Automatic generation of configurable test-suites for software product lines

Vanderson Hafemann Fragal

Supervisors:
Mohammad Reza Mousavi
Adenilso da Silva Simao
Abstract

Software Product Line Engineering (SPLE) is an approach used in the development of similar products, which aims at systematic reuse of software artifacts. The SPLE process has several activities executed to assure software quality. Quality assurance is of vital importance for achieving and maintaining a high quality for various artifacts, such as products and processes. Testing activities are widely used in industry for quality assurance. However, the effort for applying testing is usually high, and increasing the testing efficiency is a major concern. A common means of increasing efficiency is automation of test design. Several techniques, processes, and strategies were developed for SPLE testing, but still many problems are open in this area of research. The challenge in focus is the reduction of the overall test effort required to test SPLE products. Test effort can be reduced by maximizing test reuse using models that take advantage of the similarity between products. The thesis goal is to automate the generation of small test-suites with high fault detection and low test redundancy between products. To achieve the goal, equivalent tests are identified for a set of products using complete and configurable test-suites. Two research directions are explored, one is product-based centered, and the other is product line-centered. For test design, test-suites that have full fault coverage were generated from state machines with and without feature constraints. A prototype tool was implemented for test design automation. In addition, the proposed approach was evaluated using examples, experimental studies, and an industrial case study for the automotive domain. The results of the product-based centered approach indicate a reduction of 36% on the number of test cases that need to be concretized. The results of the product line-centered approach indicate a reduction of 50% on the number of test cases generated for groups of product configurations.
Acknowledgments

First of all, I would like to thank my advisors Prof. Mohammad Reza Mousavi and Prof. Adenilso Simao for excellent guidance. Thanks for encouraging and challenging me and also for good cooperation. I would also like to thank my other colleagues at the Department of Computer Science and especially the Software Engineering Research Group at Sao Paulo University and the Model-Based Testing Group at Halmstad University for the inspiring and developing work environment. Finally, thanks to my family, my parents, my brother, and sisters for constant encouragement.
List of Papers


# Contents

Abstract i  

Acknowledgments iii  

List of Papers v  

List of Figures xi  

I Background 1  

1 Introduction 3  
1.1 Motivation ............................................. 3  
1.2 Problem Setting ........................................ 4  
1.3 Thesis Statement ....................................... 4  
1.4 Contributions .......................................... 5  
1.5 Outline and Contribution Statement ..................... 6  

2 Preliminaries and Definitions 9  
2.1 Introduction to Software Testing ........................... 9  
2.1.1 Faults and Test-cases .................................. 10  
2.1.2 Test Process and Phases ................................. 11  
2.1.3 Regression Testing ..................................... 12  
2.1.4 Coverage Criteria ..................................... 13  
2.1.5 Test Techniques ....................................... 14  
2.1.6 Test Automation and Quality ............................ 14  
2.2 Model Based Testing ...................................... 15  
2.2.1 Model-Based Testing Process ........................... 16  
2.2.2 Test-case Concretization ............................... 17  
2.2.3 Test-case Generation for Finite State Machines ........... 18  
2.3 Software Product Lines ................................... 24  
2.3.1 Development Process .................................. 25  
2.3.2 Feature Diagram ...................................... 26
## CONTENTS

2.3.3 Feature Constraint ............................................. 27  
2.3.4 Feature Model .................................................. 28  
2.3.5 Software Product Line Testing ............................. 28  
2.4 Concluding Remarks ............................................. 33  
2.5 Summary of Contributions ..................................... 34

### II Papers

3 Reducing the Concretization Effort 39

3.1 Introduction ...................................................... 40  
3.2 Background ....................................................... 41  
  3.2.1 Software Product Lines ................................... 41  
  3.2.2 Finite State Machines ................................... 42  
  3.2.3 Test Properties ........................................... 43  
  3.2.4 Concretization Effort ................................... 44  
3.3 Testing Products Incrementally ............................... 45  
  3.3.1 Test Reuse Strategy ...................................... 45  
  3.3.2 Selection Algorithm ...................................... 46  
3.4 Experimental Study ............................................. 48  
  3.4.1 Experimental Setup ...................................... 48  
  3.4.2 Analysis of Results and Discussion ..................... 49  
3.5 Related Work ..................................................... 52  
3.6 Conclusion ....................................................... 52

4 Featured Finite State Machines 55

4.1 Introduction ...................................................... 56  
4.2 Background ....................................................... 57  
  4.2.1 Software Product Lines ................................... 57  
  4.2.2 Finite State Machine ................................... 58  
4.3 Featured Finite State Machines ............................... 59  
  4.3.1 Basic Definitions ........................................ 59  
  4.3.2 Product Derivation ...................................... 61  
  4.3.3 Validation Properties ................................... 62  
4.4 Implementation ................................................... 65  
4.5 Experimental Study ............................................. 67  
  4.5.1 Experimental Setup ...................................... 67  
  4.5.2 Analysis and Threats to Validity ......................... 68  
4.6 Related Work ..................................................... 70  
4.7 Conclusion ....................................................... 70
## 5 Extending HSI Test Generation Method

5.1 Introduction ........................................................................ 74
5.2 Background ........................................................................ 76
  5.2.1 Harmonized State Identifier Method ............................... 76
  5.2.2 Feature Model .............................................................. 78
  5.2.3 Featured Finite State Machines ..................................... 79
5.3 Configurable Test Design ..................................................... 81
  5.3.1 Configurable Test suites ............................................... 81
  5.3.2 Test case Derivation .................................................... 82
  5.3.3 State Coverage .......................................................... 82
  5.3.4 Transition Coverage .................................................. 84
  5.3.5 Full Fault Coverage ................................................... 86
5.4 Tool Support ....................................................................... 89
  5.4.1 Eclipse Platform ......................................................... 90
  5.4.2 Satisfiability Modulo Theories Solvers ......................... 90
5.5 Experimental Study ............................................................ 91
  5.5.1 Experimental Setup ................................................... 92
  5.5.2 Analysis and Threats to Validity .................................. 94
5.6 Body Comfort System Case Study .................................... 100
5.7 Related Work ................................................................... 104
5.8 Conclusions and Future Work .......................................... 106

## 6 Hierarchical Featured State Machines

6.1 Introduction ...................................................................... 108
6.2 Background ...................................................................... 110
  6.2.1 Feature Diagram ....................................................... 110
  6.2.2 Feature Constraint .................................................... 110
  6.2.3 Feature Model .......................................................... 112
6.3 Featured Finite State Machines ....................................... 112
  6.3.1 Basic Definitions ...................................................... 112
  6.3.2 Model Derivation ...................................................... 114
  6.3.3 Validation Properties ................................................ 115
6.4 Hierarchical Featured State Machine ............................... 115
  6.4.1 Syntax ..................................................................... 115
  6.4.2 Semantics .................................................................. 123
6.5 Tool Support .................................................................... 129
  6.5.1 HFSM Syntax Validation ........................................... 130
  6.5.2 Semantic Validation ................................................ 132
  6.5.3 Model Derivation ...................................................... 133
6.6 Body Comfort System Case Study ................................ 134
  6.6.1 Results .................................................................... 135
  6.6.2 Discussion of the Results ......................................... 137
6.7 Related Work ................................................................... 138
6.8 Conclusion ....................................................................... 139
CONTENTS

Bibliography 141

Glossary 153
# List of Figures

1.1 Directions of research in SPL test design. ......................... 5
1.2 Directions of research and chapters. ............................... 8

2.1 Testing terms in Software Engineering ............................ 10
2.2 The V-model (adapted from [25]). ................................ 12
2.3 Application fields of model-based testing (adapted from [110]). 15
2.4 Model-based testing overview (adapted from [85]). .............. 16
2.5 Abstract FSM $M$. .............................................. 19
2.6 Testing tree of a transition cover set. ............................ 21
2.7 Abstract FSM $M'$. ............................................. 22
2.8 Software product line development process (adopted from [83]). 26
2.9 AGM Feature Diagram (adapted from [95]). ......................... 27
2.10 The W-model for SPL testing (adapted from [51]). ............ 29
2.11 3D SPL test process model with dimensions of evolution (adapted from [33]). ...................................................... 30
2.12 Classic SPL testing strategies applied to the 3D model (adapted from [90]). ...................................................... 32
2.13 SPL testing artifacts interaction (adapted from [71]). ........ 33

3.1 (a) Derivation of SPL products; and (b) Overview of our contributions. 41
3.2 AGM Feature Model (adapted from [95]). .......................... 42
3.3 FSM of the third product configuration of AGM. .................. 43
3.4 (a) IRT-SPL test reuse strategy, and (b) selection algorithm. .... 46
3.5 Test case sets: (a) defined test cases $D$ for $M_3$; (b) $n$-complete test suite $T$ for $M_3$; (c) selected $n$-complete test suite $S$ for $M_3$; and (d) test case set $R$ for retest unchanged behavior. ............... 48
3.6 (a) Mobile Media feature model; and (b) derived products from Mobile Media SPL. ...................................................... 50
3.7 Accumulated effort per designed product when concretization cost is $x$ times the cost of execution: (a) the increment of features for $x = 10$; (b) the increment of features for $x = 100$; (c) the decrement of features for $x = 10$; (d) the decrement of features for $x = 100$; (e) random features for $x = 10$; and (f) random features for $x = 100$. 51

4.1 AGM Feature Model (adapted from [95]). 58
4.2 FSM of the first product configuration of AGM. 59
4.3 FFSM for the AGM SPL. 60
4.4 SMT file generated to check some conditional states and part of the completeness property. 66
4.5 Reduced feature model of BCS. 67
4.6 FFSM for AS and CLS. 68
4.7 Execution time for each case per number of non-mandatory features. 69

5.1 Abstract FSM $M$. 77
5.2 CAS Feature Model (adapted from [112]). 78
5.3 FFSM for the CAS SPL. 80
5.4 Conditional transition tree for AGM. 85
5.5 ConFTGen tool graphical interface for CAS SPL. 90
5.6 SMT parts for checking a conditional prefix relation. 92
5.7 Test suite size of the core specification. 95
5.8 Number of new tests for an FFSM and FSMs. 96
5.9 Time required to execute the HSI method for one FFSM and some FSMs. 97
5.10 Configurable test suite size per kind of feature model. 98
5.11 Number of FFSM conditional transitions per kind of feature model. 99
5.12 Time required to execute the extended HSI method per kind of feature model. 100
5.13 Feature model configuration selection for BCS. 101
5.14 FFSM of 4 composed components of BCS. 102
5.15 FFSM derived for 3 configurations with 3 composed components. 103
5.16 FFSM derived for one configuration and 2 composed components. 104

6.1 SPL validation workflow using HFSMs. 109
6.2 AGM Feature Diagram (adapted from [95]). 111
6.3 FFSM for the AGM SPL. 113
6.4 Alternative FFSM for the AGM SPL. 113
6.5 HFSM for AGM. 117
6.6 State structure for AGM HFSM. 118
6.7 Semantic variation for composing a compAnd state (part 1). 125
6.8 Semantic variation for composing a compAnd state (part 2). 127
6.9 HFSM for AGM SPL on the implemented tool. 130
6.10 Invalid states (left) and transition (right) in HFSM for AGM SPL. 131
6.11 HFSM parts for AGM SPL with a deterministic error. . . . . . . . . . . 132
6.12 HFSM parts for AGM SPL with an initially connected error. . . . . 133
6.13 HFSM parts for AGM SPL with minimal error(bottom). . . . . . . . 133
6.14 Derived HFSM for AGM SPL. . . . . . . . . . . . . . . . . . . . . . . 134
6.15 Adapted Feature Model of the Body Comfort System [60]. . . . . . 135
6.16 HFSM of 4 components of BCS. . . . . . . . . . . . . . . . . . . . . 136
6.17 HFSM derived for 3 configurations with 3 components. . . . . . . . 137
Part I

Background
Chapter 1

Introduction

1.1 Motivation

In the face of the increasing complexity, software industries moved from craftsmanship to industrialization where they customize and assemble components to produce similar products with low cost while satisfying different customer demands [41].

Software design has evolved, and new requirements for customizable/extensible software emerged while the expected release time was reduced. To satisfy such necessities new paradigms in the software engineering appeared. Software Product Line Engineering (SPLE) is a paradigm to develop software where a family of related products (a Software Product Line - SPL) is built out of a common set of core assets, thus reducing development costs for each product [83]. In SPL, products are built, step-by-step, by incrementally adding or removing functionalities, which alleviate software complexity and improve quality [58].

The SPL development process uses a product line architecture to perform a systematic reuse of requirements, architecture artifacts, components, and tests separated into two levels: domain engineering and application engineering. The domain engineering is product line-centered which develop the product line architecture with reusable/configurable artifacts. The application engineering is product-centered which develop products by instantiating the product line architecture [58].

Similar to the development of single systems, the SPL process also has several activities that are executed to ensure software quality. Testing, including various verification and validation activities check software functionalities and minimize risks. Despite the systematic software artifact reuse that increases productivity, new challenges arise in the testing activities for SPL.

Testing activities represent a large share of overall project costs and are even more challenging in SPL than for single systems [99]. In several domains new techniques are yet to be developed to test several product configurations efficiently in a systematic
manner. For example, the standard ISO 26262\(^1\) for safety-critical automotive software states that each developed product configuration should be tested using model-based techniques with a high degree of test coverage under some test criterion [15].

Several techniques, processes, and strategies [104, 112, 76, 67, 77] were developed for SPLE, but still many problems are open in this area of research. First of all, testing every single product configuration individually by using the traditional testing techniques is not acceptable for large SPLs. In general, testing products on-demand is unacceptable, due to the scarce time available for product assembly and testing. In addition, there are other challenges in SPLE including artifact management and test redundancy [90, 33].

This thesis is focused on reducing test redundancy of functional model-based conformance testing for SPLs. Functional conformance testing compares a software system to an abstract specification to check whether the expected behavior does match. The Model-Based Testing (MBT) approach [78] can automate the test process using a formal test model. By automating the test process, project costs are reduced, requirements have evolution support, and tests can achieve high fault detection rate [56].

1.2 Problem Setting

This thesis explores the problem of test redundancy in SPL and proposes a solution such that functional requirements are expressed by test models based on state machines for test design. The main research question is:

Is it possible to develop a test design method that provides low test redundancy and high fault detection for an SPL?

The main test artifacts produced in our solution are configurable test models and configurable test-suites. A configurable test model can represent the whole functional behavior of an SPL, and both configurable test-suites and test models can be instantiated using product configurations. Configurable test artifacts can use feature-based product configurations of the SPL to derive test artifacts by pruning elements (also called negative variability) [15, 52, 17].

1.3 Thesis Statement

A test design method can be developed to create reusable functional test artifacts that provide low test redundancy and high fault detection for an SPL, in which these artifacts can be configured to test a set of products derived from the SPL.

\(^{1}\)https://www.iso.org/standard/43464.html
1.4 Contributions

Several contributions lead to the proposed solution. This thesis focuses on a functional model-based testing for SPLs to reduce test-case redundancy, in two directions. Figure 1.1 provides an abstract overview of the two directions of research presented in this thesis. In the first research direction, designed as Application Engineering, the test design is focused on the product while the second direction, designed as Domain Engineering focuses on the product line architecture with configurable artifacts for behavioral conformance [36]. Dashed arrows represent the dependencies/derivation of artifacts of each step. The flow begins at the requirements and finishes at the derived test-suite. Both directions (starting in [1] and [2]) can derive test-suites. However, the second direction may achieve a better cost-benefit trade-off.

In the first direction, we explore a test-case reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL). The IRT-SPL can reduce test costs of a newly derived product in the advanced development stages based on regression testing and the P method [98]. The P method provides a reuse algorithm that minimize the number of new test-cases by incrementing test suites. This research direction enables the efficient reuse of test-cases where only a few existing test-cases...
are selected and incremented to test a newly derived product using fewer resources. The IRT-SPL strategy has three main contributions:

- an incremental test-case reuse strategy;
- a test-case selection algorithm; and
- experimental evaluation using a case study.

In the second direction, we explore test design in the early development stages. Specifically, we propose a solution named Configurable Feature-based full Coverage testing of State Machines (CFC-SM). The CFC-SM approach has the following contributions:

- proposing new configurable family-based test models for SPL;
- proposing family-based validation criteria for the full fault coverage criteria and proving them to coincide with their product-based counterparts;
- proposing the extension of a test-case generation method and proving them to coincide with their product-based counterparts;
- implementing a model-based test generation tool with a graphical interface to support validation, derivation, and generation of family-based test artifacts; and
- evaluating the approach experimentally using a realistic SPL as a case study.

1.5 Outline and Contribution Statement

The thesis is structured as follows. In Chapter 2, preliminaries of the thesis are presented such as software testing, model-based testing, and SPLs. Chapter 3 presents our first paper regarding the first research direction. In this paper, we introduce the IRT-SPL strategy. The contributions of this paper are: an incremental test-case reuse strategy; a test-case selection algorithm; and experimental evaluation using a case study. We present all contributions of the main idea, developed the technical material and experimental setup, carried out the experiments, gathered and analysed the data. The author produced the first write-up of the paper. The role of the supervisors (2nd-4th co-authors; the 3rd author is a former collaborator) was confined to help with the formalisation and presentation of the concepts.

Chapter 4 presents our second paper regarding the second research direction. We introduce the configurable test model with validation properties. The contributions


of this paper are: proposing new configurable family-based test models for SPL; proposing family-based validation criteria for the full fault coverage criteria and proving them to coincide with their product-based counterparts; and evaluating the approach experimentally using a realistic SPL as a case study. This paper focuses on model definitions and property validation. The author produced the first write-up of the paper and all technical developments. The role of the supervisors was mainly to check and review the developments and steer them if technical issues were observed.

Chapter 5 presents our third paper regarding the second research direction. We introduce an extension of the HSI test generation method for our configurable test model. The contributions of this paper are: proposing the extension of the HSI test-case generation method and proving them to coincide with their product-based counterparts; implementing a model-based test generation tool with a graphical interface to support validation, derivation, and generation of family-based test artifacts; and evaluating the approach experimentally using a realistic SPL as a case study. This paper complements the previous paper with test case generation. The author produced the first write-up of the paper, all technical developments, the experiments and their analysis. The main ideas were jointly discussed among the supervisors (and the additional co-author who was a visitor to Halmstad University).

Chapter 6 presents our fourth paper regarding the second research direction. We introduce a hierarchical version of the previous configurable test model. The contributions of this paper are: proposing new configurable family-based test models for SPL; implementing a model-based test generation tool with a graphical interface to support validation, derivation, and generation of family-based test artifacts; and evaluating the approach experimentally using a realistic SPL as a case study. This is a substantial extension of the previous paper (with about 20% overlap) that was invited by the FACS conference organisers to be submitted to a special issue. The author produced the major write-up of the paper, technical material, implemented the tool presented in this paper, and applied it to a case study. The supervisors provided reviews and comments.

Figure 1.2 depicts the relation of chapters presented with the explored research directions in this thesis.

---


Figure 1.2: Directions of research and chapters.
Chapter 2

Preliminaries and Definitions

This chapter contains the basic definitions that are required throughout the thesis.

The remainder of this chapter is organized as follows. Section 2.1 introduces basic concepts of software testing and test design. Section 2.2 presents the concepts with regard to model-based testing and test design automation. Finally, Section 2.3 presents the basic notions about software product lines.

2.1 Introduction to Software Testing

A software development process typically comprises several activities, techniques, tools and methods that may be used to increase software quality. Testing (including verification and validation) activities are used to minimize software risks and errors. Verification checks whether the results obtained during a development phase satisfy the requirements established for that phase [74]. Validation checks whether the developed software (program) satisfies user requirements [74]. Testing detects the presence of faults in the software by observing its execution [74]. Results obtained in a testing activity are also useful for maintenance and debugging. The maintenance process releases new versions of the developed software by performing updates to fix functionalities. During maintenance, regression testing can be used to verify whether new faults were not inserted after performing software modifications. On the other hand, the debugging process aims at finding faults that result in failures. Testing detects the presence of a fault in the software while debugging uses this information to try to find where the fault is located.

Basic notions of software testing followed by concepts of regression testing, coverage criteria, and test approaches are presented in this section.
CHAPTER 2. PRELIMINARIES AND DEFINITIONS

2.1.1 Faults and Test-cases

There is some divergence on software testing terms with regard to error, fault, and failure. The IEEE standard 610.12-1990 [50] provides the following explanation of software engineering terms related to the testing activity:

- **Mistake**: an incorrect action performed by a human, which produces incorrect results (e.g., a wrong action taken by the programmer);

- **Fault**: a step, procedure, or incorrect data definition (e.g., an incorrect command or instruction in the program code);

- **Error**: a difference between expected value and obtained value (e.g., a wrong intermediate result of a variable in the software execution);

- **Failure**: a wrong output produced from software execution compared to the expected output (e.g., a wrong result of user-visible events).

Figure 2.1 shows the relation among the given testing terms, in which a mistake introduces faults in the software. A fault produces errors that may be not visible, an error is propagated to an output result which may cause a failure. Errors can be classified into domain errors and computational errors. Domain errors are caused by executing an incorrect path (sequence of commands) that is different from the expected path. Computation errors are caused by an incorrect computation, while the executed path is the same as the expected path.

A specification is an artifact with an abstract representation of the system created from requirements. A specification is useful to identify failures since the expected results are determined by its analysis. A fault can also be caused by [71]:

- **Lack of requirements**: when the specification is incomplete due to a missing behavior definition. Requirements inspection can detect such faults;

- **System and specification discrepancy**: when an implemented functionality does not behave as the specification (a.k.a. functional faults). Testing can detect such faults;

- **Lack of performance, security, scalability or compatibility**: when the software execution does not satisfy a non-functional requirement. Requirements analysis and testing can detect such faults.
Testing often involves identifying faults by turning them into failures.

In general, a software system \( (\text{program}) \) has a set of values named \( \text{input domain} \), which can be accepted as input. Assume a program \( P \) that accepts Boolean values, then, the input domain of \( P \) is \( D = \{\text{true}, \text{false}\} \).

**Definition 2.1.1.** A program \( P \) with a input domain \( D \) is correct for a specification \( S \), if the program behaves as the expected behavior of the specification for all input domain values, i.e., \( \forall d \in D \cdot S(d) = P(d) \). Given two programs \( P_1 \) and \( P_2 \), if for all \( d \in D \) such that \( P_1(d) = P_2(d) \), then, \( P_1 \) and \( P_2 \) are equivalent.

The analysis of a model (specification) can be used to produce a test-case which is an acceptable input event paired with the expected output behavior of the system [71].

**Definition 2.1.2.** A test-case (or just a test) is a tuple \( (d, S(d)) \) such that \( d \in D \) is the input and \( S(d) \) is the output. A test-suite is a set of test-cases.

In general, there are two levels of abstraction for test-cases. Abstract test-cases have abstract input and output values that are executed in the specification, while concrete test-cases are executed in the real program.

In software testing, a test oracle decides whether the expected outputs matches the obtained outputs. An oracle can be a tester, developer, or another program which decides whether the output of the system under test is correct or not.

### 2.1.2 Test Process and Phases

In software development, there are several abstraction levels with well-defined phases starting from requirements down to implementation. As said before, testing can be used for the validation of the software that is being built. There are many different process models describing how to relate development and testing phases. For example, the classic V-model [25] states that each development phase has a corresponding testing phase. Figure 2.2 shows the V-model. Starting on the left-hand side there are software development phases (top-down) and on the right-hand side the testing phases (bottom-up).

Software testing performs a dynamic software analysis divided into four phases/levels [23, 1]:

1. **Unit**: testing small modules of the software project that may be components, classes, methods, functions, or procedures;

2. **Integration**: testing module interfaces and their interactions. Incomplete modules may be simulated via drivers and stubs (also called mocking in practice). *Drivers* emulate module interfaces while *stubs* simulate the behavior of a module;

3. **System**: testing the interaction of system under test with user interactions and other external (sub)systems. We check non-functional requirements (non-functional tests) and implemented functionalities (functional tests) using the software specification;
4. **Acceptance**: testing the whole system by the user to check the requirements, including functional and non-functional tests.

### 2.1.3 Regression Testing

In general, requirement changes or detected faults lead to a new version of the program. Regression testing can ensure that faults are not re-introduced in the program by reusing and executing subsets of test-suites from previous versions to test the newest version [113]. Let $P$ be a program, $P'$ the new version of $P$, and $T$ a test-suite for $P$. Test regression techniques result in a subset of test-cases $T' \subseteq T$ to verify whether $P'$ can run without new faults. By reducing the size of $T'$ we can reduce test costs. Regression testing uses three test-case reuse techniques [113]:

1. **Selection**: a subset of test-cases in $T$ is selected and put into $T'$. Selection is justified when the cost to select and execute $T'$ is smaller than re-executing the whole set $T$.

2. **Prioritization**: test-cases of $T'$ are ordered for execution according to some criteria. For example, test-cases may be ordered with regard to the chance of fault detection.

3. **Minimization**: redundant test-cases in $T'$ are removed. Test-cases are redundant when there another equivalent to serve a test goal, and they are not required for other test goals.

In general, test selection techniques identify valid test-cases related to the modified parts of the software. A test-case is *valid* with regard to a specification when the input sequence of the test-case is present (defined) in the specification. For example, a test-case $a \in T$ created from a specification $S$ used to test a program $P$ may not be
valid to test a program $P'$ based on a specification $S'$. In regression testing, test-cases are classified as [113]:

- **Reusable tests**: valid test-cases of a non-modified part of the software that are not re-executed;
- **Retestable tests**: valid test-cases which must be re-executed that are directly or indirectly related to the changed parts. For example, some test-cases are executed to reach the modified parts;
- **Obsolete tests**: invalid test-cases from $T$ for a new version of a program $P'$.

Regression testing can be corrective or progressive [107]. Regression testing is corrective when actions are executed without modifying requirements. In general, corrections are made when non-functional requirements are not satisfied by the specification. For example, performance issues may require another design pattern which does not need new test-cases. Regression testing is progressive when requirements are modified, and new test-cases are required to test modified parts of the software.

Regression testing can be used in the development and maintenance processes [73]. On maintenance, the testing approach is applied to the new version of the program while in the development a new version of a module can also be tested. Regression testing can be applied to source code and architectural artifacts [10]. When regression testing is applied to models, the change impact can be analyzed before implementation to allow better planning and cost estimation. When test-cases are traceable to the specification, it is possible to automate the identification of re-testable tests.

### 2.1.4 Coverage Criteria

Testing a program on all possible inputs is often infeasible due to the combinatorial explosion of all input combinations. Another factor that increases costs is the huge number of paths in the control flow of the program. As a consequence, other means to measure quality are explored. Test coverage criterion is a well-known heuristic for test quality measurement. A test coverage criterion (or just test criteria) decides whether a test-suite $T$ is suitable to test a program $P$. A test criterion $C$ has test requirements and test-case selection methods that select test-cases for $T$.

There is a test criterion hierarchy which determines relations of inclusion and complement [87]. One test criterion $C_1$ include $C_2$ if for any program $P$ and any test-suite $T_1$ that is $C_1$-suitable, $T_1$ is also $C_2$-suitable and exists a test-suite $T_2$ that is $C_2$-suitable but not $C_1$-suitable. For example, the specification $S$ of a program $P$ contains a connected graph with nodes and edges. A test criterion $C_1$ covers all edges while $C_2$ covers all nodes. Test criterion $C_1$ include $C_2$ because covering all edges include the coverage of all nodes.
2.1.5 Test Techniques

There are three main groups of test techniques: functional, structural, and error-based. Such techniques differ based on the kind of information used in the evaluation and generation of test-cases. Test techniques can be classified in [49]:

- **Functional testing** (black-box testing): decides whether the program satisfies functional and non-functional requirements according to the specification. The source code is unavailable and only the functionalities described in a specification are known;

- **Structural testing** (white-box testing): decides whether the source code implementation contains faults;

- **Error-based**: uses information about common mistakes found in the software development process to derive test-cases. Some basic criteria are error seeding and mutant analysis [74].

These techniques are complementary and can be combined to create an efficient, low-cost test strategy. For example, gray-box test techniques [59] combine black-box and white-box techniques where functional test-cases can be designed with access to the source code. In this thesis, we focus on black-box testing.

2.1.6 Test Automation and Quality

There is a trade-off between how much to test and how much to spend on testing. In general, there is a large amount of information that must be handled in testing, and at the same time, we need to avoid human mistakes and improve the testing quality. One solution to this problem is to automate the testing activity with the support of tools. Tools can execute costly manual tasks automatically with little effort in a systematic way, thus improving testing productivity. At the same time, they increase testing quality as they avoid human mistakes. A step that is frequently automated is test-case generation. However, full automation of the testing activity seems to be impossible due to some undecidable problems, e.g., whether two programs are equivalent to each other. Thus, manual and automatic tasks in the testing activity are seen as complementary.

Automation of the testing activity can reduce application costs, minimize human errors, and ease regression testing [1]. Regression testing tools can manage the reuse of test-cases. Tools that implement techniques for impact analysis and test-case traceability reduce costs to generate and compare test results.

The availability of testing tools facilitates the technology transfer for industries and contributes to a continuous evolution of such environments. A testing framework automates the test process, execute test-suites, and generate test reports. For example, the JUnit framework [39] can be used to define, integrate and execute unit tests.
2.2. MODEL BASED TESTING

In software development, a model describes the ideal situation in an abstract and simplified representation for a given purpose [89]. A model is formal when it has a precise, comprehensible, and unambiguous meaning [103]. Model-Based Testing (MBT) is an approach that uses a formal test model to derive test-cases automatically for functional testing [11]. A test model is a formal model derived from the requirements which represents the behavior of the software system. A test model can be seen as an abstract representation of a detailed behavioral model created in the development phases that is meaningful to generate significant test-cases.

MBT can be used in any testing phase and some approaches [4, 92] extend MBT to non-functional tests. Figure 2.3 shows the basic application areas of MBT. The prism represents the initial usage of MBT regarding kind of testing, source of test generation and level of testing. As testing approaches advances the prism may extends even further, e.g. for acceptance testing.

Some advantages of model-based testing are [110, 103]:

- test models are usually small and easy to understand, verify, and modify;
- test models can provide traceability between requirements and test-cases;

---

Figure 2.3: Application fields of model-based testing (adapted from [110]).
• test models can be used for (semi)-automatic test-case generation given a test coverage criterion;

• testing can be performed before (test-first approaches) and after the implementation of the system (traditional approach).

### 2.2.1 Model-Based Testing Process

The MBT process uses several test artifacts from requirements for the system under test. A **System Under Test (SUT)** is composed of the code implementation and the required infrastructure to run it. Figure 2.4 shows an overview of the relation between MBT artifacts with solid arrows. First, the requirements are captured in the top-left corner. Then, from requirements, the development model, test model and test-case specifications (based on a test coverage criterion) are generated. From both test model and test-case specifications, test-cases are created/reused, concretized/selected and then executed on the SUT. Finally, an oracle compares the results which generate a verdict. The dashed arrows between development model and test model represent the possibility of generating test models from development models using, e.g., model transformation techniques [63].

A typical MBT process has seven steps [85, 103]:

1. **Requirement understanding**: the tester needs to understand how the software works in a given environment following some guidelines:

   (a) Identify the characteristics or components to be tested;

   (b) Create communication channels between development groups to allow reuse and model adaptation;

   (c) Enumerate inputs and outputs for automation;

   (d) Identify input sequences that need to be modeled to ease model design;

   (e) Constantly update the requirement model for a better understanding.
2.2. MODEL BASED TESTING

2. **Modeling**: the tester creates a test model to represent the behavior of a component, subsystems, or the whole SUT. In general, the test model is an abstract representation of the desired behavior of the SUT;

3. **Test criteria definition**: the tester needs to select suitable test criterion and a tool to support test-case generation;

4. **Test-case generation**: the tester needs to select a method for automatic test-case generation using the test model and test criterion;

5. **Test-case concretization**: the tester needs to transform (concretize) abstract test-cases into executable test-cases for the SUT. The test model and the SUT are at different abstraction levels. Thus, to concretize a test-case, the tester can transform test-cases into test scripts, designs an adapter or use both scripts and an adapter. An adapter can translate inputs and outputs between a test model and an SUT using a concretization and an abstract function, respectively;

6. **Test execution**: the tester needs to execute concrete test-cases in the SUT using the off-line or on-line approach. The off-line approach separates test-case generation from test-case execution. First, all test-cases are generated and then they are executed. The on-line approach generates test-cases and executes them dynamically (on-the-fly) where the next generated test-case is dependent on the result of the previous test-case;

7. **Result analysis**: the oracle checks the results provided from the test-cases execution to generate verdicts. An oracle is a tester or a program that can automatically compare the abstract expected output from the test model specification to the real output provided from the SUT. A verdict is the result of a test-case execution that may be pass, fail, or inconclusive. A test-case is inconclusive when the execution ends early, but no failure is found.

2.2.2 Test-case Concretization

One important step in the MBT test process is the concretization of abstract test-cases [102]. Generated abstract tests are augmented with concrete implementation-specific data making them executable. According to case studies [102, 40], the cost to manually concretize a test-case is several (around 200) times greater than the cost of executing the same concretized test. To tackle this problem, adapters can be developed to automate the concretization process. However, the adapters often need to be modified for new versions/products. For example, systems that constantly evolve (e.g., graphical user interface) cannot afford to update adapters of each new version of the system, which often takes more time than manually testing the system in the first place [26].
2.2.3 Test-case Generation for Finite State Machines

The test model must be an abstract version of the desired SUT behavior or at least be easier to verify, modify and maintain. Some formal models used in MBT are: Finite State Machines, Extended Finite State Machines, Labeled Transition Systems, and Input/Output Transition Systems \[103, 78\]. Testing based on state machines has been explored extensively in the last decades \[14, 38, 69, 82, 57, 79, 112, 97, 52\]; however, there are some remaining challenges to consider. In this thesis, we investigate the generation of configurable test-suites using test models based on finite state machines by extending the formalism and the test-case generation methods to software product lines.

In this section, we present the finite state machine formalism, followed by state, transition, and full fault coverage criterion, and finally describing related test-case generation methods.

