Model-Based Testing of Reactive Software Product Line with Formal Methods

by

Ali Irannezhadi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
January 2023

Abstract

With the increasing complexity of software in recent years model-based testing has attracted more attention in industry and research, and the task of functional test of the reactive software product lines is increasingly challenging and time consuming. In model-based testing test cases are automatically generated to verify the correctness of the implementation of a system according to the model that describes the expected behavior of that system. In this research, a model-based test method is presented for the functional test of the reactive software product lines. In the proposed method, the system specification model is described by an extension of the finite state machine model. In other words, a formal method is presented to apply variability in finite state machines. Then, one of the efficient test methods on finite state machines is selected and refined to be used for production line testing. The conducted tests show that the obtained method reduces the time required to generate the requisite test cases compared to the original method.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Stefan D. Bruda. This work would not have been possible without his support, patience, and guidance.

I would like to thank the Computer Science department at Bishop's University for giving me the opportunity to pursue a Master's degree.

I would like to thank all other professors in the Department of Computer Science at Bishop's University, especially Dr. Mohammed Ayoub Alaoui Mhamdi from whom I learned a lot.

Above all I would like to express my very profound gratitude my wife, Nasim, thanks for all your love, support, motivation and being always on my side. But most of all, thank you for being my best friend. I owe you everything.

I would like to extend my sincere thanks to my parents. Thank you for your love, support, and unwavering belief in me. Without you, I would not be the person I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Various needs of customers have led to mass customization in many industries. Providing customized products at a reasonable cost drives artisans to product line engineering. Product line engineering reduces development costs and time, also increasing product quality compared to single system development. Due to the wide variety of products, product quality testing is a complex and costly matter. Also, due to the reduction of the development cost of each product, the share of the test cost in relation to the total cost in product line engineering has increased and made the test issue more critical.

The purpose of a software product line is to produce efficient and disciplined products. Software product line engineering provides the possibility of producing products with a lower cost in a shorter time and with higher quality. A basic concept in software product line engineering is reuse, which has led to its division into two processes: domain engineering and application engineering. In product line engineering as defined by ISO26550:2015, Domain Engineering is complemented by Application Engineering which takes care of the life cycle of the individual products derived from the product line [1]. Reusable components are generated in domain engineering and used in application engineering to produce a customized product [2]. The description of a product in a product line consists of a constant and a variable part. The constant part describes the common aspects between all products and the variable part describes the different aspects between different products, called variability [3]. The products of a software product line are distinguished from each other by their features. In fact, each feature adds functionality to the product. A distinct combination of features defines each product [4].

Any business that spends a significant portion of its budget on software development must implement effective testing strategies. In an organization using software product lines (SPL) testing strategies are even more crucial since the share of testing costs increases as the development costs for each product decreases. Testing of a software product line is a complex and costly task since

the variety of products derived from the product platform is huge. In addition to the complexity of stand-alone product testing, product line testing also includes the dimension of what should be tested in the platform and what should be tested in separate products [5]. Testing consumes up to 50% of the total effort in single system engineering. This percentage increases in software product line engineering, because the effort of constructing applications decreases due to comprehensive reuse [6].

Systematic testing is one of the most important and widely used techniques to check the quality of software. The current tendency is that the effort spent on testing is still increasing due to the continuing quest for better software quality, and the ever-growing size and complexity of systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. One of the new approaches to meet the challenges imposed on software testing is model based testing. In model based testing a model of the desired behavior of the implementation under test (IUT) is the starting point for testing. The main virtue of model-based testing is that it allows test automation that goes well beyond the mere automatic execution of manually crafted test cases. It allows for the algorithmic generation of large amounts of test cases, including test oracles starting from the model of required behavior [7]. Several methods for testing software product line model have been developed. An overview is given in Section 3.

Reactive systems are software and hardware systems with a (usually) non-terminating behavior that interact through visible events such as Web servers, communication protocols, operating systems, smart cards, processors, etc. Model-based testing is one of the common test methods for reactive software [8]. In this thesis we develop a model-based test method for product line of reactive systems, which is further explained in the next section.

1.1 Motivation and Research Problem

Several methods in the field of model-based testing of software product line have been developed, but the models that are considered to describe the characteristics of the product line are not based on formal notation. Indeed, most of the test methods use Unified Modeling Language (UML) models. We attempt to remedy this by using formal specification models. Software product lines are widely used

in industrial fields, and formal models, including finite state machines, are used to model systems in various fields, including industry. We also use finite state machines in our effort to model product lines of reactive systems, since this kind of automata are one of the suitable models for reactive systems. It should be noted that one of the other methods for modeling reactive systems is the use of Input Output Labeled Transition System (IOLTS) [8]. The relation of input-output conformance, called input output conformance testing (IOCO testing) is a method of model-based testing for IOLTS. In this method, an infinite number of different test cases are generated from the system description model [7]. Therefore, it is not possible to apply all test cases, and it is further impossible to estimate the quantity of particular test cases. The number and length of test cases are used instead in evaluating the effectiveness of the method. According to what was mentioned before and also the finiteness of the test theory [8], we argue that finite state machines are a more suitable choice for our model. The field of work in our research area is therefore limited to systems that can be modeled with a finite state machine. To use a finite state machine, we further need to provide a way to import variability into the machine.

We know that a fundamental concept in the software product line is reuse. In our work we want to use this concept in the product line test. For this purpose, we chose the *W* test method [9], which is one of the efficient test methods for finite state machines, and by applying necessary changes, we obtain a new variant capable of testing software product lines. In the continuation of this thesis, we will see how we reuse the generated test cases to test different products of the software product line. Our method aims to reduce the cost of the testing process. In order to evaluate this, we compare our method with the process of applying the *W* test method to each product of the product line. The details of the experiments and their results are given in Chapter 5.

1.2 Summary of research achievements

Among the achievements of this research, the following can be mentioned:

- Importing variability in a finite state machine: The starting point of model-based testing is to have a model of the desired and expected behavior of the system. Since in this research the finite state machine is chosen for modeling, it is necessary to expand it in such a way that it also shows the variability of the product line.

- Presenting an efficient model-based test method, along with applying the concept of reuse: In this research, we change the *W* test method in such a way that it produces reusable test cases for testing the product line of reactive software.

By performing the tests and observing the results, we will see that the production time of the test cases required for testing all the products of the software product line has been reduced by using our method compared to the *W* method. The cost of product line testing includes the cost of producing test items and also the time necessary to apply them to products. By reducing the first component we reduce the overall cost of the testing process.

1.3 Thesis structure

We organized this thesis in 6 chapters. Chapter 2 is dedicated to the background of the research, which includes model-based testing, finite state machine and its testing methods, and the software product line. In Chapter 3, we summarize the previous work related to our research topic. Specifically, we first introduce several software product line modeling methods, then discuss software product line test methods, and finally, we outline the test-based model for reactive systems. In Chapter 4 we describe our method. Chapter 5 is devoted to the evaluation of the proposed test method. Finally, we summarize our work as well as state some future research directions opened by our effort.

Chapter 2

Preliminaries

The concepts and methods used in this research are briefly described in this chapter. The first part describes model-based testing. In the second part we first discuss the definitions and basic concepts related to finite state machines, and then we explain the *W* test method, which is an efficient method for testing finite state machines [8], [9], [10], [11]. In the last part we cover the software product line, variability, and the feature model. It should be noted that the examples presented in this chapter are not adopted from any source and therefore are not accompanied by any citation.

2.1 Model-based testing

Model-based testing is one of the most efficient techniques to deal with software testing challenges. In model-based testing, the implementation under test (IUT) is tested for conformance to a model that specifies the expected behavior of that implementation. Therefore, the start point of this process is having a model of the expected behavior of the implementation under test. The main advantage of model-based testing is test automation. Model-based testing allows many test cases to be generated algorithmically and automatically from a given model. If the model is valid and accurately describes what the system under test should do, then all the generated tests will also be valid.

From the industrial point of view, model-based testing is a promising technique for improving the quality and efficiency of testing and reducing its cost. Test automation focuses only on the automated execution of test cases. The purpose of model-based testing however is to automatically generate high-quality test cases from the models, so it completes the automatic execution of the tests. Model-based testing is an extension of formal methods and verification techniques. Model-based testing and formal verification pursue complementary goals. With formal verification techniques, we can prove that the model of a system satisfies a number of desired characteristics [7]. The model-based test is

based on a valid model of the system; therefore, it shows that the actual and physical implementation of the system behaves in conformance with this model. It should be noted however that due to the inherent limitations of the tests, including the limited number of tests that can be performed and the lack of completeness of the tests, model-based testing can only show the presence of faults, not their absence. Model-based testing is generally not complete. However, model-based testing of finite automata which we did in this thesis is complete.

