complete results of the mutant (1st order) execution time (in μ-seconds) for the assessment described in Section 7.5. The tables contain minimal, maximal, median, mean, standard deviation time for every test case on all live and killed mutants of the enumerative method and of the FMM. Mutation score (MS) of the all actions and random test sets are provided for each product.

B.1 S.V.Mach.

Soda V. Mach. product (all act. MS: 0.85 ; random MS: 0.85)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	57	21	20	1.1
Max.	442	154	83	5.3
Median	113	79	38	2.5
Mean	120	78	39	2.5
S.Dev.	43	35	7.1	6.1

B.2 Minepump

Minepump product (all act. MS: 0.60 ; random MS: 0.82)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	441	43	26	1.7
Max.	623	212	64	9.7
Median	533	108	41	8
Mean	530	100	42	7.6
S.Dev.	35	49	6.5	34

B.3 Claroline

Claroline product (all act. MS: 0.07 ; random MS: 0.27)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	40,314	236	26	9.1
Max.	103,346	4,091	19,282	5.4
Median	53,951	652	58	380
Mean	57,000	870	280	100
S.Dev.	13,000	710	1,400	22

B.4 AGE-RR

AGE-RR product (all act. MS: 0.66 ; random MS: 0.27)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	598,520	5,644	39	140
Max.	3,948,806	99,754	46,839	84
Median	910,000	9000	110	3,300
Mean	1.1e+06	14,000	200	2,900
S.Dev.	590,000	12,000	790	870

B.5 Elsa-RR

Elsa-RR product (all act. MS: 0.75 ; random MS: 0.49)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	20,743	775	104	7.5
Max.	59,237	13,400	3,109	19
Median	22,676	918	191	89
Mean	27,000	1,300	230	62
S.Dev.	8,500	1,500	170	84

B.6 Elsa-RRN

Elsa-RRN product (all act. MS: 0.77 ; random MS: 0.30)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	34,999	1,286	93	14
Max.	166,433	34,498	36,158	4.6
Median	45,000	1,600	180	200
Mean	52,000	2,400	300	90
S.Dev.	17,000	3,200	1,600	17

B.7 Random

Random model (all act. MS: 0.16 ; random MS: 0.63)				
	Live m.	*Killed m.*	*FMM*	*Speedup*
Min.	327,418	24,675	875	28
Max.	2,552,363	140,917	60,354	42
Median	1.3e+06	55,000	1,500	160
Mean	1.3e+06	63,000	1,800	370
S.Dev.	560,000	29,000	3,500	210

APPENDIX C

MUTANTS EQUIVALENCE ANALYSIS RESULTS

This appendix presents the results of the different weak (Weak Mutation (WM)) and strong mutations (Strong Mutation (SM)) ALEs/BSs/RSs algorithms. For each algorithm, a table gives the recall, the average execution time (**time**), and the standard deviation (σ).

C.1 S.V.Mach.

			Weak Mutation			**Strong Mutation**		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	98%	0.02	0.03	91%	0.26	1.00
	1e-10	0.10	97%	0.02	0.02	91%	0.04	0.06
	1e-05	0.10	97%	<0.01	0.02	91%	0.03	0.05
	0.10	0.10	98%	0.01	0.02	91%	0.02	0.04
RS	1e-10	0.01	97%	0.02	0.03	N/A	N/A	N/A
	1e-10	0.10	96%	0.01	0.02	N/A	N/A	N/A
	1e-05	0.10	97%	<0.01	0.01	N/A	N/A	N/A
	0.10	0.10	97%	0.01	0.03	N/A	N/A	N/A

C.2 C.P.Term.

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	97%	0.49	9.05	91%	0.21	0.76
	1e-10	0.10	96%	0.02	0.10	91%	0.04	0.05
	1e-05	0.10	97%	0.01	0.05	91%	0.03	0.05
	0.10	0.10	96%	0.01	0.03	91%	0.03	0.04
RS	1e-10	0.01	97%	0.49	9.04	N/A	N/A	N/A
	1e-10	0.10	96%	0.02	0.11	N/A	N/A	N/A
	1e-05	0.10	97%	<0.01	0.05	N/A	N/A	N/A
	0.10	0.10	96%	0.01	0.04	N/A	N/A	N/A

C.3 Minepump

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	<0.01	<0.01
BS	1e-10	0.01	98%	0.40	8.54	92%	0.21	0.80
	1e-10	0.10	98%	0.02	0.15	92%	0.04	0.06
	1e-05	0.10	99%	<0.01	0.04	92%	0.03	0.05
	0.10	0.10	98%	0.01	0.04	92%	0.03	0.04
RS	1e-10	0.01	98%	0.39	8.43	N/A	N/A	N/A
	1e-10	0.10	98%	0.02	0.15	N/A	N/A	N/A
	1e-05	0.10	98%	<0.01	0.06	N/A	N/A	N/A
	0.10	0.10	98%	0.01	0.05	N/A	N/A	N/A

C.4 Claroline

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	0.02	0.02	100%	0.10	0.12
BS	1e-10	0.01	99%	3.62	49.96	98%	0.59	2.00
	1e-10	0.10	99%	0.09	0.57	98%	0.17	0.42
	1e-05	0.10	99%	0.07	0.32	98%	0.17	0.28
	0.10	0.10	99%	0.05	0.12	98%	0.18	0.71
RS	1e-10	0.01	96%	29.99	139.34	N/A	N/A	N/A
	1e-10	0.10	95%	0.39	1.52	N/A	N/A	N/A
	1e-05	0.10	94%	0.23	0.80	N/A	N/A	N/A
	0.10	0.10	94%	0.10	0.25	N/A	N/A	N/A

C.5 Elsa-RR

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	1.05	0.67
BS	1e-10	0.01	99%	0.06	0.05	95%	0.96	3.86
	1e-10	0.10	100%	0.04	0.04	95%	0.15	0.27
	1e-05	0.10	99%	0.05	0.04	95%	0.13	0.19
	0.10	0.10	100%	0.02	0.03	95%	0.09	0.16
RS	1e-10	0.01	88%	73.03	209.50	N/A	N/A	N/A
	1e-10	0.10	86%	0.92	2.56	N/A	N/A	N/A
	1e-05	0.10	86%	0.51	1.38	N/A	N/A	N/A
	0.10	0.10	87%	0.13	0.33	N/A	N/A	N/A

C.6 Elsa-RRN

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	0.01	0.01	100%	3.64	2.29
BS	1e-10	0.01	100%	0.05	0.05	90%	2.93	10.34
	1e-10	0.10	100%	0.04	0.04	90%	0.18	0.25
	1e-05	0.10	99%	0.04	0.04	90%	0.16	0.21
	0.10	0.10	100%	0.03	0.03	90%	0.10	0.11
RS	1e-10	0.01	97%	19.24	100.73	N/A	N/A	N/A
	1e-10	0.10	95%	0.37	1.42	N/A	N/A	N/A
	1e-05	0.10	95%	0.22	0.75	N/A	N/A	N/A
	0.10	0.10	94%	0.08	0.21	N/A	N/A	N/A

C.7 AGE-RR

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	0.64	0.94	100%	21.18	13.70
BS	1e-10	0.01	100%	0.06	0.08	90%	9.38	42.87
	1e-10	0.10	100%	0.05	0.10	90%	0.24	0.45
	1e-05	0.10	100%	0.04	0.08	90%	0.18	0.47
	0.10	0.10	100%	0.03	0.04	89%	0.09	0.25
RS	1e-10	0.01	96%	38.68	188.18	N/A	N/A	N/A
	1e-10	0.10	94%	0.68	2.50	N/A	N/A	N/A
	1e-05	0.10	95%	0.35	1.27	N/A	N/A	N/A
	0.10	0.10	94%	0.14	0.47	N/A	N/A	N/A

C.8 AGE-RRN

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	0.21	0.22	100%	75.29	51.92
BS	1e-10	0.01	100%	0.04	0.07	95%	7.10	32.32
	1e-10	0.10	100%	0.05	0.05	95%	0.32	0.46
	1e-05	0.10	100%	0.04	0.04	95%	0.27	0.43
	0.10	0.10	100%	0.04	0.04	95%	0.21	0.31
RS	1e-10	0.01	90%	117.21	362.41	N/A	N/A	N/A
	1e-10	0.10	88%	1.98	4.78	N/A	N/A	N/A
	1e-05	0.10	87%	1.12	2.63	N/A	N/A	N/A
	0.10	0.10	85%	0.41	0.87	N/A	N/A	N/A