2.2.3.1 Finite State Machines

The classic Finite State Machine (FSM) formalism is often used due to its simplicity and rigor for specifying systems such as communication protocols and reactive systems \[11\].

**Definition 2.2.1.** An FSM \( M \) is defined by a 5-tuple \((S, s_0, I, O, T)\), where \( S \) is a finite set of states, \( s_0 \in S \) is the initial state, \( I \) is the set of inputs, \( O \) is the set of outputs, and \( T \) is the set of transitions in the form of \( t = (s, x, o, s') \in T \), where \( s \in S \) is the source state, \( x \in I \) is the input label, \( o \in O \) is the output label, and \( s' \in S \) is the target state.

The FSM model can be used as a graph-based test model for test design where paths are selected to be executed. Moreover, paths are essential to define some FSM properties due to the tuple definition that binds FSM elements.

**Definition 2.2.2.** Given an input sequence \( \alpha = (x_1, ..., x_k) \), where \( x_i \in I \), for all \( 1 \leq i \leq k \), a path from state \( s_1 \) to \( s_{k+1} \) exists when there are transitions \( t_i = (s_i, x_i, o_i, s_{i+1}) \in T \), for each \( 1 \leq i \leq k \). A path \( \upsilon \) is a 3-tuple \((\tau, \alpha, \beta)\), where

1. \( \tau = (s_1, ..., s_{k+1}) \in S^* \) is the state sequence,
2. \( \beta = (o_1, ..., o_k) \in O^* \) is the output result.

Notation \( \Omega(s) \) is used to denote all paths that start at state \( s \in S \) and \( \Omega_M \) is used to denote \( \Omega(s_0) \). Given a path \((s_0, ..., s), \alpha, \beta) \in \Omega_M \), \( s \) can be reached using the input sequence \( \alpha \).

2.2.3.2 Validation Properties

Methods that generate test-cases from FSMs ([14, 69, 82, 97]) usually require FSMs to possess some of the properties defined below.
2.2. MODEL BASED TESTING

**Figure 2.5: Abstract FSM M.**

**Definition 2.2.3.** The following validation properties are defined for FSMs:

1. **Deterministic:** if two transitions leave a state with a common input, then both transitions reach the same state and produce the same output:

\[ \forall (s, x, o, s'), (s, x, o', s'') \in T \cdot s' = s'' \land o = o' \]

2. **Complete (not required for some methods):** every state has at least one transition for each input:

\[ \forall s \in S, x \in I \cdot \exists o \in O, s' \in S \cdot (s, x, o, s') \in T \]

3. **Initially Connected:** there is a path to every state from the initial state:

\[ \forall s \in S \cdot \exists \alpha \in I^*, \tau \in T^*, \beta \in O^* \cdot ((s_0, \ldots, s), \alpha, \beta) \in \Omega_M \]

4. **Minimal:** all pairs of states must behave differently (be distinguishable) by producing different sequences of outputs for some sequence of inputs.

\[ \forall s_a, s_b \in S \cdot \exists ((s_{a_0}, \ldots, s_{a'}), \alpha, \beta_a) \in \Omega(s_a), ((s_{b_0}, \ldots, s_{b'}), \alpha, \beta_b) \in \Omega(s_b) \cdot \beta_a \neq \beta_b \]

**Example 1.** Figure 2.5 presents a deterministic, initially connected, complete, and minimal FSM \( M = (S, s_0, I, O, T) \), where \( S = \{1, 2, 3\} \), \( s_0 = 1 \), \( I = \{a, b, c\} \), \( O = \{0, 1\} \), and \( T = \{(1, a, 1, 2), (1, b, 0, 1), (1, c, 0, 1), (2, a, 0, 2), (2, b, 1, 3), (2, c, 1, 1), (3, a, 1, 2), (3, b, 0, 3), (3, c, 1, 1)\} \). For determinism, there is only one transition leaving each state for any given input. For initial connectedness, two transitions \((1, a, 1, 2)\) and \((2, b, 1, 3)\) (highlighted) connect the initial state to states 2 and 3. For minimality, the input sequence \(a\) results in different output behavior for state pairs \((1; 2)\) and \((2; 3)\), and the input sequence \(c\) for \((1; 3)\).

**2.2.3.3 Test-cases**

To detect the presence of faults in conformance testing, test-cases are used to verify the implemented behavior.

**Definition 2.2.4.** Given an FSM \( M = (S, s_0, I, O, T) \), an input sequence \( \alpha \in I^* \) is **defined** for \( M \) on state \( s_0 \) when there is a path \(((s_0, \ldots, s), \alpha, \beta) \in \Omega_M \) (Definition 2.2.2) where \( \alpha \) reaches \( s \). A **test-case** (input part) of \( M \) is a defined input sequence \( \alpha \in I^* \).
Next, we present preliminaries to define a prefix-closed set of test-cases.

**Definition 2.2.5.** Given input sequences $\alpha, \beta, \gamma \in I^*$, an input sequence $\alpha$ is the *prefix* of an input sequence $\beta$ when $\beta = \alpha \gamma$ for the input sequence $\gamma$, and $\gamma$ is a *suffix* of $\beta$. An input sequence $\alpha$ is a *proper prefix* of $\beta$ when $\beta = \alpha \omega$ for some $\omega \neq \varepsilon$, where $\varepsilon$ is the *empty sequence*. Moreover, we say that a sequence $\alpha \in A \subseteq I^*$ is *maximal* in $A$ if there is no sequence $\beta \in A$ such that $\alpha$ is a proper prefix of $\beta$.

**Definition 2.2.6.** Given a set of input sequences $A \in \mathcal{P}(I^*)$ and an input sequence $\beta \in A$, the set of prefixes of $\beta$ is denoted by $\text{pref} (\beta)$. Similarly, $\text{pref}(A)$ is the set of all prefixes of all input sequences $\beta \in A$, i.e., $\text{pref}(A) = \bigcup_{\beta \in A} \text{pref} (\beta)$. When $A = \text{pref}(A)$ the set $A$ is called *prefix-closed*. The prefix-closed set of test-cases of $M$ is called a *test-suite* of $M$.

### 2.2.3.4 State Coverage Criterion

The *state coverage criterion* require defined input sequences that can reach each and every state. We assume FSMs that are deterministic and initially connected.

**Definition 2.2.7.** Given an FSM $M = (S, s_0, I, O, T)$ and a state $s \in S$, the test-suite $TS \subseteq \mathcal{P}(I^*)$ *covers* $s$ if there exists a path $((s_0, \ldots, s), \alpha, \beta) \in \Omega_M$ to reach $s$ such that $\alpha \in TS$. The test-suite $TS$ is a *state cover set* (for $M$) if it covers every state of $M$:

$$\forall s \in S \cdot \exists ((s_0, \ldots, s), \alpha, \beta) \in \Omega_M \cdot \alpha \in TS$$

**Example 2.** Following the state coverage criterion, the set $TS = \text{pref}(\{ab\})$ is a state cover set for the FSM $M$ presented in Figure 2.5.

### 2.2.3.5 Transition Coverage Criterion

The *transition coverage criterion* require defined input sequences that can reach the source state of each and every transition and followed by its input.

**Definition 2.2.8.** Given an FSM $M = (S, s_0, I, O, T)$, and a state cover set $TS \subseteq \mathcal{P}(I^*)$ for $M$, the test-suite $TS$ *covers* a transition $((s_0, \ldots, s), \alpha, \beta) \in \Omega_M$ if there exists a path $((s_0, \ldots, s), \alpha, \beta) \in \Omega_M$ from state $s_0$ to $s$, where $\alpha \in TS$ is an input sequence to reach $s$, $\beta$ is the output sequence, and $\alpha x \in TS$. The set $TS$ is a *transition cover* test-suite of $M$ if it covers every transition of $M$:

$$\forall (s, x, o, s') \in T \cdot \exists \alpha \in TS \cdot \exists ((s_0, \ldots, s), \alpha, \beta) \in \Omega_M \cdot \alpha x \in TS$$

A breadth-first search algorithm can be used to produce the state and transition cover sets.
Example 3. Figure 2.6 presents a testing tree generated by the transition cover set $TS = \text{pref}(\{b, c, ac, aa, aba, abb, abc\})$ for $M$ of Figure 2.5. Starting from the initial state, we identify a set of transitions that use the selected state as the source state. For each selected target state a new branch is created for the next tree level using unvisited transitions. Note that the transition cover set extends the state cover set.

2.2.3.6 Full Fault Coverage Criterion

To define the full fault coverage criterion, we use adequacy conditions based on convergence and divergence properties based on a fault domain.

Definition 2.2.9. Given an FSM $M = (S, s_0, I, O, T)$, two test-cases (Definition 2.2.4) $\alpha$ and $\beta$ of $M$ are convergent when both test-cases reach the same state, and they are divergent when they reach different states.

Test convergence and divergence with respect to a single FSM are complementary, i.e. any two tests are either convergent or divergent. However, when a set of FSMs $\Sigma$ is considered, some tests are neither $\Sigma$-convergent nor $\Sigma$-divergent.

Definition 2.2.10. Given a test-suite $T$ and a set $\Sigma$ of $k$ ($k \geq 2$) FSMs, $T$ is $\Sigma(T)$-convergent when for each pair of test cases $\alpha, \beta \in T$ they are convergent in each pair of FSMs of $\Sigma$. $T$ is $\Sigma(T)$-divergent when for each pair of test cases $\alpha, \beta \in T$ they are divergent in each pair of FSMs of $\Sigma$.

The $\Sigma$-convergence relation is reflexive, symmetric and transitive, i.e. it is an equivalence relation over the set of tests. On the other hand, the $\Sigma$-divergence relation is irreflexive and symmetric [98].


Example 4. Consider the FSMs $M$ and $M'$ in Figs 2.5 and 2.7, respectively. The tests $aa$ and $ba$ are $\{M, M'\}$-convergent, whereas the tests $bb$ and $aa$ are $\{M, M'\}$-divergent. On the other hand, tests $bb$ and $ab$ are neither $\{M, M'\}$-convergent nor $\{M, M'\}$-divergent since they are $M'$-convergent and $M$-divergent.

The notions of convergence and divergence are extended to sets of FSMs defined as a fault domain.

Definition 2.2.11. Assume an FSM $M = (S, s_0, I, O, T)$ with $n$ states. The fault domain of $M$, denoted by $\mathcal{F}$, is the set of all FSMs that all are: (i) deterministic (Definition 2.2.3 item 1); (ii) have the same input alphabet as $M$; (iii) and include all defined input sequences (Definition 2.2.4) of $M$ (i.e., $\forall N \in \mathcal{F} \implies \forall ((s_0, ..., s), \alpha, \beta) \in \Omega_M \implies \exists((q_0, ..., q), \alpha', \beta') \in \Omega_N \implies \alpha = \alpha'$). Moreover, $\mathcal{F}_n$ is the set of FSMs from $\mathcal{F}$ with $n$ states.

Distinguishing two FSMs uses input sequences that are applied to their initial states.

Definition 2.2.12. Given a test-suite $T$, FSMs $M$ and $N$ are $T$-equivalent when all test-cases of $T$ applied to $M$ and $N$ return the same output sequence. The subset $\mathcal{F}_n(T) \subseteq \mathcal{F}_n$ denotes all FSMs of $\mathcal{F}_n$ which are $T$-equivalent to $M$. Moreover, given two test-cases $\alpha, \beta \in T$ they are $T$-separated when there are test-cases $\alpha \gamma, \beta \gamma \in T$ that return different output sequences for $M$ and $N$ that return different output sequences for $M$ and $N$.

Thus, $T$-separated test-cases diverge in all FSMs that are $T$-equivalent to $M$.

Lemma 1 ([98]). Given a test-suite $T$ of an FSM $M$, $T$-separated tests are $\mathcal{F}(T)$-divergent.

We refer to [98] for detailed proofs of the results presented in this section.

Lemma 2 ([98]). Given a test-suite $T$ and $\alpha \in T$, let $K$ be an $\mathcal{F}_n(T)$-divergent set with $n$ tests and $\beta \in K$ be a test $M$-convergent with $\alpha$. If $\alpha$ is $\mathcal{F}_n(T)$-divergent with each test in $K \setminus \{\beta\}$, then $\alpha$ and $\beta$ are $\mathcal{F}_n(T)$-convergent.

To define the completeness of a test-suite $T$ the notion of preserving convergence between $M$ and FSMs of $\mathcal{F}_n$ is used.
Definition 2.2.13 ([98]). Given a test-suite $T$ of an FSM $M$, a set of tests is $\mathcal{S}_n(T)$-convergence-preserving (or, simply, convergence-preserving) if all its $M$-convergent tests are $\mathcal{S}_n(T)$-convergent.

The following theorem summarizes the main results from [98] where the full fault coverage criterion is established based on convergence and divergence properties.

Theorem 1 ([98]). Given a test-suite $T$ for an FSM $M$ with $n$ states, $T$ is $n$-complete when for all FSMs $N \in \mathcal{S}_n$ there exist tests in $T$ that distinguish $M$ and $N$ (Definition 2.2.12). If $T$ has an $\mathcal{S}_n(T)$-convergence-preserving transition cover set for $M$ that includes the empty symbol $\varepsilon$ (i.e., it is initialized), then $T$ is an $n$-complete test-suite for $M$.

A test-suite $T$ satisfies the full fault coverage criterion when it is $n$-complete for an FSM $M$. By executing an $n$-complete test suite $T$, we are capable of detecting any fault in all FSM implementations $N \in \mathcal{S}_n(T)$.

2.2.3.7 Test-case Generation Methods

In the Model-Based Testing (MBT) approach we select a test criterion to design test-cases using a behavioral test model. The resulting test-suite execution must be able to detect as much faults as possible. Thus, a suitable test criterion should strike the right balance between test costs and fault detection. The full fault coverage is one of the test criterion with such balance.

Most automatic test-case generation proposals use heuristics to find good test-suites since finding the best solution is a hard problem. There exist several methods to generate $n$-complete test-suites [14, 69, 98] for the full fault coverage criterion. For example, the incremental P method [98] uses two input parameters: a deterministic, initially connected, and minimal FSM $M$; and an initial test-suite $T$. The initial set $T$ can be empty, and new test-cases are added/incremented (if necessary) until an $n$-complete test-suite for $M$ is produced. Therefore, the P method checks if all implementations $N \in \mathcal{S}_n$ can be distinguished from $M$ using $T$, and decides if more sequences need to be added to $T$. Experimental evaluation indicates that the P method often results in smaller $n$-complete test-suites compared with other methods [31].

In this thesis, we investigate the HSI and P methods to generate test-cases for the full fault coverage criterion in two directions of research explained in the contributions section. We briefly presented some basic notions of the P method in Section 2.2.3.6, and we refer to [98] for a detailed explanation of the P method. Next, we introduce the basic notions of the HSI method explained in [69].

The HSI method extends the W method [14], which uses a characterizing set to select Harmonized State Identifiers (HSI) sets to distinguish pairs of states in the FSM.

Definition 2.2.14. Given an FSM $M = (S, s_0, I, O, T)$ with state set $S = \{s_1, \ldots, s_n\}$, the set $W \in \mathcal{P}(I^*)$ is a characterizing set if and only if for all $1 \leq i, j \leq n$ with $i \neq j$ there exists an input sequence (separating sequence) $\gamma \in W$ that distinguishes $s_i$ and $s_j$: 
∀s_i, s_j ∈ S • ∃(s_i, ..., s_i′, γ, β_i) ∈ Ω(s_i), ((s_j, ..., s_j′), γ, β_j) ∈ Ω(s_j) • β_i ≠ β_j

The sets H_1, ..., H_n ⊆ W are harmonized state identifiers if and only if for all 1 ≤ i, j ≤ n with i ≠ j there exist an input sequence (common prefix) γ ∈ H_i ∩ H_j that distinguishes s_i and s_j.

To generate test-cases the HSI method concatenates a transition cover set with HSI sets and hereby constructs the final test-suite.

Definition 2.2.15. Given a transition cover set CV for M and harmonized state identifiers sets H_i, the HSI method returns a test-suite TS by concatenating CV with every H_i set for each s_i ∈ S such that only tests of CV that reach s_i are concatenated:

∀s_i ∈ S • ∀α ∈ CV • ∃((s_0, ..., s_i), α, β) ∈ Ω_M • ∀h ∈ H_i • αh ∈ TS

Example 5. The characterizing set W for FSM M presented in Figure 2.5 is W = {a, c}, while the HSI sets are: H_1 = {a, c}, H_2 = {a}, and H_3 = {a, c}. The complete test-suite is obtained by concatenating CV = pref({b, c, ac, aa, aba, abb, abc}) with H_i sets, which results in TS = pref({ba, bc, ca, cc, aca, acc, aaa, abaa, abba, abbc, abc, abca, abc}).

2.3 Software Product Lines

Software design has evolved, and new requirements for custom/extensible software have emerged while the expected release time has been reduced. To satisfy such necessities new approaches in the software engineering have appeared. The Software Product Line Engineering (SPLE) approach is one such approach that aims at systematic reuse of core assets represented by software artifacts to instantiate, generate or assemble multiple similar systems that form a software product line [19].

A Software Product Line (SPL) is a set of software programs that share artifacts to satisfy a specific domain [19]. An SPL is derived from a reusable software architecture resulting in a product family. Products can vary regarding behavior, quality attributes, platform, physical configuration and middleware [58]. The SPL software architecture defines requirements, components and processes that are shared, reused and managed by commonalities and variabilities. Commonalities manage product similarities while variabilities manage product differences using features. A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system [53]. A feature can be classified into three types [58]:

- **Common**: this type of feature describe a characteristic that is present in every derivable product;

- **Variable**: this type of feature describes a characteristic that is present in only some products but not all of them. If there is a module that implements such a feature, then, it must be reusable;
• **Product-specific**: this type of feature describes a characteristic that may be present in only one product. The client may ask for a new specific feature, and the SPL architecture must be able to support it.

For example, assume an SPL for mobile phones. A common feature can be a fixed communication module that is present in every cell phone. A variable feature can be a video camera provided by different brands where some models use a specific camera. A product-specific feature can be a TV module which was not initially considered in the SPL and is added for the sake of a specific product.

Developing an SPL from scratch requires more effort than developing a single software system. However, the SPL infrastructure allows the systematic derivation of several products to increase productivity, reduce development costs, and ease product evolution. In general, the extra effort to develop an SPL is compensated after deriving the third product [58].

### 2.3.1 Development Process

The SPL development process performs a systematic reuse on requirements, architecture artifacts, components, and tests to develop other products. The SPL development process is separated into two levels:

- **Domain engineering** (platform development): development of common and reusable components;
- **Application engineering**: development of products by instantiating/assembling reusable components.

The domain engineering level has four steps [83]:

1. **Domain analysis**: analyze domain characteristics and represent features in a feature model.
2. **Core asset development**: design, implement, test common (core) and reusable artifacts, then, store them in a repository. In general, these artifacts contain features that are mapped to model elements.
3. **Production plan**: elaborate guidelines to derive individual products from core artifacts which may include model transformation, code generation, and compilation.
4. **Product management**: create maintenance routines that describe methods and strategies to manage variability. For example, a common feature may be updated and turn into a variable feature.

The application engineering level has three steps [83]:

1. **Product characterization**: select features from a feature model to characterize the product that is going to be built.
2. **Product synthesis**: retrieve reusable artifacts from the repository which are required to build the product.

3. **Product construction**: process reusable artifacts following the production plan until the final product is built.

Figure 2.8 presents an overview of the whole SPL process and the related artifacts. Software artifacts developed at the domain engineering are reused in the application engineering. Dashed arrows illustrate the flow of software artifacts while normal arrows illustrate the process cycles. The resulting application is derived using artifacts with common, variable and product-specific parts.

### 2.3.2 Feature Diagram

A *feature diagram* [93] uses a notational convention to describe constraint-based feature relations. The basic feature relations are mandatory, optional, inclusive-OR (or), exclusive-OR (alternative), include, and exclude [54]. A noteworthy feature modeling method is the Feature-Oriented Domain Analysis (FODA) [53]. Subsequent feature modeling methods, such as the Orthogonal Variability Model (OVM) [58], extend the FODA to add new dependency relations.
Example 6. The Arcade Game Maker (AGM) [95] can produce arcade games with different game rules. Figure 5.2 shows the feature diagram of AGM. There are three alternative features for the game rule (Brickles, Pong, and Bowling) and one optional feature (Save) to save the game. Among the alternative features, exactly one must be selected.

A feature diagram is developed in the domain engineering and used as input to the application engineering level, where it is instantiated by a configuration model. A configuration model allows for the selection of features to specify a single product, and it is useful to integrate components for the product configuration process. The product configuration process (binding) derives a specific product using the reusable SPL architecture and a configuration model with selected features.

2.3.3 Feature Constraint

In general, due to the dependencies and constraints on feature combinations, only some products can be derived. Assume a set of features $F$ of a feature model. The set of all valid products $P$ of an SPL is a subset of feature combinations from the power set $\mathcal{P}(F)$ that satisfies the constraints specified by the feature model [5].

A feature constraint $\chi$ is a propositional formula that interprets the elements of the feature set $F$ as propositional variables. The set of all feature constraints is denoted by $\mathcal{B}(F)$. The relation between features and its constraints can be modeled by a feature diagram and extracted as a feature constraint following a formal semantics [93]. A product configuration $\rho \in \mathcal{B}(F)$ of a product $p \in P$ is a feature constraint of the form $\rho = (\bigwedge_{f \in p} f) \land (\bigwedge_{f \notin p} \neg f)$, i.e., the conjunction of all features present in $p$ and the conjunction of the negation of all features absent from $p$. The set $\Lambda \subseteq \mathcal{B}(F)$ denotes all valid product configurations of the SPL. Given a feature constraint $\chi \in \mathcal{B}(F)$, a product configuration $\rho \in \Lambda$ satisfies $\chi$ (denoted by $\rho \models \chi$), if and only if the feature constraint $\rho \land \chi$ is satisfiable.

Example 7. Given the feature diagram of Figure 5.2 the extracted feature set is $F = \{G, V, R, C, A, Y, P, S, B, N, W, M, L\}$ where $O = \{G, V, R, C, A, Y, P, M, L\} \subseteq
F is the subset of mandatory features. The extracted feature constraint that represents the relation of all features is:

\[ \chi = ((\bigwedge_{f \in O} f) \land (S \implies V) \land (B \lor N \lor W) \land \neg(B \land N) \land \neg(B \land W) \land \neg(N \land W)) \in B(F) \]

There are only six product configurations that satisfy \( \chi \), namely, those specified below:

\[
\begin{align*}
\rho_1 &= ( \bigwedge_{f \in O} f ) \land B \land \neg N \land \neg W \land \neg S, \\
\rho_2 &= ( \bigwedge_{f \in O} f ) \land B \land \neg N \land \neg W \land S, \\
\rho_3 &= ( \bigwedge_{f \in O} f ) \land \neg B \land N \land \neg W \land \neg S, \\
\rho_4 &= ( \bigwedge_{f \in O} f ) \land \neg B \land N \land \neg W \land S, \\
\rho_5 &= ( \bigwedge_{f \in O} f ) \land \neg B \land \neg N \land W \land \neg S, \\
\rho_6 &= ( \bigwedge_{f \in O} f ) \land \neg B \land \neg N \land W \land S.
\end{align*}
\]

In a textual representation, the logical operators on feature constraints are denoted by & (and), || (or), and ! (not).

### 2.3.4 Feature Model

A feature model \([83]\) specifies the structure of an SPL in terms of its feature and feature constraints. It can serve as the underlying structural model for other formalisms modeling the behavior of an SPL, e.g., for the purpose of testing.

**Definition 2.3.1.** A feature model \( FM \) is a tuple \((F, \chi)\), where \( F \) is the set of features and \( \chi \) is the feature constraint.

### 2.3.5 Software Product Line Testing

The SPL testing aims to examine core artifacts that are shared with many products, reusable modules, and their interactions. Thus, SPL testing encompasses activities for the SPL architecture and its products \([71]\).

The main advantages of the SPL approach are effective, low-cost mass production and customization of products. The SPL development process has a systematic software artifact reuse that increases productivity. However, new challenges arise in the testing activity. Testing is more challenging in SPLE than for individual systems \([99]\). In general, testing large SPLs demands substantial effort, and effective reuse is a challenge \([58]\). Some SPL testing challenges are:

- **Variability management:** suitable feature models, and variability representation in the product line architecture. Design and test models contain extra elements which represent all reusable elements. In general, a non-executable single model is created and then pruned for single products. Testing the whole model may be costly without proper testing techniques \([15]\);

- **Number of products:** a large number of products that can be generated. Instead of testing all derivable products, only a representative subset of products are selected to testing the whole SPL \([66]\) (see Section 2.3.5.3);
• **Dimensions of evolution:** redundant tests exist in all three dimensions of evolution: level, version, and variant. In general, the SPL testing performs redundant tests in another test level (e.g., integration and system testing), previous versions of the same product, and previously designed products [90].

### 2.3.5.1 Test Process Model

SPL testing concerns core and reusable artifacts at two levels: domain testing; and application testing. *Domain testing* aims at detecting faults in artifacts produced in the domain engineering level and creates testing artifacts to be reused in application testing. The domain testing is similar to testing a single system. However, most core artifacts contain features that are mapped to some model elements and stubs are required to simulate unimplemented modules. *Application testing* reuses testing artifacts from domain testing to design and execute product-specific test-cases in the SUT. Domain and application testing levels are interdependent as a fault detected in the application may be a fault introduced in the domain.

Figure 2.10 presents the W-model conceptual test model for SPLs [51]. The W-model is incorporated in both SPL development levels. Assets produced by domain and application engineering are tested on the levels of the V-model. The left-hand side of the two V-models include the common abstraction levels of the software. The right-hand side of the model binds testing levels to the software abstraction levels.

### 2.3.5.2 Dimensions of Evolution

Faults may be introduced in reusable or core artifacts. A fault introduced in a core artifact that was detected in a product leads to changes in many products. Faults introduced in reusable artifacts must be traced back to the group of affected products. Moreover, the test plan must manage scope, resources, and time in three SPL dimensions of evolution [33]:
• **Level**: test-cases are concretized and executed in several abstraction levels. Functionalities that are tested using abstract test models may be re-concretized and re-executed for another test model. The lack of traceability between test models in different test levels may result in test-case redundancy. Reuse-based strategies can be developed for test levels and consequently reduce test effort [80];

• **Time**: test-cases are concretized and executed in different versions of a product over time. Regression testing approaches can be used to reuse test-cases for new versions and reduce test redundancy;

• **Variants**: test-cases are concretized and executed in various similar products. The SPL architecture has core artifacts that are present in every product turning them similar to each other. Thus, regression approaches can be adapted to reuse some test-cases between products.

Figure 2.11 presents the 3D SPL test process model with three dimensions of evolution. SPL testing might have repetitive activities across version, variants, and levels (unit, integration, and system tests) as well redundant test artifacts (including test-cases) for requirements, design, interfaces, and implementation.

Test-case reuse strategies in SPL are essential to reduce testing costs. Ideally, a new derived product can be tested with little effort using automatic test-case reuse strategies [86].
2.3.5.3 Test Strategies

Exhaustively testing a large number of products is often not viable because of project costs. In SPLs, a proper test-case reuse strategy is crucial due to the increasing test costs compared to the decreasing development costs. Thus, some products may be selected to test the SPL, or new derived products may be tested on demand. One naive strategy is to collect all test-cases of previous products and remove all obsolete and redundant tests. However, such an approach is not economically viable either in large SPLs. There are three categories of SPL testing strategies [90]: the anti-SPL philosophy; reuse-based; and combinatorial.

The anti-SPL testing strategies (traditional test-case generation methods, i.e., [14, 69]) do not reuse artifact and are useful for small SPLs, which are:

- **Brute force**: test the whole SPL architecture and all possible SPL products;
- **Pure application** (product-by-product - PBP): test every product independently without the reuse of test artifacts.

The reuse-based strategies reuse, increment, or adapt test artifacts created in the domain engineering level or from previously derived products, which are:

- **Sample application**: The most representative product (core product) is selected to be the first, and other products are tested by analyzing differences using regression-based approaches. Some strategies are the incremental testing [105] and the delta modeling [61];
- **Commonality and reuse**: test all modules of common features and individually test modules of variable features. Modules that are not present in all products are tested once a product is about to be derived and deployed. MBT testing approaches use this strategy as test models can be created in the domain testing and reused in the application testing level. However, this strategy does not reuse test-cases between products. Thus, it can be combined with the sample example strategy;
- **Division of responsibility**: test techniques are divided into domain and applications testing levels. For example, unit testing can be executed in the domain testing level while integration and system tests may be executed in the application testing level.

Combinatorial strategies identify and test representative subsets of products using heuristics, which are:

- **Pairwise testing**: test every combination of two features that interact with each other [76].
- **T-wise testing**: test every combination of several T (more than 2) features that interacts with each other [81].
Some SPL test strategies can be combined to reduce testing costs. Large SPLs may combine more than one strategy to create a test plan. Figure 2.12 illustrates four classic SPL test strategies due to Pohl’s [83]. In the brute force strategy, all related SPL test artifacts are tested, while in pure application each vertical arrow represents the testing of a product. In the sample application strategy one product is tested, and other products reuse test artifacts from the first, while commonality and reuse strategy tests all common artifacts and then tests reusable artifacts on demand in each product.

2.3.5.4 Regression-based SPL Testing

The literature proposes some strategies for improving test efficiency but contains very few real life evaluations [32]. Research on regression testing, which addresses similar problems, offers some empirically evaluated approaches. With existing regression testing approaches, it is possible to provide automated decision support in a few specific cases. Figure 2.13 shows the generic interaction between the SPL architecture and products for testing. Test-cases and the test plan are created for the SPL architecture and then specialized for each product.

Some SPL testing approaches [64, 65] make use of techniques inspired by regression testing to test new products. Such approaches explore version and variant dimensions of evolution. Few other approaches such as the one by Silveira Neto et al. [96] developed regression testing approaches based on architecture specification. They identify that the effort to apply the corrective scenario is greater than the progressive scenario. Moreover, some thesis [55, 46, 88, 67] publish tools, environment, and frameworks for SPL-based regression testing during variability modeling to instantiate products.
2.4 Concluding Remarks

Software Product Lines (SPLs) [58] provide an efficient mass customization approach at the cost of an initial extra effort to build the required architecture. As a consequence, testing the SPL becomes a challenging task due to the high number of different product configurations and other factors. One important factor is the test redundancy where several similar products share some components, and without a proper analysis of those products, part of test-suites are re-concretized and re-executed in vain.

One formalism used in the representation of the behavior of software systems is the classic Finite State Machine (FSM). There are several test-case generation techniques that were developed to generate test-suites for strong test criteria. A test-suite generated from strong test criteria once executed can detect several faults in the system compared to another test-suite from other test criteria. There is a hierarchy where one of the best is the full fault coverage for FSMs, and it was not yet fully extended to more expressive test models such as those based on labeled transition systems. Thus, in this thesis, one technique of the full fault coverage for FSMs is extended to the SPL context with the proposed FFSM.

The presented thesis is focused on model-based testing for SPLs. In detail, two test design approaches were proposed to derive test-cases avoiding redundancy and using strong coverage criteria. In our first test design approach, IRT-SPL can provide effort savings starting from 5% when we have more than 4 product configurations and the concretization value is 10, and goes up to 36% when we have 24 product configurations and the concretization value is 100 compared to other test reuse strategies.
In our second test design approach, we explore the variability to generate a small configurable test-suite for the entire SPL where most products reuse equivalent test-cases to avoid redundancy. We conducted an experimental study with random FFSMs and feature models, and for each feature model we selected 20 random products. We decided to randomize feature models and FFSMs to avoid bias of specific domains. A first test-suite was generated using the extended HSI method for an FFSM and compared to a second unified test-suite generated using the original HSI method for the selected 20 individual configurations. The results indicate a reduction in approximately 50% of the number of new tests required for testing using the first test-suite compared to the second. Also, the reduction percentage may decreases for less than 20 products and may increases for more than 20. We illustrate and evaluate our approach and tool by means of a case study from the automotive domain, using the Body Comfort System (BCS) for the VW Golf SPL [60]. The results indicate a reduction in approximately 25% of the number of new tests required for testing a slice of the BCS for 6 products. Moreover, our examples indicates a reduction in approximately 35% of the number of new tests required for testing the AGM SPL for 6 products.

To conclude, we bring our research question again.

Is it possible to develop a test design method that provides low test redundancy and high fault detection for an SPL?

Yes, we can generate small test-suites for the full fault coverage (strong test criterion) in either test design approach developed in both research directions: the product-centered and the product line-centered. The drawback is that it requires the representation of the system(s) behavior using FSM-based specifications. In summary, according to our experimental results, our first test design approach favors the testing of few SPL products, while the second test design approach favors a large number of valid configurations that require testing.

2.5 Summary of Contributions

This thesis focuses on a functional model-based testing approach for SPLs to reduce test-case redundancy using the full fault coverage in two directions. In the first direction, we explore a test-case reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL). The IRT-SPL can reduce test costs of a new derived product in the advanced development stages based on regression testing and the P method [98]. This research direction enables the efficient reuse of test-cases where only a few existing test-cases are selected and incremented to test a new derived product using fewer resources. The IRT-SPL strategy has three main contributions:

- an incremental test-case reuse strategy;
- a test-case selection algorithm; and
- experimental evaluation using a case study of Mobile Media SPL.
2.5. SUMMARY OF CONTRIBUTIONS

In the second direction, we explore test design in the early development stages. Specifically, we propose a solution named Configurable Feature-based full Coverage testing of State Machines (CFC-SM). CFC-SM is a model-based test design approach that enables the automatic generation of reusable and customizable test artifacts for an SPL, which can be configured to test a set of products. The test engineer uses CFC-SM to create configurable feature-based state machines and generates test-cases for the full fault coverage criteria, specifically extending the HSI method [69]. The CFC-SM approach has some contributions:

- proposing new configurable family-based test models for SPL;
- proposing family-based validation criteria for the full fault coverage criteria and proving them to coincide with their product-based counterparts;
- proposing the extension of the HSI test-case generation method and proving them to coincide with their product-based counterparts;
- implementing a model-based test generation tool with a graphical interface to support validation, derivation, and generation of family-based test artifacts.
- experimentally evaluating using random models and the automotive Body Comfort System SPL as a case study.
Part II

Papers
Reducing the Concretization Effort in FSM-Based Testing of Software Product Lines

Vanderson Hafemann Fragal (vanderson.fragal@gmail.com)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

Adenilso Simao (adenilso@icmc.usp.br)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

André Takeshi Endo (andreendo@utfpr.edu.br)
Federal Technological University of Paraná - Brazil

Mohammad Reza Mousavi (mm789@le.ac.uk)
Department of Informatics, University of Leicester, UK.