There are different types of model-based tests, depending on the models which are used, qualitative aspects which are tested, the desired level of formality, and the level of accessibility and observability of the system. In this thesis we consider model-based testing to be formal, specification-based, and black-box. The basis and starting point of the tests is the specification that determines what the system under test should or should not do. The specification is given in the form of a behavioral model, which is assumed to be correct and valid. In addition, the test is a black-box, that is the implementation under test is considered a black-box without internal details and can only be accessed and viewed through its external interface. The test is formal because the specification that determines the desired and expected behavior of the system is defined by a formal language whose syntax and semantics are defined in details. Of course, in addition to the formal specification the method includes the formal definition of the meaning of conformance for the system under test as well as an algorithm to generate the tests.

A formal specification-based testing framework is based on several concepts. The first one is the implementation under test. An implementation can be an actual physical object such as a hardware component, a computer program with all its libraries that is running on a processor, an embedded system consisting of software embedded in a physical device, or a process control system with its sensors and actuators. Since the test is a black-box, the implementation is also treated as a black-box. This means that the implementation interacts with its environment, but there is no information about its internal structure. The only way a tester can control or observe an implementation is through its interface. The correctness of an IUT is defined as its conformance to a specification. To check the conformance of the IUT with the desired specification we need to formally define the concept of conformance [7]. Input-output conformance (IOCO) testing [7] is worth mentioning as a practical and important model-based testing framework. In this test, the specification is modeled using an input/output labeled

transition system.

2.2 Finite state machine

Finite state machines (FSM) are used to model systems in various fields, including sequential circuits and communication protocols in networks. The need for reliability in these systems has led to extensive research in the field of testing finite state machines. These machines are also widely used in modeling reactive systems. The formal definition of finite state machine is as follows.

Definition 2.1 [8]: *A finite state machine is the 6-tuple* $(I, O, S, \delta, \lambda)$*, where* I*,* O*, and* S *are finite and non-empty sets of input symbols, output symbols, and states, respectively.* δ *is the state transition function,* $\delta: S \times I \rightarrow S$ *and* λ *is the output function,* $\lambda: S \times I \rightarrow O$.

Each FSM can be represented by a directed graph called transition diagram whose nodes and edges correspond to the states and transitions of the machine, respectively. Each edge is labeled with the input and output of the corresponding transition. Finite state machine test problems are classified into two types. In the first type the transition diagram of the machine is known, but the current state of the machine is not known. By applying an input sequence and examining its input/output behavior, some information about the current state can be obtained. The test sequences used to solve this problem are called distinguishing sequences. The other type of test is the conformity test, in which a finite state machine with a known transition diagram is given as the specification. The implementation is a black-box with only its input and output behavior visible. In this type of test, the conformance of the implementation machine with the given specification machine is checked.

Although the FSM is a simple model, the conformance test for this machine is significant and useful in practice. Indeed, FSMs are used to describe a variety of systems including digital circuits, embedded control systems, and protocols. In addition, many formal notations for describing communication protocols including state diagrams in unified modeling language (UML) and specification and description language (SDL) [12] are very similar to a finite state machine.

In what follows a number of definitions related to the finite state machine are presented, and then two conformity test methods are given for this machine.

2.2.1 Basic definitions

Let $M = (I, O, S, \delta, \lambda)$ be a finite state machine. The operation of M in state s_1 on input sequence $x=a_1a_2...a_k \in I^*$ takes the machine to the states $s_2, s_3, ..., s_{k+1}$ and generates the output sequence of $b_1b_2...b_k \in O^*$, so that for $i = 1, 2, ..., k$, the states, and outputs are $s_{i+1} = \delta(s_i, a_i)$ and $b_i = \lambda(s_i, a_i)$, respectively. Output and state transition functions can be extended from an input symbol to a sequence of input symbols using these recursive definitions, respectively:

$$\delta(s, \epsilon) = s,\ \delta(s, ax) = \delta(\delta(s, x), a)$$
$$\lambda(s, \epsilon) = \epsilon,\ \lambda(s, ax) = \lambda(s, x)\,\lambda(\delta(s, x), a)$$

In addition, these functions can be extended over a set of states instead of just one state as follows, with $Q \subseteq S$ a set of states:

$$\delta(Q, x) = \{\delta(s, x): s \in Q\}$$
$$\lambda(Q, x) = \{\lambda(s, x): s \in Q\}$$

Two states s_i and s_j of machine M are equivalent if and only if for every input sequence applied to s_i and s_j the machine produces the same output sequence. In other words, for any arbitrary input sequence x, we have $\lambda(s_i, x) = \lambda(s_j, x)$. Otherwise, the two states are not equivalent and are separated by a separating sequence. The formal description of a separating sequence is given in Definition 2.2. The definition of equivalence for two states in different machines with the same set of input and output symbols will be similar. Two machines are equivalent if and only if for every state in M there is an equivalent state in M' and vice versa.

Definition 2.2 [8]: *The sequence $x \in I^*$ is the separating sequence for $s_i, s_j \in S$, if* $\lambda(s_i, x) \neq \lambda(s_j, x)$.

We know that in a minimal finite state machine no two states are equivalent. In the following we present more properties of minimized machines. Every two states in a minimized machine with n states have a separating sequence of length at most $n-1$. In order to generate separating sequences for the states of a machine, we first form the partition $p_0, p_1, ...$ of the set of states. Two states s and t are placed in the same class p_i if and only if they have no separating sequence of length i. In other words, for each sequence $x \in I^*$ that $|x| \leq i$, $\lambda(s, x) = \lambda(t, x)$.

Generally, $p_0 = \{S\}$ and p_{i+1} is obtained from p_i.

Lemma 2.1 [8]: *If $\rho_{i+1} = \rho_i$ for some i, then the rest of the sequence of partitions is constant, i.e., $\rho_j = \rho_i$ for all $j > i$.*

Proof [8]**.** We prove the equivalent, contrapositive form: $\rho_{i+1} \neq \rho_i \Rightarrow \rho_i \neq \rho_{i-1}$ for all $i \geq 1$. If $\rho_{i+1} \neq \rho_i$ then there are two states $s, t \in S$ with a shortest separating sequence of length $i+1$, say $ax \in I^{i+1}$ (i.e., a is the first letter and x the tail of the sequence). Since ax is separating for s and t but a is not, x must be separating for $\delta(s, a)$ and $\delta(t, a)$. It is also a shortest separating sequence, because if $y \in I^*$ was shorter than x, then ay would be a separating sequence for s and t, and shorter than ax. This proves that there are two states $\delta(s, a), \delta(t, a)$ with a shortest separating sequence of length i, so $\rho_i \neq \rho_{i-1}$. ■

In a minimized machine these classes will eventually contain only single states. Since the maximum length of the separating sequence in a machine with n states is $n - 1$, these classes do not change from $i = n-1$ onwards. To generate separating sequences for the states of a finite state machine, we first form $p_0, p_1, \ldots p_r$, where r is the smallest index such that p_r contains only single states. Two states $s, t \in S$ belong to different classes of p_1, if and only if there is an input symbol $a \in I$, So that $\lambda(s, a) \neq \lambda(t, a)$. In this way, p_i is calculated. For $i > 1$, ρ_i is obtained from ρ_{i-1}. Two states $s, t \in S$ belong to different classes of p_i, if and only if for an input symbol $a \in I$ the states $\delta(t, a)$ and $\delta(s, a)$ belong to different classes of p_{i-1}. In this way, all the p_i can be calculated for $i > 1$.

To obtain a separating sequence with the shortest length for states' s and t, we find the smallest index i such that s and t belong to different classes of p_i. As mentioned in the proof of Lemma 2.1, in the separating sequence of the form ax, the sequence x is the shortest separating sequence for $\delta(t, a)$ and $\delta(s, a)$. So, we choose the input symbol a that takes s and t into different classes of p_{i-1}. That is, $\delta(t, a)$ and $\delta(s, a)$ should be placed in different p_{i-1} classes. We continue this process until we reach p_0. The concatenation of these input symbols forms the separating sequences s and t. In order to better understand what was said, let us consider an example.

We want to find a shortest separating sequence for states s_2 and s_3 from Figure 2.1. It is necessary to first partition the set of states. As we know, $p_0 = \{S\}$. The states s_0 and s_1 generate the same outputs for both inputs in $I = \{a, b\}$ and thus are placed in the same class of p_1. States s_2 and s_3 are also placed in the same

class for the same reason. In this case, we have: $p_1 = \{\{s_0, s_1\}, \{s_2, s_3\}\}$. Because δ (s_0, a) and δ (s_1, a) are in different classes of p_1, s_0 and s_1 each form a singleton class in p_2. While states s_2 and s_3 are still in the same class as p_2. As we know, states s_2 and s_3 with input b go to states s_0 and s_1, respectively, which are in different classes of p_2, so these two states are also separated in p_3. Sections p_0, ..., p_3 are given below.

$$p_0 = \{\{s_0, s_1, s_2, s_3\}\} \qquad p_1 = \{\{s_0, s_1\}, \{s_2, s_3\}\}$$
$$p_2 = \{\{s_0\}, \{s_1\}, \{s_2, s_3\}\} \qquad p_3 = \{\{s_0\}, \{s_1\}, \{s_2\}, \{s_3\}\}$$

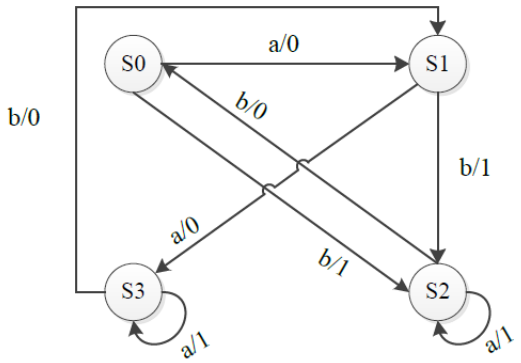


Figure 2.1. Finite state machine for an example system

Note that p_3 is the first partition where s_2 and s_3 are in different classes. By applying the input symbol b to these states, the machine goes to states s_0 and s_1, which are in different classes of p_2. In this way we found the first symbol of the separating sequence. States s_0 and s_1 also go to s_1 and s_3, respectively with input a, which are located in separate classes in p_1. So, a is the second input symbol of the separating sequence. s_1 and s_3 produce different outputs with both input symbols a and b. In this case, the last input symbol can be a or b, which leads us to the sesparating sequences *baa* and *bab* for states s_2 and s_3.