C.9 Random models

C.9.1 Random 1

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	448.61	339.19
BS	1e-10	0.01	100%	0.08	0.07	92%	0.78	2.50
	1e-10	0.10	100%	0.07	0.07	92%	0.09	0.08
	1e-05	0.10	100%	0.07	0.07	92%	0.09	0.06
	0.10	0.10	99%	0.07	0.07	92%	0.07	0.05
RS	1e-10	0.01	100%	0.03	0.07	N/A	N/A	N/A
	1e-10	0.10	100%	0.03	0.11	N/A	N/A	N/A
	1e-05	0.10	100%	0.03	0.09	N/A	N/A	N/A
	0.10	0.10	99%	0.03	0.08	N/A	N/A	N/A

C.9.2 Random 2

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	412.37	168.90
BS	1e-10	0.01	100%	0.11	0.06	89%	1.22	3.35
	1e-10	0.10	100%	0.10	0.06	89%	0.14	0.09
	1e-05	0.10	100%	0.11	0.07	89%	0.14	0.08
	0.10	0.10	100%	0.11	0.06	89%	0.12	0.07
RS	1e-10	0.01	100%	0.04	0.10	N/A	N/A	N/A
	1e-10	0.10	100%	0.03	0.07	N/A	N/A	N/A
	1e-05	0.10	100%	0.03	0.07	N/A	N/A	N/A
	0.10	0.10	99%	0.03	0.09	N/A	N/A	N/A

C.9.3 Random 3

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	367.99	154.80
BS	1e-10	0.01	100%	0.11	0.06	91%	1.04	3.20
	1e-10	0.10	100%	0.09	0.04	91%	0.23	0.15
	1e-05	0.10	100%	0.09	0.04	91%	0.23	0.14
	0.10	0.10	100%	0.09	0.05	91%	0.19	0.12
RS	1e-10	0.01	100%	0.03	0.10	N/A	N/A	N/A
	1e-10	0.10	100%	0.03	0.16	N/A	N/A	N/A
	1e-05	0.10	99%	0.03	0.12	N/A	N/A	N/A
	0.10	0.10	99%	0.02	0.07	N/A	N/A	N/A

C.9.4 Random 4

			Weak Mutation			Strong Mutation		
	δ	ϵ	Recall	$\overline{time}$	σ	Recall	$\overline{time}$	σ
ALE			100%	<0.01	<0.01	100%	306.37	127.02
BS	1e-10	0.01	100%	0.11	0.06	91%	1.09	3.23
	1e-10	0.10	100%	0.10	0.05	91%	0.22	0.14
	1e-05	0.10	100%	0.10	0.05	91%	0.23	0.12
	0.10	0.10	100%	0.09	0.04	91%	0.19	0.11
RS	1e-10	0.01	100%	0.04	0.25	N/A	N/A	N/A
	1e-10	0.10	99%	0.03	0.25	N/A	N/A	N/A
	1e-05	0.10	100%	0.03	0.10	N/A	N/A	N/A
	0.10	0.10	99%	0.02	0.09	N/A	N/A	N/A

BIBLIOGRAPHY

[1] AHK. AutoHotKey. http://ahkscript.org, 2016.

[2] Bernhard K Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korošec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-Based Mutation Testing of an Industrial Measurement Device. In Martina Seidl and Nikolai Tillmann, editors, **Tests and Proofs**, volume 8570 of **LNCS**, pages 1–19. Springer, 2014.

[3] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. **Software Testing, Verification and Reliability**, 25(8):716–748, dec 2015.

[4] Bernhard K Aichernig and Elisabeth Jobstl. Towards Symbolic Model-Based Mutation Testing: Pitfalls in Expressing Semantics as Constraints. In **2012 IEEE Fifth International Conference on Software Testing, Verification and Validation**, pages 752–757. IEEE, apr 2012.

[5] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. **Science of Computer Programming**, 97:383–404, jan 2015.

[6] Airbus Defence & Space. Sferion TM. http://www.defenceandsecurity-airbusds.com/fr/sferion, oct 2014.

[7] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. Mutation Operators for Preprocessor-Based Variability. In **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16**, VaMoS '16, pages 81–88. ACM Press, 2016.

[8] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective product-line testing using similarity-based product prioritization. **Software and Systems Modeling**, pages 1–23, 2016.

[9] ALL4TEC. MaTeLo. http://www.all4tec.net/MaTeLo/homematelo.html, 2014.

[10] K Altisen, F Maraninchi, and D Stauch. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. **Science of Computer Programming**, 63(3):297–320, 2006.

[11] P E Ammann, P E Black, and W Majurski. Using model checking to generate tests from specifications. In **Formal Engineering Methods, 1998. Proceedings. Second International Conference on**, pages 46–54, 1998.

[12] J H Andrews, L C Briand, and Y Labiche. Is mutation an appropriate tool for testing experiments? In **Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on**, pages 402–411. IEEE, 2005.

[13] J.H. Andrews, L.C. Briand, Y Labiche, and A.S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. **IEEE Transactions on Software Engineering**, 32(8):608–624, aug 2006.

[14] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin. In **Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange - eclipse '04**, pages 67–72. ACM, ACM Press, 2004.

[15] Apache. HTTP Server Version 2.2 documentation. https://httpd.apache.org/docs/2.2/en/logs.html, feb 2017.

[16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. **Feature-Oriented Software Product Lines**. Springer, 2013.

[17] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. Detecting Dependences and Interactions in Feature-Oriented Design. In **2010 IEEE 21st International Symposium on Software Reliability Engineering**, pages 161–170. IEEE, nov 2010.

[18] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Paolo Vavassori. A novel use of equivalent mutants for static anomaly detection in software artifacts. **Information and Software Technology**, 81:52–64, jan 2017.

[19] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Generating Tests for Detecting Faults in Feature Models. In **2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**, pages 1–10, Graz, Austria, apr 2015. IEEE.

[20] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automatic Detection and Removal of Conformance Faults in Feature Models. In **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**, pages 102–112. IEEE, apr 2016.

[21] Andrea Arcuri and Lionel Briand. Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing. **IEEE Transactions on Software Engineering**, 38(5):1088–1099, sep 2012.

[22] Patrizia Asirelli, Maurice H ter Beek, Alessandro Fantechi, and Stefania Gnesi. **A Model-Checking Tool for Families of Services**, pages 44–58. Springer, 2011.

[23] Patrizia Asirelli, Maurice H ter Beek, Alessandro Fantechi, and Stefania Gnesi. A Compositional Framework to Derive Product Line Behavioural Descriptions. In Tiziana Margaria and Bernhard Steffen, editors, **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I**, pages 146–161. Springer, 2012.

[24] Patrizia Asirelli, Maurice H ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Design and validation of variability in product lines. In **Proceeding of the 2nd international workshop on Product line approaches in software engineering - PLEASE '11**, PLEASE '11, page 25. ACM Press, 2011.

[25] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Formal Description of Variability in Product Families. In **2011 15th International Software Product Line Conference**, pages 130–139. IEEE, aug 2011.

[26] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. Decision support for the software product line domain engineering lifecycle. **Automated Software Engineering**, 19(3):335–377, 2012.

[27] Christel Baier and Joost-Pieter Katoen. **Principles of model checking**. MIT Press, 2008.

[28] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. **Model-Driven Testing: Using the UML Testing Profile**. Springer, 2007.

[29] Richard Baker and Ibrahim Habli. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. **IEEE Transactions on Software Engineering**, 39(6):787–805, jun 2013.

[30] Sebastien Bardin, Mickael Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and Quasi-Complete Detection of Infeasible Test Requirements. In **2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**, volume 7, pages 1–10, Graz, Austria, apr 2015. IEEE.

[31] J Bartholdt, M Medak, and R Oberhauser. Integrating Quality Modeling with Feature Modeling in Software Product Lines. In **2009 Fourth International Conference on Software Engineering Advances**, pages 365–370, sep 2009.