Abstract

To test a Software Product Line (SPL), the test artifacts and the techniques must be extended to support variability. In general, when new SPL products are developed, more tests are generated to cover new or modified features. A dominant source of extra effort for such tests is the concretization of newly generated tests. Thus, minimizing the amount of new non-concretized tests required to perform conformance testing on new products reduces the overall test effort. In this paper, we propose a test reuse strategy for conformance testing of SPL products that aims at reducing test effort. We use incremental test generation methods based on finite state machines (FSMs) to maximize test reuse. We combine these methods with a selection algorithm used to
identify non-redundant concretized tests. We illustrate our strategy using examples and a case study with an embedded mobile SPL. The results indicate that our strategy can save up to 36% of test effort in comparison to current test reuse strategies for the same fault detection capability.

**Keywords:** Conformance Testing, Test Case Reuse, Model-Based Testing, Finite State Machine, Software Product Line.

### 3.1 Introduction

Software Product Lines (SPLs) address variability by capturing various software features in an organized structure. The delta-oriented approach [67] is a software product line engineering approach, where the SPL is designed in terms of a core module and a set of delta modules. The core module is a set of features for a basic product and the delta modules add, remove, or modify features from the core module to design new products. Changing the specification of a product or deploying a similar product may require substantial effort for conformance testing. We study this problem in the context of Model-Based Testing (MBT), where models are used to steer the test process effort with the goal of making it more structured and more efficient.

One important step in the MBT test process is concretization of abstract test cases [102]. Generated abstract tests are augmented with concrete implementation-specific data making them executable in the system under test. Checking the conformance of an SPL product requires extra effort for each new test that needs to be concretized. According to case studies [102, 40], the cost to manually concretize a test case is several times (around 200) greater than the cost of executing the same concretized test. To tackle this problem, adapters can be developed to automate the concretization process. However, the adapters often need to be modified for new versions/products. For example, systems that constantly evolve (e.g., graphical user interface systems) cannot afford to update adapters of each new version of the system, which often take more time than manually testing the system in the first place [26].

In this paper, we propose a test reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL) that aims at reducing the test effort of newly designed SPL products by reducing the number of new tests that need to be concretized for conformance testing. To this end, we maximize the reuse of tests by processing concretized tests of all old products and incrementing some of them to obtain a small set of tests to concretize. We use finite state machines as test models, which are fundamental semantic models for reactive systems [11].

The contributions of this paper are: (i) the test reuse strategy, (ii) the test case selection algorithm, and (iii) their practical evaluation against other common approaches using an SPL case study. Figure 3.1 presents an overview of contributions. We improve the reuse of tests by incrementing existing concretized tests from all old products for the new behavior. In this figure, Product 5 is the one that is recently developed and hence, new test cases need to be generated and concretized for its test suite. To this end, we defined a new test case selection and reuse strategy that takes the set of all
3.2 Background

In this section, we present basic definitions regarding SPLs, FSMs and some of their test properties, and finally, the concretization effort analysis.

3.2.1 Software Product Lines

A feature is an atomic unit used to differentiate the products of an SPL. Let $F$ be the set of features. The feature structure can be represented by a feature diagram [93]. In
a feature diagram, some notational conventions are used to represent commonalities and variabilities of an SPL (e.g., mandatory, optional, and alternative features). A product configuration $p$ is defined by a set of features $p \subseteq F$ that the conjunction of literals expressed as Boolean variables satisfies all feature constraints. To illustrate the concepts throughout the paper, we use the following SPL.

**Example 8.** The Arcade Game Maker (AGM) [95] can produce arcade games with different game rules. Figure 3.2 shows the feature diagram of AGM. There are three alternative features for the game rule (Brickles, Pong, and Bowling) and one optional feature (Save) to save the game.

### 3.2.2 Finite State Machines

FSMs, as defined below, have been traditionally used for modeling and model-based testing different types of reactive hardware [57] and software systems [48, 42, 106].

**Definition 3.2.1.** FSM Model. An FSM $M$ is a 7-tuple $(S, s_0, I, O, D, \delta, \lambda)$, where $S$ is a finite set of states with the initial state $s_0$, $I$ is a finite set of inputs, $O$ is a finite set of outputs, $D \subseteq S \times I$ is a specification domain, $\delta : D \rightarrow S$ is a transition function, and $\lambda : D \rightarrow O$ is an output function.

A tuple $(s, x) \in D$ determines uniquely a defined transition of $M$. A sequence $\alpha = x_1, ..., x_k, \alpha \in I^*$ is a defined input sequence at state $s \in S$, if there exist states $s_1, ..., s_{k+1} \in S$, where $s = s_1$ such that $(s_i, x_i) \in D$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq k$. Notation $\Omega(s)$ is used to denote all defined input sequences for state $s \in S$ and $\Omega_M$ denotes $\Omega(s_0)$. We extend the transition function from input symbols to defined input sequences, including the empty sequence $\varepsilon$, assuming $\delta(s, \varepsilon) = s$ for $s \in S$. Given a set $C \subseteq \Omega(s) \cap \Omega(s')$, states $s$ and $s'$ are $C$-equivalent if
3.2. BACKGROUND

\[ \lambda(s, \gamma) = \lambda(s', \gamma) \text{ for all } \gamma \in C. \]
Otherwise, if there exists a sequence \( \gamma \in C \) such that \( \lambda(s, \gamma) \neq \lambda(s', \gamma) \), then \( s \) and \( s' \) are \( C \)-distinguishable.

According to Definition 3.2.1, \( M \) is deterministic. If \( D = S \times I \), then \( M \) is a complete FSM; otherwise, it is a partial FSM. An FSM \( M \) is said to be initially connected, if for each state \( s \in S \), there exists an input sequence \( \alpha \in \Omega_M \), such that \( \delta(s_0, \alpha) = s \), \( \alpha \) is called a transfer sequence for state \( s \). Unreachable states need to be removed to make the FSM valid for test case generation. An FSM \( M \) is minimal (or reduced), if every pair of distinct states \( s, s' \in S \) can be distinguished by a set \( C \subseteq \Omega(s) \cap \Omega(s') \).

**Example 9.** There are six possible product configurations that can be derived from the AGM FM. The FSM \( M_3 \) of the third configuration is shown in Figure 3.3. This test model is an abstracted version of the design model where observable events are represented by inputs and the correspondent outputs. The inputs are in-game commands, while the outputs 0 and 1 are abstract captured responses. We selected the Pong\([N]\] rule and discarded the Save\([S]\] option. It is straightforward to check that \( M_3 \) is a deterministic, complete, initially connected and minimal FSM.

### 3.2.3 Test Properties

In this paper, we adapt the P method, a test generation method based on fault domain for FSMs [98]. We use the notion of test suite completeness with respect to a given fault domain and sufficiency conditions based on convergence and divergence properties introduced in [98].

Given sequences \( \alpha, \beta, \gamma \in I^* \), a sequence \( \alpha \) is prefix of a sequence \( \beta \), denoted by \( \alpha \leq \beta \), if \( \beta = \alpha \gamma \), for some sequence \( \gamma \), and \( \gamma \) is a suffix of \( \beta \). A sequence \( \alpha \) is proper prefix of \( \beta \), \( \alpha < \beta \), if \( \beta = \alpha \omega \) for some \( \omega \neq \varepsilon \). We denote by \( \text{pref}(\beta) \) the set of prefixes of \( \beta \), i.e., \( \text{pref}(\beta) = \{ \alpha | \alpha \leq \beta \} \). For a set of sequences \( A \), \( \text{pref}(A) \) is the union of \( \text{pref}(\beta) \) for all \( \beta \in A \). If \( A = \text{pref}(A) \), then we say that \( A \) is prefix-closed. Moreover, we say that a sequence \( \alpha \in A \) is maximal in \( A \) if there is no sequence \( \beta \in A \) such that \( \alpha \) is a proper prefix of \( \beta \).

**Definition 3.2.2.** [98] Test case and test suite. A defined input sequence of FSM \( M \) is called a test case of \( M \). A test suite of \( M \) is a finite prefix-closed set of test cases of \( M \).
A set $C \subseteq \Omega_M$ is a state cover for an FSM $M$ if, for each state $s \in S$, there exist sequences $\alpha \in C$ such that $\delta(s_0, \alpha) = s$. The set $C \subseteq \Omega_M$ covers a transition $(s, x)$ when there exist sequences $\alpha \in C$ such that $\delta(s, \alpha) = s$ and $\alpha x \in C$. The set $C$ is a transition cover (for $M$) if it covers every defined transition of $M$. A set of sequences is initialized if it contains the empty sequence.

Throughout this paper, we assume that $M = (S, s_0, I, O, D, \delta, \lambda)$ and $N = (Q, q_0, I, O', D', \Delta, \Lambda)$ are a specification and an implementation, respectively. Moreover, $n$ is the number of states of $M$. We denote by $\mathcal{Z}$ the set of all deterministic FSMs (different from $M$) with the same input alphabet as $M$ for which all sequences in $\Omega_M$ are defined, i.e. for each $N \in \mathcal{Z}$ it holds that $\Omega_M \subseteq \Omega_N$. The set $\mathcal{Z}$ is called a fault domain for $M$ and $\mathcal{Z}_n$ is the set of FSMs from $\mathcal{Z}$ with $n$ states.

The distinguishability of FSMs is defined as the corresponding relation of their initial states. Thus, test cases are assumed to be applied in the initial state. Given a test suite $T$, $k$ FSMs are $T$-equivalent if for every test case of $T$ all $k$ FSMs return the same output sequence. Let $N \in \mathcal{Z}$ and $\mathcal{Z}(T)$ be the subset from $\mathcal{Z}$ such that $N$ and $M$ are $T$-equivalent, and $\mathcal{Z}_n(T) = \mathcal{Z}_n \cap \mathcal{Z}(T)$ be the set of FSMs from $\mathcal{Z}$ in which are $T$-equivalent to $M$ and have at most $n$ states.

The main results from [98] establish the full fault coverage criterion based on convergence and divergence properties. Convergent test cases reach the same state of $M$, while divergent test cases reach different states in $M$. A test suite $T$ is $n$-complete for an FSM $M$ with $n$ states if $T$ contains a specific convergence-preserving transition cover set for $M$. Two convergent test cases of $T$ satisfy the convergence-preserving property if they are convergent in all FSMs of the set $\mathcal{Z}_n(T)$. When a test suite $T$ is $n$-complete for an FSM $M$ (satisfies the full fault coverage criterion), then, by executing $T$ we are capable of detecting all faults in any FSM implementation $N \in \mathcal{Z}_n(T)$.

There exist several methods to generate $n$-complete test suites [14, 69, 98]. For example, the P method [98] uses two input parameters: a deterministic, initially connected, and minimal FSM $M$; and an initial test suite $T$. The initial set $T$ can be empty, and new test cases are added/incremented (if necessary) until an $n$-complete test suite for $M$ is produced. Therefore, the P method checks if all implementations $N \in \mathcal{Z}_n$ can be distinguished from $M$ using $T$, and decides if more sequences need to be added to $T$. Experimental evaluation indicates that the P method often results in smaller $n$-complete test suites compared with other methods [31].

The reuse of test cases is important to save test effort in several domains that develop similar systems. In this paper, we demonstrate how this can be exploited in the testing of SPLs.

### 3.2.4 Concretization Effort

Given a new product configuration to test, first, we check the changed behavior to ensure that they behave as intended by concretizing and executing a set of new test cases. Then, to ensure that the unchanged behavior was not affected by modifications we execute a set of test cases to retest such behavior.
### Definition 3.2.3. Test case and test suite size. The size of a test case $\alpha \in I^*$ denoted by $|\alpha|$ is calculated by the number of inputs that it contains, i.e., $|\alpha| = k, \alpha = x_1, ..., x_k$. Similarly, $|T|$ is the size of a test suite $T$ calculated by the sum of all test cases plus the reset operation for each maximal test case, i.e., $|T| = \sum (|\alpha| + 1), \alpha \in T, \not\exists \beta \in T \bullet \beta = \alpha \gamma \land \gamma \neq \epsilon$.

**Example 10.** Given a test suite $T = \{a, b, c; a, c\}$ the size of $T$ is $|T| = (|a, b, c| + 1) + (|a, c| + 1) = 7$.

**Definition 3.2.4.** Test effort. The effort to test a new product configuration is the sum of test cases that have to be executed plus those that have to be concretized times a value $x$ (manual concretization value over execution), i.e., $\text{effort} = (\text{concrete} \times x) + \text{execution}$.

Case studies [102, 40] show that the value $x$ is around 200. Execution cost is calculated based on the number of test cases that have to be executed for both changed and unchanged behavior. Concretization cost is calculated from new test cases.

**Definition 3.2.5.** Execution and concretization costs. Given a prefix-closed test suite $T$, a set of new test cases $NT \subseteq T$ and a set of concretized test cases $D = T \setminus \{NT\}$, the execution cost is equivalent to the size of $|T|$, i.e., $\text{execution} = |T|$. The concretization cost is calculated from new test cases $\alpha \in NT$ that have to be concretized. If a proper prefix $\beta$ of a new test case $\alpha = \beta \gamma, \gamma \neq \epsilon$ was already concretized before, i.e., $\beta \in D$, then we reuse $\beta$ and the concretization cost is the sum of the size of all suffixes $\gamma$, i.e., $\text{concrete} = \sum |\gamma|$.

**Example 11.** Given a test suite $T = \text{pref} \{a, b, c; a, c; a, d\}$ and a subset of new test cases $NT = \{a, d\}$, the concretization cost is $\text{concrete} = |d| = 1$.

### 3.3 Testing Products Incrementally

In this section, we present our test reuse strategy and the selection algorithm.

#### 3.3.1 Test Reuse Strategy

The Incremental Regression-based Testing for Software Product Lines (IRT-SPL) strategy was inspired by earlier approaches in this domain [98, 105, 12, 65] and developed to improve the reuse of tests cases to reduce concretization effort. We use incremental test generation methods (to increment concretized tests) for full fault coverage criterion explained in Section 3.2.3. Figure 3.4 (a) presents the main steps of IRT-SPL.

Given a new product configuration $p \in F$ that require testing, to check its conformance we design the test model as an FSM $M$, obtain all defined test cases $D \subseteq \Omega_M$ that were concretized in old products, and execute the following sequence of steps and conditions:
Our Test Reuse Strategy

Select tests using our algorithm
All concretized tests \( D \) for \( M \)
Small set of non-concretized tests \( NT \)
Selected concretized tests \( \mathcal{R} \)

Condition 1: Is set \( D \) \( n \)-complete for \( M \)? When the answer is false, move to Step 2; otherwise, copy \( D \) to set \( T \) and move to Step 3.

Step 2: Increment test cases using a test generation method. Incremental test generation methods use a cost calculation that decides which new test case gives a small increment based on test cases of \( D \). Thus, we used the P method \([98]\) for this step. New test cases are incremented from \( D \) and put in \( NT \), resulting in an \( n \)-complete test suite \( T = D \cup NT \).

Step 3: Obtain test cases using our selection algorithm. Execute the selection algorithm using \( M \) and \( T \) as parameters, obtain the \( n \)-complete test suite \( S \subseteq T \) and return \( R = S - NT \) as the set of selected concretized test cases and \( NT \) as the set of non-concretized test cases.

In general, incremental test case generation algorithms check available test cases that can be incremented. Sometimes two equivalent test cases (with the same size for a given test criteria) can be used to increment and generate a new test case. Selecting one of such tests might result in a larger/smaller test suite at the end of the process. Thus, we pick the first test case (greedy) as generating a minimal test suite (even by incrementing test cases) can lead to an exponential number of situations.

### 3.3.2 Selection Algorithm

The selection algorithm proposed was developed on the last step of our IRT-SPL strategy. Given an FSM \( M \), an \( n \)-complete test suite \( T \) for \( M \), and an initialized
3.3. TESTING PRODUCTS INCREMENTALLY

convergence-preserving transition cover set $O \subseteq T$, we select non-redundant test cases of $O$ to obtain the $n$-complete test suite $S \subseteq T$. Set $O$ is the one that controls model coverage regarding the full fault coverage which might contain redundant tests. Each test in $O$ requires a set of tests cases in $T$ to maintain the set property considering $M$. Thus, we select a subset $G \subseteq O$ resulting set $S \subseteq T$.

The main steps are:

**Step 1:** Identify all redundant test cases. All test cases of $O$ that cover each transition of $M$ are identified. Also, the resulting test suite $S$ and set $G \subseteq O$ are initialized.

**Condition 1:** Is set $S$ $n$-complete for $M$? The set $G \subseteq O$ must be a transition cover set with a convergence-preserving property [98]. When the answer is true, return $S$; otherwise, move to Step 2.

**Step 2:** Select one transition $t$ of $M$. Only transitions not covered by $G$ are selected. Select a test case $a \in O \setminus G$ that covers $t$. Sometimes there are several redundant test cases in $O$ that cover $t$. Then, select the test case that gives the smallest increment of test cases for $S$. For equivalent test cases (which lead to the same size) we just select the first in the line (greedy).

**Step 3:** Given a test case $a$ from Step 2, identify the set of related test cases $E \subseteq T$ required to cover $t$. Then, $a$ is included in $G$ and $E$ in $S$.

**Theorem 2.** Given an $n$-complete test suite $T$ for an FSM $M$, the selection algorithm terminates with an $n$-complete test suite $S \subseteq T$.

An extended version of this paper with the proof and detailed analysis on the selection algorithm can be found elsewhere.¹

**Example 12.** Assume that two products were already tested for Brickles, with and without the Save option. Figure 3.5 shows four test case sets generated by IRT-SPL for the third product presented in Example 9: (a) defined test cases $D$ for $M_3$ that were already concretized before; (b) $n$-complete test suite $T$ for $M_3$ generated by P method by incrementing $D$; (c) a selected $n$-complete test suite $S \subseteq T$ for $M_3$ generated by our selection algorithm; and (d) test case set $R$ for retest unchanged behavior. Test cases were simplified for readability and each input corresponds to: (i) SG - Start; (ii) PS - Pause; (iii) EX - Exit; and (iv) SV - Save.

Note that the difference between (a) and (b) is the addition of Line 13 on (b) and only the last inputs are in bold. Since all four test case sets are prefix-closed, every prefix is also a test case to be counted. Notice that the prefix $SG, PS, PS$ of Line 13 (b) is already present on Line 5 as a prefix that can be reused. On (c), the algorithm to select concretized test cases is executed using as input $M_3$ and (b), such that it keeps new test cases and reduce the test cases set of (a) as some of them are redundant to cover the unchanged behavior. Thus, the set of new test cases is composed by Line 8 (c) (i.e., SG,PS,PS,PS) and the test case set for the retest is (d).

¹The detailed algorithm can be found in
https://drive.google.com/open?id=0B8ERV2K1aAvvcm1Y2dIz1ZMUE
3.4 Experimental Study

To evaluate the applicability of IRT-SPL and the reuse efficiency of our selection algorithm, we conducted a study to compare the effort required to test new products to other test case reuse strategies. Our research question is: How much test effort can be saved using IRT-SPL to test a set of new SPL products compared to existing test case reuse strategies?

3.4.1 Experimental Setup

The setup of our experiment consists of designing several SPL products in different orders and comparing the total effort required to test all products. We compare the...
effort of IRT-SPL to other reuse strategies found in the literature. A survey on some approaches [30, 79, 13, 105, 12, 65] showed two generic reuse strategies, which are described as follows:

1. The first reuse strategy TSPL was named after the FSM-TSPL approach from Capellari et al. [12]. They only reuse test cases of the last product for conformance testing of a new product where they increment test cases.

2. The second reuse strategy DIATP was named after the Delta-oriented Incremental Architecture-based Testing Process found in the approach of Lochau et al. [65]. They reuse test cases of all previous products selecting test cases to verify the unchanged behavior of the new product, but they generate new test cases for the changed behavior without the increment of test cases.

To obtain a fair comparison on the effort of our strategy, TSPL, and DIATP we setup similar environments also using the P method for TSPL and DIATP.

The embedded camera Mobile Media SPL [35] was used to compare all three reuse strategies. The Mobile Media SPL contains several features, such as photo manipulation, music, and videos on mobile devices which results in total 56 product configurations where we selected 24 relevant product configurations. Figure 3.6 (a) presents the feature diagram with three Or features (Photo(MP), Music(MM), and Video(MV)) and three optional (Favourites(F), Copy Media(CM), and SMS Transfer(SMS)) used to characterize all possible configurations of the SPL. Figure 3.6 (b) presents 24 configurations of Mobile Media used to design corresponding products. Note that each product of the order 1 to 24 increases the number of features compared to the previous products.

For each product, an FSM was modeled with varying number of states from three to six, and with fixed eight inputs and two outputs. All FSMs share some properties: complete, deterministic, reduced, and initially connected.

### 3.4.2 Analysis of Results and Discussion

The collected data from the experiments is shown in Figure 3.7. The variable \( x \) is the manual concretization value (from literature \( x = 200 \)) from the effort formula \( \text{effort} = (\text{concrete} \ast x) + \text{execution} \) presented in Section 3.2.4. On (a) and (b) we have the derivation of products with an increasing number of features when \( x = 10 \) and \( x = 100 \), respectively. We pick values 10 and 100 to check how it scales for low and high values of \( x \). On (c) and (d) we have the derivation of products with decreasing number of features when \( x = 10 \) and \( x = 100 \), respectively. Finally, on (e) and (f) we have the derivation of products with a random number of features when \( x = 10 \) and \( x = 100 \), respectively.

We noticed that the total effort required to test the Mobile Media SPL vary according to the value \( x \). First, noticed that for our case study the effort saving percentage when \( x = 100 \) and \( x = 200 \) is approximately the same and when \( x \) is below 10 our approach does not provide significant saving. Also, the number of newly designed
products should be considered as for few products there is no significant difference of effort below 4 product configurations. Moreover, only a few design product random orders were considered and relatively small feature models. These are threats to validity of this study.

Our approach assumes new products to be similar to old ones. In the worst case scenario, the new product to be tested is completely different from any product developed before. However, in the SPL approach, this is unlikely as commonalities are propagated throughout the entire family.

In summary, we conclude that our approach provides effort savings starting from 5% when we have more than 4 product configurations and the $x$ value is over 10 and goes up to 36% when we have 24 product configurations and the $x$ value is over 100 compared to other test reuse strategies.

The random order the time to execute every strategy depends on the complexity of the P method. Traditional test case generation methods for the full fault coverage that use FSMs as test models (e.g. W [14], HSI [69]) are not incremental. Thus, they cannot increment test cases for new specifications based on old test cases to improve reuse. However, generated new test cases can be compared to old test cases for reuse resulting in a weaker reuse strategy as some of these new test cases may be equivalent to existing old test cases.

Figure 3.6: (a) Mobile Media feature model; and (b) derived products from Mobile Media SPL.
Figure 3.7: Accumulated effort per designed product when concretization cost is $x$ times the cost of execution: (a) the increment of features for $x = 10$; (b) the increment of features for $x = 100$; (c) the decrement of features for $x = 10$; (d) the decrement of features for $x = 100$; (e) random features for $x = 10$; and (f) random features for $x = 100$. 
Our experiment has been limited mostly to flat FSMs with few states and few new products. One of the issues regarding the P method is the increasing time required to generate test cases based on the number of states times inputs plus the size of the test input set to start with. Both of these issues (few subjects and time) are threats to the validity of our results for real-world cases. We plan to mitigate these threats by analyzing some realistic case studies as a benchmark for our future research. Realistic FSMs use hierarchy to sustain scalability. Thus, an extension of FSM-based test generation methods for full fault coverage is required. We plan to investigate this further in the near future.

3.5 Related Work

Much recent research has been devoted to developing efficient testing techniques for SPLs by exploiting variability in a systematic manner; we refer to [34, 77, 100] for recent surveys of the field.

There are several incremental test approaches [30, 79, 105, 12, 3, 111, 108] devoted to generating, reusing, and optimizing test suites for SPLs. El-Fakih et al. [30] adapted FSM-based test generation methods for conformance testing that allow the generation of test cases only for the modified parts of an evolving specification. Pap et al. [79] extended their work and designed a bounded incremental algorithm that maintains two sets based on the HSI method [69]. They utilize existing test cases of the previous version of the system to generate test cases for the modified version. Similarly, Capellari et al. [12] explored the FSM-based Testing of SPLs (FSM-TSPL) testing strategy where the P method is used to design new test cases based on the last product derived. Uzuncaova et al. [105] also developed an incremental test generation approach that uses SAT-based analysis to develop tests suites for every product of an SPL, while Baller and Lochau [3] focused on test suite optimization. Moreover, recent delta-oriented approaches [67, 65, 106] developed regression-based SPL approaches to design and reuse test artifacts.

In contrast to current approaches, our work introduces a test reuse strategy focused on reducing test effort of a set of new SPL products. We analyze concretized test cases of derived products to generate a small set of new test cases to concretize for conformance testing. To our knowledge, there is no proposal that reuse test cases from all previous products to reduce concretization effort for new SPL products using incremental test case generation methods.

3.6 Conclusion

This paper proposed a test reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL) that aims at reducing test effort on checking conformance of several SPL products. Test cases of previously designed products can be efficiently reused for a newly designed product using incremental test case.
3.6. CONCLUSION

generation methods to reduce the number of required test cases for concretization. We assume that manual concretization of test cases (as seen in some case studies [102, 40]) is several times (around 200) more expensive than executing the same test case. Thus, the effort required to test a new product is directly related to concretization costs.

Finite State Machines were used to represent the abstract behavior of the products as test models. To maximize reuse of test cases, all test cases that were concretized before are analyzed and some of them are selected to retest the unchanged behavior of the new product under test. Thus, our strategy also contains a selection algorithm to perform the selection of non-redundant concretized test cases.

To illustrate our strategy, we used examples and a case study of an embedded Mobile Media SPL [35]. The results indicate that our approach can save 5% up to 36% test effort for 24 selected products when the manual concretization cost is 10 up to 100 times, respectively, more expensive than execution compared to current test reuse strategies for the same fault detection capability. We found out that the effort saving percentage stabilizes after 100 times for small specifications due to the small number of tests cases for retest.

The problem and the approach described above are very much inspired by a similar problem and approach in regression testing. Regression testing concerns testing software evolution in time (in versions) while SPL testing is about testing software evolution in space (in features) [34]. Hence, we believe our approaches and the obtained results will also be applicable to the regression testing setting.

As future work, we plan to investigate test models with hierarchy and adapt test generation methods for such models to handle scalability problems. Also, we intend to investigate more studies regarding formal representations of SPLs to perform incremental reuse of test artifacts.
Paper 4

Validated Test Models for Software Product Lines: Featured Finite State Machines

Vanderson Hafemann Fragal\textsuperscript{1} (vanderson.fragal@gmail.com)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

Adenilso Simao (adenilso@icmc.usp.br)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

Mohammad Reza Mousavi\textsuperscript{2} (mm789@le.ac.uk)
Department of Informatics, University of Leicester, UK.

Abstract

Variants of the finite state machine (FSM) model have been extensively used to describe the behaviour of reactive systems. In particular, several model-based testing techniques have been developed to support test case generation and test case executions from FSMs. Most such techniques require several validation properties to hold for the underlying test models. In this paper, we propose an extension of the FSM test model for software product lines (SPLs), named featured finite state machine (FFSM). As the first step towards using FFSMs as test models, we define feature-oriented variants of basic test model validation criteria. We show how the high-level validation

\textsuperscript{1}The work of V. Hafemann has been partially supported by the Science Without Borders project number 201694/2015-8.

\textsuperscript{2}The work of M. R. Mousavi has been partially supported by the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation project number 20140312.
properties coincide with the necessary properties on the product FSMs. Moreover, we
provide a mechanised tool prototype for checking the feature-oriented properties using
satisfiability modulo theory (SMT) solver tools. We investigate the applicability of our
approach by applying it to both randomly generated FFSMs as well as those from a
realistic case study (the Body Comfort System). The results of our study show that for
random FFSMs over 16 independent non-mandatory features, our technique provides
substantial efficiency gains for the set of proposed validity checks.

Keywords: Formal Modelling, Model Validation, Software Product Line, Finite
State Machine.

4.1 Introduction

Motivation. Different forms of finite state machines (FSMs) have been extensively
used as the fundamental semantic model for various behavioural specification lan-
guages and design trajectories. In particular, several test case generation techniques
have been developed for hardware and software testing based on FSMs; an overview
of these techniques can be found in [57, 11, 47]. All FSM-based testing techniques
require the underlying test models to satisfy some basic validation criteria such as
connectedness and minimality.

Software Product Lines (SPLs) [58] are used for systematic reuse of artefacts
and are effective in mass production and customisation of software. However, testing
large SPLs demand substantial effort, and effective reuse is a challenge. Model-Based
Testing (MBT) approaches need to be adapted to the SPL domain (see [77] for a survey
of existing approaches).

There are a few recent attempts [64, 106] to extend the FSM-based testing tech-
niques to SPLs, mostly using the delta-oriented approach to SPL modelling. We are
not aware of any prior work that addresses the basic test model validation criteria for
SPLs at the family-wide level. The present paper aims at bridging this gap. To this
end, we first propose a product-line extension of FSMs, named Featured Finite State
Machine (FFSM). An FFSM unifies the test models of the valid product configurations
in a family into a single model. Our aim is to extend FSM-based test case generation
techniques [82, 98] to generate test suites for groups of SPL products. As the first step
to this end, we define feature-oriented family-based validation criteria that coincide
with the necessary conditions of such test case generation techniques at the product
level.

Our family-based validation criteria are implemented in a tool using Java and the
Z3 [24] tool. A case study from the automotive domain concerning the Body Comfort
System [60] was performed to show the applicability of our criteria and tool. Our
research question is: How large does an FFSM have to be in order to save time in the
validation of the FFSM instead of its valid product FSMs? To this end, we performed
an empirical study on randomly generated FFSMs with various parameters. The results
indicate that for random FFSMs with over 10 independent non-mandatory features,
we have substantial efficiency gains for the set of proposed validity checks.
Contributions. The main contributions of this paper are summarised below:

1. Proposing family-based validation criteria for FSM-based test models and proving them to coincide with their product-based counterparts, and

2. Implementing efficient family-based validation techniques and investigating their applicability by applying them to a large set of examples.

Also as a carrier for these contributions, we propose a feature-oriented extension of FSMs.

Organisation. The remainder of this paper is organised as follows. Section 4.2 presents some preliminary notions and concepts regarding SPL testing and FSMs. Section 4.3 describes the FFSM formalism, the proposed validation properties, and the associated theoretical results. Section 4.4 provides an overview of the implementation used for property checking in Java and Z3. Section 4.5 illustrates the experimental study and the analysis of results. Section 4.6 provides an overview of the related works and a comparison among the relevant approaches in the literature. Section 4.7 concludes the paper and presents the directions of our future work.

4.2 Background

This section recapitulates the basic concepts and definitions of SPLs and FSMs that we are going to use through the rest of the paper.

4.2.1 Software Product Lines

A feature is an atomic unit used to differentiate the products of an SPL. Let $F$ be the set of features. A product $p$ is defined by a set of features $p \subseteq F$. The feature structure can be represented by a feature diagram \cite{93}. In a feature diagram, some notational conventions are used to represent commonalities and variabilities of an SPL (e.g., mandatory, optional, and alternative features). To illustrate the concepts throughout the paper, we use the following SPL.

Example 13. The Arcade Game Maker (AGM) \cite{95} can produce arcade games with different game rules. The objective of the player in any game is to get more points. Figure 4.1 shows the feature diagram of AGM. There are three alternative features for the game rule (Brickles, Pong and Bowling) and one optional feature (Save) to save the game.

In general, not all combinations of features are valid. Dependencies and constraints on feature combinations reduce the power set $\mathcal{P}(F)$ of all potential feature combinations to a subset of valid products $P \subseteq \mathcal{P}(F)$ \cite{5, 22}. A feature constraint is a propositional formula generated by interpreting the elements of the set $F$ as propositional variables. We denote by $B(F)$ the set of all feature constraints.
A product configuration \( \rho \) of a product \( p \in P \) is the feature constraint that uses all features of \( F \), where all features in \( p \) are true, i.e., \( \rho = ( \bigwedge_{f \in p} f ) \land ( \bigwedge_{f \notin p} \neg f ) \).

We denote by \( \Lambda \) the set of all valid product configurations. Given a feature constraint \( \chi \in B(F) \), a product configuration \( \rho \in \Lambda \) satisfies \( \chi \) (denoted by \( \rho \models \chi \)), if the assertion \( \rho \land \chi \) is not false.

Consider a feature model \( FM \), let \( F \) be a set of features extracted from \( FM \). Given \( F = \{ Y, S \} \), we know from the \( FM \) that feature Play\([Y]\) is mandatory and Save\([S]\) is optional; an example feature constraint involving both features is \( (Y \land \neg S) \in B(F) \), which specifies the products in which \( Y \) is included and \( S \) is excluded.

### 4.2.2 Finite State Machine

The classic Finite State Machine (FSM) formalism is often used due to its simplicity and rigour for systems such as communication protocols and reactive systems [11]. In this study, we use the following definition of FSM.

**Definition 4.2.1.** An FSM \( M \) is a 5-tuple \((S, s_0, I, O, T)\), where \( S \) is a finite set of **states** with the **initial state** \( s_0 \), \( I \) is a finite set of **inputs**, \( O \) is a finite set of **outputs**, and \( T \) is a set of **transitions** \( t = (s, x, o, s') \in T \), where \( s \in S \) is the source state, \( x \in I \) is the input label, \( o \in O \) is the output label, and \( s' \in S \) is the target state.