2.2.2 Equivalence of finite state machines

Suppose $M = (I, O, S, \delta, \lambda)$ and $M' = (I, O, S', \delta', \lambda')$ are two finite state machines with the same set of input and output symbols. The homomorphism Θ from M to M' is a mapping from S to S', such that for each state $s \in S$ and each input symbol $a \in I$ the following relation holds:

$$\delta'(\Theta(s), a) = \Theta(\delta(s, a))$$
$$\lambda'(\Theta(s), a) = \lambda(s, a)$$

If Θ is a one-to-one function, it is called an isomorphism. Machines M and M' are isomorphic if there is an isomorphic mapping from one to the other. In this case, the two machines have the same number of states and have the same behavior regardless of the differences in state names. As a result, it can be said that isomorphic finite state machines are equivalent.

Equivalence of machines is an equivalence relation on all FSMs with the same sets of input and output symbols. Each equivalence class contains a machine with a minimum number of states, called a minimized machine. A finite state machine is minimized if and only if no two states in it are equivalent. All minimized machines in an equivalence class have the same number of states. For any two minimized machines in a class, there is also a one-to-one correspondence between the equivalent states, which defines a one-to-one isomorphism between them.

For each finite state machine, its equivalent minimized machine can be obtained. Equality of states is an equivalence relation on the set of states that divides them into equivalence classes. An algorithm for classifying equivalent states exists [11]. Each class contains equivalent states and the states of different classes are inequivalent.

To obtain the minimized machine of a finite state machine, we first obtain the equivalence classes for states. Suppose that for the machine M = (I, O, S, δ, λ) classes $B_1, ..., B_r$ have been determined. Note that $\{B_1, ..., B_r\}$ is a partition on S, meaning that $B_i \cap B_j = \emptyset$ $(i \neq j)$ and $U_{i=1}^{r} B_i = S$ holds. Each state of S is in only one of these classes. In the case that s_i and s_j are in the same class, they produce the same output for each $a \in I$, and also $\delta(s_i, a)$ and $\delta(s_j, a)$ are in the same class. To build the minimized machine M', we consider each class B_i as one state. In this case, the set of states of the minimized machine will be $S' = \{B_i | \ i = 1, ..., r\}$. We know that all the states in B_i go to states that are all in the same block B_j on an input symbol a, and also produce the same output o. Therefore, we have: $\delta'(B_i, a) = t$ and $\lambda'(B_i, a) = o$. In this way, the output and transition functions of the minimized machine M' are obtained.

2.2.3 Conformance testing

This section is dedicated to the problem of finite state machine conformance testing. In the definition of conformance testing, the finite state machines M_S and M_I describe the specification and implementation of the system, respectively. It is

assumed that the transition diagram of M_S is known, while in the case of M_I only its input-output behavior is visible. In other words, the machine under test is a black-box. We want to check whether M_I implements M_S correctly, that is, whether M_I is in conformance with M_S or not. Conformance testing is also called fault detection, because the goal is to detect points where M_I has not implemented M_S correctly. Conformance is defined as an equality or an isomorphic relationship between two machines that specify and implement the system. We say that M_I conforms to M_S if and only if their initial states are equivalent, that is, they produce the same output for each input sequence. In order to check the conformity, a set of input sequences is generated from machine M_S and applied to machine M_I. If for each input sequence M_I produces the expected output sequence (that is, the output produced by M_S), we say that M_I conforms to M_S.

Each pair of input sequence and expected output sequence is considered as a test case. The collection of test cases is called a test suite. Applying the test cases one after the other is equivalent to applying the sequence obtained from the concatenation of these test cases. This input sequence is called checking sequence. Its formal definition is given as follows.

Definition 2.3 [8]: *A checking sequence for a finite state machine M_S is a sequence of input symbols that distinguishes the class of machines equivalent to M_S from other machines.*

The common goal of all testing methods is to check the conformity of M_I with M_S by generating a checking sequence or a set of input sequences as test cases. The difference between these methods is in the cost of producing test cases and the fault detection ability. The shortness of the test cases increases the applicability of the test method. On the other hand, a test suite should cover the implementation as much as possible and discover its faults. The main difference between these methods is in the assumptions that each makes about M_S and M_I machines. Some methods can only be used under certain assumptions. while others are used with more general assumptions, but they produce a very long checking sequence. The following four conditions must be met for each test method:

- M_S is minimized. Equivalent machines have the same input-output behavior, so it is not possible to distinguish them by observing the outputs.

For this reason, any machine that describes system specifications must be minimized.

- M_S machine is completely specified. This means that its state transition and output functions are defined for each state $s \in S$ and each input $a \in I$.
- M_S is strongly connected. That is, all states can be obtained from all other states through one or more transitions. In some methods, it is enough that all states can be obtained from the initial state. These methods require a reset message to reset the machine, otherwise a dead-end may cause the test to stop. In this situation, the existence of the reset message replaces the strong connection requirement in the specification machine.
- M_I does not change during the test. In addition, its set of input and output symbols is the same as M_S.

These four conditions are mandatory and are assumed in all test methods. However, there are other conditions that are not mandatory, but help the testing process. These conditions are listed below.

- Equality of the number of states: the number of states in M_I is equal to M_S. In this case, it can be concluded that possible faults do not increase the number of states. Based on this assumption, possible faults in M_I are output errors and transition errors. An output fault occurs when a transition in the implementation machine produces incorrect output. A transition error is when the implementation machine goes to the wrong destination state. A more general assumption is that the number of states of machine M_I has an upper bound m, which can be larger than the number of states of M_S.
- Reset message: Each of the M_I and M_S machines has a special input called reset, which takes the machine from any state to the initial state and does not produce any output. In other words, for each $s \in S$ we have δ (*s, reset*) = s_1 and λ (s, reset) = -.
- Status message: M_I and M_S machines have a special input called status, which returns their current state as its output. The state of the machine does not change by applying this message. In other words, for each $s_i \in S$ we have δ (s_i, *status*) = s_i and λ (s_i, *status*) = i.

- Set message: When the set s_j message is received in the initial state, the machine goes to state s_j and does not produce any output. In other words, for each $t \in S$ we have δ (*reset*, *set*(t)) = t and λ (s, *set*(t)) = -.

Assuming that all the conditions mentioned above are met, a simple conformance test using the set message is given below.

Algorithm 2.1 [8]: Conformance test using set message.
The following steps are repeated for each $s \in S$ and each $a \in I$:

1. By applying the reset message, we transfer the M_I machine to the initial state.
2. By applying the set message, we move the M_I machine to the s state.
3. We apply the input symbol a.
4. We check the conformance of the generated output with the output of M_S.
5. By applying the status message, we check the conformance of the destination state with the expected state, i.e., $\delta_S(s, a)$.

Algorithm 2.1 checks whether M_I has correctly implemented M_S or not. This algorithm detects any output and transition faults in Steps 4 and 5, respectively. In addition to the input symbols belonging to I, it is necessary to test the *set*, *reset* and *status* messages as well. In order to test the *status* message, in every s_i state, after using *set* s_i, *status* should be applied twice. First, it must be applied in Step 3, to ensure that the *status* is in state s_i and gives the correct output i. If the implementation of *set* is wrong and takes the machine to s_j instead of s_i, and the *status* message in s_j gives the wrong output i; this fault will be discovered during the test of s_j. The second use of *status* is in Step 5, in order to check that the first use of *status* did not change the status. Once we are sure of the correct implementation of this message, we can test *set* and *reset* by applying them to any state and checking the correctness of the destination state by *status*.

The checking sequence of Algorithm 2.1 is obtained from the concatenation of *reset*, *set*(s), a and *status* for each $s \in S$ and each $a \in I$. Its length is $4pn$ where $p = |I|$ is the number of input symbols and $n = |S|$ is the number of states of the machine. The main weakness of Algorithm 2.1 is the need for the *set* message, which is not always available. Sequences that traverse the machine and go through each state and transition at least once can be used in place of *set*.