[32] Don Batory. Feature Models, Grammars, and Propositional Formulas. In **Software Product Lines**, volume 3714, pages 7–20. Springer, 2005.

[33] Don Batory and Egon Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. **Journal of Universal Computer Science**, 14(12):2059–2082, 2008.

[34] F Belli and M Beyazit. Event-Based Mutation Testing vs. State-Based Mutation Testing - An Experimental Comparison. In **Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual**, pages 650–655. IEEE, jul 2011.

[35] Fevzi Belli, Christof J Budnik, Axel Hollmann, Tugkan Tuglular, and W Eric Wong. Model-based mutation testing - Approach and case studies. **Science of Computer Programming**, 120:25–48, 2016.

[36] Fevzi Belli, Christof J Budnik, and W Eric Wong. Basic Operations for Generating Behavioral Mutants. In **Proceedings of the Second Workshop on Mutation Analysis**, MUTATION '06, pages 9—-, Washington, DC, USA, nov 2006. IEEE.

[37] Harsh Beohar and Mohammad Reza Mousavi. Input-output Conformance Testing Based on Featured Transition Systems. In **Proceedings of the 29th Annual ACM Symposium on Applied Computing**, SAC '14, pages 1272–1278. ACM Press, 2014.

[38] Harsh Beohar and M.˜R. Mousavi. Spinal Test Suites for Software Product Lines. **ArXiv e-prints**, 2014.

[39] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. **Science of Computer Programming**, jul 2015.

[40] Danilo Beuche. Modeling and building product lines with pure::variants. In **Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops**, SPLC '13 Workshops, page 147. ACM Press, 2013.

[41] P.E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In **Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering**, pages 81–88. IEEE, 2000.

[42] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. **ACM SIGPLAN Notices**, 48(1):457–468, jan 2013.

[43] Filippo Bonchi and Damien Pous. HKC Library v.˜1.0. https://perso.ens-lyon.fr/damien.pous/hknt/, 2013.

[44] Lionel C Briand and Yvan Labiche. A UML-Based Approach to System Testing. In **Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools**, pages 194–208, London, UK, UK, 2001. Springer.

[45] Dimo Brockhoff, Tobias Friedrich, and Frank Neumann. Analyzing Hypervolume Indicator Based Algorithms. In Günter Rudolph, Thomas Jansen, Nicola Beume, Simon Lucas, and Carlo Poloni, editors, **Parallel Problem Solving from Nature – PPSN X**, volume 5199 of **LNCS**, pages 651–660. Springer, 2008.

[46] Renée C Bryce and Charles J Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. **Information and Software Technology**, 48(10):960–970, 2006.

[47] Renée C Bryce and Charles J Colbourn. The density algorithm for pairwise interaction testing. **Software Testing, Verification and Reliability**, 17(3):159–182, 2007.

[48] Renée C. Bryce and Charles J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. **Software Testing, Verification and Reliability**, 19(1):37–53, mar 2009.

[49] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. **Computer Languages**, 10(1):63–73, jan 1985.

[50] Timothy Alan Budd. **Mutation Analysis of Program Test Data**. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[51] Timothy Alan Budd and Dana Angluin. Two notions of correctness and their relation to testing. **Acta Informatica**, 18(1):31–45, 1982.

[52] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. **Computer Networks**, 41(1):115–141, jan 2003.

[53] Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of LTL properties—A case study. **Formal Methods in System Design**, 28(3):213–261, may 2006.

[54] Andrea Calvagna and Angelo Gargantini. **A Logic-Based Approach to Combinatorial Testing with Constraints**, pages 66–83. Springer, 2008.

[55] Andrea Calvagna, Angelo Gargantini, and Paolo Vavassori. Combinatorial Interaction Testing with CITLAB. In **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**, pages 376–382. IEEE, mar 2013.

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

[57] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T.H. Tse. Adaptive Random Testing: The ART of test case diversity. **Journal of Systems and Software**, 83(1):60–66, jan 2010.

[58] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based Coverage-driven Test Suite Generation for Software Product Lines. In **Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems**, MODELS'11, pages 425–439. Springer, 2011.

[59] Edmund M Clarke, Orna Grumberg, and Doron Peled. **Model Checking**. MIT Press, 1999.

[60] Andreas Classen. Modelling with FTS: a Collection of Illustrative Examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[61] Andreas Classen. **Modelling and Model Checking Variability-Intensive Systems**. PhD thesis, PReCISE Research Center, Faculty of Computer Science, University of Namur (FUNDP), 2011.

[62] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. **Science of Computer Programming**, 76(12):1130–1143, dec 2011.

[63] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Jean-François Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. **IEEE Transactions on Software Engineering**, 39(8):1069–1089, aug 2013.

[64] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In José Luiz Fiadeiro and Paola Inverardi, editors, **Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'08)**, volume 4961 of **LNCS**, pages 16–30. Springer, 2008.

[65] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic Model Checking of Software Product Lines. In **33rd International Conference on Software Engineering, ICSE 2011, May 21-28, 2011, Waikiki, Honolulu, Hawaii, Proceedings**, pages 321–330. ACM Press, 2011.

[66] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In **32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings**, pages 335–344. ACM Press, 2010.

[67] D M Cohen, S R Dalal, M L Fredman, and G C Patton. The AETG system: an approach to testing based on combinatorial design. **IEEE Transactions on Software Engineering**, 23(7):437–444, 1997.

[68] M B Cohen, C J Colbourn, and A C H Ling. Augmenting simulated annealing to build interaction test suites. In **14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.**, pages 394–405. IEEE, nov 2003.

[69] M.B. Cohen, M.B. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A

Greedy Approach. **IEEE Transactions on Software Engineering**, 34(5):633–650, sep 2008.

[70] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. **Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06**, pages 53–63, 2006.

[71] J.J. Colao. With 60 Million Websites, WordPress Rules The Web. So Where's The Money? **Forbes**, September, 2012.

[72] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In **Proceedings of the 17th International Software Product Line Conference Co-located Workshops**, SPLC '13 Workshops, pages 141–146. ACM Press, 2013.

[73] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Towards an incremental automata-based approach for software product-line model checking. In **Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1**, SPLC '12, page 74. ACM Press, 2012.

[74] Maxime Cordy, Marco Willemart, Bruno Dawagne, Patrick Heymans, and Pierre-Yves Schobbens. An Extensible Platform for Product-Line Behavioural Analysis. In **Proceedings of the 18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers**, SPLat@SPLC'14, pages 102–109, Florence, Italy, 2014. ACM Press.

[75] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. **Introduction to algorithms**, volume 6. MIT press Cambridge, 2001.

[76] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. **Proceedings - 11th International Software Product Line Conference, SPLC 2007**, pages 23–32, sep 2007.

[77] Jacek Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In **Proceedings of 24th Pacific Northwest Software Quality Conference**, volume 82, 2006.

[78] Márcio Eduardo Delamaro and José Carlos Maldonado. Proteum/IM 2.0: An Integrated Mutation Testing Environment. In W Eric Wong, editor, **Mutation Testing for the New Century**, pages 91–101. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[79] Richard A DeMillo and A Jefferson Offutt. Experimental results from an automatic test case generator. **ACM Transactions on Software Engineering and Methodology**, 2(2):109–127, apr 1993.

[80] Xavier Devroey, Maxime Cordy, Gilles Perrouin, Eun-Young Kang, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Benoit Baudry. A Vision for Behavioural Model-Driven Validation of Software Product Lines. In Margaria T., Steffen B., and Merten M., editors, **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Proceedings, Part I**, volume 7609 of **LNCS**, pages 208–222, Heraklion, Crete, Greece, 2012. Springer.

[81] Xavier Devroey, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. State machine flattening, a mapping study and tools assessment. In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 1–8, Graz, Austria, apr 2015. IEEE.

[82] Xavier Devroey and Gilles Perrouin. Variability Intensive system Behavioural teSting framework (VIBeS), 2016.

[83] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, and Pierre-Yves Schobbens. A variability perspective of mutation analysis. In **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014**, pages 841–844, Hong Kong, 2014. ACM Press.

[84] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Statistical prioritization for software product line testing: an experience report. **Software & Systems Modeling**, 16(1):153–171, feb 2017.