Given an input sequence \( \alpha = (x_1, ..., x_k), x_i \in I, 1 \leq i \leq k \), a **path** from state \( s_1 \) to \( s_{k+1} \) exists when there are transitions \( t_i = (s_i, x_i, o_i, s_{i+1}) \in T \), for each \( 1 \leq i \leq k \). A path \( \nu \) is a 5-tuple \((s_1, \alpha, \tau, \beta, s_{k+1})\), where

1. \( s_1 \in S \) is the source state where the path begins,
2. \( \alpha \in I^* \) is the defined input sequence,
3. \( \tau \in T^* \) is the transition sequence, i.e., \( \tau = (t_1, ..., t_k) \),
4. \( \beta \in O^* \) is the output result, i.e., \( \beta = (o_1, ..., o_k) \)
5. \( s_{k+1} \in S \) is the target state where the path ends.

Notation \( \Omega(s) \) is used to denote all paths that start on state \( s \in S \). \( \Omega_M \) is used to denote \( \Omega(s_0) \).
Figure 4.2: FSM of the first product configuration of AGM.

Test case generation methods such as the Harmonised State Identification (HSI) [82] method, and the Fault Coverage-Driven Incremental (P) [98] method require FSMs with some of the semantic properties defined below.

**Definition 4.2.2.** The following validation properties are defined for FSMs:

1. **Deterministic**: if two transitions leave a state with a common input, then both transitions reach the same state, i.e., \( \forall (s, x, o, s', (s, x, o', s'')) \in T \bullet s' = s'' \);

2. **Complete** (required only by some algorithms): every state has one transition for each input, i.e., \( \forall s \in S, x \in I \bullet \exists o \in O, s' \in S \bullet (s, x, o, s') \in T \);

3. **Initially Connected**: there is a path to every state from the initial state, i.e., \( \forall s \in S \bullet \exists \alpha \in I^*, \tau \in T^*, \beta \in O^* \bullet (s_0, \alpha, \tau, \beta, s) \in \Omega_M \);

4. **Minimal**: all pairs of states are distinguishable, i.e., \( \forall s_a, s_b \in S \bullet \exists (s_a, \alpha, \tau_a, \beta_a, s_a') \in \Omega(s_a), (s_b, \alpha, \tau_b, \beta_b, s_b') \in \Omega(s_b) \bullet \beta_a \neq \beta_b \).

**Example 14.** There are six possible products that can be derived from the AGM FM. The FSM \( M_1 \) of the first configuration is presented in Figure 4.2. This test model is an abstracted version of the design model where observable events are represented by inputs and the correspondent outputs. The inputs are in-game commands, while the outputs 0 and 1 are abstract captured responses. We selected the \( Pong[N] \) rule and discarded the \( Save[S] \) option represented by \( \rho_1 = (G \land V \land R \land C \land A \land M \land L \land Y \land P \land N \land \neg B \land \neg W \land \neg S) \in \Lambda \). It is straightforward to check that \( M_1 \) is a deterministic, complete, initially connected and minimal FSM.

### 4.3 Featured Finite State Machines

A Featured Finite State Machine (FFSM) is an extension of a Finite State Machine (FSM) by annotating states and transitions with feature constraints.

This section presents the basic definitions for FFSMs, followed by the notion of product derivation, and the high-level validation properties required for test case generators.

#### 4.3.1 Basic Definitions

The simplified syntax (with conditions) of an FFSM is defined as follows.
Definition 4.3.1. An FFSM is a 7-tuple \((F, \Lambda, C, c_0, Y, O, \Gamma)\), where

1. \(F\) is a finite set of features,
2. \(\Lambda\) is the set of product configurations,
3. \(C \subseteq S \times B(F)\) is a finite set of conditional states, where \(S\) is a finite set of state labels, \(B(F)\) is the set of all feature constraints, and \(C\) satisfies the following condition:
   \[
   \forall (s, \varphi) \in C \bullet \exists \rho \in \Lambda \bullet \rho \models \varphi
   \]
4. \(c_0 = (s_0, \text{true}) \in C\) is the initial conditional state,
5. \(Y \subseteq I \times B(F)\) is a finite set of conditional inputs, where \(I\) is the set of input labels,
6. \(O\) is a finite set of outputs,
7. \(\Gamma \subseteq C \times Y \times O \times C\) is the set of conditional transitions satisfying the following condition:
   \[
   \forall ((s, \varphi), (x, \varphi'), o, (s', \varphi'')) \in \Gamma \bullet \exists \rho \in \Lambda \bullet \rho \models (\varphi \land \varphi' \land \varphi'')
   \]

The components of FFSM are self-explanatory; the above-given two conditions ensure that every conditional state and every transition is present in at least one valid product of the SPL. A conditional transition from conditional state \(c\) to \(c'\) with conditional input \(y\) and output \(o\) is represented by quadruple \(t = (c, y, o, c')\), or alternatively by \(c \overset{y}{\rightarrow}^o c'\).

Example 15. Figure 4.3 shows the FFSM for the AGM SPL. The notation of a conditional state in the model is \(s(\varphi) \equiv (s, \varphi) \in C\), the transition line by \(x(\varphi)/o \equiv \frac{x, \varphi}{o}\) \(\in Y \times O\), and the operators of feature constraints are denoted by \& (and), || (or), and ! (not). Omitted feature conditions mean that the condition is true, i.e., for states \(s \equiv (s, \text{true}) \in C\), and transitions \(x \overset{y}{\rightarrow}^o (x, \text{true})\).

Next, we define auxiliary definitions on FFSMs that are used to describe transfer sequences; they are subsequently used in expressing the FFSM validation properties.
Definition 4.3.2. Given a conditional input sequence $\delta = (y_1, ..., y_k), y_i \in Y, 1 \leq i \leq k$, a conditional path from conditional state $c_1$ to $c_{k+1}$ exists when there are conditional transitions $t_i = (c_i, y_i, o_i, c_{i+1}) \in \Gamma$, for each $1 \leq i \leq k$. A conditional path $\sigma$ is a 6-tuple $(c_1, \delta, \nu, \gamma, \omega, c_{k+1})$, where:

1. $c_1 \in C$ is the conditional state where the path begins,
2. $\delta \in Y^*$ is the conditional input sequence,
3. $\nu \in \Gamma^*$ is the conditional transition sequence, i.e., $\nu = (t_1, ..., t_k)$,
4. $\gamma \in O^*$ is the output result, i.e., $\gamma = (o_1, ..., o_k)$
5. $\omega \in B(F)$ is the resulting path condition, i.e., $\omega = (\varphi_1, ..., \varphi_{k+1}) \land (\theta_1, ..., \theta_k), y_i = (x_i, \theta_i), c_i = (s_i, \varphi_i)$
6. $c_{k+1} \in C$ is the conditional state where the path ends.

Notation $\Theta(c)$ is used to denote the set of all conditional paths that start at conditional state $c \in C$. $\Theta_{FF}$ is used to denote $\Theta(c_0)$.

We also define a valid transfer sequence that is used to transfer the machine from one conditional state to another.

Definition 4.3.3. Given two conditional states $c, c' \in C$, a conditional input sequence $\delta \in Y^*$ is a valid transfer sequence if there are at least one path $(c, \delta, \nu, \gamma, \omega, c') \in \Theta(c)$ and one product that satisfies the path condition, i.e., $\exists \rho \in \Lambda \bullet \rho \models \omega$.

Example 16. Consider the FFSM of Figure 4.3. Note that a transfer sequence $\delta = (Start, Pause)$ of a conditional path $(Start, \delta, (StartGame \xrightarrow{Start} \text{Brickles}(B), \text{Brickles}(B) \xrightarrow{Pause} \text{PauseGame}), (1, 1), (B), \text{PauseGame}) \in \Theta_{FF}$ has a transfer condition $\omega = (B)$ and only two products can satisfy $\omega$, namely, $\rho_5 = (G \land V \land R \land C \land A \land M \land L \land Y \land P \land B \land \neg N \land \neg W \land \neg S) \in \Lambda$ and $\rho_6 = (G \land V \land R \land C \land A \land M \land L \land Y \land P \land B \land S \land \neg N \land \neg W) \in \Lambda$. Thus, $\omega$ is not satisfied by valid product $\rho_1 = (G \land V \land R \land C \land A \land M \land L \land Y \land P \land N \land \neg B \land \neg W \land \neg S)$.

### 4.3.2 Product Derivation

We define a product derivation operator, reminiscent of the operator in [7, 8], that is parameterised by feature constraints. Given a feature constraint, the product derivation operator reduces an FFSM into an FSM representing the selection of products.

Definition 4.3.4. Given a feature constraint $\phi \in B(F)$ and an FFSM $FF = (F, \Lambda, C, c_0, Y, O, \Gamma)$, if exactly one product $\rho \in \Lambda$ satisfies $\phi$, i.e., $\exists! \rho \in \Lambda \bullet \rho \models \phi$, then the product derivation operator $\Delta_\phi$ induces an FSM $\Delta_\phi(FF) = (S, s_0, I, O, T)$, where:

1. $S = \{s | (s, \varphi) \in C \land \rho \models (\varphi \land \phi)\}$ is the set of states;
2. $s_0 = s, c_0 = (s, \varphi) \in C$ is the initial state;
3. $T = \{(s, x, o, s') | (s, \varphi) \xrightarrow{x, \varphi''_o} (s', \varphi') \in \Gamma \land \rho \models (\varphi \land \varphi' \land \varphi'' \land \phi)\}$ is the set of transitions.
The set of all valid products of $FF$ is the set of all induced FSMs. Figure 4.3 shows the FFSM generated for the AGM SPL that can induce six products. Using the feature constraint $\phi = N \land \neg S$ the FFSM is projected into the FSM presented in Figure 4.2.

### 4.3.3 Validation Properties

To adopt FFSMs as test models, first, we need to validate the product-line-based specification with properties used for FSMs. Next, we define the high-level counterparts of the four basic properties, namely, determinism, completeness, initially connected-ness, and minimality, and show that they coincide with the aforementioned properties for their valid FSM products.\(^3\)

**Definition 4.3.5.** An FFSM $FF$ is **deterministic** if for all conditional states when exists two enabled conditional transitions with the same input for a product $\rho$, then both transitions lead to the same state, i.e., $\forall \rho \in \Lambda \cdot \rho \not\models (\phi \land \phi' \land \phi'' \land \varphi_a \land \varphi_b) \lor s' = s''$.

Next, we state and prove that an FFSM is deterministic when all its valid product FSMs are deterministic.

**Theorem 3.** An FFSM $FF$ is deterministic if and only if all derived product FSMs $\Delta_{\phi}(FF)$ are deterministic.

**Proof.** We break the bi-implication in the thesis into two implications and prove each by contradiction. For the implication from left to right, assume that FFSM $FF$ is deterministic, but there is a derived FSM $\Delta_{\phi}(FF)$ for a product $\rho$ which is non-deterministic; we obtain a contradiction. Let FFSM $FF = (F, \Lambda, C, c_0, Y, O, \Gamma)$ be deterministic and a derived FSM $\Delta_{\phi}(FF) = (S, s_0, I, O, T)$ be non-deterministic for a product $\rho \in \Lambda$ on state $s \in S$. As $\Delta_{\phi}(FF)$ is non-deterministic, then by the negation of Definition 4.2.2 item 1 there is an input $x \in I$ such that two transitions $(s, x, o, s')$, $(s, x, o, s'') \in T$ reach different states $s' \neq s''$. By Definition 4.3.4 item 3 if $\Delta_{\phi}(FF)$ has two transitions $(s, x, o, s')$ and $(s, x, o, s'')$, then both were induced from conditional transitions $(s, \varphi) \xrightarrow{x, \varphi'} (s', \varphi_a)$, $(s, \varphi) \xrightarrow{x, \varphi''} (s'', \varphi_b) \in \Gamma$ of $FF$ and $\rho \models (\varphi \land \varphi' \land \varphi_a \land \varphi_b)$. However, $FF$ is deterministic and by Definition 4.3.5 the condition $\rho \not\models (\varphi' \land \varphi'' \land \varphi_a \land \varphi_b) \lor s' = s''$ holds for all pairs of conditional transitions, which is a contradiction as there is a pair of conditional transitions that the negation of the condition $\rho \models (\varphi \land \varphi' \land \varphi_a \land \varphi_b) \land (s' \neq s'')$ also holds.

Likewise, for the implication right to left, assume that $\Delta_{\phi}(FF)$ is deterministic for $\rho$, but $FF$ is non-deterministic; we obtain a contradiction. Let $FF = (F, \Lambda, C, c_0, Y, O, \Gamma)$ be non-deterministic on conditional state $(s, \varphi) \in C$, $\rho \models \varphi$, and $\Delta_{\phi}(FF) = (S, s_0, I, O, T)$ is deterministic for $\rho$. As $FF$ is non-deterministic, then by the negation of Definition 4.3.5 there is an input $x \in I$ such that two

\(^3\)Due to space limitation proof sketches are provided below; detailed proofs of correctness for these properties is available at [http://ceres.hh.se/mediawiki/Vanderson_Hafemann](http://ceres.hh.se/mediawiki/Vanderson_Hafemann)
conditional transitions \((s, \varphi) \xrightarrow{\varphi'} (s', \varphi_a), (s, \varphi) \xrightarrow{\varphi''} (s'', \varphi_b) \in \Gamma\) are satisfied by \(\rho \models (\varphi \land \varphi' \land \varphi_a \land \varphi_b)\) and reach different states \(s' \neq s''\). As \(\rho \models \phi\) and by Definition 4.3.4 item 3 each transition of \(FF\) that satisfies \(\phi\) is induced to \(\Delta_\phi(FF)\), thus \((s, x, o, s')\), \((s, x, o, s'') \in T\). However, \(\Delta_\phi(FF)\) is deterministic and by Definition 4.2.2 item 1 the condition \(s' = s''\) is true for all pairs of transitions \((s, x, o, s'), (s, x, o, s'') \in T\), which is a contradiction as there is a pair of transitions \((s, x, o, s')\), \((s, x, o, s'') \in T\) such \(s' \neq s''\).

**Definition 4.3.6.** An FFSM \(FF\) is **complete** if for all conditional states in a product there is an outgoing valid transition for each and every input, i.e., \(\forall (s, \varphi) \in C \land \forall \rho \in \Lambda \land \exists x \in I \land \rho \not\models \varphi \lor \exists (x, \varphi') (x, \varphi') \in \Gamma \land \rho \models \varphi' \land \varphi''\).

Next, we state and prove that an FFSM is complete when all its valid product FSMs are complete.

**Theorem 4.** An FFSM is complete if and only if all derived product FSMs are complete.

**Proof.** We break the bi-implication in the thesis into two implications and prove each by contradiction. For the implication left to right, assume that FFSM \(FF\) is complete, but there is a derived FSM \(\Delta_\phi(FF)\) for a product \(\rho\) which is non-complete; we obtain a contradiction. Let FFSM \(FF = (F, \Lambda, C, Y, O, \Gamma)\) be complete and a derived FSM \(\Delta_\phi(FF) = (S, s_0, I, O, T)\) be non-complete for a product \(\rho \in \Lambda\) on state \(s \in S\) for input \(x \in I\). As \(\Delta_\phi(FF)\) is non-complete, then, by the negation of Definition 4.2.2 item 2 there is no transition \((s, x, o, s') \in T\) on \(s\) with input \(x\). By Definition 4.3.6 if \(FF\) is complete, then for all products \(\rho \in \Lambda\) that satisfies a conditional state \((s, \varphi) \in C \land \rho \models \varphi\) and for all inputs \(x \in I\) there are conditional transitions \((s, \varphi) \xrightarrow{\varphi''} (s', \varphi') \in \Gamma\) such \(\rho \models \varphi' \land \varphi''\). However, by Definition 4.3.4 item 3 every conditional transition \((s, \varphi) \xrightarrow{\varphi''} (s', \varphi') \in \Gamma\) in \(FF\) that satisfies \(\rho \models \phi\) induces a transition \((s, x, o, s') \in T\) in \(\Delta_\phi(FF)\), which is a contradiction as \(\Delta_\phi(FF)\) does not have a transition \((s, x, o, s') \in T\) on state \(s\) for input \(x\).

Likewise, for the implication right to left, assume that \(\Delta_\phi(FF)\) is complete for \(\rho\), but \(FF\) is non-complete; we obtain a contradiction. Let \(FF = (F, \Lambda, C, Y, O, \Gamma)\) be non-complete on conditional state \((s, \varphi) \in C\) for input \(x \in I\), \(\rho \models \varphi\), and \(\Delta_\phi(FF) = (S, s_0, I, O, T)\) is complete for \(\rho\). As \(FF\) is non-complete, then by the negation of Definition 4.3.6 on conditional state \((s, \varphi) \in C\) there is no conditional transition \((s, \varphi) \xrightarrow{\varphi''} (s', \varphi') \in \Gamma\) with input \(x \in I\) for \(FF\), or it exists but is not satisfied \(\rho \not\models \varphi' \land \varphi''\). By Definition 4.3.4 item 3 if a conditional transition \((s, \varphi) \xrightarrow{\varphi''} (s', \varphi') \in \Gamma\) does not exist in \(FF\), or it exists but \(\rho \not\models \varphi' \land \varphi''\), then there is no transition \((s, x, o, s') \in T\) induced in \(\Delta_\phi(FF)\). However, \(\Delta_\phi(FF)\) is complete and by Definition 4.2.2 item 2 for all states \(s \in S\) and for all inputs \(x \in I\) there
are transitions \((s, x, o, s') \in T\), which is a contradiction as there is no transition \((s, x, o, s') \in T\) in \(\Delta_\phi(FF)\) for state \(s\) and input \(x\).

**Definition 4.3.7.** An FFSM \(FF\) is **initially connected** if there exist transfer sequences from the initial conditional state to every conditional state for every satisfiable product, i.e., \(\forall c=(s, \varphi) \in C \cdot \forall \rho \in \Delta \cdot \rho \models \varphi \implies \exists (c_0, \delta, \nu, \gamma, \omega, c) \in \Theta_{FF} \cdot \rho \models \omega\).

Next, we state and prove that an FFSM is initially connected when all its valid product FSMs are initially connected.

**Theorem 5.** An FFSM is initially connected if and only if all derived product FSMs are initially connected.

**Proof.** We break the bi-implication in the thesis into two implications and prove each by contradiction. For the implication left to right, assume that FFSM \(FF\) is initially connected, but there is a derived FSM \(\Delta_\phi(FF)\) for a product \(\rho\) which is non-initially connected; we obtain a contradiction. Let FFSM \(FF = (F, \Lambda, C, c_0, Y, O, \Gamma)\) be initially connected and a derived FSM \(\Delta_\phi(FF) = (S, s_0, I, O, T)\) be non-initially connected for a product \(\rho \in \Lambda\) on state \(s_k \in S\). As \(\Delta_\phi(FF)\) is non-initially connected, then, by the negation of Definition 4.2.2 item 3 there is no path \(v \in \Omega_{\Delta_\phi(FF)}\) to \(s_k\) from the initial state \(s_0\). By Definition 4.3.7 if \(FF\) is initially connected, then there is a path \(\sigma_k \in \Theta_{FF}\) to every conditional state \((s_k, \varphi_k) \in \Omega\) from the initial conditional state \(c_0\), and \(\rho\) satisfies the path condition \(\omega\). However, by Definition 4.3.3 every conditional transition \((s_i, \varphi_i) \xrightarrow{(x_i, o_i)} (s_{i+1}, \varphi_{i+1}) \in \Gamma, 0 \leq i \leq k\) forms a path to reach \((s_k, \varphi_k)\) which is satisfied by \(\rho\). As \(\rho \models \varphi\), and by Definition 4.3.4 item 3 every conditional transition \((s_i, \varphi_i) \xrightarrow{(x_i, o_i)} (s_{i+1}, \varphi_{i+1}) \in \Gamma\) is induced to \((s_i, x_i, o, s_{i+1}) \in T\) that forms a path to reach \(s_k\), which is a contradiction as there is no path for \(v \in \Omega_{\Delta_\phi(FF)}\) to reach state \(s_k\).

Likewise, for the implication right to left, assume that \(\Delta_\phi(FF)\) is initially connected for \(\rho\), but \(FF\) is non-initially connected; we obtain a contradiction. Let \(FF = (F, \Lambda, C, c_0, Y, O, \Gamma)\) be non-initially connected on conditional state \((s, \varphi) \in C\), \(\rho \models \varphi\), and \(\Delta_\phi(FF) = (S, s_0, I, O, T)\) is initially connected for \(\rho\). As \(FF\) is non-initially connected, then by the negation of Definition 4.3.7 there is no path \(\sigma \in \Theta_{FF}\) to reach \((s_k, \varphi_k)\) from the initial conditional state \(c_0\). By Definition 4.2.2 item 3 if \(\Delta_\phi(FF)\) is initially connected, then there is a path \(v \in \Omega_{\Delta_\phi(FF)}\) to reach every state \(s \in S\) from the initial state \(s_0\). As \(\rho \models \varphi\), and by Definition 4.3.4 item 3 every transition \((s_i, x_i, o, s_{i+1}) \in T\) was induced from a conditional transition \((s_i, \varphi_i) \xrightarrow{(x_i, o_i)} (s_{i+1}, \varphi_{i+1}) \in \Gamma\) and \(\rho \models \varphi_i \land \varphi_i' \land \varphi_{i+1}\) that forms a path to reach \((s_k, \varphi_k)\), which is a contradiction as there is no path \(\sigma \in \Theta_{FF}\) to reach \((s_k, \varphi_k)\).

**Definition 4.3.8.** An FFSM \(FF\) is **minimal** if for all pairs of conditional states of all satisfiable products there are common valid transfer sequences that distinguish both conditional states, i.e., \(\forall c_a=(s_a, \varphi_a), c_b=(s_b, \varphi_b) \in C \cdot \forall \rho \in \Delta \cdot \rho \models \varphi_a \land \varphi_b \Rightarrow \exists (c_a, \delta, \nu_a, \gamma_a, \omega_a, c_a') \in \Theta(c_a) \cdot (c_b, \delta, \nu_b, \gamma_b, \omega_b, c_b') \in \Theta(c_b) \cdot \gamma_a \neq \gamma_b \land \rho \models (\omega_a \land \omega_b)\).
Next, we state and prove that an FFSM is minimal when all its valid product FSMs are minimal.

**Theorem 6.** An FFSM is minimal if and only if all its derived product FSMs are minimal.

**Proof.** We break the bi-implication in the thesis into two implications and prove each by contradiction. For the implication left to right, assume that FFSM \( FF \) is minimal, but there is a derived FSM \( \Delta_\varphi(FF) \) for a product \( \rho \) which is non-minimal; we obtain a contradiction. Let FFSM \( FF \) \( = (F, \Lambda, C, c_0, Y, O, \Gamma) \) be minimal and a derived FSM \( \Delta_\varphi(FF) = (S, s_0, I, O, T) \) be non-minimal for a product \( \rho = \Lambda \) on states \( s_a, s_b \in S \). As \( \Delta_\varphi(FF) \) is non-minimal, then, by the negation of Definition 4.2.2 item 4 there is no common input sequence \( \alpha \in I^* \) of two paths \( \nu_a \in \Omega(s_a), \nu_b \in \Omega(s_b) \) that distinguish states \( s_a \) and \( s_b \). By Definition 4.3.8 if \( FF \) is minimal, then for every pair of conditional states \( c_a = (s_{a0}, \varphi_{a0}), c_b = (s_{b0}, \varphi_{b0}) \in C \) and for all products \( \rho \in \Lambda \) that satisfy the condition \( \varphi_{a0} \land \varphi_{b0} \) there are two paths with a common a distinguishing sequence \( \delta \in \gamma^* \) and \( \rho \) also satisfies both path conditions \( \omega_a \land \omega_b \). However, by Definition 4.3.3 every pair of conditional transitions \((s_{ai}, \varphi_{ai}) \xrightarrow{(x_i, \varphi_i')} (s_{a_{i+1}}, \varphi_{a_{i+1}}), (s_{bi}, \varphi_{bi}) \xrightarrow{(x_i, \varphi_i'')} (s_{b_{i+1}}, \varphi_{b_{i+1}}) \in \Gamma, 0 \leq i \leq k \) of the distinguishing sequence \( \delta \) is satisfied by \( \rho \). As \( \rho \models \varphi \), and by Definition 4.3.4 item 3 every pair of conditional transitions \((s_{ai}, \varphi_{ai}) \xrightarrow{(x_i, \varphi_i')} (s_{a_{i+1}}, \varphi_{a_{i+1}}), (s_{bi}, \varphi_{bi}) \xrightarrow{(x_i, \varphi_i'')} (s_{b_{i+1}}, \varphi_{b_{i+1}}) \in \Gamma \) is induced to \((s_{ai}, x_i, o, s_{a_{i+1}}), (s_{bi}, x_i, o', s_{b_{i+1}}) \in T \) in \( \Delta_\varphi(FF) \) that distinguishes \( s_a \) and \( s_b \), which is a contradiction as there is no distinguishing sequence \( \alpha \in I^* \) for states \( s_a \) and \( s_b \).

Likewise, for the implication right to left, assume that \( \Delta_\varphi(FF) \) is minimal for \( \rho \), but \( FF \) is non-minimal; we obtain a contradiction. Let \( FF = (F, \Lambda, C, c_0, Y, O, \Gamma) \) be non-minimal on conditional state \( c_a = (s_{a0}, \varphi_{a0}), c_b = (s_{b0}, \varphi_{b0}) \in C, \rho \models \varphi \), and \( \Delta_\varphi(FF) = (S, s_0, I, O, T) \) is minimal for \( \rho \). As \( FF \) is non-minimal, then by the negation of Definition 4.3.8 there is no common input sequence \( \delta \in \gamma^* \) that distinguish conditional states \( c_a \) and \( c_b \). By Definition 4.2.2 item 4 if \( \Delta_\varphi(FF) \) is minimal, then there are two paths with a common a distinguishing sequence \( \alpha \in I^* \) for every pair of states \( s_a \) and \( s_b \). As \( \rho \models \varphi \), and by Definition 4.3.4 item 3 every pair of transitions \((s_{ai}, x_i, o, s_{a_{i+1}}), (s_{bi}, x_i, o', s_{b_{i+1}}) \in T \) were induced from \((s_{ai}, \varphi_{ai}) \xrightarrow{(x_i, \varphi_i')} (s_{a_{i+1}}, \varphi_{a_{i+1}}), (s_{bi}, \varphi_{bi}) \xrightarrow{(x_i, \varphi_i'')} (s_{b_{i+1}}, \varphi_{b_{i+1}}) \in \Gamma \) and \( \rho \) satisfies both conditional paths that distinguishes \( c_a \) and \( c_b \), which is a contradiction as there is no distinguishing sequence \( \delta \in \gamma^* \) for \( c_a \) and \( c_b \). \( \square \)

### 4.4 Implementation

It is well-known that feature models can be translated into propositional formulas; see, e.g., [5, 20]. This translation enables mechanising the analysis of feature-based specifications using existing logic-based tools, such as SAT solvers. In our approach the Z3 tool [24] was used to check propositional formulas for FFSM properties.
We implemented a tool in Java to parse and process FFSMs in an adapted version of KISS format \cite{29} and subsequently generate assertions in the SMT format that correspond to the initial syntactical checks on the FFSM definition (Definition 4.3.1) and the semantic FFSM validation properties in Section 4.3.3.

To check the initial FFSM conditions on the FFSM of Figure 4.3, we: (i) transform the feature model of Figure 4.1 into a propositional formula; and (ii) generate assertions to check feature constraints of conditional states and transitions. Subsequently, we check the validity conditions on the generated propositional formulae. The validation process is progressive, starting with validating conditional states and transitions, and proceeding with checking determinism, completeness, initially connected and then minimality.

**Example 17.** Figure 4.4 presents parts of the generated SMT files to check validity of conditional states and completeness, where: (a) all features are declared as Boolean variables; (b) the root mandatory and also the feature model propositional formula are asserted; (c) conditional states \textit{Brickles (B)} and \textit{Save (S)} are verified; and (d) a completeness check on the conditional state \textit{Save (S)} for input \textit{Pause} is verified (see Figure 4.3). To check conditional states we combine and execute parts (a), (b), and (c), while to check completeness (a), (b), and (d) are combined and executed. In the end, for every \textit{(check − sat)} command we have an answer that we connect back to Java.

In Z3, \textit{push} and \textit{pop} commands are used to temporarily set the context (e.g., with assertions), and once a verification goal is discharged the context can be reset. The \textit{(check − sat)} command is used to evaluate the assertions which returns \textit{(sat} or \textit{unsat}). If a conditional state check yields \textit{unsat}, then there is no product that will ever have this state and hence, the FFSM is invalid.
4.5 Experimental Study

To evaluate the applicability and the efficiency of our approach, we conducted an experiment to evaluate and compare the time required to check properties of FFSMs with the Product by Product (PbP) approach. Our research question is: How large does an FFSM have to be in order to save time in validation of the FFSM instead of its valid product FSMs? In the future, we plan to use the same setup (extended with more case studies), to evaluate the test case generation methods on FFSMs.

4.5.1 Experimental Setup

The setup of our experiment consists of generating random FFSMs varying the number of conditional states from 8 to 70. Every FFSM uses different types of feature models and the arrangement of the features (structure) defines the number of configurations. Initially, we manually inspected a large sample of generated FFSMs, their underlying FSMs and their validation times.

We also modeled the Body Comfort System (BCS) that is used on the VW Golf SPL [60] to reduce the threats to validity and contrast the results from randomly generated (F)FSMs with their real-world counterparts. The original BCS system has 19 non-mandatory features and can have 11616 configurations. In order to manage its complexity, we picked a subset of the feature model (without unresolved dependencies) with 13 non-mandatory features and 8 independent features at the leafs of the feature model. (We plan to introduce hierarchy into our models and treat the complete example in the future.)

Figure 4.5 shows the feature model of a selected part of features for the BCS. Figure 4.6 shows the FFSM for a small part of this specification featuring the Alarm System (AS) with an optional Interior Monitoring (IM) function; and (ii) the Central Locking System (CLS) with an optional Automatic Locking (AL) function. This FFSM turns out to be deterministic, complete, initially connected and minimal.
The implementation of our experiments is explained in Section 4.4. We also implemented a random generator for feature models and (F)FSMs. We designed the random generator to map features to conditional states of the FFSMs. The number of conditional states is hence designed to be proportional to the number of features. We used FeatureIDE [101] to visualise and inspect the feature models and gain insight into the complexity with respect to their structure. The running environment used Windows 7 (64 bit) on an Intel processor i5-5300U at 2.30GHz.

4.5.2 Analysis and Threats to Validity

The collected data after running our experiments is visualised in Figure 4.7. The total validation time is calculated in milliseconds, and we stopped our experiments around 2 million milliseconds (approximately 30 minutes) for 68 non-mandatory features when checking FFSMs.

As an immediate observation, we noticed that the number of non-mandatory features is the dominant factor in the complexity of validation time in both approaches. This was an early observation that was verified by inspecting several data points and resulted in the way we visualised the data in Figure 4.7.

Additionally, the FSM-based analysis (the product-by-product approach) is very sensitive to the structure of the feature model: the number of independent optional features (optional features appearing in different branches of the feature model) play a significant role in the number of products and hence, the validation time. Thus,
we classified the FSM-based data regarding the relative number of independent non-mandatory features in Figure 4.7; the worst-case time is where all the non-mandatory features are independent; the average case is where half of the non-mandatory features are independent, and the best case is where all non-mandatory features are dependent (form a line in the feature model).

To summarise, we conclude that for random SPLs with more than 16 independent non-mandatory features, the FSM-based approach fails to perform within a reasonable amount of time (e.g., ca. 30 minutes), while the FFSM-based approach scales well (regardless of the feature model structure) for up to 70 non-mandatory features.

Regarding our BCS case study, we have obtained similar results regarding the difference between the FFSM and the FSM-based approaches. Namely, for the resulting FFMSM with 13 non-mandatory features (of which 8 are independent) and 50 conditional states, the validation takes approx. 500 seconds (~8 minutes) while for its 384 configurations (FSMs) we have approx. 700 seconds (~11 minutes). We expect the scalability of the FFSM-based approach to improve if more structural aspects, e.g., hierarchy, are taken into account. We plan to investigate this further in the near future.

Our experiment has been limited mostly to random feature models and (F)FSMs with a given mapping from feature models to FFMSMs. We have only included one realistic case-study for comparison. Both of these issues (the actual structure of feature models and of FFMSMs) are threats to the validity of our results for real-world cases. We plan to mitigate this threat by analyzing a number of realistic case studies as a benchmark for our future research. Regarding feature models, as our current results suggest, the FFMSM-based approach is not very sensitive to the structure of the feature model and hence, our results are not likely to change much for realistic feature models. Regarding realistic FFMSMs, it is common that the flat (i.e., non-hierarchical) FFMSMs are the result of the composition of parallel features and hence, their number of states grows exponentially with the number of independent non-mandatory features.
Hence, for realistic FFSMs, using the hierarchical structure in the validation process is necessary for sustaining scalability.

4.6 Related Work

There have been several proposals for the behavioural modelling of SPLs in the literature; we refer to [9, 21, 91] for recent surveys and Thüm et al.’s recent survey [101] for a classification of different SPL analysis techniques. A number of behavioural models proposed in the literature, e.g., these in [62, 2, 18] are based on Finite State Machines or Labeled Transition Systems. They are mainly used to provide the formal specification of SPLs and their formal verification using model checking.

In Feature-Annotated State Machines, some approaches [20, 43] (e.g. the 150% test model) propose a pruning-based approach to UML modelling of SPLs, separating variability from the base models using mapping models. Similar approaches [62, 68] use Statecharts to model reusable components and in their approaches, the instances can also be derived syntactically by pruning. Recent approaches [52, 17] encode feature annotations into transition guards to project model elements. In [52], the authors use model slicing to generate tests for parts of the model to reduce complexity. In Featured Transition Systems [17], model fragments are annotated with presence conditions, i.e., Boolean expressions that define to which products a fragment belongs. However, in none of these approaches, the authors deal with semantic issues in FSMs/LTSs, such as the validation properties considered in our approach, and only verify the syntactical correctness of possible valid products. Moreover, there is a sizable literature focusing on product-based analysis techniques such as syntactic consistency, type checking and model-checking of SPLs [2, 6, 101].