In the method that will be presented below the *status* message is not used to

specify the current state of the machine. This method assumes that the machines do not have a *status* message (but have a *reset* message), and use separating sequences to detect a state. It worth noting that because M_S is minimal no two of its states are equivalent. As a result, for both states s_i and s_j there is an input sequence x, which is considered a separating sequence and differentiates these two states by generating different outputs i.e., $\lambda (s_i, x) \neq \lambda (s_j, x)$.

2.2.4 The *W* test method

The *W* method is one of the most widely used methods for testing systems modeled as finite state machines. This method only needs the *reset* message, and it uses a covering transition set and a separating set, which are used to test transitions and recognize states, respectively. The minimality of the machine that describes the specification (M_S), is necessary to calculate the separating set and is considered one of the necessary conditions for using the *W* method. Another condition is that M_S and M_I must be deterministic and completely specified, and all their states can be accessed from the initial state. The number of states of M_I has an upper limit of m, which can be greater than n, which is the number of states of M_S.

In the *W* method, all the transitions of M_I are traversed using the transition covering set. In each transition, in addition to comparing the output with the output of the corresponding transition in M_S, the correct destination of the transition must also be checked. For this purpose, the separating set is used. The transition cover set is formally defined as follows.

Definition 2.4 [8]: *The transition cover set P of the finite state machine M_S is a set of input sequences such that for each $s \in S$ and $a \in I$, there is a sequence x in P that starts from the initial state and ends at s with transition a. In other words,*

$$\forall\ s \in S \text{ and } \forall\ a \in I, \exists\ x \in P \text{ such that } x = y.a \text{ and } \delta(s_1, y) = s$$

The set P can be obtained using a breadth-first traversal of the transition diagram of M_S. This set is closed under prefix, meaning that if x belongs to P, then every prefix of x also belongs to P. One way to obtain the set P is to construct a test tree T from M_S and then extract partial paths from it. A partial path is a sequence of edges starting from the root of the tree and ending at a terminal or non-terminal node. Since each edge of T is labeled with an input

symbol, a partial path will be a sequence of input symbols. Therefore, P is a set of input sequences. Note that ϵ (the empty input sequence) is a member of every set P. Algorithm 2.2 shows the steps of constructing the test tree.

Algorithm 2.2 [8]: Test tree construction

1. Set the initial state of M_S as the root of the T tree. This is considered as level 1 of the tree.
2. Suppose the tree is built up to level k. Then level $k+1$ is built as follows:
   (a) From left to right, consider each node t of level k.
   (b) If the node t is the same as one of the level j, where $j \leq k$, it is considered as the last node.
   (c) Otherwise, suppose the label of this node is s_i. For each input x, if the machine M_S has a transition from s_i to s_j, an edge with label x and destination s_j is connected to node t.

The W method uses the set P to test each M_I transition and the separating set to check the destination node of that transition. The separating set or in short W is formally defined as follows:

Definition 2.5 [8]: *The separating set of finite machine M_S is a set of W input sequences such that for each separate state s and t in S, there is an input sequence in W, which applied to these two states produces different outputs. In other words, we have:*

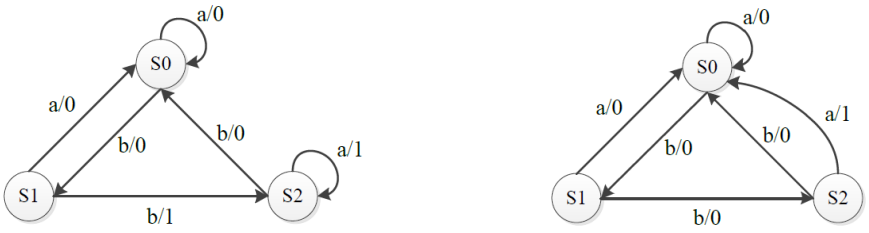
$$\forall\, s,\, t \in S, \exists\, x \in W \rightarrow \lambda\,(s,\, x) \neq \lambda\,(t,\, x)$$

The set W exists for every minimized finite state machine. This set is not unique and the fewer the sequences, the longer their lengths. To obtain the set, it is necessary to iteratively partition the state set of the machine into B_i blocks. Initially, $w = \emptyset$, $B_1 = S$, and $i = 1$. We then iteratively choose two separate states s and t from B_i and obtain a separating sequence x for them. We add the sequence x to W and break the block B_i into smaller blocks based on the output of each of its states according to sequence x. so that all states that produce the same output for x are placed in the same block. We continue this process until all the blocks become singletons, and thus obtain the set W.

For each two separate states, the set W contains a separating sequence that

distinguishes these two states from each other. So, the outputs produced by one state by applying sequences *W* are different from the outputs of another state. Due to this feature, the *W* method uses this whole set instead of the *status* message to check the conformance of the destination state of each transition with what is expected. Since the set *W* can contain several sequences, the machine must go to each destination state several times to apply all these sequences to each destination. For this purpose, *reset* message and the set *P* are used. As a result, the input sequences that the *W* method produces as test sequences are obtained by concatenating each sequence *P* with each sequence *W*. It should be noted that at the beginning of each test sequence it is mandatory to use the *reset* message to go to the initial state. In other words, the set of test sequences produced by the *W* method is equal to $\{reset\}.P.W$. Each output fault is detected by a sequence of *P* and each exit fault is detected by a sequence of *W*. If no error occurs when applying these sequences to M_I and the set of outputs produced is the same as the outputs of M_S, then the implementation is proven to be correct.

To better understand the performance of the *W* method, we will use the following example. Figure 2.2(a) shows a specification finite state machine M_s. In this machine $I = \{a, b\}$ and $O = \{0, 1\}$. To obtain the test sequences, we first calculate the separating set. The sequence *a*, distinguishes the states s_0 and s_2, as well as the states s_1 and s_2. To distinguish s_0 from s_1, *b* must also be included in the separating set. As a result, we have $W = \{a, b\}$.



(a) specification machine (b) implementation machine

Figure 2.2 Finite state machines describing the specification and implementation of a system

To obtain the transition cover set, we have built the test tree using Algorithm 2.2, as seen in Figure 2.3. The partial paths of this tree form the transition cover set and therefore $P = \{\epsilon, a, b, ba, bb, bba, bbb\}$. Now, having the covering and separating sets, we can obtain the test sequences. Table 2.1 shows the test

sequences along with their output in M_S and the transitions tested. In this table, *r* represents *reset*.

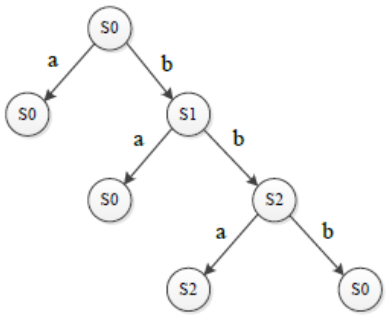


Figure 2.3 A test tree for M_S machine

Figure 2.2(b) shows an incorrect implementation of M_S. By applying the test sequences from Table 2.1 to this implementation machine and comparing its outputs with the expected outputs of each sequence, implementation faults are discovered. The transition fault in this machine, which goes from state s_2 with input *a* to state s_0 by mistake, is detected by the *rbbaa* test sequence and output fault $s_1 \xrightarrow{b/0}$ is detected by the *rbb* test sequence.

P	ϵ		*a*		*b*		*ba*		*bb*		*bba*		*bbb*	
Trans.	$\overset{\epsilon}{s_0 \rightarrow}$		$\overset{a/0}{s_0 \longrightarrow s_0}$		$\overset{b/0}{s_0 \longrightarrow s_1}$		$\overset{a/0}{s_1 \longrightarrow s_0}$		$\overset{b/1}{s_1 \longrightarrow s_2}$		$\overset{a/1}{s_2 \longrightarrow s_2}$		$\overset{b/0}{s_2 \longrightarrow s_2}$	
r.P.W	*ra*	*rb*	*raa*	*rab*	*rba*	*rbb*	*rbaa*	*rbab*	*rbba*	*rbbb*	*rbbaa*	*rbbab*	*rbbba*	*rbbbb*
output	0	0	00	00	00	01	000	000	011	010	0111	0110	0100	0100

Table 2.1 Test sequences for testing specification machine.

2.3 Product line engineering

The way that goods are produced has changed significantly in the course of time. In the past, products were handcrafted for individual customers. Over time, the number of people who could afford to buy various kinds of products increased and the production lines emerged, which enabled mass production. This mass production reduced costs compared to customized production and at the same time reduced the possibilities for diversification. Customers were content with standardized mass products for a while, but all the people did not have same reasons for buying products and therefore, all of them did not want the same type of products. For example, it can be said that some people want an automobile

suitable for urban life and others for living in the countryside. Some customers need a small size automobile and others need a larger family size automobile. Therefore, the industry faced an increased demand for customized products and this was the beginning of mass customization, which means taking into account the needs of customers and preparing the products that they want. In fact, mass customization is the large-scale production of goods tailored to individual customers' needs. From the customer's point of view, mass customization means the ability to have an individualized product. For a company however, mass customization means higher technological investments which leads to higher prices for the individualized products and/or to lower profit margins for the company. Both effects are undesirable, so many companies started to introduce common platforms considering common parts in different products. The use of a common platform for different products reduced the cost of production. In general, a platform is any base of technology on which other technologies or processes are built. The combination of mass customization and a common platform allows one to reuse a common base of technology and, at the same time, to bring out products in close accordance with customers' wishes. The result of this combination is product line engineering [2].