[85] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards statistical prioritization for software product lines testing. In Andrzej Wasowski and Thorsten Weyer, editors, **Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems - VaMoS '14**, pages 1–7, Nice, France, 2013. ACM Press.

[86] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-yves Schobbens, and Patrick Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In Tiziana Margaria and Bernhard Steffen, editors, **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Proceedings, Part I**, volume 8802 of **LNCS**, pages 336–350, Corfu, Greece, 2014. Springer.

[87] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations. In **Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '15**, pages 59–66, Hildesheim, Germany, 2015. ACM Press.

[88] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based Similarity-driven Behavioural SPL Testing. In **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16**, pages 89–96, Salvador, Brazil, jan 2016. ACM Press.

[89] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In **Proceedings of the 38th International Conference on Software Engineering - ICSE '16**, pages 655–666, Austin, Texas, USA, may 2016. ACM Press.

[90] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation. In **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**, pages 424–429, Tokyo, Japan, 2017. IEEE.

[91] Xavier Devroey, Gilles Perrouin, and Pierre-Yves Schobbens. Abstract test case generation for behavioural testing of software product lines. In **Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14**, volume 2, pages 86–93, Florence, Italy, 2014. ACM Press.

[92] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On strategies for testing software product lines: A systematic literature review. **Information and Software Technology**, 56(10):1183–1199, oct 2014.

[93] Brigitte Doucet. IBA: quand la qualité logicielle devient vitale. Au sens strict du terme. **Regional-IT**, 2014.

[94] Laurent Doyen and Jean-François Raskin. Antichain Algorithms for Finite Automata. In **Tools and Algorithms for the Construction and Analysis of Systems, {TACAS}**, pages 2–22, 2010.

[95] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. **Theoretical Computer Science**, 276(1):51–81, apr 2002.

[96] W. Dulz. MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. **Third International Conference on Quality Software, 2003. Proceedings.**, pages 336–342, 2003.

[97] EMVCo. EMV 4.3 Specifications. Technical report, EMVCo, 2011.

[98] Emelie Engström and Per Runeson. Software product line testing – A systematic mapping study. **Information and Software Technology**, 53(1):2–13, jan 2011.

[99] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. **Evolutionary Search-Based Test Generation for Software Product Line Feature Models**, pages 613–628. Springer, 2012.

[100] L Etxeberria and G Sagardui. Evaluation of Quality Attribute Variability in Software Product Families. In **15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)**, pages 255–264, 2008.

[101] S.C.P.F. Fabbri, J C Maldonado, and M E Delamaro. Proteum/FSM: a tool to support finite state machine validation based on mutation testing. In **Computer Science Society, 1999. Proceedings. SCCC '99. XIX International Conference of the Chilean**, pages 96–104, 1999.

[102] S.C.P.F. Fabbri, J C Maldonado, T Sugeta, and P C Masiero. Mutation testing applied to validate specifications based on statecharts. In **Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on**, pages 210–219, 1999.

[103] Alessandro Fantechi and Stefania Gnesi. Formal Modeling for Product Families Engineering. In **2008 12th International Software Product Line Conference**, pages 193–202, Washington, DC, USA, sep 2008. IEEE.

[104] Abderrahmane Feliachi and Hélène Le Guen. Generating Transition Probabilities for Automatic Model-Based Test Generation. In **Third International Conference on Software Testing, Verification and Validation**, pages 99–102, 2010.

[105] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In **Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06**, ROSATEA '06, pages 39–48. ACM Press, 2006.

[106] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In **Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-9**, ESEC/FSE-9, page 152. ACM Press, 2001.

[107] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**, ESEC/FSE '11, pages 416–419. ACM Press, 2011.

[108] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. **Empirical Software Engineering**, 20(3):783–812, jun 2015.

[109] Froglogic. Squish GUI Tester. https://www.froglogic.com/squish/, 2016.

[110] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. **Computers & Security**, 28(1):18–28, 2009.

[111] B J Garvin, M B Cohen, and M B Dwyer. An Improved Meta-heuristic Search for Constrained Interaction Testing. In **2009 1st International Symposium on Search Based Software Engineering**, pages 13–22. IEEE, 2009.

[112] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An Improved Meta-heuristic Search for Constrained Interaction Testing. **2009 1st International Symposium on Search Based Software Engineering**, pages 13–22, may 2009.

[113] BradyJ. Garvin, MyraB. Cohen, and MatthewB. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. **Empirical Software Engineering**, 16(1):61–102, 2011.

[114] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In **Proceedings of the 36th International Conference on Software Engineering**, ICSE '14, pages 277–287, Hyderabad, India, 2014. ACM Press.

[115] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In **Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013**, page 302. ACM Press, 2013.

[116] Ali Gondal, Michael Poppleton, and Michael Butler. **Composing Event-B Specifications - Case-Study Experience**, pages 100–115. Springer, 2011.

[117] S.-D. Gouraud, A Denise, M.-C. Gaudel, and B Marre. A new way of automating statistical testing methods. In **Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)**, pages 5–12. IEEE, nov 2001.

[118] Maria Fernanda Granda, Nelly Condori-Fernandez, Tanja E.J. Vos, and Óscar Pastor. Mutation Operators for UML Class Diagrams. **CAiSE 2016**, 3:325–341, 2016.

[119] J Greenyer, A M Sharifloo, M Cordy, and P Heymans. Efficient consistency checking of scenario-based product-line specifications. In **2012 20th IEEE International Requirements Engineering Conference (RE)**, pages 161–170, sep 2012.

[120] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. **Software Testing, Verification and Reliability**, 15(3):167–199, sep 2005.

[121] Iris Groher and Markus Voelter. Aspect-oriented model-driven software product line engineering. In Shmuel Katz, Harold Ossher, Robert France, and Jean-Marc Jézéquel, editors, **Transactions on Aspect-Oriented Software Development VI**, volume 5560 of **LNCS**, pages 111–152. Springer, 2009.

[122] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. **Modeling and Model Checking Software Product Lines**, pages 113–131. Springer, 2008.

[123] J Guerrero-Garcia, J M Gonzalez-Calleros, J Vanderdonckt, and J Munoz-Arteaga. A Theoretical Survey of User Interface Description Languages: Preliminary Results. In **2009 Latin American Web Congress**, pages 36–43. IEEE, nov 2009.

[124] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**, pages 301–311. IEEE, nov 2013.

[125] Dan Gusfield. **Algorithms on strings, trees and sequences: computer science and computational biology**. Cambridge university press, 1997.

[126] Axel Halin and Alexandre Nuttinck. **Sampling & Testing all configurations: The JHipster Case Study**. master thesis, University of Namur, 2017.

[127] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. Yo variability! JHipster: A Playground for Web-Apps Analyses. In **Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems - VAMOS '17**, pages 44–51, Eindhoven, Netherlands, feb 2017. ACM Press.

[128] Mark Harman, Yue Jia, Pedro Reales Mateo, and Macario Polo. Angels and monsters. In **Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14**, pages 397–408, Vasteras, Sweden, 2014. ACM Press.

[129] Alan Hartman. **Software and Hardware Testing Using Combinatorial Covering Suites**, pages 237–266. Springer US, Boston, MA, 2005.

[130] H Hemmati and L Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In **Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on**, pages 141–150, nov 2010.

[131] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. **ACM Transactions on Software Engineering and Methodology**, 22(1):1–42, feb 2013.

[132] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining Multi-Objective Search and Constraint Solving for Configuring Large

Software Product Lines. In **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**, pages 517–528. IEEE, may 2015.

[133] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In Claire Le Goues and Shin Yoo, editors, **Search-Based Software Engineering**, volume 8636 of **LNCS**, pages 92–106. Springer, 2014.

[134] Christopher Henard, Mike Papadakis, and Yves Le Traon. Flattening or not of the combinatorial interaction testing models? In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 1–4. IEEE, apr 2015.

[135] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. **IEEE Transactions on Software Engineering**, 40(7):650–670, jul 2014.

[136] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Y Le Traon, and Yves Le Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. **Software Testing Verification and Validation Workshop, IEEE International Conference on**, 0:188–197, 2013.

[137] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective Test Generation for Software Product Lines. In **Proceedings of the 17th International Software Product Line Conference**, SPLC '13, pages 62–71. ACM Press, 2013.