Our proposed test model and validation criteria can be classified as a family-based and feature-oriented specification and analysis method. To our knowledge, however, there only a few pieces of research that extend test models, test case generation and test case execution to the family-based level; examples of such work include earlier delta-oriented techniques such as [64, 65, 106] and feature-oriented approaches [7, 8, 28]. However, the approach proposed in [7, 8] exploits a non-deterministic test case generation algorithm (with no fault model or finite test suite) and hence, validation of test models is not an issue in their approach. Thus, we are not aware of any prior study one extending the FSM-based test-model validation techniques to the family-based setting.

4.7 Conclusion

In this paper, we presented the Featured Finite State Machine (FFSM) model as a behavioural test model for software product lines (SPLs). Validation properties were specified for adopting FFSMs as input models for test case generation algorithms and we showed that they coincide with their corresponding properties for the product FSM
models. Moreover, a framework for validation of test models using Java and Z3 was implemented.

We conducted an experimental study comparing the validation time of FFSM properties with the accumulated time of validating all FSM product models (both using randomly generated models and a case study for the Body Comfort System). We found that checking collective FFSM models can save significant amount of time for SPLs that have 16 or more independent non-mandatory features.

As future work, we plan to use FFSMs to extend FSM-based test case generation methods to SPLs. Moreover, we plan to extend the FFSM model to Hierarchical FFSMs (using concepts from Statecharts and UML State Machines) to handle the state explosion problem identified in the case study and apply validation (and test case generation) on hierarchical models. Also, in addition to the validation issues, other aspects of the FFSM model can be explored such as applicability, maintainability and the relation between semantic properties such as determinism and minimality.
Paper 5

Extending HSI Test Generation Method for Software Product Lines

Vanderson Hafemann Fragal\textsuperscript{1} (vanderson.fragal@gmail.com)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

Adenilso Simao (adenilso@icmc.usp.br)
Institute of Math. and Computer Sciences - ICMC, University of São Paulo, Brazil.

Mohammad Reza Mousavi\textsuperscript{2} (mm789@le.ac.uk)
Department of Informatics, University of Leicester, UK.

Uraz Cengiz Turker (urazc@sabanciuv.edu)
Computer Engineering Department, Gebze Technical University, Turkey

Abstract

Featured Finite State Machines (FFSMs) were proposed as a modeling formalism that represents the abstract behavior of an entire software product line (SPL). Several model-based testing techniques have been developed to support test case generation for SPL specifications, but none support the full fault coverage criterion for SPLs at the family-wide level. In this paper, we propose an extension of the Harmonized

\textsuperscript{1}The work of V. Hafemann has been partially supported by the Science Without Borders project number 201694/2015-8.

\textsuperscript{2}The work of M. R. Mousavi has been partially supported by the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation project number 20140312.
State Identifiers (HSI) method, an FSM-based testing method supporting full fault coverage. By extending the HSI method for FFMSs we are able to generate a single configurable test suite for groups of SPL products that can be instantiated using feature constraints. We implement a graphical tool named ConFTGen to guide the design, validation, derivation, and test case generation for state, transition, and full fault coverage of FFMSs. Experimental results indicate a reduction of approximately 50% on the number of test cases required to test 20 random SPL products. Also, we investigate the applicability of our method by applying it to a case study from the automotive domain, namely, the Body Comfort System.

**Keywords:** Formal Modeling; Test case Generation; Software Product Line; Featured Finite State Machine.

### 5.1 Introduction

Software Product Line Engineering (SPLE) is a paradigm to develop software where a family of related products (a Software Product Line - SPL) is built out of a common set of core assets, thus reducing the development cost for each individual product [83]. In SPLE, products are built, step-by-step, by adding or removing functionalities, to alleviate software complexity and improve quality.

Similar to the development of single systems, the SPLE process also has several activities that are executed to ensure software quality. Testing is an example of such activities that is performed to ensure quality and minimize risks. Testing activities have a large share of overall project costs and are even more challenging in SPLE than for single systems [99]. In several domains, it is not highly non-trivial to follow development standards to efficiently test various product configurations in a systematic manner. For example, the standard ISO 26262 for safety-critical automotive software states that each developed product configuration should be tested using model-based techniques with a high degree of test coverage under some test criterion [15].

Several techniques, processes, and strategies [104, 112, 76, 67, 77] were developed for testing SPLs, but many problems are still open in this area of research. First of all, testing every single product configuration individually by using common testing techniques is not feasible for large SPLs due to the huge number of possible configurations. Secondly, testing products on-demand is unacceptable, due to the scarce time available for product assembly. Moreover, there are other challenges in SPLE including variability, artifact management, test redundancy, overlay, and gaps [90, 33].

In the Model-Based Testing (MBT) approach, we select a test criterion to design test cases using a behavioral test model. The resulting test suite execution must be able to turn as many faults into failures as possible [50]. Thus, good test criteria are the results of compromises between testing cost and fault detection capability. The full fault coverage is a test criterion with such a balance. There are a number of MBT methods [14, 69, 98] that use the full fault coverage to generate test suites from Finite State Machine.
5.1. INTRODUCTION

Machines (FSMs). We focus on the Harmonized State Identifiers (HSI) method [69] as it is an improvement of the W method [14]. We do not use incremental test generation approaches such as the P method [98] as they are substantially more complex and our product-line-centered approach, does not need such level of complexity. Moreover, the HSI method has better performance than the P method and can generate small test suites compared to other non-incremental methods [31].

In this paper, we propose an extension of the HSI method [69] for Featured Finite State Machines (FFSMs) [37]. We bring the HSI method to the family level, by extending several FSM-based notions of coverage and sequences to the family-level setting. By extending the HSI method, we can generate a single configurable test suite for the SPL. Such test suite can be pruned using feature constraints for a (group of) product configuration(s). A feature constraint can also be used in the derivation of sub-FFSM models for specific sets of SPL product configurations. We implemented a graphical support tool named Configurable Full Test Generator (ConFTGen) to guide the design of FFSMs. The tool performs validation, product model derivation, and test case generation for state, transition, and full fault coverage. The Z3 SMT tool [24] efficiently process feature constraints.

Moreover, we conducted an experimental study with random FFSMs and feature models, and for each feature model we selected 20 random products. We decided to randomize feature models and FFSMs to avoid the bias of specific domains. First, a test suite was generated using the extended HSI method for an FFSM and compared to a second unified test suite generated using the original HSI method for the selected 20 individual configurations. The results indicate a reduction in approximately 50% of the number of new tests required for testing using the first test suite compared to the second. The reduction percentage decrease for less than 20 products and increases for more than 20. Moreover, we illustrate and evaluate our approach and tool by means of a case study from the automotive domain, using the Body Comfort System (BCS) for the VW Golf SPL [60]. The results indicate 25% reduction in the number of new tests required for testing a slice of the BCS for 6 products.

The main contributions of this paper are summarized below:

1. Proposing an extension of the HSI test case generation method and proving it to coincide with its product-based counterpart;

2. Implementing a model-based test generation tool with a graphical interface to support design, validation, derivation, and generation of family-based test artifacts;

3. Experimentally assessing the proposal with both a set of random SPLs and a realistic SPL case study.

The remainder of this paper is organized as follows. Section 5.2 presents some preliminary notions and concepts regarding the HSI method, feature models, and FFSMs. Section 5.3 presents our approach which extends the MBT concepts for a configurable test design including state, transition, and full fault coverage, followed by the HSI
extension. Section 5.4 presents the implementation details of the ConFTGen tool and usage of SMT Solvers. Section 5.5 illustrates the experimental study with random models and result analysis. Section 5.6 provides a case study regarding the Body Comfort System. Section 5.7 provides an overview of the related work and a comparison of relevant approaches in the literature. Section 5.8 concludes the paper and presents the directions of our future work.

5.2 Background

This section recapitulates the basic concepts and definitions of the HSI method, feature models, and FFSMs (mostly from [37]) that we are going to use through the rest of the paper.

5.2.1 Harmonized State Identifier Method

The Harmonized State Identifiers (HSI) method can generate a test suite which attains the full fault coverage to test the behavior of a software system represented by a finite state machine.

5.2.1.1 Finite State Machines

The classic Finite State Machine (FSM) formalism, presented below, is often used due to its simplicity and rigor for modeling systems such as communication protocols and reactive systems [11].

**Definition 5.2.1.** An FSM \( M \) is defined as a 5-tuple \( (S, s_0, I, O, T) \), where \( S \) is a finite set of states, \( s_0 \in S \) is the initial state, \( I \) is the set of inputs, \( O \) is the set of outputs, and \( T \) is the set of transitions in the form of \( t = (s, x, o, s') \in T \), where \( s \in S \) is the source state, \( x \in I \) is the input label, \( o \in O \) is the output label, and \( s' \in S \) is the target state.

The HSI method requires an FSM with the following 3 properties: (a) determinism - for each pair of state and input, there is at most one outgoing transition; (b) initially connectedness - there is a sequence of transitions to every state from the initial state; and (c) minimality - all pairs of states must behave differently (be distinguishable) by producing different sequences of outputs for some sequence of inputs.

**Example 18.** Figure 5.1 presents a deterministic, initially connected, and minimal FSM \( M = (S, s_0, I, O, T) \), where \( S = \{1, 2, 3\} \), \( s_0 = 1 \), \( I = \{a, b, c\} \), \( O = \{0, 1\} \), and \( T = \{(1, a, 1, 2), (1, b, 0, 1), (1, c, 0, 1), (2, a, 0, 2), (2, b, 1, 3), (2, c, 1, 1), (3, a, 1, 2), (3, b, 0, 3), (3, c, 1, 1)\} \). For determinism, there is only one transition leaving each state for a given input. For initial connectedness, two transitions \((1, a, 1, 2)\) and \((2, b, 1, 3)\) (highlighted) connect the initial state to states 2 and 3. For minimality, the input sequence \( a \) results in different output behavior for state pairs \((1; 2)\) and \((2; 3)\), and the input sequence \( c \) for \((1; 3)\).
5.2. BACKGROUND

5.2.1.2 Full Fault Coverage Criterion

The full fault coverage criterion is defined using a fault domain.

**Definition 5.2.2.** Consider an FSM $M = (S, s_0, I, O, T)$ with $n$ states and its implementation $N = (Q, q_0, I, O', T')$. The symbol $\mathcal{F}$ denotes the fault domain that is the set of all FSMS that are: (i) deterministic; (ii) have the same input alphabet as $M$; and (iii) include all defined input sequences of $M$. Moreover, $\mathcal{F}_n$ is the set of FSMS from $\mathcal{F}$ with $n$ states.

The following definition summarizes the main results from Simao and Petrenko [98] where the full fault coverage criteria is established based on specific properties.

**Definition 5.2.3.** Given a test-suite $T$ for an FSM $M$ with $n$ states, $T$ is $n$-complete when for all FSMs $N \in \mathcal{F}_n$ there exist tests in $T$ that distinguish $M$ and $N$.

A test suite $T$ satisfies the full fault coverage criterion when it contains an $n$-complete test suite for an FSM $M$; in that case, by executing $T$ we are capable of detecting any fault (from a failure) in all FSM implementations $N \in \mathcal{F}_n(T)$.

5.2.1.3 Test case Generation

The HSI method is based on the W method [14]; both methods use a characterizing set to distinguish pairs of states in the FSM.

**Definition 5.2.4.** Given an FSM $M = (S, s_0, I, O, T)$ with state set $S = \{s_1, ..., s_n\}$, the set $W \in \mathcal{P}(I^*)$ is a characterizing set if and only if for all $1 \leq i, j \leq n$ with $i \neq j$ there exists an input sequence $\gamma \in W$ that distinguishes $s_i$ and $s_j$.

The HSI method uses the W set to obtain subsets for each state.

**Definition 5.2.5.** Given an FSM $M = (S, s_0, I, O, T)$ with state set $S = \{s_1, ..., s_n\}$, the sets $H_1, ..., H_n \subseteq W$ are harmonized state identifiers if and only if for all $1 \leq i, j \leq n$ with $i \neq j$ there exist an input sequence (common prefix) $\gamma \in H_i \cap H_j$ that distinguishes $s_i$ and $s_j$.

To generate test cases, we need inputs sequences that reach all transitions (a transition cover set) and the distinguishing set(s). The original W method concatenates each input sequence with the whole W set. The HSI method selects a HSI set for each input sequence, resulting in less test cases.

![Figure 5.1: Abstract FSM $M$.](image)
Example 19. The characterizing set $W$ for FSM $M$ presented in Figure 5.1 is $W = \{a, c\}$, while the HSI sets are: $H_1 = \{a, c\}$, $H_2 = \{a\}$, and $H_3 = \{a, c\}$. The $n$-complete test suite is obtained by concatenating a transition cover set $CV = \text{pref}(\{b, c, ac, aa, aba, abb, abc\})$ with $H_i$ sets, which results in $TS = \text{pref}(\{ba, bc, ca, cc, aka, acc, aaa, abaa, abba, abbc, abca, abc\})$.

5.2.2 Feature Model

A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system [53]. Feature models [83] define feature relations based on commonalities and variabilities using graphical models such as feature diagrams. A feature diagram [93] uses a notational convention to describe constraint-based feature relations. The basic feature relations are mandatory, optional, inclusive-OR (or), exclusive-OR (alternative), include, and exclude [54]. A noteworthy feature modeling method is the Feature-Oriented Domain Analysis (FODA) [53]. Subsequent feature modeling methods extended the FODA to add new dependency relations like the Orthogonal Variability Model (OVM) [58].

Example 20. We use in this paper a simplified version of the Car Audio System (CAS) that was presented as a running example in [112]. The CAS SPL can produce different audio systems for cars and provides playbacks and controls. Figure 5.2 shows the feature diagram of CAS based on FODA. There are two alternative features for the Player module (CD and Cassette) and one optional feature (USB) for the Playback module. One and only one alternative feature must be selected, and the optional feature may or may not be included.

In general, due to the dependencies and constraints on feature combinations, only some products can be derived. Assume a set of features $F$ of a feature model. The set of all valid products $P$ of an SPL is a subset of feature combinations from the power set $\mathcal{P}(F)$ that satisfies the constraints specified by the feature model [5].

A feature constraint $\chi$ is a propositional formula that interprets elements of the feature set $F$ as propositional variables. The set of all feature constraints is denoted
by $B(F)$. Feature relation and constraints of a feature diagram can be extracted as a feature constraint following a formal semantics [93]. A product configuration $\rho \in B(F)$ of a product $p \in P$ is a feature constraint of the form $\rho = (\bigwedge_{f \in p} f) \land (\bigwedge_{f \notin p} \neg f)$, i.e., the conjunction of all features present in $p$ and the conjunction of all features absent from $p$. The set $\Lambda \subseteq B(F)$ denotes all valid product configurations of the SPL. Given a feature constraint $\chi \in B(F)$, a product configuration $\rho \in \Lambda$ satisfies $\chi$ (denoted by $\rho \models \chi$), if and only if the feature constraint $\rho \land \chi$ is satisfiable.

**Example 21.** Given the feature diagram of Figure 5.2 the extracted feature set is $F = \{A, B, M, L, N, W, Y, U, C, T\}$, where $O = \{A, B, M, L, N, W, Y\} \subseteq F$ is the subset of mandatory features. The extracted feature constraint that represents all valid products is $\chi = (\bigwedge_{f \in O} f) \land (U \implies B) \land (C \lor T) \land \neg (C \land T)) \in B(F)$. There are only four product configurations that satisfy $\chi$,

\[
\rho_1 = (\bigwedge_{f \in O} f) \land C \land \neg T \land \neg U, \quad \rho_2 = (\bigwedge_{f \in O} f) \land C \land \neg T \land U,
\]

\[
\rho_3 = (\bigwedge_{f \in O} f) \land \neg C \land T \land \neg U, \quad \rho_4 = (\bigwedge_{f \in O} f) \land \neg C \land T \land U.
\]

The feature diagram graphically represents the feature relation while a feature model can be used to represent relevant information for testing. A feature model is defined as follows.

**Definition 5.2.6.** A feature model $FM$ is a triple $(F, \chi, \Lambda)$, where $F$ is the set of features, $\chi$ is the feature constraint of all feature relations, and $\Lambda$ is the set of product configurations that satisfies $\chi$.

Given two feature constraints $\omega_a$ and $\omega_b$, and $\Lambda_a, \Lambda_b \subseteq \Lambda$ satisfying $\omega_a$ and $\omega_b$, respectively, we say that $\omega_a$ and $\omega_b$ are *equivalent* under $FM$ if $\Lambda_a \subseteq \Lambda_b$ and $\Lambda_b \subseteq \Lambda_a$.

### 5.2.3 Featured Finite State Machines

A Featured Finite State Machine (FFSM) [37] is an extension of a Finite State Machine (FSM) in which states and transitions are annotated with feature constraints. The syntax of an FFSM is defined as follows.

**Definition 5.2.7.** An FFSM is a 6-tuple $(FM, C, c_0, Y, O, \Gamma)$, where

1. $FM = (F, \chi, \Lambda)$ is a feature model (Definition 5.2.6),
2. $C \subseteq S \times B(F)$ is a finite set of *conditional states*, where $S$ is a finite set of state labels, $B(F)$ is the set of all feature constraints, and $C$ satisfies the following condition:

\[
\forall (s, \varphi) \in C \cdot \exists \rho \in \Lambda \cdot \rho \models \varphi
\]

3. $c_0 = (s_0, true) \in C$ is the *initial conditional state*,
4. $Y \subseteq I \times B(F)$ is a finite set of *conditional inputs*, where $I$ is the set of input labels,
Figure 5.3: FFSM for the CAS SPL.

5. \( O \) is a finite set of outputs,
6. \( \Gamma \subseteq C \times Y \times O \times C \) is the set of conditional transitions satisfying the following condition:

\[
\forall ((s,\varphi),(x,\varphi''),o,(s',\varphi')) \in \Gamma \quad \exists \rho \in \Lambda \quad \rho \models (\varphi \land \varphi' \land \varphi'')
\]

The above-given two conditions ensure that every conditional state and every conditional transition is present in at least one valid product of the SPL. A conditional state \( c = (s,\varphi) \in C \) is alternatively denoted by \( s(\varphi) \). A conditional transition from conditional state \( c \) to \( c' \) with conditional input \( y = x(\varphi'') \) and output \( o t = (c, y, o, c') \) is alternatively denoted by \( x(\varphi'') /o \) or \( c \xrightarrow{o} \varphi' c' \). The operators of feature constraints are denoted by \&\& (and), || (or), and ! (not). Omitted feature conditions mean that the condition is true, i.e., state \( s \) is equivalent to \( (s,\text{true}) \in C \), and \( \xrightarrow{o} \) is equivalent to \( (x,\text{true}) \rightarrow_o \).

Example 22. Figure 5.3 shows an FFSM for the CAS SPL for the feature model presented in Figure 5.2. The playback behavior begins with radio turned off, and once it is turned on it cycles between playbacks starting with radio, then alternatively CD or cassette, and then USB if it was included, otherwise, it gets back to radio again. This is a simple example in which history is not included, thus, the first command is to execute the radio module. From any conditional state except \( \text{Off} \) the radio can be turned off that can be noticed by the shutdown output. Alternative modules such as CD (green) and Cassette (dark blue) are combined into a single abstract state independent of the selected feature. Specific transitions can represent the specific behavior of each module, i.e., the transitions from \( \text{Radio} \) to \( \text{CD} | \text{Cassette} \) for \( \text{CD} \) and \( \text{Cassette} \) features, respectively.

We can use a sequence of conditional transitions to form a conditional path to generate test cases. This concept is formalized in the following definition.

Definition 5.2.8. Given a conditional input sequence \( \alpha = (y_1, \ldots, y_k) \in Y^* \), where \( y_i = (x_i, \theta_i) \in Y \) for \( 1 \leq i \leq k \), a conditional path from conditional state \( c_1 = \)
(s₁, φ₁) to cₖ₊₁ exists when there are conditional transitions tᵢ = (cᵢ, yᵢ, oᵢ, cᵢ₊₁) ∈ Γ, for 1 ≤ i ≤ k. A conditional path σ is a 4-tuple (ν, δ, γ, ω), where:
1. ν = (c₁, ..., cₖ₊₁) ∈ C* is the conditional state sequence,
2. δ = (x₁, ..., xₖ) ∈ I* is the input sequence,
3. γ = (o₁, ..., oₖ) ∈ O* is the output result,
4. ω = (φ₁ ∧ ... ∧ φₖ₊₁) ∧ (θ₁ ∧ ... ∧ θₖ) ∈ B(F) is the resulting path condition.

A conditional path is valid if there is at least a product configuration ρ ∈ Λ that can satisfy the path condition ω, i.e., ∃ρ∈Λ • ρ ⊨ ω. Notation Θ(c) is used to denote the set of all conditional paths that start at conditional state c ∈ C. Θ_FF is used to denote Θ(c₀).

Further details on model derivation and validation properties for FFSM models can be found in [37].

5.3 Configurable Test Design

In this section, we extend basic test definitions used in FSM-based test case generation for state coverage, transition coverage, and the HSI method [69] for FFSMs. Only FFSM specifications that are deterministic, initially connected, and minimal (as presented in [37]) are used for test case generation.

5.3.1 Configurable Test suites

To generate conditional test cases, we use sequences of inputs that are valid in at least one product configuration. A configurable test suite, also defined below, is a set of conditional tests.

**Definition 5.3.1.** Given an FFSM FF = (FM, C, c₀, Y, O, Γ) such that FM = (F, χ, Λ), a conditional test case (or simply a conditional test) of FF is a tuple (δ, ω) ∈ I* × B(F), where δ is an input sequence of a valid conditional path ((c₀, ..., cₖ), δ, γ, ω) ∈ Θ_FF, and ω is the feature constraint of the path. A configurable test suite CTS ⊆ P(I* × B(F)) of FF is a finite set of conditional tests of FF.

To determine whether a conditional test (δₐ, ωₐ) is a conditional prefix of another conditional test (δₖ, ωₖ) we use the feature model to compare the configurations satisfied by each feature constraint.

**Definition 5.3.2.** Given an FF = (FM, C, c₀, Y, O, Γ) such that FM = (F, χ, Λ), a conditional test (δₐ, ωₐ) is a conditional prefix of (δₖ, ωₖ) if: (i) δₐ is a prefix of δₖ; (ii) there exist configurations that satisfy both feature constraints, i.e., ∃ρ∈Λ • ρ ⊨ (ωₐ ∧ ωₖ); and (iii) Λₐ ⊆ Λₖ, where Λₐ, Λₖ ⊆ Λ are the subsets of configurations satisfying ωₐ and ωₖ, respectively.
We denote by $c_{\text{pref}}(\delta, \omega)$ the set of prefixes of $(\delta, \omega)$, and $c_{\text{pref}}(CTS)$ for the prefixes of all tests in a configurable test suite. Moreover, given two conditional tests $(\delta_a, \omega_a)$ and $(\delta_b, \omega_b)$, if $\delta_a = \delta_b$, then they can be merged into a conditional test $(\delta_a, (\omega_a \lor \omega_b))$. Please see Section 5.4.2 for more details about the prefix check using the Z3 tool.

5.3.2 Test case Derivation

The product derivation operator $\Delta_\phi$ induces an FSM from a given FFSM and a given feature constraint $\phi$ [37]. Similarly, we define the derivation of test suites from a configurable test suite of an FFSM. The feature constraint $\phi$ is able to filter/prune test cases from the test suite in the same way a configuration model is used for test models.

**Definition 5.3.3.** Let $FF = (FM, C, c_0, Y, O, \Gamma)$ be an FFSM, a configurable test suite $CTS$ for $FF$, and a product configuration $\rho \in \Lambda$. The derivation operator $\Delta_\rho$ induces a test suite $\Delta_\rho(CTS) \subseteq P(I^*)$ for an FSM, where:

$$\Delta_\rho(CTS) = \{\delta | (\delta, \omega) \in CTS \land \rho \models \omega\}$$

5.3.3 State Coverage

To define the state coverage criterion for FFSMs, we need to check whether a given FFSM is initially connected. If the given FFSM does not have the property, then we cannot generate a configurable state cover set for it. Otherwise, if it is initially connected, then we generate conditional tests from valid conditional paths found for each conditional state. Note that only FFSMs that are deterministic, initially connected, and minimal are used for test models. Please see [37] for more details about FFSM model validation.

**Definition 5.3.4.** Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$ and a conditional state $c = (s, \varphi) \in C$, the set $CTS \subseteq P(I^* \times B(F))$ covers $c$ for the state coverage criterion if for all product configurations that satisfy $\varphi$ there is a valid conditional path $(((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF}$ to reach $c$ and $(\delta, \omega) \in CTS$. The set $CTS$ is a configurable state cover test suite if it covers every conditional state of $FF$:

$$\forall c = (s, \varphi) \in C \bullet \forall \rho \in \Lambda \bullet \rho \models \varphi \implies \exists ((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF} \bullet \rho \models (\omega \land \varphi) \land (\delta, \omega) \in CTS$$

**Example 23.** We use the FFSM $FF$ presented in Figure 5.3 to identify valid conditional paths for generating a configurable state cover set $CTS$. To cover state $Off$ we need a conditional test with the empty sequence $(\varepsilon, (true))$. For state $Radio$ we can use a single conditional test $(on, (true))$. For state $CD|Cassette$, we use 2 conditional tests $((on, switch), (C))$ and $((on, switch), (T))$, which can be combined into a single conditional test $((on, switch), (C \lor T))$. Finally, for state $USB$ we need two conditional tests $((on, switch, switch), (C \land U))$ and
5.3. CONFIGURABLE TEST DESIGN

((on, switch, switch), (\(T \wedge U\))), which can be combined into a single conditional test
((on, switch, switch), ((C \lor T) \wedge U)). Thus, the resulting configurable state cover set
is \(cpref(CTS) = \{(on, switch), (C \lor T), (on, switch), ((C \lor T) \wedge U)\}\). Applying the derivation operator for the product configuration \(\rho_3 = (\ldots \wedge C \wedge T \wedge \neg U)\)
on \(CTS\) we derive a test suite \(cpref(\Delta_\rho(CTS)) = \{(on, switch)\}\).

Algorithm 1 generates a configurable state cover set. We identify conditional
paths using a breadth-first search looking for different paths combinations excluding
self-loops transitions and those that create a loop in the current path resulting in paths
no larger than the number of states minus one.

Algorithm 1 Configurable state cover set generation.

1: procedure \text{CSTATECOVER}(FF)
2: \hspace{1em} paths ← \text{findNoLoopPaths}(FF)
3: \hspace{1em} validPaths ← \text{checkSat}(paths)
4: \hspace{1em} CTS ← \{\}
5: \hspace{1em} for CState cs : FF.getCStates() do
6: \hspace{2em} reach ← \{\}
7: \hspace{2em} for Path path : validPaths do
8: \hspace{3em} if cs == path.end() then
9: \hspace{3em} \hspace{1em} reach ← reach \cup \{path\}
10: \hspace{2em} if checkCov(cs, reach) == false then
11: \hspace{3em} \hspace{1em} return null
12: \hspace{2em} CTS ← CTS \cup reach.getCSequences()
13: \hspace{2em} CTS ← ReduceRedundantPaths(CTS, cs)
14: \hspace{1em} return CTS

Next, we state and prove that a configurable state cover set is a state cover set for
all its valid product FSMs.

**Theorem 7.** If the test suite \(CTS \subseteq P(I^* \times B(F))\) is a configurable state cover
for an FFSM \(FF\), then \(CTS\) contains a state cover set \(\Delta_\rho(CTS) \subseteq P(I^*)\) for all
derived product FSMs \(\Delta_\rho(FF)\).

**Proof.** We prove the implication by contradiction. Let the set \(CTS \subseteq P(I^* \times B(F))\)
be a configurable state cover for an FFSM \(FF = (FM, C, c_0, Y, O, \Gamma)\) and assume that
the set \(\Delta_\rho(CTS) \subseteq P(I^*)\) does not cover a state \(s \in S\) for a product configuration \(\rho\)
under a derived FSM \(\Delta_\rho(FF) = (S, s_0, I, O, T)\). By Definition 5.3.4 for every product
configuration \(\rho \in \Lambda\) that can be satisfied by the feature constraint of a conditional
state \(c = (s, \varphi) \in C\) there exist a valid conditional path ((c_0, ..., c), \delta, \gamma, \omega) \in \Theta_{FF}
that reaches \(c\). By Definition 5.3.3 the conditional input sequence (\(\delta, \omega\)) \in CTS
derives an input sequence \(\delta \in \Delta_\rho(CTS)\) for \(\rho\) that reaches \(s\), which leads to a
contradiction as \(\delta \notin \Delta_\rho(CTS)\). \(\square\)
5.3.4 Transition Coverage

To extend the transition coverage from FSMs to FFSMs, first we redefine the transition cover set for FSMs. The transition coverage criterion uses an input sequence that can reach the source state of a transition, and then concatenate the transition input to the input sequence. A test suite is a transition cover set for an FSM if it can cover all transitions of the FSM.

**Definition 5.3.5.** Given an FSM $M = (S, s_0, I, O, T)$, and a state cover set $TS \subseteq \mathcal{P}(I^*)$ for $M$, the test suite $TS$ covers a transition $(s, x, o, s') \in T$ if there exist a path $((s_0, ..., s), \alpha, \beta) \in \Omega_M$ from state $s_0$ to $s$, where $\alpha \in TS$ is the input sequence to reach $s$, $\beta$ is the output sequence, and $\alpha x \in TS$. The set $TS$ is a transition cover test suite of $M$ if it covers every transition of $M$:

$$\forall (s, x, o, s') \in T \cdot \exists \alpha \in TS \cdot \exists ((s_0, ..., s), \alpha, \beta) \in \Omega_M \cdot \alpha x \in TS$$

To define the transition coverage criterion for FFSMs we use valid conditional paths to reach conditional source states of conditional transitions for all valid products, and then include each and every outgoing transition.

**Definition 5.3.6.** Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, a configurable state cover $CTS \subseteq \mathcal{P}(I^* \times B(F))$ for $FF$ (Definition 5.3.4), and a conditional transition $t = (c_a, (x, \varphi), o, (s_b, \omega_b)) \in \Gamma$, the test suite $CTS$ covers $t$ if for all conditional tests $((\delta, \omega)) \in CTS$ that reach $c_a$ such that exists a product configuration $\rho \in \Lambda$ that satisfies $\varphi_t = (\omega \land \varphi \land \omega_b)$, then $(\delta x, \varphi_t) \in CTS$. The set $CTS$ is a configurable transition cover test suite if it covers every conditional transition of $FF$:

$$\forall t \in \Gamma \cdot \forall ((\delta, \omega)) \in CTS \cdot \exists \rho \in \Lambda \cdot \rho \models \varphi_t \Rightarrow (\delta x, \varphi_t) \in CTS$$

**Example 24.** Consider the FFSM $FF$ and the configurable state cover set $CTS$ for $FF$ presented in Example 23. Its configurable transition cover set is an extension of $\text{cpref}(CTS)$. Figure 5.4 presents a conditional testing tree generated of $CTS$. Starting from left to right, the tree shows 11 (the number of leafs in the tree) valid conditional paths. Thus, after merging the conditional tests, substituting long feature constraints with smaller equivalent ones, and removing conditional test prefixes the resulting set is:

$$\{\{(\text{off}, \text{true})\}, \{(\text{switch}, \text{true})\}, \{(\text{on}, \text{off}), \text{true}\}, \{(\text{on}, \text{switch}, \text{off}), \text{true}\}, \{(\text{on}, \text{switch}, \text{switch}), \text{true}\}, \{(\text{on}, \text{switch}, \text{switch}, \text{off}), U\}, \{(\text{on}, \text{switch}, \text{switch}, \text{switch}), U\}\}$$
Applying the derivation operator for the product configuration \( \rho_3 = (...) \land \neg C \land T \land \neg U \) on \( CTS \) we derive a test suite \( cpref(\Delta_\rho(CTS)) = \{(\text{off}), (\text{switch}), (\text{on}, \text{off}), (\text{on}, \text{switch}, \text{off}), (\text{on}, \text{switch}, \text{switch})\} \).

Algorithm 2 generates a configurable transition cover set. We reuse a subset of the configurable state cover set to reach the source state of a conditional transition, and then we concatenate the conditional transition to each reachable path, check whether it forms a valid conditional path, and finally add new conditional tests in \( CTS \).

To state and prove that a configurable transition cover set is a transition cover set for all its valid product FSMs, first we redefine a transition cover set for an FSM, then, we move to FFSMs.

**Theorem 8.** If the set \( CTS \subseteq \mathcal{P}(I^* \times B(F)) \) is a configurable transition cover for an FFSM \( FF \), then \( CTS \) contains a transition cover set \( \Delta_\rho(CTS) \subseteq \mathcal{P}(I^*) \) for all derived product FSMs \( \Delta_\rho(FF) \).

**Proof.** We prove the implication by contradiction. Let the set \( CTS \subseteq \mathcal{P}(I^* \times B(F)) \) be a configurable transition cover for an FFSM \( FF = (FM, C, c_0, Y, O, \Gamma) \) and assume that the set \( \Delta_\rho(CTS) \subseteq \mathcal{P}(I^*) \) does not cover a transition \( t = (s_a, x, o, s_b) \in T \) for a product configuration \( \rho \) under a derived FSM \( \Delta_\rho(FF) = (S, s_0, I, O, T) \). By Definition 5.3.4 and 5.3.6 for every product configuration \( \rho \in \Lambda \) that can be satisfied by the feature constraint of a conditional state \( c_a = (s,a, \varphi_a) \in C \) there exist a valid conditional path \(((c_0, \ldots, c_a), \delta, \gamma, \omega) \in \Theta_{FF} \) that reaches \( c_a \), and a
Algorithm 2 Configurable transition cover set generation.

```plaintext
1: procedure CTransitionCover(FF)
2:      CTS ← CStateCover(FF)
3:      paths ← recoverValidPaths()
4:      for CState cs : FF.getCStates() do
5:          out ← cs.getOut()
6:          for CTransition t : out do
7:              for Path path : paths.getReach(cs) do
8:                  addNewPath(paths, path, t)
9:              reach ← checkSat(paths.getTReach(t))
10:             if checkTCov(t, reach) == false then
11:                 return null
12:             CTS ← CTS ∪ reach.getSequences()
13:             CTS ← ReduceRedundTPaths(CTS, t)
14:      return CTS
```

transition \( t = (c_a, (x, \varphi), o, (s_b, \varphi_b)) \in \Gamma \) such that \( (\delta x, (\omega \land \varphi \land \varphi_b)) \in CTS \). By Definition 5.3.3 the conditional test \( (\delta x, (\omega \land \varphi \land \varphi_b)) \in CTS \) derives an input sequence \( \delta x \in \Delta_\rho(CTS) \) for \( \rho \) that reaches \( s_a \) concatenated with \( x \), which leads to a contradiction as \( \delta x \notin \Delta_\rho(CTS) \).