2.3.1 Creating the Platform

In single-system engineering products are regarded as independent, self-contained items this means having distinct projects for developing distinct products. But developing by product line engineering requires the creation of a platform that suits all products. For this purpose, attention is paid first to what is common between the products and then to their differences. In the first step artefacts that can be reused for all products are provided. The products of a product line can be different in the functions they provide, the requirements they fulfill, and even their architecture. These differences should be identified and described during the development process. Creating flexibility in reusable artifacts allows for mass customization.

Different automobiles in an automobile production line can have different windshield washers, so machines should be designed to have a common approach in supporting different engines for these washers. Along with this flexibility comes a series of limitations. For example, in an automobile with the possibility of opening the roof, it is necessary to comply with the limitation that when the

roof is open the rear window washer is disabled [2].

Flexibility is a precondition for mass customization; it also means that we can pre-define what possible realizations shall be developed. In addition, it means that we define exactly the places where the products can differ so that they can have as much in common as possible. For example, in an automobile manufacturing product line, there is a limited number of window washers that must be identified in advance. In the context of software product line this flexibility is called variability. This variability is the basis of mass customization. Further explanations regarding variability are given below.

The variation point is a point in description of the product line which is different for different products. Different initial values to the variation point result in different products. A configuration of the product line defines specific values for the variation points.

2.4 Software product line engineering

Software product line engineering is the development of applications using platforms and mass customization. Development of applications using platforms means planning ahead for reuse, building reusable parts, and reusing components that are built for this purpose. Development of an application for mass customization means applying the concept of managed variability. It means that both the commonalities and the differences of product line applications must be modeled.

Using product line engineering principles for developing software reduces the cost of software development, because platform artifacts can be reused in several different systems. Due to the testing of artifacts in different products, the quality of the software also increases. In addition when changes are made in an artifact (for example, to correct an error), these changes are propagated to all products that use that artifact, and as a result maintenance costs are reduced.

Adapting product line engineering to software development has always encountered obstacles. Certain prerequisites are needed to overcome these obstacles. For a long time, a major obstacle to adapting product line engineering was the lack of needed technology for simple implementation of the rules of product line engineering. One of the most important technologies is object-oriented programming. This technology makes it easy to use concepts such as encapsulation that are necessary to realize managed variability. Another important

achievement is the introduction of component technology, which makes it possible to encapsulate software in parts with low connectivity. Component technology supports the realization of managed variability by limiting the range in which variability is possible. Late binding and dynamic binding methods allow delays in configuration related decisions. Therefore variability can be designed and implemented without worrying about the final shape of various configurations. The use of these techniques facilitates the implementation of platforms and provides a simple way to realize mass customization.

2.4.1 Software product line engineering framework

A software product line engineering framework is a combination of the core concepts of product line engineering, namely the use of platforms and the provision of mass customization. In terms of software, a platform is a collection of reusable artifacts that include all types of software development products, including requirement models, architectural models, software components, and various test plans. The concept of variability of platform must then be introduced. As a result, the artifacts that are different in various products of the software product line are modeled using variability. There are two development processes in software product line engineering:

- Domain engineering: In domain engineering, product line similarities and differences are identified and then a reusable platform is developed.
- Application engineering: Application engineering is a process of software product line engineering in which product line applications are built by reusing domain artifacts and applying product line variability. In other words, this process is responsible for extracting product line applications from the platform obtained in domain engineering.

The advantage of separating these two processes is to separate the two categories of (a) development of a strong platform and (b) building special customized applications in the shortest time.

2.4.2 Variability

The definition and implementation of variability throughout the various phases of

the software product line cycle is supported by the concept of managed variability. The time to decide on variability is called the variability binding time. In common language, the word variability stands for the ability to change or the tendency to change. However, the variability that we are considering is purposeful rather than accidental. Answering three questions helps define product line variability:

1. What changes? Answering this question means identifying parts and features of the environment that change. In fact, this question leads us to the definition of variability as a variable part of the existing world or a variable feature of such a part.
2. Why does the subject of variability change? There are various reasons for changing a part or its feature. including different needs of stakeholders, different local laws, technical reasons, etc. Additionally, if there is a dependency between different parts, then the reason for variation of one part can be the variability of another part.
3. How does the subject of variability change? This question is related to different forms that a variability subject can take. To identify the different forms of a variability subject, a variability object is defined. A variability object is a special instance of a variability subject.

Considering these three questions plays an important role in thinking about variability. Being aware of variability and acting consciously on it is an important prerequisite in modeling variability. In the field of software product line engineering, a variation point is a representation of the variability subject in the product domain, enriched with context information.

2.4.3 Variability models

Variability can be defined as a part of software development artifacts, or as a separate model. The variability model represents the common characteristics and artifacts of a product line and is actually used to manage variability in the product line. Many modeling methods for variability have been proposed over the recent years, and each one uses its own concepts for modeling variability. Among these methods, component-based models such as Koala [13], feature model [14], orthogonal variability model [2], and unified modeling language models [15] like use case model [16] are worth mentioning. In this thesis we use the feature model

to model the variability of the product line. This model is described below.

2.4.3.1 Feature model

The feature model is widely used in software product lines to manage shared and variable features. The products of a software product line are distinguished from each other through their features. In fact, a unique product is defined by features and a product line is defined by a feature model [17]. The feature model was proposed for the first time in 1990 [14]. The following is one definition of feature [14]: "Feature means an outstanding or distinctive aspect, quality or characteristic of a software system or systems, which can be seen by the end user".

Note that the feature is defined as a visible property for the user [14]. However, elsewhere features are considered for each stakeholder including customers, analysts, architects, developers, etc., and therefore a feature can represent any functional or non-functional property at the level of requirement, architecture, component, platform or any other level [18].

If we choose the feature model as the variability model in the software product line, a configuration will be a set of features. Also, a valid configuration is a configuration that satisfies the constraints and limitations of the feature model. Each valid configuration results in the production of one product from the product line. The presence of a feature in a configuration makes the product corresponding to that configuration have the desired feature. In Chapter 4, where our proposed method is presented, configuration means valid configuration.

A feature information model represents all possible products of a software product line as features and relationships between them. A feature model is a set of features that are arranged hierarchically according to the relationships between them. These relationships are divided into two categories:

- Relationships that can exist between the parent feature (or compound) and its child features (or sub-features).
- Cross-tree (or cross-hierarchy) relationships or limitations that usually include a requirement relationship or an exclusion relationship

Feature charts can be defined more precisely [19]. In fact, a feature model consists of a feature graph and additional information for each feature. A feature graph consists of a set of nodes, a set of directed edges, and a set of arcs for the edges. Arcs connect a subset of edges related to a node and classifies the sub-

nodes of a node. Table 2.2 shows the graphical symbols of the feature diagram. The description of these symbols is given below.

Symbol						
Description	Mandatory Feature	Optional Feature	Alternative Feature	Or-Feature	Requires	Excludes

Table 2.2 Graphical symbols of the feature diagram [19].

- Mandatory: A child feature has a mandatory relationship with its parent if the child feature exists in all products in which the parent attribute appears. The features at the root are always mandatory and are present in all products.
- Optional: A child feature has an optional relationship with its parent if the child feature can optionally exist in all products in which the parent attribute appears.
- Alternative: A set of child features have alternative relation with their parent, if only one of them can appear in the product of which the parent feature is a part.
- Or: A set of child features have an or relationship with their parent, whenever one or more of them can appear in the product of which the parent feature is a part.

In addition to relationships between parent and child features, a feature model can also include constraints between features. These restrictions include:

- Requires: If feature A requires feature B, choosing A in a product requires choosing B in that product.
- Excludes: If feature A excludes feature B, both features cannot be part of the same product.

In addition to these limitations, more complex relationships can exist between features in the form of logic formulas [20]. Figure 2.4 shows a simple feature model of a mobile phone.

Based on the feature model in Figure 2.4, all mobile phones must be able to support calls and also include a screen for basic, high-resolution or colorful

information display. Phones may also support GPS and multimedia devices such as cameras or music players or both. Phones that include a camera need a high-resolution screen. GPS and basic display are incompatible features and mutually exclusive.