[138] Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. A bibliometric analysis of 20 years of research on software product lines. **Information and Software Technology**, 72:1–15, apr 2016.

[139] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate Probabilistic Model Checking. In Bernhard Steffen and Giorgio Levi, editors, **Verification, Model Checking, and Abstract Interpretation**, volume 2937 of **LNCS**, pages 73–84. Springer, 2004.

[140] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In **2011 IEEE 22nd International Symposium on Software Reliability Engineering**, number i, pages 120–129. IEEE, nov 2011.

[141] André Heuer, Vanessa Stricker, Christof J. Budnik, Sascha Konrad, Kim Lauenroth, and Klaus Pohl. Defining variability in activity diagrams and Petri nets. **Science of Computer Programming**, 78(12):2414–2432, dec 2013.

[142] Patrick Heymans. Formal Methods for the Masses. In **Proceedings of the 16th International Software Product Line Conference - Volume 1**, SPLC '12, page 4, Salvador, Brazil, 2012. ACM Press.

[143] Robert M Hierons, Mark Harman, and Sebastian Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. **Software Testing, Verification and Reliability**, 9(4):233–262, 1999.

[144] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. **Journal of Systems and Software**, 82(11):1804–1818, nov 2009.

[145] W.E. Howden. Weak Mutation Testing and Completeness of Test Sets. **IEEE Transactions on Software Engineering**, SE-8(4):371–379, jul 1982.

[146] William E Howden. Reliability of the Path Analysis Testing Strategy. **IEEE Transactions on Software Engineering**, 2(3):208–215, 1976.

[147] IEEE. Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. jan 2014.

[148] Laura Inozemtseva and Reid Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In **Proceedings of the 36th International Conference on Software Engineering**, ICSE 2014, pages 435–445. ACM Press, 2014.

[149] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. **Bulletin del la Société Vaudoise des Sciences Naturelles**, 37:547–579, 1901.

[150] David Jackson and Martin R Woodward. Parallel Firm Mutation of Java Programs. In W Eric Wong, editor, **Mutation Testing for the New Century**, pages 55–61. Springer, Boston, MA, 2001.

[151] Y Jia and M Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In **2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation**, pages 249–258. IEEE, sep 2008.

[152] Yue Jia and M Harman. An Analysis and Survey of the Development of Mutation Testing. **Software Engineering, IEEE Transactions on**, 37(5):649–678, sep 2011.

[153] Elisabeth Jöbstl. **Model-Based Mutation Testing with Constraint and SMT Solvers**. Ph.d. thesis, Graz University of Technology, 2014.

[154] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, **Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings**, number Section 3, pages 638–652. Springer, 2011.

[155] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In **Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1**, volume 1 of **SPLC '12**, page 46. ACM Press, 2012.

[156] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In **Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems**, MODELS'12, pages 269–284. Springer, 2012.

[157] René Just, Michael D Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In **Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014**, pages 315–326. ACM Press, 2014.

[158] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are Mutants a Valid Substitute for Real Faults in Software Testing? In **Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**, FSE 2014, pages 654–665. ACM Press, 2014.

[159] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A S Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University, Software Engineering Institute, 1990.

[160] Kalpesh Kapoor and Jonathan P Bowen. Ordering Mutants to Minimise Test Effort in Mutation Testing. In **{FATES} 2004**, pages 195–209, 2005.

[161] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-aware Testing. In **Proceedings of the 4th International Workshop on Feature-Oriented Software Development**, FOSD '12, pages 1–8. ACM Press, 2012.

[162] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared Execution for Efficiently Testing Product Lines. In **2012 IEEE 23rd International Symposium on Software Reliability Engineering**, pages 221–230, Dallas,Texas, nov 2012. IEEE.

[163] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D'Amorim. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013**, ESEC/FSE 2013, page 257. ACM Press, 2013.

[164] Marinos Kintis and Nicos Malevris. {MEDIC:} {A} static analysis framework for equivalent mutant identification. **Information {&} Software Technology**, 68:1–17, 2015.

[165] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Employing second-order mutation for isolating first-order equivalent mutants. **Software Testing, Verification and Reliability**, 25(5-7):508–535, aug 2015.

[166] Anneke G Kleppe, Jos B Warmer, and Wim Bast. **MDA explained: the model driven architecture: practice and promise**. Addison-Wesley Professional, 2003.

[167] Alexander Knapp, Markus Roggenbach, and Bernd-Holger Schlingloff. On the Use of Test Cases in Model-based Software Product Line Development. In **Proceedings of the 18th International Software Product Line Conference - Volume 1**, SPLC '14, pages 247–251. ACM Press, 2014.

[168] Willibald Krenn and Rupert Schlick. Mutation-driven Test Case Generation Using Short-lived Concurrent Mutants – First Results. **CoRR**, abs/1601.0:19, jan 2016.

[169] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard Aichernig, Elisabeth Jobstl, and Harald Brandl. MoMut::UML Model-Based Mutation Testing for UML. In **Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on**, pages 1–8, 2015.

[170] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In **Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering - SIGSOFT '04/FSE-12**, SIGSOFT '04/FSE-12, page 137. ACM Press, 2004.

[171] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. Software fault interactions and implications for software testing. **IEEE Transactions on Software Engineering**, 30(6):418–421, jun 2004.

[172] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical Combinatorial Testing: Beyond Pairwise. **IT Professional**, 10(3):19–23, may 2008.

[173] Orna Kupferman and Moshe Y Vardi. Verification of fair transition systems. In **Computer Aided Verification**, pages 372–382. Springer, 1996.

[174] Hartmut Lackner and Martin Schmidt. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. In **Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2**, SPLC '14, pages 62–69. ACM Press, 2014.

[175] Beatriz Pérez Lamancha, Macario Polo, and Mario Piattini. PROW: A Pairwise algorithm with constRaints, Order and Weight. **Journal of Systems and Software**, 99:1–19, 2015.

[176] William B Langdon and Mark Harman. Optimizing Existing Software With Genetic Programming. **IEEE Transactions on Evolutionary Computation**, 19(1):118–135, feb 2015.

[177] William B Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. **Journal of Systems and Software**, 83(12):2416–2430, 2010.

[178] Kim G Larsen, Ulrik Nyman, and Andrzej Wąsowski. **Modal I/O Automata for Interface and Product Line Theories**, pages 64–79. Springer, 2007.

[179] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality Assurance in the Presence of Variability. In Selmin Nurcan, Camille Salinesi, Carine Souveyet, and Jolita Ralyté, editors, **Intentional Perspectives on Information Systems Engineering**, pages 319–333. Springer, 2010.

[180] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In **2009 IEEE/ACM International Conference on Automated Software Engineering**, pages 269–280. IEEE, nov 2009.

[181] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. **IEEE Transactions on Software Engineering**, 38(1):54–72, jan 2012.

[182] Axel Legay, Gilles Perrouin, Xavier Devroey, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. On Featured Transition Systems. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, **SOFSEM 2017: Theory and Practice of Computer Science**, volume 10139 of **LNCS**, pages 453–463, Limerick, Ireland, 2017. Springer.

[183] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A General Strategy for T-Way Software Testing. In **14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)**, pages 549–556. IEEE, mar 2007.

[184] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. **Software Testing, Verification and Reliability**, 18(3):125–148, 2008.

[185] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool Support for Feature-oriented Software Development: FeatureIDE: an Eclipse-based Approach. In **Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange**, eclipse '05, pages 55–59. ACM Press, 2005.

[186] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In **Soviet physics doklady**, volume 10, pages 707–710, 1966.

[187] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying Cross-cutting Features As Open Systems. **SIGSOFT Softw. Eng. Notes**, 27(6):89–98, 2002.

[188] Harry C Li, Shriram Krishnamurthi, and Kathi Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. **Automated Software Engineering**, 12(3):349–382, jul 2005.

[189] H.C. Li, S Krishnamurthi, and K Fisler. Interfaces for modular feature verification. In **Proceedings 17th IEEE International Conference on Automated Software Engineering,**, pages 195–204. IEEE, 2002.

[190] Nan Li, Anthony Escalona, and Tariq Kamal. Skyfire: Model-Based Testing with Cucumber. In **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**, pages 393–400. IEEE, apr 2016.