### 5.3.5 Full Fault Coverage

To extend the HSI method for FFSMs we are going to use two sets: (i) a configurable transition cover set \( CTS \); and (ii) a configurable characterizing set \( CW \) to distinguish conditional states. We have defined the configurable transition cover set \( CTS \) in Definition 5.3.6. Now we need to define parametrized separating sets, define a configurable characterizing set, and then build conditional HSI sets for state identification of FFSMs.

#### 5.3.5.1 Parametrized Separating Set

To distinguish a pair of states of an FSM we only need one separating sequence. However, to distinguish a pair of conditional states of an FFSM we need a set of separating sequences.

**Definition 5.3.7.** Let \( c_a = (s_a, \varphi_a), c_b = (s_b, \varphi_b) \) be two distinct conditional states of an FFSM \( FF \) and also let \( \rho \in \Lambda \) be a product configuration for \( FF \) under a given SPL. If (i) \( \rho \models \varphi_a \land \varphi_b \) (ii) there exists two valid conditional paths \( ((c_a, \ldots, c_a'), \delta, \gamma_a, \omega_a) \in \Theta_{FF} \) and \( ((c_b, \ldots, c_{b'}), \delta, \gamma_b, \omega_b) \in \Theta_{FF} \) such that \( \rho \models \omega_a \land \omega_b \) and \( \gamma_a \neq \gamma_b \), then a conditional test \( (\delta, (\omega_a \land \omega_b)) \in I^* \times B(F) \) is called a separating sequence for \( c_a \) and \( c_b \) under the product \( \rho \).

Note that for a given FFSM, there might be a single separating sequence that can distinguish a pair of states for every satisfiable product configuration. However, we need as many separating sequences as the number of valid product configurations.

**Example 25.** Given the FFSM \( FF \) presented in Figure 5.3 to distinguish the pair of conditional states \((Radio, true)\) and \((CD|Cassette, (C \lor T))\) we use four transitions:

\[
\begin{align*}
(Radio, true) \xrightarrow{(\text{switch}, C)} & (CD|Cassette, (C \lor T)) \\
(Radio, true) \xrightarrow{(\text{switch}, T)} & (CD|Cassette, (C \lor T)) \\
(CD|Cassette, (C \lor T)) \xrightarrow{(\text{switch}, \neg U)} & (Radio, true) \\
(CD|Cassette, (C \lor T)) \xrightarrow{(\text{switch}, \neg U)} & (USB, U)
\end{align*}
\]

First, the set of product configurations that we need to cover using separating sequences is \( \text{true} \land (C \lor T) \) which is equivalent to all product configurations of \( \Lambda \). Then, from the transitions we obtain four separating sequences \( s_1 = (\text{switch}, (C \land \neg U)), \ s_2 = (\text{switch}, (C \land U)), \ s_3 = (\text{switch}, (T \land \neg U)), \) and \( s_4 = (\text{switch}, (T \land U)). \) Next, we can optionally combine/merge those separating sequences into one \( s = (\text{switch}, true) \). This is the worst case where there are 4 product configurations and 4 separating sequences. However, this is unlikely to happen frequently due to the SPL commonalities. For example, for another pair of conditional transitions such as \((Radio, true)\) and \((\text{Off}, true)\) we need a single separating sequence \((\text{off}, true)\).

Given an FFSM and an SPL, there may be exponentially many number of valid product configurations, hence it may not be practical to derive a separating sequence for each valid product configuration. Instead, we compute separating sequences for sets of product configurations and put in a parametrized separating set.

**Definition 5.3.8.** Let \( c_a = (s_a, \varphi_a), c_b = (s_b, \varphi_b) \) be two conditional states of an FFSM \( FF \) such that there exists at least one product configuration \( \rho \in \Lambda \) that satisfying both feature constraints, i.e., \( \rho \models \varphi_a \land \varphi_b \). The set \( PS_{ab} \in \mathcal{P}(I^* \times B(F)) \) is called a parametrized separating set for \( c_a \) and \( c_b \) if the disjunction of conditions of every separating sequence \( (\delta, \omega) \in PS_{ab} \) (Definition 5.3.7) is equivalent to \( \varphi_a \land \varphi_b \). Thus, \( PS_{ab} \) can distinguish \( c_a \) and \( c_b \).

**Example 26.** From example 25 we know that \( c_a = (Radio, true), c_b = (CD|Cassette, (C \lor T)) \) which results in \( PS_{ab} = \{(\text{switch}, (C \land \neg U)), (\text{switch}, (C\land U)), (\text{switch}, (T\land \neg U)), (\text{switch}, (T\land U))\} \), or simply \( PS_{ab} = \{(\text{switch}, true)\} \).

### 5.3.5.2 State Identification

The configurable characterizing set \( CW \) for state identification of FFSMs is built using parametrized separating sets for every conditional state pair of the FFSM.
Definition 5.3.9. Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, with conditional state set $C = \{c_1, \ldots, c_n\}$, the set $CW \in \mathcal{P}(I^* \times B(F))$ is a configurable characterizing set, if and only if, for all $1 \leq i, j \leq n$ with $i \neq j$ and $\Lambda_i \cap \Lambda_j \neq \emptyset$, there exists a parametrized separating set $PS_{ij} \subseteq CW$ (Definition 5.3.8).

Example 27. In example 26 we obtained the parametrized separating set for conditional states $c_a = (\text{Radio}, \text{true}), c_b = (\text{CD|Cassette}, (C \lor T))$. Combining other pairs of conditional states $s_c = (\text{Off}, \text{true})$ and $s_d = (\text{USB}, U)$ we have:

- $PS_{ab} = \{ (\text{switch}, \text{true}) \}$, $PS_{ac} = \{ (\text{off}, \text{true}) \}$,
- $PS_{ad} = \{ (\text{switch}, U) \}$, $PS_{bc} = \{ (\text{off}, \text{true}) \}$,
- $PS_{bd} = \{ (\text{switch}, U) \}$, $PS_{cd} = \{ (\text{off}, \text{true}) \}$.

The resulting configurable characterizing set (after removing conditional prefixes - Definition 5.3.2) is $\text{cpref}(CW) = \{ (\text{switch}, \text{true}), (\text{off}, \text{true}) \}$.

The conditional HSI sets for state identification of FFSMs are built using the configurable characterizing set $CW$.

Definition 5.3.10. Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, with conditional state set $C = \{c_1, \ldots, c_n\}$, the sets $CH_1, \ldots, CH_n \in \mathcal{P}(I^* \times B(F))$ are conditional harmonized state identifiers, if and only if for all $1 \leq i, j \leq n$ with $i \neq j$ and $\Lambda_i \cap \Lambda_j \neq \emptyset$, there exists a common parametrized separating subset $PS_{ij} \subseteq CW_i \cap CW_j$ (Definition 5.3.8) with conditional prefixes from $CW$ that distinguishes $c_i$ and $c_j$ using separating sequences.

Example 28. The conditional characterizing set $CW$ of Example 27 for the FFSM $FF$ presented in Figure 5.3 is: $CW = \{ (\text{off}, \text{true}), (\text{switch}, \text{true}) \}$. The conditional HSI sets for the FFSM $FF$ are:

- $CH_{\text{Off}} = \{ (\text{off}, \text{true}) \}$,
- $CH_{\text{Radio}} = \{ (\text{off}, \text{true}), (\text{switch}, \text{true}) \}$,
- $CH_{\text{CD|Cassette}} = \{ (\text{off}, \text{true}), (\text{switch}, \text{true}) \}$, and
- $CH_{\text{USB}} = \{ (\text{off}, \text{true}), (\text{switch}, U) \}$.

5.3.5.3 Extended HSI Method

Now we have sets $CTS$ for transition coverage and conditional HSI sets for state identification. Then, the final configurable test suite $CTS$ is the concatenation of $CTS$ with every $CH_i$.

Definition 5.3.11. Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, the configurable transition cover set $CTS$, and the conditional harmonized state identifier $CH_i$ sets, the extended conditional HSI method defines a complete configurable test suite $CTS$ for $FF$ by concatenating every tuple of $CTS$ with every $CH_i$ set for each $c_i \in C$ such that only conditional tests of $CTS$ that reach $c_i$ are concatenated and the conjunction of constraints is satisfied:

$$\forall c_i \in C \bullet \forall (\delta, \omega) \in CTS \bullet \exists ((c_0, \ldots, c_i), \delta, \gamma, \omega) \in \Theta_{FF} \bullet \forall (\beta, \omega') \in CH_i \bullet \exists \rho \in \Lambda \bullet \rho \models (\omega \land \omega') \implies (\delta \beta, (\omega \land \omega')) \in CTS$$
Example 29. The complete configurable test suite is obtained by concatenating CTC of Example 24 with the conditional HSI sets presented in Example 28. The following set is a complete configurable test suite for \( FF \).

\[
\text{cpref}(CTS) = ((\text{off, off}), \text{true}),
((\text{switch, off}), \text{true}),
((\text{on, off, off}), \text{true}),
((\text{on, switch, off, off}), \text{true}),
((\text{on, switch, switch, off}), \text{true}),
((\text{on, switch, switch, switch}), \text{true}),
((\text{on, switch, switch, switch, off}), \text{U}),
((\text{on, switch, switch, switch, off}), \text{U}),
((\text{on, switch, switch, switch, off, off}), \text{U}),
\}
\]

Applying the derivation operator for the product configuration \( \rho_3 = (\ldots \land \neg C \land T \land \neg U) \) on CTC we derive a test suite \( \text{cpref}(\Delta_{\rho}(CTC)) = \{(\text{switch, off}), \}
\]

Next, we state and prove that a complete configurable test suite is a complete test suite for all its valid product FSMs.

**Theorem 9.** If the set \( CTS \subseteq \mathcal{P}(I^* \times B(F)) \) is an \( n \)-complete configurable test suite for an FFSM \( FF \), then \( CTS \) contains an \( n \)-complete test suite \( TS \subseteq \mathcal{P}(I^*) \) for all derived product FSMs \( \Delta_{\rho}(FF) \).

**Proof.** We prove the implication by contradiction. Let the set \( TS \subseteq \mathcal{P}(I^* \times B(F)) \) be an \( n \)-complete configurable test suite for an FFSM \( FF = (F, \Lambda, C, c_0, Y, O, \Gamma) \) and the set \( \Delta_{\rho}(TS) \subseteq \mathcal{P}(I^*) \) be not an \( n \)-complete configurable test suite derived from a product configuration \( \rho \) under a derived FSM \( \Delta_{\rho}(FF) = (S, s_0, I, O, T) \). As \( \Delta_{\rho}(TS) \) is not an \( n \)-complete configurable test suite for state \( \Delta_{\rho}(FF) \), then does not exist a test \( \delta h \in \Delta_{\rho}(TS) \) such \( \delta \) exists in a path to reach a state \( s \) and \( h \) distinguishes \( s \) with another state \( s' \). By Definition 5.3.10 and 5.3.11 for each conditional state \( c_i \in C \) we have at least one conditional test \( (\delta h, (\omega \land \omega')) \in TS, \rho \models (\omega \land \omega') \) where \( (\delta, \omega) \in CTC \) reaches \( c_i \) and \( (h, \omega') \in CH_i \) distinguishes \( c_i \) with another conditional state \( c_j \in C, i \neq j \). By Definition 5.3.3 the conditional input sequence \( (\delta h, (\omega \land \omega')) \in TS \) derives an input sequence \( \delta h \in \Delta_{\rho}(TS) \) for \( \rho \), which is a contradiction as \( \delta h \notin \Delta_{\rho}(TS) \).

### 5.4 Tool Support

This paper reports on the design, implementation, and application of a test design tool **Configurable Full Test Generator (ConFTGen)**\(^4\). The ConFTGen has a graphical editor

\(^4\)ConFTGen tool [https://github.com/vhfragal/ConFTGen-tool](https://github.com/vhfragal/ConFTGen-tool)
based on the Eclipse platform that was extended from the Yakindu GitHub Project⁵ (Eclipse Public Licence) and integrated with FeatureIDE [101] (Lesser General Public Licence - LGPL) and Z3 SMT Solver [24] (MIT licence) for constructing feature models and analyzing feature constraints, respectively. ConFTGen supports the automatic validation and derivation of FFSM models, and automatic generation of test suites for state, transition, and full fault coverage. Figure 5.5 shows the FFSM for CAS SPL in the ConFTGen tool. Inputs, outputs, and some features are declared on the left, and behavior represented inside conditional states are self-loop transitions.

### 5.4.1 Eclipse Platform

The Eclipse Platform is a framework that provides an open source software development environment. Eclipse has a core architecture that supports the integration of tools and other development environments. Moreover, Eclipse projects are implemented in Java language and can be run in several operating systems including Linux, Mac OS X, and Windows.

The functional unit of Eclipse are plug-ins. Plug-ins are combined with the core architecture and can be integrated to build complex tools. As stated before, the ConFTGen tool extends plug-ins from the Yakindu Project. The core plug-ins used to develop the ConFTGen tool were `org.eclipse.core`, `org.eclipse.ui`, `org.eclipse.gef`, `org.eclipse.emf`, `org.eclipse.emf`, `org.eclipse.emf`

### 5.4.2 Satisfiability Modulo Theories Solvers

The ConFTGen tool has been implemented on the Eclipse platform, and we used the Z3 Satisfiability Modulo Theory (SMT) Solver [24] to process feature constraints. It is well-known that feature models can be translated into propositional formulas; see, e.g., [5, 20]. This translation enables mechanizing the analysis of feature-based specifications using existing logic-based tools, such as SMT solvers. We present how SMT solvers are used for test design by checking prefix and equivalence relations

---

⁵Open Source Yakindu Project [https://github.com/Yakindu/statecharts](https://github.com/Yakindu/statecharts)
5.5. EXPERIMENTAL STUDY

based on feature constraints. We used the Java language to parse and process FFSMs and subsequently generate assertions in the SMT file format.

The Z3 tool is used for checking conditional prefixes and equivalence relations for a pair of conditional tests $(\delta_a,\omega_a), (\delta_b,\omega_b) \in I^* \times B(F)$, where $\Lambda_a \subseteq \Lambda$ is the subset of configurations that the constraint $\omega_a$ can satisfy, and $\Lambda_b \subseteq \Lambda$ for $\omega_b$. To check whether $(\delta_a,\omega_a)$ is a conditional prefix of $(\delta_b,\omega_b)$, we perform the following steps:

1. If $\delta_a$ is a prefix of $\delta_b$, then we assert the feature constraint $\chi$ of the feature model $FM = (F, \chi, \Lambda)$ and the logical conjunction of feature constraints $(\omega_a \land \omega_b)$. If it returns unsatisfiable, then there is no common configuration between both constraints $\Lambda_a \cap \Lambda_b = \emptyset$ and we cannot derive/deduce the prefix relation.

2. If the first check returns satisfiable, then another check identify the prefix relation $\Lambda_a \subseteq \Lambda_b$. We assert the first condition $\omega_a$ followed by the negation $\neg \omega_b$ to check whether $\Lambda_a \cap (\Lambda \setminus \Lambda_b) = \emptyset$ is unsatisfiable. If the proposition turns out to be satisfiable, then there are configurations for $\omega_a$ which $\neg \omega_b$ was not able to invalidate, i.e., $\Lambda_a \not\subseteq \Lambda_b$. Otherwise, when the result is unsatisfiable, then, $\neg \omega_b$ was able to invalidate all configurations of $\omega_a$, i.e., $\Lambda_a \subseteq \Lambda_b$.

Example 30. Figure 5.6 presents parts of the generated SMT file used to check whether a conditional test $(\delta_a,\omega_a) = (a, (B \land S))$ is a conditional prefix of $(\delta_b,\omega_b) = (aa, (B))$, where: (a) all features are declared as Boolean variables, (b) the feature constraint of the feature model is asserted, and (c) the prefix relation checks are asserted. In Z3, push and pop commands are used to temporarily set the context (e.g., with assertions), and once a verification goal is discharged the context can be reset. The (check − sat) command is used to evaluate the assertions which return sat or unsat. First, we notice that $a$ is a prefix of $aa$, then, checking the conjunction of both conditions ($(B \land S) \land B) \equiv (B \land S)$ results in sat. Next, we assert the first constraint $(B \land S)$ and also assert the negation of the second constraint $\neg(B)$ to eliminate configurations of the first, which results in unsat. Thus, $(a, (B \land S))$ is a conditional prefix of $(aa, (B))$.

The Z3 tool is also used for checking feature constraints in most definitions in this paper. The usage is a variation of the Example 30 where we change the (c) part of the SMT file. Given a feature constraint $\varphi \in B(F)$ that requires checking, we use (assert $\varphi$) in (c) to check if it returns sat or unsat.

5.5 Experimental Study

To evaluate the applicability and the efficiency of our approach, we conducted an experiment to evaluate random FFSMs and random feature models. We focus on the characteristics of the models, the comparison of test suite size and test generation time against the product-by-product approach.
Focusing on the full fault coverage, we are aware of some incremental test case generators for FSMs \[30, 79, 98\]. These methods can process an existing test set and increment few tests for a new product configuration. Despite the good results on the few number of new tests, these methods have to process the test set for every new product configuration. Moreover, they are quite sensitive to the size of the test set used as input. Hence, they may not scale well to increment large test sets for large specifications.

We aimed at answering the following research questions:

- Q1- Is there a difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM?

- Q2- In which scenario do we reduce the number of test cases using an FFSM instead of FSMs?

- Q3- In which scenario do we have smaller test generation times using an FFSM instead of FSMs?

- Q4- Is there a relation between the feature model and the configurable test suite size?

### 5.5.1 Experimental Setup

Our experiment aims at generating test suites (full fault coverage) for random FFSMs and random feature models. We choose to use random FFSMs to avoid bias of specific domains. Different feature models also were randomized to use in combination with a respective FFSM. Different feature structures may result in different number of product configurations which we may check whether it affects validation and generation or
5.5. EXPERIMENTAL STUDY

The adaptable parameters (independent variables) include random feature models that are generated with the following parameters required for the BeTTy tool [94]:

- Number of features (20);
- Percentage of custom constraints (0);
- Probability of a feature being mandatory (0);
- Probability of a feature being optional (33);
- Probability of a feature being in an or-relation (33);
- Probability of a feature being in an alternative-relation (33);
- Maximal branching factor (10);
- Maximal number of children in a set relationship (10).

The probability parameters use numbers from 0 to 100 (in percentage) and their sum must not exceed 100. We decided to use no custom constraints and mandatory features. Custom constraints may be redundant to constraints defined by the feature structure, and they are not used in order to simplify the analysis. About mandatory features, we have only one that is the root feature which represent the core specification. Adding extra mandatory features may result in SPLs with fewer product configurations as were using a fixed number of 20 features. Moreover, we only use feature constraints with non-mandatory features in the FFSM model. Thus, all mandatory behavior is implicitly represented in the core specification. The maximal branching factor limits the number of branches in the feature model structure. The number of children in a set relationship limits the number of features that may be in an alternative/or/optimal subset relation.

To visualize some characteristics for the relation/influence of feature models and FFSMs we divided 20 random feature models into two groups. The first group has feature models with less than the median of independent features. The second group has feature models with more than the median of independent features. An independent feature is placed in an or-relation or set in parallel with other optional features, which makes the number of product configurations grow exponentially.

To answer our research questions we executed three groups of FFSMs. Each group has 20 random FFSMs and 20 random feature models. The FFSM groups are:

1. FFSMs with 12 conditional states, 10 to 15 inputs/outputs and 100-450 transitions.
2. FFSMs with 18 conditional states, 15 to 20 inputs/outputs and 225-800 transitions.
3. FFSMs with 24 conditional states, 20 to 25 inputs/outputs and 400-1250 transitions.
Each random FFSM is generated by randomizing the target state of conditional transitions. We assume that only \( 1/3 \) of the conditional states and transitions have random feature constraints. Thus, \( 2/3 \) of the behavior is part of the core specification. We use FFSMs that are deterministic, initially connected, and minimal.

The fixed parameters (controlled variables) are:

- Each random FFSM is linked (for feature mapping) to one random feature model;
- The number of inputs and outputs are the same as the number of FFSM states;
- One derived FSM (from the FFSM) represents the core specification of the SPL;
- The test generation method HSI used for FSM and FFSMs.

Then, we measure (dependable variables):

1. The number of new different tests: we measure the number of new tests that are required to test all products using an FFSM-based test suite against 20 random products with individual FSM-based test suites;

2. Test suite size for the FSM core: we measure the size of the test suite pruned for the FSM core specification from an FFSM-based test suite against generating the test suite directly from the FSM core specification;

3. Checking and generation time: we measure the amount of time it takes to generate tests for an entire FFSM against one FSM and a set of FSMs.

The running environment used Ubuntu 15.04 (64 bit) operating system on an Intel processor i7-5500U at 2.40GHz with 12GB of RAM.

5.5.2 Analysis and Threats to Validity

To answer our questions, we generated complete configurable test suites for all 3 groups of FFSMs. As stated before, each group has 20 random FFSMs, such that each group has larger FFSM specifications than the previous group. For each FFSM we derived an FSM which represents the SPL core specification (core product). The collected data in our experiments is analyzed below.

5.5.2.1 Q1- Is there a difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM?

Results. To answer this question, we used a one sided test assuming that the null hypothesis is true. Our null hypothesis (H0) is: \( mu = a \) the true mean of complete test suites for the core product using the HSI method. Our alternative hypothesis is: \( mu > a \) the mean of complete test suites for the core product pruned from a configurable complete test suite generated using an FFSM is larger than \( a \). Figure 5.7
shows the results on the test suite size comparing our H0 and H1 for all three FFSM groups. The p-value of a normal distribution of all three groups (12, 18, 24 states) are respectively: 0.149, 0.182, and 0.140.

**Analysis.** We noticed that the results from the first FFSM group continue in the other two groups as it increases the FFSM size. There is a small difference favoring the direct application of the HSI on the core product over pruning a configurable test suite from the extended HSI method for the same core product. Thus, our experiment does not indicate a statistically meaningful difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM.

**Threats.** The number of FFSMs in each group. Larger samples may reduce the p-value.

**5.5.2.2 Q2- In which scenario do we reduce the number of test cases using an FFSM instead of FSMs?**

**Results.** To answer this question, we selected 20 random valid configurations of each FFSM. For every selected configuration an FSM was derived and a complete test suite was generated using the HSI method. Figure 5.8 shows the results on the accumulated number of new tests comparing the (FFSM) configurable test suite size with three cases: (FSM.core) complete test suite size for the core product; (10FSMs) merged set of complete test suites from the first ten selected configurations; (20FSMs) merged set of complete test suites from the all selected configurations. In the first
FFSM group, the means are: (FFSM) 1422; (FSM.core) 660; (10FSMs) 2199; and (20FSMs) 3782. In the second FFSM group, the means are: (FFSM) 3810; (FSM.core) 1757; (10FSMs) 6819; and (20FSMs) 8946. In the third FFSM group, the means are: (FFSM) 9191; (FSM.core) 3002; (10FSMs) 13924; and (20FSMs) 16588.

**Analysis.** We noticed that the results from the first FFSM group continue in the other two groups as it increases the FFSM size. There is a significant difference comparing the first and third cases as we need to test more FSMs. An FFSM is linked to a feature model ranging from $2^{21}$ to $2^{20}$ valid configurations depending on the feature structure. Selecting up to the minimum number of valid configurations shows us how powerful it is to exploit commonalities between products compared to the product-by-product approach. For example, an FFSM of the first group with 12 conditional states may represent $2^{10}$ valid configurations and may have a configurable test suite of size 1422 (the mean), and to test 20 out of $2^{10}$ configurations individually we require an average set of tests with size 3782. When we create a test case without checking test suites of similar products we may end up generating a different but equivalent test case. In our approach, one test case may be reused across all valid products minimizing redundant test cases. In all three groups the difference is above 50%. Thus, our experiment indicates a statistically meaningful difference of approximately 50% in the number of new test cases when we have more than 20 random valid configurations to test individually compared to an entire SPL using a configurable test suite generated from an FFSM.
5.5. EXPERIMENTAL STUDY

Threats. Very similar specifications may not accumulate so many different test cases with so few products.

5.5.2.3 Q3- In which scenario do we have smaller test generation times using an FFSM instead of FSMs?

Results. To answer this question, we selected 100 random valid configurations of each FFSM. For every selected configuration an FSM was derived and a complete test suite was generated using the HSI method. Figure 5.9 shows the results on the time required to generate complete test suites comparing the (FFSM) time to generate a configurable test suite with three cases: (FSM.core) time to generate a complete test suite for the core product; (50FSMs) time to generate complete test suites from the first fifty selected configurations; (100FSMs) time to generate complete test suites from the all selected configurations. In the first FFSM group, the means are (in minutes): (FFSM) 0.66; (FSM.core) 0.00017; (50FSMs) 0.008; and (100FSMs) 0.017. In the second FFSM group, the means are (in minutes): (FFSM) 1.92; (FSM.core) 0.012; (50FSMs) 0.63; and (100FSMs) 1.27. In the third FFSM group, the means are (in minutes): (FFSM) 4.99; (FSM.core) 0.11; (50FSMs) 5.96; and (100FSMs) 11.92.

Analysis. We noticed that the results vary as it increases the FFSM size. There is a significant difference comparing the first and third cases in the last FFSM group. Assume that we have to generate a configurable complete test suite for an FFSM which may have $2^{10}$ valid configurations or individually 100 valid configurations. If our FFSM have more than 24 conditional states and 450 conditional transitions and we
need to test more than 100 valid configurations we may consider a better generation time using our approach.

**Threats.** The number of required paths for test generation due to complex feature models may be different in specific FFSMs.

### 5.5.2.4 Q4- Is there a relation between the feature model and and the configurable test suite size?

**Results.** To answer this question, we divided each FFSM group in two subgroups of 10 FFSMs. The median of the valid feature configurations was used to separate the FFSMs into two types of feature models. In the first group of FFSMs with 12 conditional states the median of product configurations found from all feature models was 6276, then 12808 for the second group, and 25608 for the third group. Figure 5.10 shows the results for the relation between the configurable test suite size and types of features models.

**Analysis.** We noticed that the size of complete configurable test suites is larger in those FFSMs which have feature models with more than the median of valid configurations in each group. As we stated before, each random FFSM is generated by randomizing the target state of conditional transitions. We assume that only 1/3 of the conditional states and transitions have random feature constraints. Thus, 2/3 of the behavior is part of the core specification. Also, only one feature is used on each conditional state/transition. The first FFSM group (12 states) has smaller specifications.
5.5. EXPERIMENTAL STUDY

than other FFSM groups, and we first believe that less features are used (out of 20) resulting in a smaller feature configuration median. To check that belief, we also checked the number of conditional transitions of each subgroup. Figure 5.11 shows the results for the relation between the number of transitions of FFSMs and types of feature models.

We noticed that FFSMs linked to feature models with more configurations than the median has fewer conditional transitions. When an FFSM has fewer conditional transitions the extended HSI algorithm has fewer options to find common conditional paths, resulting in more different conditional tests.

We also checked the relation between the feature model and the extended HSI test generation time. Figure 5.12 shows the results for the relation between the time required to generate a configurable test suite size and types of features models.

We noticed that the time required to generate configurable test suites does not have a pattern from the 3 FFSM groups. Despite the larger configurable test suites for those FFSMs with more than the median of valid configurations presented in Figure 5.10, the time required to generate them do not follow. Also, it does not have a direct relation to the FFSM size as presented in Figure 5.11. We believe that this is caused by the irregular number of required conditional paths and separating sequences. Thus, our experiment indicates no influence of the feature model for the time required to generate configurable test suites.

Threats. Our experiment indicates a tendency to have larger configurable test suites for FFSMs when the feature model has many valid configurations but possibly affected by the number of fewer transitions. This indication may be a threat to validity. The
irregular distribution of valid configurations may be a coincidence or a consequence of the FFSM random generator.

5.6 Body Comfort System Case Study

We illustrate and evaluate our approach in a prototypical implementation using a case study from the automotive domain, namely, Body Comfort System (BCS) for the VW Golf SPL [60]. We use the BCS to reduce the threats to validity and contrast the results from randomly generated FFSMs with their real-world counterparts. The FeatureIDE tool [101] was used to elaborate feature models and their configurations. The original BCS system has 19 non-mandatory features and 11616 configurations.

Our first observation was that flattening the whole system into an FFSM was infeasible due to a large number of conditional states (in theory more than 50000 states). Thus, we used a simplified version (slice) which integrates some components, and we performed a manual flattening of the slice. We selected a part of the feature model with 4 non-mandatory features and 6 possible configurations for 4 components: Finger Protection (FP) blocking the window movement when a finger is clamped in a window, Manual Power Window (ManPW) or alternatively Automatic Power Window (AutPW), and Central Locking System (CLS) with optional Automatic Locking (AL) when the car is driving. Figure 5.13 shows the selection of the Automatic Power Window component while leaving the Central Locking System and
5.6. BODY COMFORT SYSTEM CASE STUDY

Figure 5.13: Feature model configuration selection for BCS.

Automatic Locking features open which means that this model still accommodates 3 out of 6 possible configurations.

To help design FFSM models, we used the ConFTGen tool proposed in Section 5.5. Figure 5.14 presents the flat composition of four selected components of BCS and abstract inputs and outputs described below. The original 150% behavioral model of each component can be found in [60]. States 1, 2 and 3 represent the behavior of Manual and Automatic Power Window alternative components. The Finger Protection component has two states, and hence, the same behavior is repeated in states 4, 5, and 6. The last component Central Locking System also has two states which lead to the same behavior repeated in states 7 to 12.

The ConFTGen tool has two types of model derivation. The first type of model derivation (FFSM) ignores the open features and derives an FFSM for a subset of valid configurations using a feature constraint without the open features. The second type of model derivation (FSM) uses a feature constraint with all features of the model and
Figure 5.14: FFSM of 4 composed components of BCS.
negates the open features to derive an FFSM model for a single configuration that corresponds to an FSM. Figure 5.15 shows the first FFSM model derivation type for 3 product configurations. Figure 5.16 shows the second FFSM model derivation type for a single product configuration which excludes Central Locking System and Automatic Locking features.

Following our product line-centered approach, the configurable test suite obtained for the state coverage has 14 tests and size 40. The configurable test suite obtained for the transition coverage has 124 tests and size 715. The configurable test suite obtained for the full fault coverage has 433 tests and size 2311. The validation time takes approximately 10 seconds while the configurable test suite generation 1 minute.

To test individually all 6 product configurations we derived 6 FSMs and we generated 6 test suites using the original HSI method. Then, to calculate the number of new tests required for all 6 products we unified all 6 test suites and counted the number of tests that would be concretized and executed. The validation time takes approximately 1 seconds while the configurable test suite generation 5 seconds. In the end, we found out that our unified test suite has 463 tests and size 3071. Using our approach we had 433 tests and size 2311 which is a reduction of 25%.
Comparing the results with the random FFSMs with 12 conditional states, we found out that the number of new test cases is in accordance with question 2 and test suite generation is in accordance with question 3.

5.7 Related Work

Usually, an SPL can generate several similar products where only a few features vary from one to another. One challenge in SPL testing is the verification of products using a simplified behavioral model that takes advantage of the similarity between products. There are proposals [15, 17] that provide a concise formalism for representing SPL behaviors in one model. However, they are focused on model checking [17] or simple test criteria like boundary tests [15]. In this paper, we take a step forward by extending the FSM-based formalism for SPLs. The main purpose of this extension is to enable test case generation methods that use family-based FSM models to achieve comprehensive, configurable test suites using the full fault coverage.

Regarding regression-based approaches for SPLs, there are several incremental test approaches [30, 79, 105, 12, 3] devoted to generating, reusing, and optimizing test suites for SPLs. El-Fakih et al. [30] adapted FSM-based test generation methods for conformance testing. Their approach allows for the generation of test cases only for the modified parts of an evolving specification. Pap et al. [79] extended their work and designed a bounded incremental algorithm that maintains two sets based on the HSI method [69]. They utilize existing test cases of the previous version of the system to generate test cases for the modified version. Similarly, Capellari et al. [12] explored the FSM-based Testing of SPLs (FSM-TSPL) testing strategy where the P method is used to design new test cases based on the last product derived. Uzuncaova et al. [105] also developed an incremental test generation approach that uses SAT-based analysis to develop tests suites for every product of an SPL, while Baller and Lochau [3] focused on test suite optimization. Moreover, recent delta-oriented approaches...
5.7. RELATED WORK

[67, 65, 106] developed regression-based SPL approaches to design and reuse test artifacts.

Regarding configurable test artifacts, most modeling concepts for variability can be distinguished into three main approaches: annotative, compositional and transformational variability modeling [91]. Compositional approaches for modeling variability capture variation by selecting specific component variants. Compositional variability modeling [44] allows for a modular description of variability but limits the impact of changes to the applied composition technique. Transformational approaches represent variability by transformation of a base architectural model. Model transformation rules guide the derivation of products by performing additions, modifications or removals using variability. For example, delta modeling [16] can represent variability in the model transformation which a core system is developed, and subsequent products are derived by executing such transformations rules. Annotative approaches use variant annotations (also called 150%-models), e.g., UML stereotypes in UML models to define which model elements belong to specific product variants. In the orthogonal variability model (OVM) [84], a separate variability representation with links to the architecture model replaces direct annotations. Some approaches [20, 43] propose a pruning-based approach to UML 150% test model for SPLs, separating variability from the base models using mapping models.