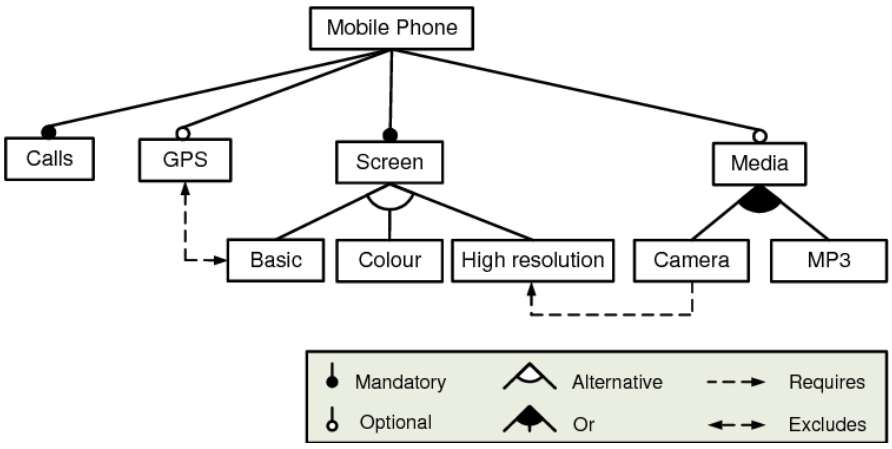


Figure 2.4 A sample feature model of mobile phones [17]

Chapter 3

Previous Work

Two essential parts of this research are behavioral modeling and reactive system product line testing. Therefore it is necessary to review and categorize the previous work that is related to each of these two concepts. Also, the position of the work done in this research, which is given in the next sections of the thesis, should be specified in this context. There is a taxonomy of the work done for model-based test case generation and prioritization [21]. In this section, the necessity of conducting this research and its advantages compared to previous works are also discussed.

3.1 Modeling product line of reactive systems

A design method based on integrated modeling language for the software product line named PLUS (Product Line UML-Based Software Engineering) was developed [15]. This method extends the UML-based modeling used for single-system to be suitable for software product line modeling. PLUS aims to model commonalities and variabilities in a software product line. The PLUS method is an informal method for modeling the entire software product line. Behavioral modeling can be used in this method to model the reactive software product line.

One of the activities performed in the PLUS method is behavior modeling with a finite state machine. Many systems are state-dependent. In such systems, the activities depend not only on the input but also on the previous state of the system. A state transition diagram, state transition table, and state chart can be used to define the finite state machines. The notation of state tables in UML is based on an existing state table notation [22]. In the PLUS method, the statechart is used. The inherited state machine and parameterized state machine can be used to model variability in the software product line.

Inheritance is one of the two main methods for modeling variability with the state machine. A specialized state machine for a product is a child state machine that inherits from the parent state machine which describes the product line

commonalities. This machine inherits all the states, transitions, and activities of the parent machine, and also by adding new states, transitions and activities, it can describe the new product. Another method to model variability with a state machine is to use a parameterized state machine. In this method, there is only one parameterized state machine that includes all states, transitions, and activities related to all features of the product line. For each feature there is a Boolean condition called the feature condition. If the Boolean condition of a feature is true, that feature is selected. If there are several features and only one of them can be present in the product, only one of the conditions can be true per product. Feature conditions are placed on transitions as transition conditions. Therefore in order for a transition to be performed, in addition to the presence of the desired input, the condition of the corresponding feature of that transition must also be true, which means that the corresponding feature is present. For each product, a number of these feature conditions are true [15].

We presented above a software product line modeling methods from the perspective of software engineering. Methods for the formal modeling of the software product line have also been provided, including modeling by transition systems [23, 24, 25, 26], Petri net [27], and process algebrae [28, 29].

A behavioral model called Extended Modal Labeled Transition System (EMLTS) [26] is proposed to model different states of variability that usually comes in the definitions of product families.

An EMLTS describes a family of products by determining the mandatory and optional transitions of each state of a system. A labeled transition system consists of a set of states and a set of actions, which are used as labels for transitions between states. An important point when modeling the behavior of a product with a labeled transition system is that different products perform different actions in the same state. So all the different facilities, which make the products belong to the family, should be included in the definition of the product family. This can be done by defining Modal I/O Automata to model variability [23]. A Modal I/O Automaton with two transition relations determine allowed (may) and required (must) behaviors. Most of the variability features can be modeled by these transition relations in the product line definitions. However, the alternative variability cannot be modeled by this machine and thus the model was extended [3], by the introduction of EMLTS.

An EMLTS introduces a family of labeled transition systems, where any labeled transition system that describes a product can be extracted from that

extended modal labeled transition system.

An *Extended Modal Labeled Transition System* (EMLTS) is a quintuple (*S, Act, s_0, □, ◇),* where *S* is a set of states, *Act* is a set of actions, $s_0 \in S$ is the initial state, $\square \subseteq S \times 2^{Act \times S}$ is the "at least k of n" transition relation, and $\diamond \subseteq S \times 2^{Act \times S}$ is the "at most k of n" transition relation [3] which is shown as:

$$sa1, \ldots, \underset{\rightarrow}{an} \square s1, \ldots, sn \text{ and } sa1, \ldots, \underset{\rightarrow}{an} \diamond s1, \ldots, sn$$

In these two relations, □ mean that in any product of the family should have at least one of the n transitions $sa\underset{\rightarrow}{i}si$, while in ◇ any product of the family should have at most one of the n transitions. It is noteworthy that in each of these two relationships there is a target state for each action. Using these two relations various variability can be modeled. For example, the relation exactly one of the n transitions, which represents alternative variability, is obtained from $\square \cap \diamond$.

A process algebra can also be used to describe the specifications of reactive software product lines [28]. In this paper, PL-CCS is introduced as an extension of CCS [30] to model the interaction of software components used in software product lines. While CCS is suitable for describing the relation of software systems, it does not support the definition of a set of systems (product family).

More specifically, CCS is extended [28] by adding the variants operator $\bigoplus$, which allows modeling the alternative behavior i.e., alternative process. This means that only one of the alternative processes will be existing in the final system. The main advantage of the modeling product line behavior with PL-CCS is that it provides automatic verification by model checking.

The models introduced so far to describe the product family have not considered the importance of features as the unit of differentiation. That is, they can model different behaviors but are not able to associate each product with its behavioral description and characteristics. In addition, they do not use the information available in the variability models such as the copresence of several features in a product or the prohibition of the presence of a feature by another feature. To deal with these challenges, one can extend the transition system with features and introduce featured transition system (FTS) to describe the behavior of the entire product line [25]. In this model, each behavior is explicitly associated with the feature that causes that behavior, and this association takes place at the transition level. For this purpose, each transition is labeled with a feature. In some cases, a feature removes transitions rather than adding them. To show this, it is necessary to define a priority relationship between transitions. A priority

relationship can be defined between two transitions that leave the same origin state and are labeled with different features. In this case, for any product that includes both features, the transition with lower priority will be removed. In general, the priority relationship is a solution for modeling situations in which a feature overrides the behavior of another feature.

We know that every product consists of a set of features. To obtain the behavior of a specific product, it is enough to project the model that specifies the entire product line i.e., the FTS on the features of that product. For this purpose, we remove all transitions labeled with features not available in that product, as well as lower-priority transitions that are overridden by higher-priority transitions. The result of the projection is a normal transition system.

3.2 Software product line testing

Efficient testing strategies are very important for any organization where the share of software development costs is high. This issue is more critical for the organizations that use a software product line. Due to the large variety of products deriving from the product line platform, software product line testing is a complex and costly practice. In the early research on product line not much attention was paid to the issue of product line testing, but over time, the need to investigate this field became evident [5].

High productivity in product line engineering requires an efficient testing method, and the testing methods are being used in single-system engineering are not good and efficient enough for this purpose. In the single-product production, testing accounts for about 50% of the total effort and cost, and this percentage increases in product line engineering. Therefore, it can be said that testing is a bottleneck in the development of product families [6]. The main challenge in software product line testing is the large number of required tests. To fully test the product line it is necessary to test all the products. We know on the other hand that the number of products grows exponentially with the increase in the number of features. Accordingly, the main problem is to reduce the redundant tests and minimize the effort required for testing through the reuse of the test artifacts.

3.2.1 Automated and model-based test generation

The idea of proactive reuse in product family testing was introduced [6], where reusable test cases generated in domain engineering are used to test different

products. The paper presents the ScenTED (Scenario based TEst case Derivation) method, which is a model-based and reuse-based method for test cases derivation. Model-based testing is a method of deriving test cases in single-system engineering and basically consists of two main steps as shown in Figure 3.1. In the first step a test model is created from the requirements, and in the second step test cases are generated using coverage criteria or other test extraction techniques.