[191] Nan Li and Jeff Offutt. A test automation language framework for behavioral models. In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, number 1, pages 1–10. IEEE, apr 2015.

[192] Xuelin Li, W Eric Wong, Ruizhi Gao, Linghuan Hu, and Shigeru Hosono. Genetic Algorithm-based Test Generation for Software Product Line with the Integration of Fault Localization Techniques. **Empirical Software Engineering**, pages 1–51, 2017.

[193] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. SAT-based analysis of large real-world feature models is easy. In **Proceedings of the 19th International Conference on Software Product Line - SPLC '15**, pages 91–100. ACM Press, jul 2015.

[194] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. **USIXML: A Language Supporting Multi-path Development of User Interfaces**, pages 200–220. International Workshop on Design, Specification, and Verification of Interactive Systems. Springer, Hamburg, Germany, 2005.

[195] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based integration testing of large-scale systems. **Journal of Systems and Software**, 91:63–84, may 2014.

[196] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. **Software Quality Journal**, 20(3-4):567–604, sep 2012.

[197] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In AchimD. Brucker and Jacques Julliand, editors, **Tests and Proofs**, volume 7305 of **LNCS**, pages 67–82. Springer, 2012.

[198] Roberto E. Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In **2013 IEEE International Conference on Software Maintenance**, pages 404–407. IEEE, sep 2013.

[199] Roberto E Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 1–10. IEEE, apr 2015.

[200] C Lott, A Jain, and S Dalal. Modeling Requirements for Combinatorial Software Testing. In **Proceedings of the 1st International Workshop on Advances in Model-based Testing**, A-MOST '05, pages 1–7. ACM Press, 2005.

[201] Cucumber ltd. Cucumber. https://cucumber.io, 2016.

[202] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. **Software Testing, Verification and Reliability**, 15(2):97–133, jun 2005.

[203] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. **IEEE Transactions on Software Engineering**, 40(1):23–42, jan 2014.

[204] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In **Proceedings of the 17th International Software Product Line Conference on - SPLC '13**, page 227. ACM Press, aug 2013.

[205] Pedro Reales Mateo and Macario Polo Usaola. Reducing mutation costs through uncovered mutants. **Software Testing, Verification and Reliability**, 25(5-7):464–489, aug 2015.

[206] Aditya P. Mathur. **Foundations of software testing**. Pearson Education, India, 2008.

[207] Rui A Matnei Filho and Silvia R Vergilio. A multi-objective test data generation approach for mutation testing of feature models. **Journal of Software Engineering Research and Development**, 4(1):4, dec 2016.

[208] John D McGregor. Toward a Fault Model for Software Product Lines. In Steffen Thiel and Klaus Pohl, editors, **Software Product Lines, 12th International Conference, {SPLC} 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)**, pages 157–162. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[209] M D Mcilroy. Mass Produced Software Components. Technical report, 1969.

[210] Marcilio Mendonça. SPLAR. https://code.google.com/archive/p/splar/, 2010.

[211] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In **Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications**, OOPSLA '09, pages 761–762. ACM Press, 2009.

[212] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. **Proceedings of the 7th international conference on Generative programming and component engineering - GPCE '08**, page 13, 2008.

[213] Andreas Metzger and Klaus Pohl. Software Product Line Engineering and Variability Management: Achievements and Challenges. In **Proceedings of the on Future of Software Engineering**, FOSE 2014, pages 70–84. ACM Press, 2014.

[214] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In **Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems**, VaMoS '11, pages 82–89. ACM Press, 2011.

[215] Microsoft. Inspect. https://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx, 2016.

[216] Jean-Vivien Millo, S Ramesh, Shankara Narayanan Krishna, and Ganesh Khandu Narwane. Compositional Verification of Evolving Software Product Lines. **CoRR**, abs/1212.4, 2012.

[217] R Milner. **A Calculus of Communicating Systems**. Springer, Secaucus, NJ, USA, 1982.

[218] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. In **2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**, ICST '15, pages 1–10. IEEE, apr 2015.

[219] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation Analysis Testing for Model Transformations. In **ECMDA-FA**, pages 376–390, 2006.

[220] John D Musa, Gene Fuoco, Nancy Irving, Diane Kropfl, and Bruce Juhlin. The operational profile. **NATO ASI series F Comp. and Syst. Sc.**, 154:333–344, 1996.

[221] Radu Muschevici, Dave Clarke, and Jose Proenca. Feature Petri Nets. In Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood, editors, **Proceedings of the 14th International Software Product Line Conference (SPLC 2010)**, volume 2. Lancaster University, 2010.

[222] Peter Naur and Brian Randell, editors. **Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO**. 1969.

[223] Torsten Nelson, Donald Cowan, and Paulo Alencar. **Supporting Formal Verification of Crosscutting Concerns**, pages 153–169. Springer, 2001.

[224] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In **Proceedings of the 36th International Conference on Software Engineering - ICSE 2014**, ICSE '14, pages 907–918. ACM Press, 2014.

[225] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. Feature interaction: The security threat from within software systems. **Progress in Informatics**, 5:75–89, 2008.

[226] Changhai Nie and Hareton Leung. A survey of combinatorial testing. **ACM Computing Surveys**, 43(2):1–29, jan 2011.

[227] N Niu and S Easterbrook. Extracting and Modeling Product Line Functional Requirements. In **2008 16th IEEE International Requirements Engineering Conference**, pages 155–164. IEEE, sep 2008.

[228] A Jefferson Offutt and W M Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. **Software Testing, Verification and Reliability**, 4(3):131–154, 1994.

[229] A Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. **Software Testing, Verification and Reliability**, 7(3):165–192, sep 1997.

[230] Jeff Offutt. A mutation carol: Past, present and future. **Information and Software Technology**, 53(10):1098–1107, oct 2011.

[231] Rafael Olaechea, Uli Fahrenberg, Joanne M Atlee, and Axel Legay. Long-term Average Cost in Featured Transition Systems. In **Proceedings of the 20th International Systems and Software Product Line Conference**, SPLC '16, pages 109–118, Nashville, Tennessee, USA, 2016. ACM Press.

[232] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. Comparison of exact and approximate multi-objective optimization for software product lines. In **Proceedings of the 18th International Software Product Line Conference - Volume 1**, SPLC '14, pages 92–101. ACM Press, 2014.

[233] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In Jan Bosch and Jaejoon Lee, editors, **Proceedings of the 14th International Software Product Lines Conference: Going Beyond**, SPLC'10, pages 196–210, Jeju Island, South Korea, 2010. Springer.

[234] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. A survey of model-based software product lines testing. In Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman, editors, **Model-Based Testing for Embedded Systems**, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 338–381. CRC Press, 2011.

[235] M Papadakis, C Henard, and Y le Traon. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In **Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on**, pages 1–10, 2014.

[236] Mike Papadakis, Márcio Eduardo Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. **Science of Computer Programming**, 95:298–319, 2014.

[237] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**, pages 936–946. IEEE, may 2015.

[238] Mike Papadakis and Yves Le Traon. Metallaxis-FL: mutation-based fault localization. **Software Testing, Verification and Reliability**, 25(5-7):605–628, aug 2015.

[239] Mike Papadakis and Nicos Malevris. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In **2010 IEEE 21st International Symposium on Software Reliability Engineering**, pages 121–130. IEEE, nov 2010.

[240] Mike Papadakis and Nicos Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. **Software Quality Journal**, 19(4):691–723, dec 2011.

[241] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. **Information and Software Technology**, 54(9):915–932, sep 2012.

[242] José A. Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. Multi-objective test case prioritization in highly configurable systems: A case study. **Journal of Systems and Software**, 122:287–310, 2016.

[243] Krishna Patel and Robert M Hierons. **Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness**, pages 123–138. Springer, Cham, 2016.

[244] S Patel, P Gupta, and V Shah. Combinatorial Interaction Testing with Multi-perspective Feature Models. In **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops**, pages 321–330. IEEE, 2013.

[245] Sachin Patel, Priya Gupta, and Vipul Shah. Feature interaction testing of variability intensive systems. **2013 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE)**, pages 53–56, may 2013.