Using an annotative 150% statechart test model and transition coverage criterion, Cichos et al.’s approach [15] presents SPL test design for complete test model coverage with subsequent product subset selection for test suite execution. Weissleder et al. [112] propose an approach for automatic test suite derivation based on reusable UML state machine test models and OCL expressions. Similarly, Liu et al. [62] use statecharts to model reusable components and also through pruning, derive instances syntactically, and may be combined with Wasowski’s approach [109] to flattening statecharts without the state explosion problem. In Devroey et al.’s approach [28], they use mutation testing applied to annotated Featured Transition Systems (FTS) [17]. Moreover, Luthmann et al.’s approach [70] uses the Featured Time Automata formalism (a variation of FTS) to check real-time properties of SPLs.

Model-based testing can be used in SPL testing. We refer the reader to Oster et al. [77] for a summary of model-based SPL testing approaches, to [9, 21, 91, 6] for recent surveys, and Thum et al.’s recent survey [100] for a classification of different SPL analysis techniques. Some behavioral models proposed in the literature, e.g., those in [62, 2, 18] are based on Finite State Machines or Labeled Transition Systems. They are mainly used to provide a formal specification for SPLs and enable their formal verification using model checking.

Our proposed approach for configurable test artifacts can be classified as a family-based and feature-oriented specification. To our knowledge, however, there only a few pieces of research that extend test models, test case generation and test case execution to the family-based level; examples of such work include earlier delta-oriented techniques such as [64, 65, 106] and feature-oriented approaches [7, 27] using FTS-based formalisms. However, the approach proposed in [7, 8] exploits a non-deterministic test case generation algorithm (with no fault model or finite test
suite) and hence, semantic validation of test models is not an issue in their approach. We are not aware of any prior study on extending the FSM-based test-model validation and test case generation techniques to the family-based setting that is based on the notion of full fault coverage.

5.8 Conclusions and Future Work

In this paper, we presented an extension of the HSI test case generation method for FFSMs. An FFSM test model represents the abstract behavior of SPL components and its compositions. The HSI test generation method was originally designed to generate tests using FSMs for the full fault coverage criterion. However, FSMs used as inputs to HSI require semantic properties such as determinism, initially connected and minimal. In our previous work [37], we presented an extension of the FSM for SPLs named FFSMs, where such semantic properties were extended to FFSMs, and we showed that they coincide with their corresponding properties for the product FSM models. In this paper, the HSI method was extended from FSMs to FFSMs.

We conducted an experimental study comparing the number of new tests required to test SPL products using the extended HSI method with a random set of products using individual test suites. Random FFSMs and feature models were generated, and our implemented method was applied on them. The results indicate a significant decrease in the number of new tests when compared to the traditional product-by-product approach. The experiments showed that in general, we have more new tests from only 20 products than the whole SPL by using our approach with an FFSM. Also, the case study shows that only with 6 products we still have more new tests using the product-by-product approach. Moreover, we checked the relation between FFSMs and feature models in respect to configurable test suite size and test case generation time. We observed no strong influence on different kinds of feature models for test case generation using FFSMs.

A prototyping tool named ConFTGen was implemented to guide the design of FFSMs. The tool also performs validation, derivation, and test case generation for the state, transition, and full fault coverage. A case study for the Body Comfort System was used to present the tool and show some issues related to the current FFSM specification.

In a parallel line of work, we plan to extend the FFSM model to Hierarchical FFSMs (using concepts from Statecharts and UML State Machines) to handle the state explosion problem identified in the case study. We then apply validation (and test case generation) on hierarchical models. Another possible line of work is to improve test case generation using new concepts from regression-based incremental methods.
Abstract

Variants of the Finite State Machine (FSM) model have been extensively used to describe the behavior of reactive systems. In particular, several model-based testing techniques have been developed to support test case generation from FSMs and test case execution. Most of such techniques require several validation properties to hold for the underlying test models. The Featured Finite State Machine (FFSM) is an extension of the FSM model that represents the abstract behavior of an entire Software Product Line (SPL). By validating an FFSM, we validate all valid products configurations of the SPL looking forward configurable test suites. However, modeling a large SPL using flat FFSMs may lead to a huge hard-to-maintain specification. In this paper, we propose an extension of the FFSM model, named Hierarchical Featured State Machine (HFSM). Inspired by Statecharts and UML state machines, we introduce the HFSM
model to improve model readability which groups up FFSM conditional states and transitions into an abstracted model. Our ultimate goal is to use HFSMs as test models. To this end, we first define some syntactic and semantical validation criteria for HFSMs as prerequisites for using them as test models. Moreover, we implemented an adapted graphical Eclipse-based editor from the Yakindu Project for modeling, derivation, and checking feature-oriented properties using Satisfiability Modulo Theory (SMT) solver tools. We investigate the applicability of our approach by applying it to an HFSM for a realistic case study (the Body Comfort System). The results indicate that HFSMs can be used to compactly represent and efficiently validate the behavior of parallel components in SPLs.

Keywords: Model Validation, Software Product Line, Featured Finite State Machine, Hierarchical Featured Finite State Machine.

6.1 Introduction

In the face of the increasing complexity, software industries moved from craftsmanship to industrialization, where components are customized and assembled to produce similar products with low cost and satisfying several customer demands [41].

Software Product Line Engineering (SPLE) is a paradigm to develop software, where a family of related products (a Software Product Line - SPL) is built out of a common set of core assets, thus reducing development costs for each product [83]. In SPL, products are built, step-by-step, by incrementally adding or removing functionalities which alleviate software complexity and improve quality.

Similar to the development of single systems, the SPL process also has several activities that are executed to ensure software quality. Verification, validation, and testing are examples of such activities that are useful to check software functionalities and minimize risks. Testing is a very important activity in software engineering, which can detect faults in the software system [74]. Despite the systematic software artifact reuse that increases productivity, new challenges arise in testing activities for SPL [77, 33].

Testing activities represent a large share of overall project costs and are even more challenging in SPL than for single systems [99]. Unfortunately, several domains, such as embedded and safety-critical systems, do not strictly follow development standards (due to high test costs) to efficiently test several product configurations in a systematic manner. For example, the standard ISO 26262 for safety-critical automotive software requires the highest level of safety integrity that each developed product configuration has to be tested using model-based techniques with a high degree of test coverage under strong test criteria.

Finite State Machines (FSMs) and their variants have been extensively used as a fundamental semantic model for various behavioral specification languages. In particular, several test case generation techniques have been developed for hardware

---

3https://www.iso.org/standard/43464.html
and software testing based on FSMs; an overview of these techniques can be found in [57, 11, 47]. All FSM-based testing techniques require the underlying test models to satisfy some basic validation criteria such as connectedness and minimality.

There are recent attempts [64, 106] to extend the FSM-based testing techniques to SPLs, mostly using the delta-oriented approach to SPL modeling. We proposed Featured Finite State Machines (FFSMs) in the conference publication of this paper [37] with focus on the basic test model validation criteria for SPLs at the family-wide level. However, the scalability problem is the main issue of using the FFSM model for large and complex systems. Such a problem makes model analysis costly to execute and it also leads to a hard-to-maintain test model. To improve scalability in modeling, we propose in this paper an extension of FFSMs named Hierarchical Featured State Machines (HFSMs). The HFSM model is inspired by Statecharts [45], and its syntax is validated by checking well-formed states and transitions. To define a formal semantics for HFSMs and enable its formal analysis, we define a transformation from HFSMs to FFSMs.

Figure 6.1 shows an overview of the SPL validation workflow for HFSM with artifact dependency represented by dashed arrows. In domain engineering, we generate reusable and configurable artifacts while in application engineering those artifacts can be configured for a set of product configurations.

The main contributions of this paper are: (i) extending the FFSM model to HFSM. The introduction of FFSMs is our first contribution, but due to our focus on hierarchy, we dispense with presenting the detailed results about the basic formalism which can be found in [37]; (ii) implementing a support tool for modeling, validation, and derivation of HFSMs using the Eclipse platform, Java language, and the Z3 solver.
tool [24]; and (iii) investigating the HFSM applicability and validation using a case study. The case study is from the automotive domain concerning the Body Comfort System [60]. The results indicate that HFSMs can be used to compactly represent and efficiently validate the behavior of parallel components in SPLs.

The remainder of this paper is organized as follows. Section 6.2 presents some preliminary notions and concepts regarding SPL features. Section 6.3 introduces the FFSM formalism. Section 6.4 introduces the HFSM formalism with detailed syntax and semantics. Section 6.5 presents the supporting tool for HFSM modeling, validation, and derivation. Section 6.6 illustrates a case study with an industrially-inspired HFSM. Section 6.7 provides an overview of the related work and a comparison among the relevant approaches in the literature. Finally, Section 6.8 concludes the paper and presents the directions of our future work.

6.2 Background

This section presents the basic concepts and definitions regarding SPL features that we are going to use through the rest of the paper.

6.2.1 Feature Diagram

A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system [53]. A feature diagram [93] uses a notational convention to describe constraint-based feature relations. The basic feature relations are mandatory, optional, inclusive-OR (or), exclusive-OR (alternative), include, and exclude [54]. A noteworthy feature modeling method is the Feature-Oriented Domain Analysis (FODA) [53]. Subsequent feature modeling methods, such as the Orthogonal Variability Model (OVM) [58], extend the FODA to add new dependency relations.

Example 31. The Arcade Game Maker (AGM) [95] can produce arcade games with different game rules. Figure 6.2 shows the feature diagram of AGM. There are three alternative features for the game rule (Brickles, Pong, and Bowling) and one optional feature (Save) to save the game. One and only one alternative feature must be selected, and the optional feature is left open for selection.

A feature diagram is developed in the domain engineering and used as input to the application engineering level, where it is instantiated by a configuration model. A configuration model allows for the selection of features to specify a single product, and it is useful to integrate components for the product configuration process. The product configuration process (binding) derives a specific product using the reusable SPL architecture and a configuration model with selected features.

6.2.2 Feature Constraint

In general, due to the dependencies and constraints on feature combinations, only some products can be derived. Assume a set of features $F$ of a feature model. The set
of all valid products $P$ of an SPL is a subset of feature combinations from the power set $P(F)$ that satisfies the constraints specified by the feature model [5].

A feature constraint $\chi$ is a propositional formula that interprets the elements of the feature set $F$ as propositional variables. The set of all feature constraints is denoted by $B(F)$. The relation between features and its constraints can be modeled by a feature diagram and extracted as a feature constraint following a formal semantics [93]. A product configuration $\rho \in B(F)$ of a product $p \in P$ is a feature constraint of the form $\rho = (\bigwedge_{f \in p} f) \land (\bigwedge_{f \notin p} \neg f)$, i.e., the conjunction of all features present in $p$ and the conjunction of the negation of all features absent from $p$. The set $\Lambda \subseteq B(F)$ denotes all valid product configurations of the SPL. Given a feature constraint $\chi \in B(F)$, a product configuration $\rho \in \Lambda$ satisfies $\chi$ (denoted by $\rho \models \chi$), if and only if the feature constraint $\rho \land \chi$ is satisfiable.

**Example 32.** Given the feature diagram of Figure 6.2 the extracted feature set is $F = \{G, V, R, C, A, Y, P, S, B, N, W, M, L\}$ where $O = \{G, V, R, C, A, Y, P, M, L\} \subseteq F$ is the subset of mandatory features. The extracted feature constraint that represents the relation of all features is:

$$\chi = ((\bigwedge_{f \in O} f) \land (S \implies V) \land (B \lor N \lor W) \land 
\neg(B \land N) \land \neg(B \lor W) \land \neg(N \land W)) \in B(F)$$

There are only six product configurations that satisfy $\chi$, namely, those specified below:

- $\rho_1 = (\bigwedge_{f \in O} f) \land B \land \neg N \land \neg W \land \neg S$,
- $\rho_2 = (\bigwedge_{f \in O} f) \land B \land \neg N \land \neg W \land S$,
- $\rho_3 = (\bigwedge_{f \in O} f) \land \neg B \land N \land \neg W \land \neg S$,
- $\rho_4 = (\bigwedge_{f \in O} f) \land \neg B \land N \land \neg W \land S$,
- $\rho_5 = (\bigwedge_{f \in O} f) \land \neg B \land \neg N \land W \land \neg S$,
- $\rho_6 = (\bigwedge_{f \in O} f) \land \neg B \land \neg N \land W \land S$.

For modeling, the logical operators on feature constraints are denoted by \&\& (and), || (or), and ! (not).
6.2.3 Feature Model

A Feature model [83] specifies the structure of an SPL in terms of its feature and feature constraints. It can serve as the underlying structural model for other formalisms modeling the behavior of an SPL, e.g., for the purpose of testing.

**Definition 6.2.1.** A feature model $FM$ is a tuple $(F, \chi)$, where $F$ is the set of features and $\chi$ is the feature constraint.

6.3 Featured Finite State Machines

Variants of the FSM model have been extensively used to describe the behavior of different domains. In particular, several model-based testing approaches [77] have been developed to support test design and execution for SPLs. In this paper, we introduce Featured Finite State Machines (FFSMs) [37] and use them as the semantics model for our hierarchical model HFSM. The FFSM can represent the behavior of SPLs using a single model where product FSM properties are defined and checked in a family-wide level.

6.3.1 Basic Definitions

An FFSM combines states and transitions with feature constraints. The syntax of an FFSM is defined as follows.

**Definition 6.3.1.** An FFSM is a 6-tuple $(FM, C, c_0, Y, O, \Gamma)$, where

1. $FM = (F, \chi)$ is a feature model (Definition 6.2.1),
2. $C \subseteq S \times B(F)$ is a finite set of conditional states, where $S$ is a finite set of state labels, $B(F)$ is the set of all feature constraints, and $C$ satisfies the following condition:
   $$\forall (s, \varphi) \in C \bullet \exists \rho \in \Lambda \bullet \rho \models \varphi$$
3. $c_0 = (s_0, true) \in C$ is the initial conditional state,
4. $Y \subseteq I \times B(F)$ is a finite set of conditional inputs, where $I$ is the set of input labels,
5. $O$ is a finite set of outputs,
6. $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying the following condition:
   $$\forall ((s, \varphi), (x, \varphi'), o, (s', \varphi'')) \in \Gamma \bullet \exists \rho \in \Lambda \bullet \rho \models (\varphi \land \varphi' \land \varphi'')$$
6.3. FEATURED FINITE STATE MACHINES

Figure 6.3: FFSM for the AGM SPL.

Figure 6.4: Alternative FFSM for the AGM SPL.

The above-given two conditions ensure that every conditional state and every conditional transition is present in at least one valid product of the SPL. A conditional state \( c = (s, \varphi) \in C \) is alternatively denoted by \( s(\varphi) \). A conditional transition from conditional state \( c \) to \( c' \) with conditional input \( y = x(\varphi'') \) and output \( o t = (c, y, o, c') \) is alternatively denoted by \( x(\varphi'')/o \) or \( c y \rightarrow_o o c' \). Omitted feature conditions mean that the condition is \( \text{true} \), i.e., state \( s \) is equivalent to \( (s, \text{true}) \in C \), and \( \frac{x}{o} \) is equivalent to \( \frac{(x, \text{true})}{o} \).

**Example 33.** Figure 6.3 shows the FFSM for the AGM SPL. Transitions have abstract input and output events. For inputs, the name is intuitive and for outputs we have 0 (nothing) and 1 (beep). Feature constraints are put in brackets for conditional states and transitions. Feature constraints in conditional states generalize the feature constraints of all related conditional transitions. Product-specific transitions can be represented using specific feature constraints. Both conditional transitions \( \text{StartGame} \xrightarrow{(\text{Exit}, (W \& \& \neg S))} \) and \( \text{Bowling}(W) \xrightarrow{(\text{Exit}, (\neg S))} \) only exist in one product which have the \( W \) and not the \( S \) feature. Figure 6.4 shows an alternative FFSM for
AGM which represent mutually exclusive behavior combining *Brickles*, *Pong*, and *Bowling* in a single conditional state. States with alternative feature constraints can be composed into a single state creating a composed name (using “*”) and performing a disjunction of its constraints.

### 6.3.2 Model Derivation

To perform model derivation using an FFSM we use a specific feature constraint. To use a simplified feature constraint, we first need to define the equivalence relation between feature constraints for a given feature model \( FM \).

**Definition 6.3.2.** Given an FFSM \( FF = (FM, C, c_0, Y, O, \Gamma) \), where \( FM = (F, \chi) \), a feature constraint \( \omega_a \) is a conditional prefix of \( \omega_b \) if: (i) there exists a valid configuration that satisfies both feature constraints, i.e. \( \exists \rho \in \Lambda \cdot \rho \models (\omega_a \land \omega_b) \); and (ii) the subset of configurations \( \Lambda_a \subseteq \Lambda \) that satisfy \( \omega_a \) is a subset of configurations \( \Lambda_b \subseteq \Lambda \) that satisfy \( \omega_b \), i.e., \( \Lambda_a \subseteq \Lambda_b \). When \( \Lambda_a \subseteq \Lambda_b \) and \( \Lambda_b \subseteq \Lambda_a \) we say that \( \omega_a \) and \( \omega_b \) are equivalent under \( FM \).

Thus, we define a model derivation operator, reminiscent of the operator in [7, 8], that is parameterized by feature constraints. Given a feature constraint, the product derivation operator reduces an FFSM into an FSM representing the selection of products.

**Definition 6.3.3.** Given an FFSM \( FF = (FM, C, c_0, Y, O, \Gamma) \), and a product configuration \( \rho \in \Lambda \) or a feature constraint \( \phi \) equivalent to \( \rho \) under \( FM \) (Definition 6.3.2), the product derivation operator \( \Delta o \rho \) can induce an FSM \( \Delta o \rho (FF) = (S, s_0, I, O, T) \), where:

1. \( S = \{ s \mid (s, \varphi) \in C \land \rho \models (\varphi \land \phi) \} \) is the set of states;
2. \( s_0 = s, c_0 = (s, \varphi) \in C \) is the initial state;
3. \( T = \{ (s, x, o, s') \mid (s, \varphi) \xrightarrow{\varphi''} (s', \varphi') \in \Gamma \land \rho \models (\varphi \land \varphi' \land \varphi'' \land \phi) \} \) is the set of transitions.

By abusing the same notation, we also define how to reduce an FFSM into another FFSM that specifies a set of products (i.e., a product sub-line).

**Definition 6.3.4.** Given a feature constraint \( \phi \in B(F) \) and an FFSM \( FF = (FM, C, c_0, Y, O, \Gamma) \), if at least one product configuration \( \rho \in \Lambda \) satisfies \( \phi \), i.e., \( \exists \rho \in \Lambda \cdot \rho \models \phi \), then the product derivation operator \( \nabla o \phi \) induces a reduced FFSM \( \nabla o \phi (FF) = (FM', C', c_0, Y', O, \Gamma') \) comprising only those elements (i.e., conditional states and transitions) that satisfy \( \phi \).
6.3.3 Validation Properties

To adopt FFSMs as test models, we need to validate the product-line-based specification with properties used for FSMs. We present theorems regarding the high-level counterparts of three basic properties, namely, determinism (Theorem 3), initially connected-ness (Theorem 5), and minimality (Theorem 6). These properties coincide with properties for their valid FSM products.

6.4 Hierarchical Featured State Machine

Inspired by Harel’s Statecharts [45], several hierarchical state machine formalisms were defined to specify the behavioral aspects of reactive systems and extended to object-oriented software development methodologies such as the Unified Modeling Language (UML) [75]. The states represented in the hierarchical model can be simple states or contain an entire state machine. Systems can be specified by a stepwise refinement and visualized in different levels of granularity at the cost of complex syntax and semantics.

We introduced the Featured Finite State Machine (FFSM) formalism in [37] to extend the Finite State Machine (FSM) formalism to the Software Product Line (SPL) context. To design configurable models and tests, one must first validate the model to check properties required for some basic test criteria. Due to scalability problems, e.g., the state explosion problem, large SPLs are hard to model and maintain using the FFSM model.

In this paper, we present the Hierarchical Featured State Machine (HFSM) formalism that extends the FFSM model including hierarchy. The HFSM model improves model readability by grouping up FFSM conditional states and transitions into an abstracted view, which provides a better solution for modeling SPL-based models, e.g., as test models in a Model-Based Testing (MBT) approach. Furthermore, the HFSM model can also be pruned for subsets of product configurations.

In our approach, we use the HFSM as a front-end for modeling and syntax check while the semantics are represented by an FFSM. The syntax check verifies the well-formed state and transition structure. The semantic check verifies properties such as determinism, initially connectedness, and minimality at the SPL level. These properties are required for test-case generation using the full fault coverage (see [69]). However, one of the main issues of lifting the FFSM formalism to HFSM is how to compose orthogonal regions that we also explain in this paper.

Next, we present the detailed syntax and semantics of HFSMs (based on [72]) followed by a support tool for syntax and semantic checks, and then a case study.

6.4.1 Syntax

There are many variations of hierarchical FSMs that we can reuse to define an HFSM. We choose a definition in which there are no final states and inspired terminology from
UML [75]. Thus, an HFSM comprises states that might have a further internal structure (hierarchy) and transitions among them. States and transitions have feature constraints that must be satisfied according to a feature model. The following definitions are based on the corresponding definitions in [45] and [72].

**Definition 6.4.1.** A Hierarchical Featured State Machine \( HFSM \) is represented by a 5-tuple \( (FM, \Upsilon, I, O, T) \), where:

1. \( FM = (F, \chi) \) is a feature model (Definition 6.2.1),
2. \( \Upsilon \) is a well-formed state structure,
3. \( I \) is a set of inputs events,
4. \( O \) is a set of outputs events,
5. \( T \) is a set of well-formed transitions.

A well-formed state structure (inspired by the same notion in [45]) is a tree that represents a valid state hierarchy. The set of well-formed transitions contains valid transitions that connect sets of states in the hierarchy. Input and output sets represent the observable behavior of the machine. Not every state structure or transition is valid, e.g., a simple state must not have sub-states. Thus, some criteria are required to represent well-formed state structures and transitions. We present the formal definitions of well-formed states and transitions in subsections 6.4.1.1 and 6.4.1.2, respectively.

### 6.4.1.1 Well-formed state structure

We define the syntactic state structure of HFSMs, their restrictions, and well-formedness conditions. First, we define the state structure as follows.

**Definition 6.4.2.** A state structure \( \Upsilon \) is defined by a 6-tuple \( (S, \text{root}, \text{default}, \text{sub}, \text{type}, \text{feature}) \), where:

1. \( S \) is a finite set of states;
2. \( \text{root} \in S \) is the root state of the state structure;
3. \( \text{default} : S \to \{\text{true}, \text{false}\} \), is a total function determining whether a state is default or not;
4. \( \text{sub} : S \to \mathcal{P}(S) \) is a total function defining for each state, the set of its sub-states;
5. \( \text{type} : S \to \{\text{simple}, \text{compOr}, \text{compAnd}, \text{region}\} \) is a total function determining for each state whether it is a simple (i.e., has no sub-states), a compOr (i.e., is composite and has only one default sub-state), a compAnd (i.e., is composite and has more than one default sub-state), or a region state (i.e., is a sub-state of a composite state). Thus, composite states only have regions as their sub-states. Sub-states of a region may be of any type but region;
6. \( \text{feature} : S \to B(F) \), is a total function determining the feature constraint of a state.
A tree of conditional states represents the state structure. The root state is the unique state that encapsulates all states in the state structure. The sub-states of a state are their children in the tree structure.

**Example 34.** Figure 6.5 shows the HFSM for the AGM SPL [95]. Unlike Example 33, we have given more structure to the alternative states (i.e., Brickles, Pong, and Bowling). We have put the alternative states in region states (in order to model independent and potentially parallel behavior). Due to feature constraints applied to region states (i.e., B for R1, N for R2, and W for R3), they are exclusive and never composed. Alternatively, we could represent these alternative states by combining them as we did in Figure 6.4. However, we may have more than one state per alternative feature (i.e. 2 states with constraint B and 3 states with constraint N). Thus, such groups of alternative states can be grouped by regions. Regarding transitions, we can model transitions that start and finish in the same state (self-loop transitions) inside the state. By modeling self-loop transitions inside a state, we can combine those which have equivalent feature constraints, e.g., instead of modeling inside SaveGame[S] two self-loop transitions Save/0 and Exit/0, we can model Save, Exit/0. For simplicity, we do not treat history, deep-history, join/forks, and entry/exit points connections in this paper. Albeit important, those features require a more elaborate semantics and are deferred to future work.
A default state is a state that is automatically activated once the super-state (parent) is activated or it is the root itself. Default states are only activated when the transition does not explicitly target a state within the region. For example, the state \textit{StartGame} is not activated after taking the transition from \textit{Rules} to \textit{SaveGame} using the input \textit{Save}, which activates \textit{Menu} and \textit{RegionM}. In Figure 6.5, the default states, besides region states, are represented by default connectors, i.e., a filled circle with an outgoing arrow pointing towards the default state.

\textbf{Example 35.} Figure 6.6 shows the state structure of the HFSM presented in Figure 6.5, where \textit{AGM} is the root state, \textit{Rules} is the compAnd state, all states but \textit{PauseGame}, \textit{SaveGame}, and \textit{Rules} are default states and the sub-states of the \textit{RegionA} state is \(\text{sub}(\text{RegionA}) = \{\text{Menu}, \text{Rules}\}\).

The state structure is generic and allows for inconsistent specifications, e.g., sub-states of a region state being region states. To validate the state structure we need to define several auxiliary functions, and finally, formalize the well-formed state structure. First, we define hierarchy using descendants and ancestors. Descendants are recursively defined below to include sub-states and their sub-states and so forth.

\textbf{Definition 6.4.3.} Given a state \(s \in S\), the set of \textit{descendants} of \(s\), denoted by \text{desc}(s)\) is the smallest set satisfying the following two properties.

1. \(s \in \text{desc}(s)\); and
2. \(\forall a \in S \bullet a \in \text{desc}(s) \implies \forall b \in \text{sub}(a) \bullet b \in \text{desc}(s)\).

For a set of states \(S' \subseteq S\), we define the notion of descendant by \(\text{Desc}(S') = \bigcup_{s' \in S'} \text{desc}(s')\). Super-states and ancestors are the inverses of sub-states and descendants, respectively, and are defined as follows.

\textbf{Definition 6.4.4.} Given a state \(s \in S\), a \textit{super-state} (parent) of \(s\) is defined by a function \(\text{super} : S \rightarrow S\) where \(\forall s, s' \in S \bullet s \in \text{sub}(s') \iff s' = \text{super}(s)\). Moreover, \textit{ancestors} of \(s\) are denoted by \text{anc}(s)\) and satisfy the following condition.

\[\forall s, s' \in S \bullet s \in \text{anc}(s') \iff s' \in \text{desc}(s)\]
For a set of states \( S' \subseteq S \), we define the notion of ancestor by \( \text{Anc}(S') = \bigcap_{s' \in S'} \text{anc}(s') \).

**Example 36.** Following Figure 6.6 some descendants and ancestors in the state structure of the HFSM are:

- \( \text{desc}(\text{Menu}) = \{ \text{Menu}, \text{RegionM}, \text{StartGame}, \text{PauseGame}, \text{SaveGame} \} \);
- \( \text{anc}(\text{PauseGame}) = \{ \text{PauseGame}, \text{RegionM}, \text{Menu}, \text{RegionA}, \text{AGM} \} \);
- \( \text{Desc}(\{ \text{RegionM}, R1 \}) = \{ \text{RegionM}, R1, \text{StartGame}, \text{PauseGame}, \text{SaveGame}, \text{Brickles} \} \); and
- \( \text{Anc}(\{ \text{PauseGame}, \text{Brickles} \}) = \{ \text{RegionA}, \text{AGM} \} \).

In the FFSM representation, a conditional state generalizes the feature constraint on the transitions that leave or reach the state. In other words, transitions “inherit” feature constraints of their source and target states. The feature constraint of a transition is composed and checked using the constraints of the involved states and the constraint of the transition itself. Similarly, in the HFSM representation, the feature constraint of a state constrains the constraint of its descendants. Thus, we define a composition of constraints that are used for checking states in the hierarchy.

**Definition 6.4.5.** Given a set of states \( S' \subseteq S \), the *state feature composition*, denoted by \( f_{\text{comp}} : \mathcal{P}(S') \rightarrow B(F) \) is the conjunction of feature constraints of \( S' \):

\[
f_{\text{comp}}(S') = \bigwedge_{s \in S'} \text{feature}(s)
\]

To check the feature constraint of an HFSM state \( s \in S \), we compose the feature constraints of all ancestors of \( s \) (i.e., set \( \text{anc}(s) \)). Then, we check the constraint \( f_{\text{comp}}(\text{anc}(s)) \).

Finally, we define the concept of well-formedness of state structures extended with feature constraints (inspired by the corresponding restrictions in [45] and [72]).

**Definition 6.4.6.** The state structure \( \Upsilon = (S, \text{root}, \text{default}, \text{sub}, \text{type}, \text{feature}) \) is *well-formed*, when:

1. Simple-states have no sub-states.
   \[
   \forall s \in S \cdot \text{type}(s) = \text{simple} \implies \text{sub}(s) = \emptyset
   \]
   This is a basic assumption as simple states should not have any further structure inside them.

2. All nodes, besides root, have a unique super-state.
   \[
   \forall s \in S \setminus \{ \text{root} \} \cdot \exists! s' \in S \cdot s' = \text{super}(s)
   \]
   This is to make sure that the chain of ancestors reaches an end in the root and also to make sure that every internal behavior is encapsulated by regions states.
3. Descendance relation is asymmetric.
\[ \forall s, s' \in S \cdot s \in \text{desc}(s') \setminus \{s'\} \implies s' \notin \text{desc}(s) \setminus \{s\} \]

This constraint disallows loops in the chain of ancestors and descendants.

4. The single sub-state of a compOr state is a region state.
\[ \forall s \in S \cdot \text{type}(s) = \text{compOr} \implies \exists! s' \in S \cdot s' \in \text{sub}(s) \land \text{type}(s') = \text{region} \]

Upon entering a compOr state, we enter the single region sub-state that represent an inner machine.

5. All sub-states of compAnd states are region states.
\[ \forall s \in S \cdot \text{type}(s) = \text{compAnd} \implies |\text{sub}(s)| > 1 \land \forall s' \in \text{sub}(s) \cdot \text{type}(s') = \text{region} \]

By definition, a compAnd state has more than one region sub-state. Upon entering a compAnd state, we enter in all region sub-states to support parallelism.

6. Region states are default states, their sub-states must not be region states, and only one of their sub-states is default.
\[ \forall s \in S \cdot \text{type}(s) = \text{region} \implies \text{default}(s) \land \forall s' \in \text{sub}(s) \cdot \text{type}(s') \neq \text{region} \land \exists! s'' \in \text{sub}(s) \cdot \text{default}(s'') \]

Upon entering a composite state, we automatically enter a region state that contains an inner machine. Region states are special states used to represent inner machines, and only composite states can have region states as sub-states. One sub-state of the region state must be default to represent the “initial state” of this region.

7. Root is of type compOr and all states are descendants of the root.
\[ \exists! s \in S \cdot s = \text{root} \land \text{type}(s) = \text{compOr} \land \forall s' \in S \cdot s' \in \text{desc}(s) \]

The root state is the common ancestor state of all states in the state structure.

8. The feature constraint of every state is satisfied by at least one product (Definition 6.4.5).
\[ \forall s \in S \cdot \exists \rho \in \Lambda \cdot \rho \models f\text{comp}(\text{anc}(s)) \]

A state has a valid feature constraint based on its ancestors. A simple state inherits all feature constraints of \( \text{anc}(s) \). Thinking about an executable machine, once we activate a state (e.g., via transition), we activate all the ancestors as well. Thus, the hierarchy must not contain branches that are not satisfied by any product configuration.
6.4. HIERARCHICAL FEATURED STATE MACHINE

6.4.1.2 Well-formed transitions

A transition connects states using input events that trigger output events when its constraints are satisfied. The syntax of a transition connects a pair of states of the well-formed state structure. Next, we formalize the definition of transitions and their well-formedness criteria.

Definition 6.4.7. A transition $t$ in an HFSM is defined by a 5-tuple $(a, i, \omega, o, b)$, where:

1. $a \in S$ is the source state;
2. $i \in I$ is the input event;
3. $\omega \in B(F)$ is the feature constraint of the transition;
4. $o \in O$ is the output event;
5. $b \in S$ is the target state.

A transition $t = (a, i, \omega, o, b)$ is denoted by $a \xrightarrow{(i,\omega)} o \xrightarrow{o} b$. The source state is where the transition begins, while the target state is where it ends. The input and output events are the observable behavior of the transition. The feature constraint $\omega$ is a specific condition of the transition. When we omit the feature constraint, $\omega = true$. The feature constraints of the source and target states generalize the entire feature constraint of a transition. Thus, we compose the feature constraints of all elements of the transition, that is defined as follows.

Definition 6.4.8. Given a transition $t = (a, i, \omega, o, b)$, the transition feature composition, denoted by $t_{comp}: T \rightarrow B(F)$ is the conjunction of feature constraints of elements of $t$:

$$t_{comp}(t) = f_{comp}(\text{anc}(a)) \land \omega \land f_{comp}(\text{anc}(b))$$

A transition exists in a product configuration only if it satisfies its transition feature composition. The definition of transition is generic and allows for inconsistent specifications, e.g., transitions whose target state is a region state. As mentioned before, region states are default states that can represent an entire machine. The basic metamodels (e.g., of statecharts or UML) does not allow for transitions to connect region states.

To validate a transition, we define several auxiliary functions, and finally, define the notion of a well-formed transition. Thus, we define the subset of states that can be used as source and target in a transition.