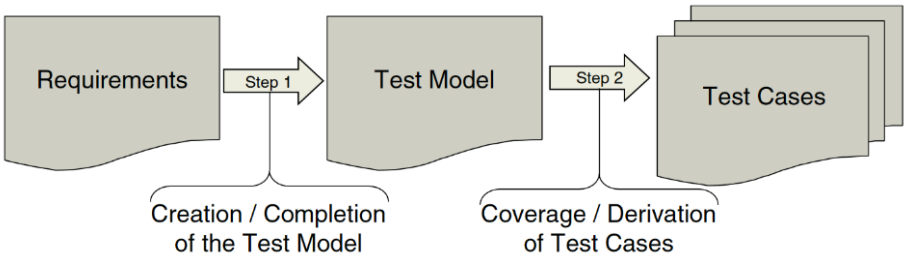


Figure 3.1 Model-based Testing in Single System Development [6]

The model-based testing technique includes several advantages. For example, test cases are generated in an organized and repeatable process with stopping rules. Therefore, model-based testing is a prerequisite for the automated generation of test cases. Another important aspect is that the test engineers will check the correctness of the requirements by creating the testing model. This means that the defects in the requirements, such as ambiguity or lack of completeness, can be discovered during the development of the testing model.

In applying model-based testing to the software product line, in addition to domain engineering, one must also consider application engineering. The test models produced in domain engineering include variability and the generated test cases include variability information. In the application engineering test a new test model is actually not developed. In other words, in application engineering the test model of each product is developed based on the requirements of that product and by reusing the domain test model. In this way, variability is removed from the domain test model and new requirements are added to it. Test cases can then be derived in two steps. First, reusable test cases are selected from domain engineering. Some of them need to be changed to adapt to the product, based on the variability selected in that product. In the second step, based on the new requirements added to the test model, new test cases are generated if necessary.

Figure 3.2 shows the application of model-based testing in a software product line.

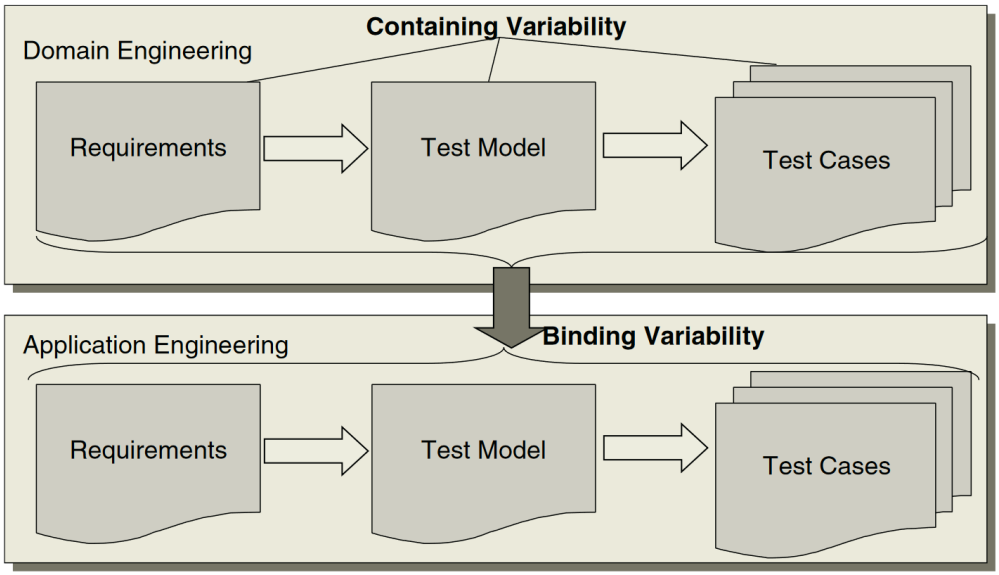


Figure 3.2 Model-based Testing in Software Product Family Engineering [6]

In the ScenTED test method it is assumed that the requirements are specified as use cases. In this method, activity diagrams are used for the test model and sequence diagrams are used for the test case scenarios. Test case scenarios describe the activities of the engineer responsible for the test and the responses of the system [6].

A model-based testing method for a software product line generates test specifications from use cases and the feature model. This test specifications are configured and used to test each product in the product line. Hasan Gomma, the provider of the PLUS method that we mentioned earlier, presents a model-based software product line testing method that reduces the number of test specifications necessary to cover all use cases, features, and all possible combinations in the product line [31]. The CADeT (Customizable Activity Diagrams, Decision Tables and Test Specifications) method presented in this paper uses both test cases and feature models to generate test specifications. Another process is proposed to generate test cases based on application cases and variability model [32].

Test case generation models were also established based on formal specifications that are described by process algebrae [33, 34]. An incremental

method for test generation using Alloy also exists [4]. Alloy [35] is a declarative language based on relational logic to describe specifications. The Alloy analyzer is then used to generate the test incrementally, which means that the analyzer is run more than once and on the partial specifications, which makes the problem easier. Li, et al. [36] utilize the information in execution traces to reduce the number of tests runs for each product in a software product line.

3.2.2 SPL-based testing methods

SPL-based testing methods generate reusable test cases from software product line requirements and use them to test products derived from the product line. Three methods are presented [37] to generate tests from product line requirements in the form of use cases, and then a coverage criterion is presented based on the use case to apply the tests. A subsequent paper [38] deals with the problem of defining and managing the relationship between the feature model and the tests by a decision model. In this paper, the decision model that is used for feature selection during product derivation is also used for selection and customization of test cases.

The issue of how to make test-ready the models generated for a product line in order to create reusable test cases for products derived from that product line was also addressed [39]. In this article, the analysis models and requirements of a product line developed with the PLUS method are prepared for testing. A test-ready model contains enough information to automatically generate test cases using one or more test strategies.

Reusable test cases can be generated during domain engineering [40, 41]. However, these methods are not model based and the test cases are derived from the requirements described in natural language. Hartmann et al. [42] use the activity diagram for the test model, which includes variability but test cases are only generated in application engineering. Therefore the method is model based, but does not consider the reuse of test cases. Finally, it is possible to not use test models and yet have specifications that are structured and include variability [43]. Test cases for each product are created based on these specifications.

3.3 Reactive system model-based testing

Reactive systems are hardware and software systems with non-terminating behaviors that interact with the external environment through visible events. Communication protocols, web servers, processors, and operational systems can be mentioned among these systems. Since the product lines of reactive systems are considered in this research, we briefly mention here some of the work that has been done so far in the field of model-based testing of such systems. Finite state machines and labeled transition systems are both widely used to model reactive systems. The concepts and definitions related to finite state machines and the conformance test of these machines were explained earlier. Labeled transition systems were introduced by Keller [44] and are used to model context-sensitive systems (concurrent and sequential programs) as well as hardware circuits.

Input-output conformance (*IOCO*) test selection method is a formal, black-box, and model-based testing method for testing functional behaviors. The specification describes the input and output interactions of the system with its environment as a labeled transition system.

The *IOCO* test method generates test cases based on the specification to determine whether the implementation under test (i) conforms to its specification (s) or not, or whether the relation $iconfs$ is established or not.

Another method for model-based testing of reactive systems is to use UML 2.0 test profiles. While UML models focus primarily on the definition of system structure and behavior, they provide only limited means for describing test objectives and test procedures. In 2001, a consortium was built by the Object Management Group (*OMG*) in order to develop a UML 2.0 profile for the testing domain. A UML profile provides a generic extension mechanism for building UML models in particular domains. The UML 2.0 Testing Profile (*U2TP*) is such an extension which is developed for the testing domain. It bridges the gap between designers and testers by providing means to use UML for both system modeling and test specification. This allows a reuse of UML design documents for testing and enables test development in an early system development phase. The UML 2.0 Testing Profile provides concepts to develop test specifications and test models for black-box testing. Four concept groups are introduced in the profile, and they cover the following areas: test architecture, test behavior, test data, and time [8].

3.4 Conclusions so far

In this thesis, we focus on modeling the reactive software product line by means of a finite state machine that is, a formal method. As mentioned, modeling with a labeled system is also one of the formal methods for describing the behavior of reactive software. However, due to the fact that in the *ioco* testing method an infinite number of tests are produced using LTS describing the specifications of the system, using labeled transition system is not suitable for our approach. Indeed, FSM testing theory is finite, and we exploit this in the evaluation of our method, where the quantity of test cases is important.

Due to the structural similarity of the finite state machine to the labeled transition system, we start from a previously developed method [25] to apply variability to finite state machines. Therefore in this thesis the entire software product line is modeled by a finite state machine labeled with features.

Chapter 4

Proposed method

4.1 Introduction

A product line allows producing products with common features. These features can be used in the design and implementation of these products. That is, instead of designing and implementing each product separately, we can use the common points of these products, and design and implement the product line. In this thesis we want to take advantage of the similarity of the products while testing the software product line. This way, we provide a method that produces the test cases required for testing all the products of that product line, based on the specification of the entire product line and according to the similarities and change points. In fact, using this method we obtain reusable test cases.

We use a small product line as a running example, designed to provide a better understanding of the method by applying the proposed test method to it. The description of this product line follows. The behavior of the product line is modeled with a finite state machine. However, it is necessary to use an extended type of finite state machine. The description of this extended finite state machine and how to model variability with it is given in Chapter 2.

4.1.1 Running example

In order to better understand the steps of the test method presented in this research, the beverage vending machine product line is introduced as a running example. The feature model of this product line is shown in Figure 4.1. In the specifications of this product line, we assumed that milk is only served with coffee. We apply this condition with the relationship *M requires C.* It should also be noted that any machine that serves tea certainly serves plain tea. Feature *F* is related to serving free soda and it excludes features *T* and *C*. This means that a machine that can serve free drinks only serves soda.