[246] Radek Pelánek. Typical Structural Properties of State Spaces. In Susanne Graf and Laurent Mounier, editors, **Model Checking Software**, volume 2989 of **LNCS**, pages 5–22. Springer, 2004.

[247] Radek Pelánek. Model Classifications and Automated Verification. In Stefan Leue and Pedro Merino, editors, **Formal Methods for Industrial Critical Systems**, volume 4916 of **LNCS**, pages 149–163. Springer, 2008.

[248] Radek Pelánek. Properties of state spaces and their applications. **International Journal on Software Tools for Technology Transfer**, 10(5):443–454, 2008.

[249] Beatriz Pérez Lamancha and Macario Polo Usaola. Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, **Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings**, pages 111–125. Springer, 2010.

[250] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Pairwise testing for software product lines: Comparison of two approaches. **Software Quality Journal**, 20(3-4):605–643, 2011.

[251] Justyna Petke, Shin Yoo, Myra B Cohen, and Mark Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013**, ESEC/FSE 2013, page 26. ACM Press, 2013.

[252] K Pohl, G Böckle, and F Van Der Linden. **Software product line engineering: foundations, principles, and techniques**. Springer, 2005.

[253] Macario Polo, Mario Piattini, and Ignacio Garc\'\ia-Rodr\'\iguez. Decreasing the Cost of Mutation Testing with Second-order Mutants. **Software Testing Verification and Reliability**, 19(2):111–131, 2009.

[254] Michael R Poppleton. **Towards Feature-Oriented Specification and Development with Event-B**, pages 367–381. Springer, 2007.

[255] S J Prowell. JUMBL: a tool for model-based statistical testing. In **System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on**, pages 9 pp.–, jan 2003.

[256] Georg Püschel, Ronny Seiger, and Thomas Schlegel. Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. **Modiquitous Workshop**, 2012.

[257] QSpin. QTaste. https://github.com/qspin/qtaste (version 2.3.0), 2016.

[258] RaiMan. Sikuli Script. http://www.sikuli.org, 2016.

[259] Awais Rashid, Jean-Claude Royer, and Andreas Rummler. **Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way**. Cambridge University Press, 2011.

[260] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10**, volume 1 of **ICSE '10**, page 445. ACM Press, 2010.

[261] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-based product-line testing: effective sample generation based on feature-diagram mutation. In **Proceedings of the 19th International Conference on Software Product Line - SPLC '15**, pages 131–140. ACM Press, 2015.

[262] G N Rodrigues, V Alves, V Nunes, A Lanna, M Cordy, P Y Schobbens, A M Sharifloo, and A Legay. Modeling and Verification for Probabilistic Properties in Software Product Lines. In **2015 IEEE 16th International Symposium on High Assurance Systems Engineering**, pages 173–180, jan 2015.

[263] Kenneth Rosen. **Discrete Mathematics and Its Applications, 7th edition**. McGraw-Hill Science, 2011.

[264] Hamideh Sabouri and Ramtin Khosravi. Efficient Verification of Evolving Software Product Lines. In **Fundamentals of Software Engineering SE - 24**, volume 7141, pages 351–358. 2012.

[265] Sahipro. Sahi: The Tester's Web Automation Tool. http://sahipro.com, 2016.

[266] Hamza Samih, Mathieu Acher, Ralf Bogusch, Hélène Le Guen, and Benoit Baudry. Deriving Usage Model Variants for Model-based Testing: An Industrial Case Study. In IEEE, editor, **2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS 2014)**, Tianjin, Chine, 2014.

[267] Hamza Samih and Ralf Bogusch. MPLM - MaTeLo Product Line Manager. In **Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2**, SPLC '14, pages 138–142. ACM Press, 2014.

[268] Hamza Samih, Hélène Le Guen, Ralf Bogusch, Mathieu Acher, and Benoit Baudry. An Approach to Derive Usage Models Variants for Model-based Testing. In **The 26th IFIP International Conference on Testing Software and Systems (2014)**, Madrid, Espagne, 2014. Springer.

[269] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and a. Gunes Koru. Prioritizing User-Session-Based Test Cases for Web Applications Testing. In **Software Testing, Verification, and Validation, 2008 1st International Conference on**, pages 141–150. IEEE, apr 2008.

[270] Ana B Sánchez, Sergio Segura, José A Parejo, and Antonio Ruiz-Cortés. Variability testing in the wild: the Drupal case study. **Software & Systems Modeling**, 16(1):173–194, feb 2017.

[271] Ana B Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. The Drupal Framework: A Case Study to Evaluate Variability Testing Techniques. In **Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems**, VaMoS '14, pages 11:1—-11:8. ACM Press, 2013.

[272] Ana B. Sanchez, Sergio Segura, and Antonio Ruiz-Cortes. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation**, number July in ICST '14, pages 41–50. IEEE, mar 2014.

[273] David Sankoff and Joseph B Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. **Addison-Wesley Readings**, 1983.

[274] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In **2013 35th International Conference on Software Engineering (ICSE)**, pages 492–501. IEEE, may 2013.

[275] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines, 2010.

[276] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. **Computer Networks**, 51(2):456–479, 2007.

[277] David Schuler and Andreas Zeller. Covering and Uncovering Equivalent Mutants. **Software Testing, Verification and Reliability**, 23(5):353–374, 2013.

[278] Sergio Segura, Ana B Sánchez, and Antonio Ruiz-Cortés. Automated Variability Analysis and Testing of an E-commerce Site.: An Experience Report. In **Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering**, ASE '14, pages 139–150. ACM Press, 2014.

[279] Selenium. SeleniumHQ Web Browser Automation. http://www.seleniumhq.org, 2016.

[280] Smartesting Solutions & Services. Smartesting. http://www.smartesting.com, 2016.

[281] Pourya Shaker and Joanne M Atlee. A Featur-Oriented Requirements Modelling Language (FORML). Technical report, Waterloo, Canada, 2012.

[282] Pourya Shaker and Joanne M Atlee. Behaviour Interactions Among Product-line Features. In **Proceedings of the 18th International Software Product Line Conference**, SPLC '14, pages 242–246, Florence, Italy, 2014. ACM Press.

[283] Pourya Shaker, Joanne M Atlee, and Shige Wang. A Feature-Oriented Requirements Modelling Language. In **20th IEEE International Requirements Engineering Conference**, pages 151–160, Chicago, Illinois, USA, 2012. IEEE.

[284] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In **Proceeding of the 33rd international conference on Software engineering - ICSE '11**, page 461. ACM Press, 2011.

[285] George Sherwood. Effective testing of factor combinations. In **Proc. Third International Conference on Software Testing, Analysis and Review (STAR'94)**, 1994.

[286] Sharon Shoham and Orna Grumberg. Multi-valued model checking games. **Journal of Computer and System Sciences**, 78(2):414–429, 2012.

[287] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015**, pages 284–294. ACM Press, aug 2015.

[288] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. **Information and Software Technology**, 55(3):491–507, mar 2013.

[289] Sonatype. Maven by Example. page 151, 2011.

[290] Jennifer Sorge, Michael Poppleton, and Michael Butler. A Basis for feature-oriented modelling in Event-B. 2009.

[291] Sara Sprenkle, Lori Pollock, and Lucy Simko. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In **Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on**, pages 230–239. IEEE, mar 2011.

[292] Sara E Sprenkle, Lori L Pollock, and Lucy M Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. **Software Testing, Verification and Reliability**, 23(6):439–464, 2013.

[293] Michaela Steffens, Sebastian Oster, Malte Lochau, and Thomas Fogdal. Industrial Evaluation of Pairwise SPL Testing with MoSo-PoLiTe. In **Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems**, VaMoS '12, pages 55–62. ACM Press, 2012.

[294] Daniel Strüber, Julia Rubin, Marsha Chechik, and Gabriele Taentzer. A Variability-Based Approach to Reusable and Efficient Model Transformations.

In **Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE '15)**, volume 9033, pages 283–298, New York, NY, USA, 2015. Springer.

[295] SwingInspector. SwingInspector. http://www.swinginspector.com/index_en.htm, 2016.

[296] M H ter Beek, A Legay, A Lluch Lafuente, and A Vandin. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In **Proceedings of the 19th International Conference on Software Product Line**, SPLC '15, pages 11–15, Nashville, Tennessee, USA, 2015. ACM Press.