Definition 6.4.9. Given the set of states $S$, the set of transition-relevant states, denoted by $R \subset S$ is the subset of states that are neither region states nor the root state:

$$\forall s \in S \bullet \text{type}(s) \neq \text{region} \land s \neq \text{root} \implies s \in R$$

Next, we define the notion of the least common ancestor. To define orthogonality between two states, we check their least common ancestor.
Definition 6.4.10. Given a subset of states $S' \subseteq S$, their least common ancestor, denoted by $lca(S')$, is the bottommost ancestor which contains all states of $S'$.

$$\exists a \in Anc(S') \land \forall b \in \bigcap_{s' \in S'} \text{anc}(s') \cdot b \in \text{anc}(a) \Leftrightarrow lca(S') = a$$

Example 37. The relevant states for a transition and the least common ancestor of PauseGame and Brickles states are:

- $R = \{\text{Menu}, \text{Rules}, \text{StartGame}, \text{PauseGame}, \text{Brickles}, \text{Pong}, \text{Bowling}\};$
- $lca(\{\text{PauseGame}, \text{Brickles}\}) = \text{RegionA}.$

Orthogonal states can be active at the same time. Transitions with source states that are orthogonal for a common input are synchronized in their execution, and they activate their target states at the same time.

Definition 6.4.11. Two states $s, s' \in S$ are orthogonal to each other when their least common ancestor (Definition 6.4.10) is a compAnd state.

Finally, we define the concept of well-formedness of transitions extended with feature constraints (inspired by the corresponding definition in [45]).

Definition 6.4.12. A transition $t = (a, i, \omega, o, b)$ is well-formed when the following conditions hold.

1. The source and target states are transition-relevant states (Definition 6.4.9).

$$a, b \in R$$

This is a basic assumption as no transition should use regions or root as their source or target states.

2. The source and target states are not orthogonal to each other (Definition 6.4.11).

$$\text{type}(lca(\{a, b\})) \neq \text{compAnd}$$

There should not be any transition involving parallel regions. Communication mechanisms trigger transitions in parallel regions.

3. The transition $t$ has to be satisfied by at least one product (Definition 6.4.8).

$$\exists \rho \in \Lambda \cdot \rho \models tcomp(t)$$

Every transition must exist in at least one product configuration.
6.4. HIERARCHICAL FEATURED STATE MACHINE

6.4.2 Semantics

The semantics of HFSMs is represented by FFsms. We transform an HFSM with a valid syntax into an FFSM to represent its semantic. We use an algorithm for model transformation that compose parallel regions creating a transformed set of state configurations (TSC).

Conditional states of the semantic FFsm are obtained from TSC, which is derived after composing all compAnd states. Then, conditional transitions are derived by processing enter and exit sets. Next, we present the required items for the model transformation.

6.4.2.1 State configurations

To transform an HFSM into an FFSM, valid conditional states and transitions are required. The set of conditional states of the FFSM is defined using the HFSM well-formed state structure.

Definition 6.4.13. Given an HFSM \( H = (FM, \Upsilon, I, O, T) \) with valid syntax, a state configuration (or just configuration) \( SC \subset S \) is a maximal orthogonal set of simple states. i.e., \( \forall s, s' \in SC \bullet (type(s) = type(s') = simple) \land (lca(\{s, s'\}) = \text{compAnd}) \).

The root state and at least one leaf of the state structure tree are always active. The initial configuration is identified using all descendants of root that are default.

Definition 6.4.14. Given a state \( s \in S \), the set of default descendants is denoted by \( ddesc(s) \) and satisfies the following condition. For every state \( s' \in S \) if \( s' \) is a descendant of \( s \) and every ancestor of \( s' \) excluding the ancestors of \( s \) are default, then \( s' \in ddesc(s) \):

\[
\forall s, s' \in S \bullet \forall s'' \in (\text{anc}(s') \setminus \text{anc}(s)) \bullet (s' \in \text{desc}(s)) \land \text{default}(s'') \iff s' \in ddesc(s)
\]

The initial conditional state of a semantic FFsm is represented by simple states of the default descendants of root \( ddesc(root) \). The state set \( Init \subset S \) denotes the initial configuration, i.e., \( \forall s \in S \bullet (s \in ddesc(root)) \land (type(s) = \text{simple}) \iff s \in Init \).

Example 38. The default descendants of the root state are \( ddesc(root) = \{AGM, RegionA, Menu, RegionM, StartGame\} \), where \( root = AGM \). The initial configuration of the state structure presented in Figure 6.6 is \( Init = \{StartGame\} \). If the state Rules was a default state instead of Menu, then our initial configuration would be \( Init = \{Brickles, Pong, Bowling\} \). Next section we show that after composing regions \( R1, R2, \) and \( R3 \) we would have a single state, i.e., \( Init = \{Brickles \ast Pong \ast Bowling\} \).

State configurations may include simple states of several orthogonal state regions. To transform HFSM states into FFsm conditional states, we need to identify every valid state configuration. Next, we define the composition of orthogonal states that derive FFsm conditional states.
6.4.2.2 Composition of Orthogonal States

The composition of orthogonal states (state composition) is a mechanism that allows a flat representation of the parallel execution of the HFSM. We identify valid (reachable) state configurations by combining pairs of regions of compAnd states. To transform a state configuration into an FFSM conditional state, we merge all parallel states of the state configuration and put it in the set of transformed state configurations \( TSC \), i.e., \( \{a, b, c\} \in SC \) implies \( \{a \ast b \ast c\} \in TSC \).

To perform state composition, we use Algorithm 3. On Line 1, we execute the recursive function \( \text{compose\_states} \) using the \( \text{root} \) state and the empty set \( TSC \) as the initial input parameter. On Line 2, we check all substates of a state \( s \), starting with \( \text{root} \). On Line 3, we check whether the substate is \( \text{compAnd} \) type. On Line 4, we make a recursive call using the \( \text{compAnd} \) state as input and resulting in an \( TSC \) set. On Line 5, we initialize the \( \text{init} \) conditional state using the \( \text{compAnd} \) state and the disjunction of the feature constraints of all regions involved. On Line 6, we use the power set of regions to decide whether a subset of regions (called a segment) can be composed or not. On Line 7, we check whether there is a product configuration that can have a specific combination of regions. A feature constraint is created and verified using a conjunction of the feature constraints of all region states in \( R \) and the negation of the feature constraints of all regions out of \( R \) (\( \text{get\_full\_conj\_const} \)). On Line 8, we call the \( \text{pairwise\_merge} \) function to perform the composition process using pairs of region states in \( R \). On Line 10, we execute the \( \text{pairwise\_merge} \) function using \( R, \text{init} \) and \( TSC \) as input. On Line 11, we remove the first region from \( R \) and initialize \( r_1 \). On Line 12, we get every other region to compose with \( r_1 \). On Line 13, we call the \( \text{compose} \) function that merge regions \( r_1 \) and \( r_2 \). The resulting composition is stored in \( r_1 \) to be composed with the next region of \( r_2 \). On Line 14, we set \( \text{init} \) as the initial state of the resulting \( r_1 \) composed (flattened) region. On Line 15, we remove from \( TSC \) all descendants of the regions of \( R \). This is required for the return of the recursion (upper \( \text{compAnd} \) states). On Line 16, we update \( TSC \) with the resulting flat region \( r_1 \). On Line 18, we compose all states and transitions of a pair of regions. On Line 21, 22, and 23, for each substate pair we create the conditional state using their name, the combined feature constraint, and store in \( \text{state} \). On Line 24, we store the composed elements of the flat region, including their conditional states \( \text{state} \) and conditional transitions. Every transition that leaves or reaches \( s_1 \) and \( s_2 \) are combined for \( \text{state} \).

**Example 39.** Figure 6.7 and Figure 6.8 show how the semantics of a compAnd state vary in terms of the feature model.

Figure 6.7.(a) shows a part of an HFSM comprising a compAnd state with three regions and two simple states with synchronous transitions for input a. We have three feature constraints for each parallel region: F1 for R1; F2 for R2 and F3 for R3. Figure 6.7.(b) shows the semantic FFSM using Feature Model A. We perform the composition in pairs of regions, i.e., region R1 with R2, then the result with R3. The pair order of the composition does not change the final behavior, but changes the composed name of the configuration and how inconsistent pairs are removed (how efficiently we
Figure 6.7: Semantic variation for composing a compAnd state (part 1).
Algorithm 3 Composition of orthogonal states.

1: function COMPOSE_STATES(s, TSC)
2:     for s' ∈ sub(s) do
3:         if type(s') = compAnd then
4:             TSC = compose_states(s', TSC)
5:             init = (s', get_disjunction_constraint(sub(s')))
6:         for R ∈ get_powerset(sub(s')) do
7:             if get_full_conj_const(R, sub(s')) = satisfiable then
8:                 TSC = pairwise_merge(R, init, TSC)
9:     return TSC
10: function PAIRWISE_MERGE(R, init, TSC)
11:     r1 = R.remove(0)
12:     for r2 ∈ R do
13:         r1 = compose(r1, r2)
14:         link_segment(r1, init)
15:         TSC = TSC\{Desc(R)}
16:         TSC = TSC ∪ r1
17:     return TSC
18: function COMPOSE(r1, r2)
19:     for s1 ∈ sub(r1) do
20:         for s2 ∈ sub(r2) do
21:             name = s1 + " * " + s2
22:             feature = fcomp(anc(s1)) + " and " + fcomp(anc(s2))
23:             state = create_state(name, Z3_cond(feature))
24:             comp = comp ∪ merge_transitions(state, s1, s2)
25:     return comp

prune the structure). Dashed states represent unreachable state configurations that are not transformed to conditional states. Due to synchronous transitions, our semantic FFSM has 4 out of 8 state configurations. In the HFSM, each region has a feature constraint that is equivalent to true based on the feature model A, which is similar to the composition of states without feature constraints, i.e., for statecharts.

Figure 6.8.(a) shows the semantic FFSM using Feature Model B. All pair of states are composed, however all features are optional which create behavior segments for specific product configurations, i.e., products that have feature F1 and not F2 and not F3, or products that have features F1 and F3, but not F2. These possibilities increase the number of valid state configurations (worst case). Note that state configurations designated in gray are the same presented in Figure 6.7.(b). Note that there are groups of state configurations with the same feature constraint which belong to the same behavior segment. Thus, we have 7 out of 8 segments that represent all possible combinations. All segments are connected to the initial state configuration that uses as feature constraint the disjunction of all three region feature constraints.
Figure 6.8: Semantic variation for composing a compAnd state (part 2).
Figure 6.8.(b) shows the semantic FFSM using Feature Model C. This scenario is where we greatly reduce the number of state configurations (best case). In this case, the number of segments is the number of alternative features, i.e., 3 segments with one state configuration designated in light gray.

After executing Algorithm 3, we add the rest of state configurations that contains a single state into $TSC$, i.e., $\forall s \in S \cdot \forall s' \in \text{anc}(s) \cdot \text{type}(s') \neq \text{compAnd} \land \text{type}(s) = \text{simple} \implies s \in TSC$. Finally, for each transformed state configuration $TSC$, we create an FFSM conditional state.

**Definition 6.4.15.** Given a set of transformed state configurations (TSC) after executing the state composition (Algorithm 3), the set of conditional states $C$ of an FFSM is defined by: $\forall s \in TSC \cdot (s, f\text{comp}(\text{anc}(s))) \in C$.

### 6.4.2.3 Creating conditional transitions

Semantically, transitions of the HFSM connect several states of the well-formed state structure; in particular, all ancestors of states of a state configuration are active.

**Definition 6.4.16.** Given a transition $t = (a, i, \omega, o, b)$ and a state configuration $SC$, the set of active states $AC$ is all ancestors of $SC$, i.e., $\forall s \in SC \cdot \text{anc}(s) \subseteq AC$.

Some states are activated/deactivated once a transition is performed. To identify those states we define the scope of a transition.

**Definition 6.4.17.** Given a transition $t = (a, i, \omega, o, b)$, the scope of $t$, denoted by $\text{scope}(t)$, is the lowest state in the state hierarchy that is a common ancestor of source and target states, i.e., $\text{scope}(t) = \text{lca}(\{a, b\})$.

When a transition is performed, some states are deactivated while some are activated. The set of states that are activated after executing the transition is called enterSet.

**Definition 6.4.18.** Given a transition $t = (a, i, \omega, o, b)$, the enterSet of $t$ is defined by: all default descendants of $b$ plus all ancestors of $b$ except the ancestors of the scope, i.e., enterSet = $\{d\text{desc}(b) \cup \{\text{anc}(b) \setminus \text{anc}(\text{scope}(t))\}\}$.

We do not include the ancestors of the scope of $t$ because they are already active. The set of states that are deactivated after executing a transition is called exitSet.

**Definition 6.4.19.** Given a transition $t = (a, i, \omega, o, b)$ and a state configuration $SC$, the exitSet of $t$ is defined by: all descendants of $a$ that are in $SC$ plus all ancestors of $a$ except the ancestors of the scope, i.e., exitSet = $\{d\text{esc}(a) \cap \text{Anc}(SC)\} \cup \{\text{anc}(a) \setminus \text{anc}(\text{scope}(t))\}$.

We do not include the ancestors of the scope of $t$ because we do not want to exit those states only to enter again. Once the transition is executed, the current state configuration may change. Simple states of the exitSet are removed and simple states of the enterSet are included in the new state configuration.
Example 40. Given the transition $t = (\text{Menu}, \text{Start}, \text{true}, 1, \text{Rules})$ (Figure 6.5) and the current state configuration $SC_1 = \{\text{PauseGame}\}$, the scope of $t$ is $\text{scope}(t) = \text{RegionA}$, the enterSet is $\{\text{Rules}, \text{R1}, \text{R2}, \text{R3}, \text{Brickles}, \text{Pong}, \text{Bowling}\}$ ($\{\text{Rules}, \text{R1}^* \text{R2}^* \text{R3}, \text{Brickles}^* \text{Pong}^* \text{Bowling}\}$ after orthogonal composition - section 6.4.2.2) and the exitSet is $\{\text{Menu}, \text{RegionM}, \text{PauseGame}\}$. After performing the transition, the new state configuration is $SC_2 = \{\text{Brickles}^* \text{Pong}^* \text{Bowling}\}$. In Algorithm 3 the merge_transition call combine transitions involving different region states. Thus, transitions that leave or reach Brickles, Pong, and Bowling are combined for Brickles $^*$. Pong $^*$. Bowling. The single resulting FFSM conditional transition is created using the name of simple states and their feature constraints: $((\text{PauseGame}, f\text{comp}(SC_1)), \text{Start}, \text{true}, 1, (\text{Brickles}^* \text{Pong}^* \text{Bowling}, f\text{comp}(SC_2)))$.

For transition $t_2 = (\text{Rules}, \text{Save}, \text{W}, 1, \text{SaveGame})$ and the current state configuration $SC_2 = \{\text{Brickles}^* \text{Pong}^* \text{Bowling}\}$, the scope of $t_2$ is RegionA, the enterSet is $\{\text{Menu}, \text{RegionM}, \text{SaveGame}\}$ and the exitSet is $\{\text{Rules}, \text{R1}^* \text{R2}^* \text{R3}, \text{Brickles}^* \text{Pong}^* \text{Bowling}\}$. After performing the transition the new state configuration is $SC_3 = \{\text{SaveGame}\}$. The single resulting FFSM conditional transition is: $((\text{Brickles}^* \text{Pong}^* \text{Bowling}, f\text{comp}(SC_2)), \text{Save}, \text{W}, 1, (\text{SaveGame}, f\text{comp}(SC_3)))$.

6.5 Tool Support

We implemented a tool \(^4\) (Eclipse Public Licence) that has a graphical editor based on the Eclipse platform, which extends the Yakindu GitHub Project \(^5\) (publicity available under Eclipse Public Licence) and is integrated with FeatureIDE \([101]\) (publicity available under Lesser General Public Licence - LGPL), and the Z3 SMT Solver \([24]\) (publicity available under MIT license) for constructing feature models and analyzing feature constraints, respectively. The tool supports modeling, validation, and derivation of HFSM models with the aid of a semantic FFSM. Figure 6.9 shows how the HFSM of Figure 6.5 is modeled in our tool.

Our tool parses HFSMs in a simple textual format and generates a flattened version, after having analyzed the corresponding feature constraints according to Algorithm 3. The resulting FFSM is stored for further analysis (please see below) in a textual format with transitions of the following shape:

```
"source@z3condition -- -- input@z3condition/output -- > target@z3condition"
```

Example 41. The resulting FFSM generated from the HFSM of Figure 6.9 has 4 states and 26 transitions, which is equivalent to the manually modeled FFSM of Figure 6.4. The source of the first transition is the root state. The first and the last FFSM transitions are:

```
"StartGame@true -- -- Exit@((and W (not S))/_1()) -- > StartGame@true"
```

\(^4\)Tool support https://github.com/vhfragal/ConFTGen-tool

\(^5\)Open Source Yakindu Project https://github.com/Yakindu/statecharts
Figure 6.9: HFSM for AGM SPL on the implemented tool.

“Brickles*Pong*Bowling@(or B N W) —— Exit@(and S W)/_1() — > PauseGame@true”

6.5.1 HFSM Syntax Validation

The syntax validation is performed automatically by the implemented tool. To validate
the syntax of an HFSM, we use the feature constraint $\chi$ of a feature model $FM$, and
assertions in the Z3 format. We execute Z3 externally by passing on the constraints in
a smt file type. The generated smt file has three parts: (i) type definitions; (ii) assertion
of the feature constraint of a given feature model; (iii) validation assertions.

Example 42. To validate the syntax of the HFSM of Figure 6.9, we extract the feature
constraints of the feature diagram of Figure 6.2. Then, we prepare the file header using
type definitions and assert $\chi$ (in Z3 format) of $FM$:

(define-sort Feature () Bool)
(declare-const G Feature) (declare-const A Feature) (declare-const M Feature)
(declare-const L Feature) (declare-const C Feature) (declare-const R Feature)
(declare-const B Feature) (declare-const N Feature) (declare-const W Feature)
(declare-const V Feature) (declare-const Y Feature) (declare-const P Feature)
(declare-const S Feature)
(assert (and G (= A G) (= M A) (= L A) (= C G) (= R G) (= (or B N W) R) (not (and B
N)) (not (and B W)) (not (and N W)) (= V G) (= Y V) (= P V) (= > S V))

To validate assertions, we include several checks using assertion blocks. In Z3,
push and pop commands can temporarily set the context (e.g., with assertions), and
once a verification goal is discharged, the context can be reset. The (check – sat)
command evaluates all assertions present in the smt file so far, which return sat or
unsat. We can complete our smt file to check one or more feature constraints following
6.5. TOOL SUPPORT

Figure 6.10: Invalid states (left) and transition (right) in HFSM for AGM SPL.

the structure:
(push)(assert Z3_CONSTRAINT)(check-sat)(pop)

Assume that we need to check two different transitions $x$ and $y$, where the $f_{comp}(x) = (B \land N)$ and $f_{comp}(y) = (W \lor \neg S)$, respectively. Inside a command block which begins with push and ends with pop we could write several assert commands. Some properties such as minimality combine these checks into a single block. However, our simple check only requires one assertion for each $f_{comp}$. Thus, we create push-pop command blocks such as:
(push)(assert (and B N))(check-sat)(pop)
(push)(assert (or W (not S)))(check-sat)(pop)

These assertions result in unsat and sat, respectively. The unsat result means that there is no product configuration that satisfy $(B \land N)$. The sat result means that there is at least one product configuration that satisfies $(W \lor \neg S)$, in this case $\rho_1, \rho_3, \rho_5, \rho_6$, which is a combination of subsets (due to $\lor$ operator), such that $\rho_1, \rho_3, \rho_5$ satisfy $\neg S$, and $\rho_5, \rho_6$ satisfy $W$ (please see Example 32).

6.5.1.1 Well-formed validation

To check the constraints of a well-formed state structure, most of the items (1-7) from Definition 6.4.2 are covered by the Yakindu implementation based on its metamodel. The metamodel ensures that those items are always valid by construction. Hence, only validation of items 6 and 8 were added in our tool. About item 6, we check whether every region state has a substate that is default. Thus, we do not allow empty regions. About item 8, every state must be satisfied by at least one product configuration, otherwise, there is no point in modeling a state that is not going to be used in the SPL.

Example 43. Figure 6.10 (left) shows a state region with an invalid feature constraint which also invalidates all of its descendants (*Brickles*). The *Brickles* state inherits the feature constraints of its ancestors, in this case, $B \& N$ of $R_1$ and true for *Rules, RegionA*, and *AGM* (root). All HFSM states are checked using their feature constraint. The resulting $f_{comp}(anc(R_1))$ is equivalent to $(B \land N)$ in Z3 format, which is unsat (Example 42) according to our *FM* for AGM.

To check well-formed transitions, only item 1 of Definition 6.4.12 is covered by the metamodel and it is valid by construction. Item 2 is checked by Yakindu and we implemented item 3 in our extended tool.

---

6Z3 online tool [https://rise4fun.com/z3](https://rise4fun.com/z3)
Example 44. Figure 6.10 (right) shows an invalid transition due to its feature constraint. The transition $t = (\text{PauseGame, Save, } (B \land N), _0(), \text{SaveGame})$ has an invalid feature constraint that results unsat in our Z3 check.

6.5.2 Semantic Validation

Once we derive a semantic FFSM of our HFSM, the basic and property validation can be performed as briefly introduced in Section 6.3.3 (we refer [37] to for full details).

To check determinism, we check all conditional states and their transitions. We select a conditional state $c = (s, \phi)$ and then select a conditional input $i$. First, we identify the set of satisfiable product configurations for the selected state, i.e., $\forall \rho \in \Lambda \bullet \rho \models \phi \implies \rho \in \Lambda_s$. If $c$ has more than one transition leaving the state with the input $i$, i.e., $t_1 = (c, i, \phi_1, o, c')$ and $t_2 = (c, i, \phi_2, o', c'')$ then we pairwise check whether the resulting feature constraint of those transitions ($t_{comp}(t_1)$ and $t_{comp}(t_2)$) have an intersection of product configurations, i.e, $\exists \rho \in \Lambda \bullet \rho \models t_{comp}(t_1) \land t_{comp}(t_2)$. If they do, then our FFSM is not-deterministic. In other words, a deterministic FFSM cannot have a product configuration with two transitions leaving the same state with the same input.

After checking determinism, we check initially connectedness and minimality, in this order. These properties are checked automatically after saving the HFSM model. To check minimality and initially connectedness, we use other checks to establish whether there are distinguishing sequences and reaching paths, respectively.

Figure 6.11 shows an example of determinism error. The deterministic check fails when we change the feature constraint $W$ to $(W||N)$ of the transition from $\text{StartGame}$ to $\text{PauseGame}$; in that case, it will be in conflict with the self-loop transition of $\text{StartGame}$ with Pause input and $\neg W$ feature constraint. The conflict occurs due to the non-empty intersection of two product configurations that have feature $N$. Thus, checking the feature model we see that both transitions are valid for products configurations with feature $N$.

Figure 6.12 shows an example of initially connectedness error. The initially connectedness check fails when for some product, there is no path from the initial state to some valid state. The conditional state $(\text{SaveGame, } S)$ has the feature constraint $S$ which is satisfied by $\rho_2$, $\rho_4$ and $\rho_6$. To reach this conditional state in all products, we need three distinct paths (from B, N, and W). Once we remove the conditional
transition \((PauseGame, true), Save, N, _1(), (SaveGame, S)\), there is no path that reaches \(SaveGame\) and is satisfied by \(\rho_4\) anymore.

Figure 6.13 shows an example of minimality error. The minimality check fails when we cannot distinguish all pairs of conditional states. The \(Pause\) input is one of the required inputs to distinguish the pair of conditional states \(StartGame\) and \(PauseGame\). Both states have the \(true\) feature constraint, then all product configurations must distinguish in this pair of states. The \(Exit\) input can distinguish the pair for configurations \(\rho_1, \rho_2, \rho_3, \rho_4, \rho_6\). Thus, we need an input that can distinguish the pair for configuration \(\rho_5\). The input \(Pause\) originally can distinguish the pair for configurations \(\rho_5\) and \(\rho_6\), then both inputs are enough to cover the state pair. However, once we change the self-loop transition of \(PauseGame\) altering the output from 0 to 1, we cannot distinguish \(\rho_5\) and \(\rho_6\) anymore as it now distinguish \(\rho_1\) to \(\rho_4\).

### 6.5.3 Model Derivation

Once the HFSM is modeled, we can use a configuration file to select product configurations for AGM. Using the product configuration \(\rho_5\) that is equivalent to \(W \land \neg S\) (Example 32) the tool can derive a pruned HFSM. Figure 6.14 shows the reduced HFSM with only satisfiable elements. The semantic FFSM of the reduced HFSM is:

- \(StartGame@true \rightarrow Start@true/_1() \rightarrow Bowling@W\)
- \(StartGame@true \rightarrow Exit@(and W (not S))/_1() \rightarrow StartGame@true\)
- \(StartGame@true \rightarrow Pause@W/_1() \rightarrow PauseGame@true\)
- \(PauseGame@true \rightarrow Start@true/_1() \rightarrow Bowling@W\)
- \(PauseGame@true \rightarrow Pause@true/_1() \rightarrow PauseGame@true\)
- \(PauseGame@true \rightarrow Exit@true/_1() \rightarrow StartGame@true\)
Figure 6.14: Derived HFSM for AGM SPL.

"Bowling@W ← Start@true/1() → Bowling@W"
"Bowling@W ← Pause@true/1() → PauseGame@true"
"Bowling@W ← Exit@(not S)/1() → Bowling@W"

The semantic FFSM of the reduced HFSM can be derived into an FSM for \( \rho_5 \) by removing the feature constraints of its elements.

### 6.6 Body Comfort System Case Study

We illustrate and evaluate our approach in a prototypical implementation using a case study from the automotive domain, a simplified Body Comfort System (BCS) for the VW Golf SPL [60]. The FeatureIDE tool [101] was used to elaborate Feature Models and their configurations. The original BCS system has 19 non-mandatory features and can have 11616 configurations.

Figure 6.15 presents an adapted version of the feature model used to handle a part of the features with 4 non-mandatory features and 6 possible configurations for 4 components: `Finger_Protection_FP` (FP) blocking the window movement when a finger is clamped in a window, `Manual_ManPW` (ManPW) or alternatively `Automatic_AutPW` (AutPW), and `Central_Locking_System_CLS` (CLS) with optional `Automatic_Locking_AL` (AL) when the car is driving. In Example 3, we show that states with alternative features can be composed for FFSMs. The behavior of `ManPW` and `AutPW` components are similar and exclusive, thus we can combine them in a single region by adding product-specific conditional transitions.

The behavior of components can be checked individually or in groups. In groups, they can be composed of parallel regions using hierarchical models or elaborated individually using flat models. Figure 6.16 presents the HFSM of four selected compo-
6.6. BODY COMFORT SYSTEM CASE STUDY

Figure 6.15: Adapted Feature Model of the Body Comfort System [60].

Two alternative components were modeled in the PowerWindow region, and product-specific transitions represent the behavior in each case. The only non-mandatory region is CentralLockingSystem which means that for different products, we have to compose: (i) all three regions; and (ii) only the first two on the left. Region composition is explained in Figure 6.7 and Figure 6.8.

6.6.1 Results

The HFSM presented in Figure 6.16 was validated regarding the syntax and semantics. To check the basic syntax and derive the semantic FFSM it took less than 2 seconds. The resulting semantic FFSM of BCS for the selected four components has 17 conditional states and 171 conditional transitions, and it took approximately 2 minutes to perform semantic checks for all three properties. The running environment used Ubuntu 15.04 (64 bit) operating system on an Intel processor i7-5500U at 2.40GHz with 12GB of RAM. Some experimental results about the validation time of such properties in FFSMs were presented in [37].

Once the HFSM was validated, we chose a product configuration to derive and validate partial specifications. We pruned our HFSM for a subgroup of product config-
Figure 6.16: HFSM of 4 components of BCS.
6.6. BODY COMFORT SYSTEM CASE STUDY

For example, in a feature model configuration file we selected the Automatic Power Window component and left the Central Locking System and Automatic Locking features unchecked, resulting 3 out of 6 possible configurations. By using a feature constraint that ignores the uncheck features we derived an HFSM for those 3 product configurations. Figure 6.17 shows the resulting HFSM for 3 product configurations. We also derived another specification for a single product configuration by excluding other features, similar to the example presented in section 6.5.3. The time that it took to derive such models was 1 second.

6.6.2 Discussion of the Results

Scalability is the main issue of our approach as for any hierarchical model. The impact in terms of scalability concerns semantic validation that may increase as we add more parallel states. The BCS is no different, as it has several parallel regions containing 1 to 9 states. Composing all regions results in more than 50000 states which is a challenge for our semantic checks, i.e. the initially connected which may require checking several paths to each state. Also, we show in Section 6.4.2.2 that composing

Figure 6.17: HFSM derived for 3 configurations with 3 components.
parallel regions with features may increase the number of semantic states. This is a threat to validity for the applicability for real-world cases.

In terms of modeling, the feature constraint usage is straightforward and easy to understand. A given element only exists for the satisfiable configurations of the feature model. Regarding the maintenance, we achieve a compact representation of the SPL behavior using an HFSM. However, very large specifications require a substantial amount of time to validate.

One approach to test the BCS is to work with parts of the specification in order to verify its behavior. The original BCS model has several components that work in parallel. One issue of modeling the whole system is the state explosion of composing all regions. Each component can be verified alone or we can combine few components for validation.

### 6.7 Related Work

Usually, an SPL can generate several similar products where only a few features vary from one to another. One challenge in SPL is the verification of products using a simplified behavioral model that takes advantage of the similarity between products. There are proposals [15, 17] that provide a concise hierarchical formalism for representing SPL behavior in one model. However, they are focused on model checking [17] or simple test criteria such as boundary tests [15]. In this paper, we lift the definition of an FFSM model to handle hierarchy with an HFSM. Thus, we provide syntax and semantic checks looking forward to test generation methods.

Regarding configurable models for SPLs, most modeling concepts for variability can be distinguished into three main approaches: annotative, compositional and transformational variability modeling [91]. Compositional approaches for modeling variability capture variation by selecting specific component variants. Compositional variability modeling [44] allows a modular description of variability but limits the impact of changes to the applied composition technique. Transformational approaches represent variability by transformation of a base architectural model. Model transformation rules guide the derivation of products by performing additions, modifications or removals using variability. For example, delta modeling [16] can represent variability in model transformation which a core system is developed, and subsequent products are derived by executing such transformations rules.

Annotative approaches use variant annotations (also called 150%-models), e.g., UML stereotypes in UML models to define which model elements belong to specific product variants. In the orthogonal variability model (OVM) [84], a separate variability representation with links to the architecture model replaces direct annotations. Some approaches [20, 43] propose a pruning-based approach to UML 150% test model for SPLs, separating variability from the base models using mapping models.

Using an annotative 150% statechart test model and transition coverage criteria, Cichos et al.’s approach [15] presents SPL test design for complete test model coverage with subsequent product subset selection for test suite execution. Weissleder et al. [112]
propose an approach for automatic test suite derivation based on reusable UML state machine test models and OCL expressions. As in Featured Transition Systems [17], model fragments are annotated with presence conditions, i.e., Boolean expressions that define to which products a fragment belongs.

Our proposed approach for configurable HFSMs is based on a specification that uses features of an SPL as feature constraints. To our knowledge, however, there are only a few pieces of research that extend formal models to the SPL level; examples of such work include approaches based on Labeled Transition Systems [64, 65, 106] and feature-oriented approaches [7, 28]. However, the approach proposed in [7, 8] exploits non-deterministic models, and semantic validation of models is not an issue in their approach. Thus, we are not aware of any prior study that uses formal models with hierarchy in the SPL context to validate properties such as determinism, initially connectedness, and minimality.

6.8 Conclusion

In this paper, we presented the Hierarchical Featured State Machine (HFSM) formalism for representing behavioral test models in the Software Product Line (SPL) context. The HFSM improves the modeling of SPL behavior compared to FFSMs by providing compact representations.

Inspired by statecharts and UML, we defined the syntax and semantics of HFSMs. In the syntactic part, we checked well-formed state structures and transitions. In the semantical part, we used an FFSM to represent our HFSM. State configurations were transformed into conditional states. We also addressed the composition of orthogonal states, which is the main problem regarding scalability. We show how using feature constraints in regions can lead to scalable models for SPLs. After the identification of all FFSM conditional states, we create the conditional transitions using enter and exit sets. Then, we perform the semantic check on the FFSM with basic validation properties, that are often prerequisites for test-case generation techniques.

To aid the validation of our HFSM, we implemented a tool by adapting the Yakindu project. We added several checks regarding syntactic and semantical validation properties. We also explained how we use the Z3 tool to verify our feature constraints. The tool performs all the checks automatically after saving the model. Moreover, the tool provides model derivation commands that are useful to create partial HFSM models for a single or a group of product configurations.

Finally, we used the Body Comfort System as a case study. We noticed that we could not analyze the whole specification due to the well-known state explosion problem: the resulting flat FFSM would have more than 50000 states. Thus, we use check parts of the specification. The results indicate that our small HFSM is able to represent the parallel SPL behavior of four components each having two to three states, which is equivalent to an FFSM with 17 states and 171 transitions.

As future work, we plan to use HFSMs to extend FSM-based test-case generation methods to SPLs. We also plan to include history, deep-history, join/forks, and
entry/exit points connections in the HFSM model. Moreover, we plan to explore the state explosion problem identified in the HFSM used in the case study.
Bibliography


[95] SEI. A framework for software product line practice, 2011. (Cited on pages xi, xii, 27, 42, 57, 58, 110, 111, and 117.)


Glossary

AGM  Arcade Game Maker
BCS  Body Comfort System
CFC-SM Configurable Feature-based full Coverage testing of State Machines
ConFTGen Configurable Full Test Generator
DSL  Domain Specific Language
EMF  Eclipse Modeling Framework
FSM  Finite State Machine
FFSM Featured Finite State Machine
GEF  Graphical Editing Framework
GMF  Graphical Modeling Framework
IDE  Integrated Development Environment
IRT-SPL Incremental Regression-based Testing for Software Product Lines
HFSM Hierarchical Featured State Machine
HSI  Harmonized State Identifier
HSM  Hierarchical State Machine
MBT  Model-Based Testing
SAT  Satisfiability
SMT  Satisfiability Modulo Theory
SPL  Software Product Line
SPLE Software Product Line Engineering
UI  User Interface

VV&T  Verification Validation and Testing