[297] Maurice H ter Beek, Alberto Lluch Lafuente, and Marinella Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In **Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops**, SPLC '13 Workshops, page 10. ACM Press, 2013.

[298] Maurice H ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. **Statistical Model Checking for Product Lines**, pages 114–133. Springer, Corfu, Greece, 2016.

[299] Maurice H ter Beek, Henry Muccini, and Patrizio Pelliccione. Guaranteeing Correct Evolution of Software Product Lines: Setting Up the Problem. In Elena A Troubitsyna, editor, **Software Engineering for Resilient Systems: Third International Workshop, SERENE 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings**, number January, pages 100–105. Springer, 2011.

[300] Pascale Thévenod-Fosse and Hélène Waeselynck. An Investigation of Statistical Software Testing. **Softw. Test., Verif. Reliab.**, 1(2):5–25, 1991.

[301] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. **ACM Computing Surveys**, 47(1):1–45, jun 2014.

[302] Jan Tretmans. Model-based testing: Property checking for real. In **Keynote address at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices**, 2004.

[303] Jan Tretmans. Model based testing with labelled transition systems. **Formal methods and testing**, pages 1–38, 2008.

[304] Jan Tretmans. Model-Based Testing and Some Steps towards Test-Based Modelling. In Marco Bernardo and Valérie Issarny, editors, **Formal Methods for Eternal Networked Software Systems**, volume 6659 of **LNCS**, chapter 9, pages 297–326. Springer, 2011.

[305] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. Fama framework. In **12th International Software Product Line Conference**, page 359. IEEE, 2008.

[306] Roland H Untch, A Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In **Proceedings of the 1993 international symposium on Software testing and analysis - ISSTA '93**, pages 139–148. ACM Press, 1993.

[307] Mark Utting and Bruno Legeard. **Practical Model-Based Testing: A Tools Approach**. Morgan Kaufmann, 2007.

[308] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. (April 2011):297–312, 2012.

[309] Engin Uzuncaova, Sarfraz Khurshid, and Don Batory. Incremental Test Generation for Software Product Lines. **Software Engineering, IEEE Transactions on**, 36(3):309–322, 2010.

[310] Axel Van Lamsweerde. **Systematic Requirements Engineering - From System Goals to UML Models to Software Specifications**. Wiley, 2008.

[311] Jeremy Vanhecke. **Concrétisation de tests abstraits avec AbsCon, un AddOn QTaste**. Master thesis, University of Namur, 2016.

[312] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based Testing of Object-oriented Reactive Systems with Spec Explorer. In Robert M Hierons, Jonathan P Bowen, and Mark Harman, editors, **Formal methods and testing**, chapter Model-base, pages 39–76. Springer, 2008.

[313] A Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. **IEEE Transactions on Information Theory**, 13(2):260–269, 1967.

[314] Jeffrey M Voas and Gary McGraw. **Software Fault Injection: Inoculating Programs Against Errors**. John Wiley & Sons, Inc., 1997.

[315] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA Model: On the Combination of Product-line Analyses. In **Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems**, VaMoS '13, pages 14:1—-14:8. ACM Press, 2013.

[316] J.A. James A Whittaker, G Thomason Michael, and Michael G M.G. Thomason. A Markov chain model for statistical software testing. **IEEE Transactions on Software Engineering**, 20(10):812–824, oct 1994.

[317] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. **Experimentation in Software Engineering**. Springer, 2012.

[318] M.R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. **Software Engineering Journal**, 8(4):211, 1993.

[319] WordPress.org. WordPress. https://wordpress.org, apr 2010.

[320] P Zave. Feature interactions and formal specifications in telecommunications. **Computer**, 26(8):20–28, 1993.

GLOSSARY

AbsCon Abstract test case Concretizer. xxi, xxiii, 25, 26, 133, 136, 146

abstract test case A finite sequence of actions in a FTS such that there exists a sequence of transitions in this FTS, labelled with the corresponding actions. xxi, 43, 44, 133, 136

AGE *Assemblée Générale des Étudiants*. 34

ALE Automata Language Equivalence. xxii, 80–82, 88, 111, 167

API Application Programming Interface. xxiii, 124–126

BDD Binary Decision Diagram. 5, 27

BS Biased Simulation. xxii, 82–84, 88, 111, 167

bug The result, during the execution of a system, of an error made by the programmer. If it propagates to the output of the system, it causes a failure. 13

CCS Calculus of Communicating Systems. 10

CIT Combinatorial Interaction Testing. v, xx, xxi, 19, 20, 54

CMS Content Management System. xxiii, 33

CNF Conjunctive Normal Form. 5, 27, 45, 50, 128, 152

concretization Process used to transform abstract test cases into executable test scripts. 25, 134, 136, 145

coverage criterion Criterion defined over a model or a piece of source code and used to drive a test case selection process. 47

DTMC Discrete-Time Markov Chain. xxii, 32, 61, 205, **see:** usage model

EMP Equivalent Mutants Problem. xxii, 80, 88

error Mistake made by a programmer during the writing of the source code of a system. 13

failure Result (at runtime) of the propagation of a bug to the output of a system. 13

fault Synonym for bug see. 13

feature expression A boolean expression over features. 9, 127, 154

feature model A model used to represent all the valid products of a product line [159]. Usually, feature models are represented using a tree structure where features are decomposed into sub-features. xvii, xix, xx, 5, 8, 11, 17, 18, 28, 29, 70, 73, 128, 149, 151, 204

featured mutant model A compact formalism to represent mutants of a system as a family (*i.e.,* a product line). v, vii, xxii, 70, 73, 87, 150, 204

featured transition system A compact formalism used to represent the behaviour of a software product line [63]. An FTS is a LTS where transitions are tagged with feature expressions specifying which products may fire the transition. v, vii, xv, xxi, 8, 9, 43, 204

FM Feature Model. xvii, xix, 5, 38, 41, **see:** feature model

FMM Featured Mutants Model. v, vii, xxii, xxiii, 70, 73, 74, 87–89, 107, 130–132, 150, 154, **see:** featured mutant model

FODA Feature Oriented Domain Analysis. 5

FTS Featured Transition System. v, vii, xvii, xxi, xxiii, 7–11, 17, 18, 20, 25, 27–29, 40, 42, 43, 49, 68, 70, 73, 74, 124–126, 128–130, 149, 152, **see:** featured transition system

labelled transition system Formalism used to represent the behaviour of a system as a set of states and transitions labelled with actions. xv, 7, 43, 204

LTS Labelled Transition System. xv, 7–10, 17, 70, 71, 124, 130, **see:** labelled transition system

MBT Model-Based Testing. 43

MHML Modal Hennessy-Milner Logic. 10

MTS Modal Transition System. 7, 10

negative abstract test case Abstract test case that cannot be executed on the FTS of the product line. 44, 45

PIN Personal Identification Number. 8, 9, 30

PL-CCS Product Line CCS. 10

POM Project Object Model. 124

positive abstract test case Abstract test case that can be executed on the FTS of the product line. 44, 45

QTaste QSpin Tailored Automated System Test Environment. xxi, 25, 26, 28, 134

RS Random Simulation. xxii, 82, 83, 88, 111, 167

SAT Boolean Satisfiability Problem. 5, 9, 10, 27

SM Strong Mutation. 81, 83, 84, 167

soda vending machine A case study representing a beverage vending machine product line that sells soda and/or tea (see Section 4.1 for the complete description). 29, 44, 47, 126

SPL Software Product Line. v, xv, xix, xx, 3–7, 10, 11, 13, 16, 18, 20, 27, 43, 86, 149, 155

SUT System Under Test. 13, 43–45, 133, 134

test suite A set of test cases, selected in order to satisfy a given criterion. 14, 149

TVL Text-based Variability Language. 5, 131

UIDL User Interface Description Language. 144

UM Usage Model. 61, **see:** usage model

usage model Model representing the usage of a system as a Discrete-Time Markov Chain (DTMC). xxii, xxiii, 61, 101, 124, 129

VIBeS Variability Intensive Behavioural teSting. v, vii, xvi, xxi, xxiii, 25–28, 123–126, 131, 132, 134, 150, 155

WM Weak Mutation. 81, 83, 84, 167