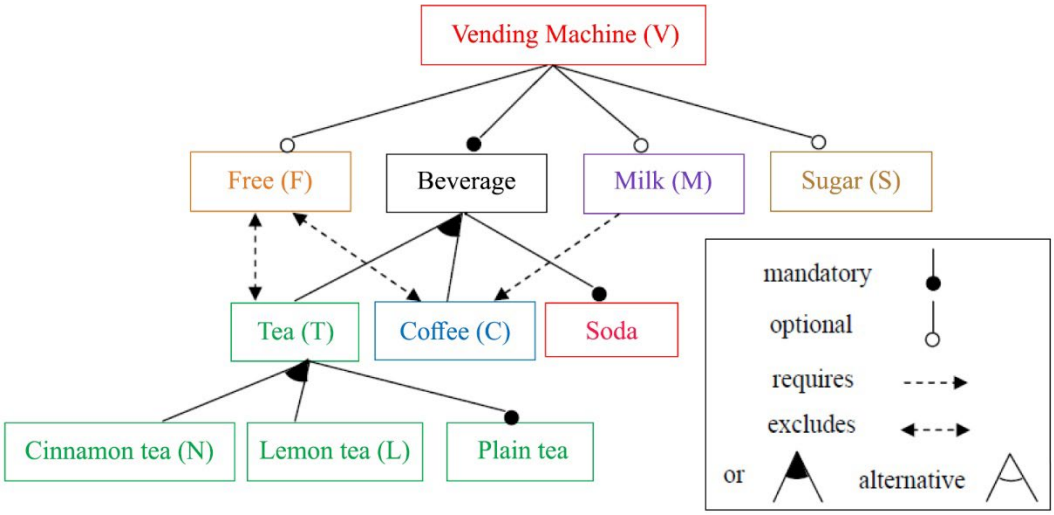


Figure 4.1 Feature model of, the beverage vending machine product line

The initial version of the drink machine delivers a soda upon receiving a coin and then goes back to receiving orders. The modeling of this drinking machine with a finite state machine is shown in Figure 4.2 (a). Among the other features that can be added to this initial version are serving tea and coffee. This machine receives a coin and returns *cack* as an output as confirmation of receiving the coin. From there, it may serve soda, tea, or coffee, depending on the desired configuration feature. To order coffee, another coin must be given to the machine. After requesting coffee, the customer can request milk to be added to the drink. The machine with the possibility of serving coffee with milk and/or sugar is modeled in Figure 4.2 (c).

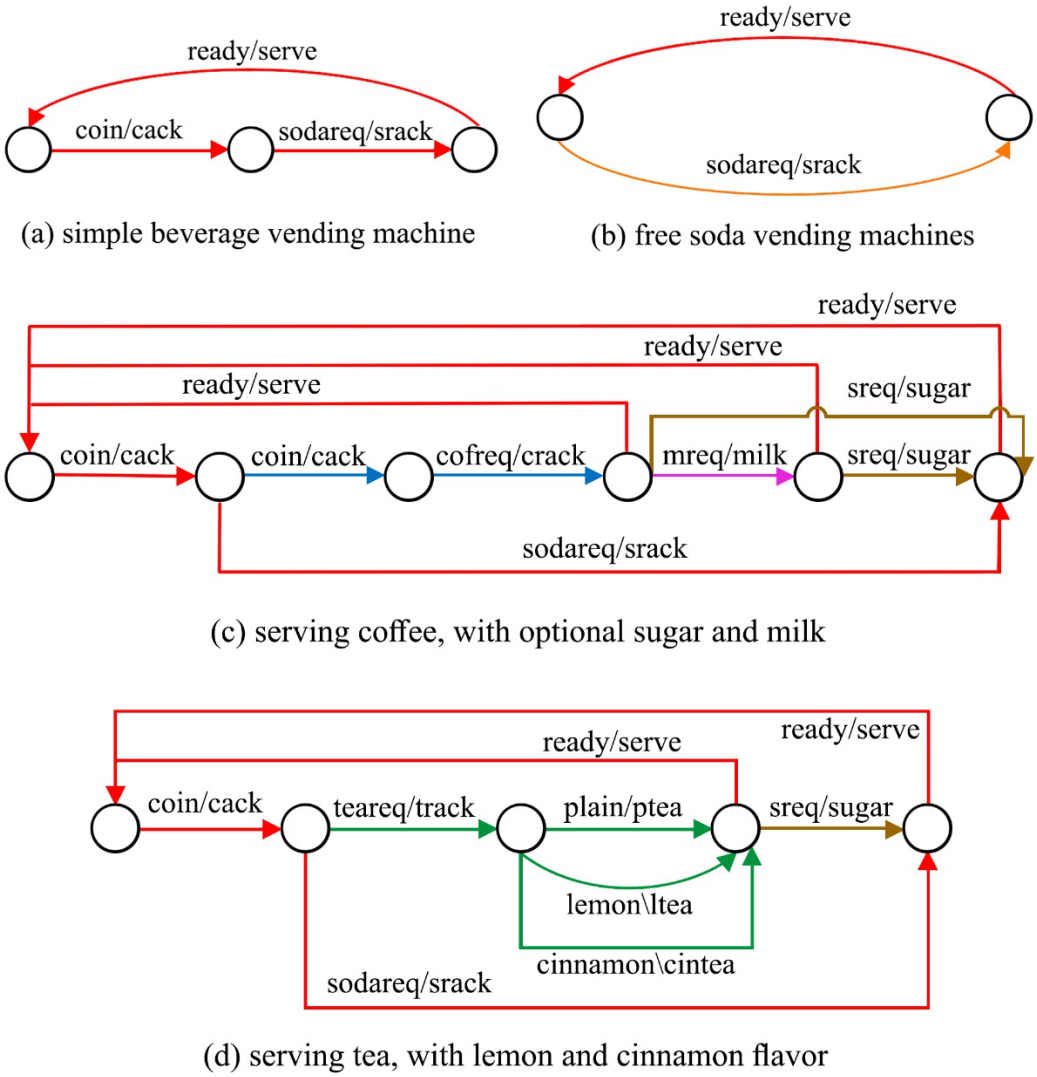


Figure 4.2 Beverage vending machines, modeled with finite state machines

4.2 Product line behavior modeling

In this thesis we intend to produce reusable test cases. That is, from the specifications of the entire product line generate a set of test cases which will be used to test each product from the product line. For this purpose, the behavior of the entire product line must first be described by means of an extended finite state machine. For this, we consider the earlier idea of modelling the product line using a featured transition system [25]. We saw more details about this method in

Chapter 3.

As we know, each function in a product line is related to a feature. In this case, any product that has that feature has the corresponding function. To illustrate this, we label each edge of the machine describing the product line with its corresponding feature. In fact, the machine we designed for modeling in this research is a featured finite state machine (F^2SM). This machine is formally introduced as follows.

Definition 4.1 Featured finite state machine
An F^2SM, is a tuple in the form of $f = (I, O, S, \delta, \lambda, d, \gamma, Prio)$, so that we have:

- $(I, O, S, \delta, \lambda)$ is a finite state machine
- d is a feature model (see Section 2.4.3.1)
- $\gamma : \delta \rightarrow F$ is a function, labeling transitions with features
- $Prio \subseteq \delta \times \delta$ Specifies the priority between some transitions

The finite state machine which is part of the F^2SM models all possible functions of all products in the product line. d is the feature model of the product line from which all valid configurations can be derived. F is the set of all the features and M_F is the set of mandatory features. The set VC (which stands for Valid Configurations) contains all the valid configurations that can be extracted from the feature model. The function γ labels each transition of the finite state machine with its corresponding property. This function is surjective because for every feature in F there is at least one transition labeled with that feature. Also, because more than one transition can be labeled with the same attribute, the function is not injective.

Defining the priority relation between transitions is used to apply conditions or to consider some assumptions about the products of the product line. Priority can be defined between two transitions $tr1$ and $tr2$ that have the same initial state and are labeled with different attributes $f1$ and $f2$, respectively. If it is $(tr1, tr2) \in Prio$, then the priority of transition $tr1$ is higher than $tr2$, and transition $tr2$ would be removed from any product that has both features $f1$ and $f2$. It should be noted that a priority must be defined between two transitions that have the same origin and input but are labeled with different features. In this case, in the products that include both features, one of the two transitions is removed and the deterministic property of the finite state machine of the product is preserved. As we know, in applying the W test method the finite state machine describing the characteristics

must be deterministic and completely specified. In the method presented in this research, which is a modification of the *W* method, it is necessary to establish these conditions. The F^2SM that describes the specifications of the product line is called M_S. In order to label each transition in M_S with its corresponding feature we obtain the set of input symbols of the machine, from the product of the set *I* with the features set *F*. In order for the machine to be completely specified, it is necessary to add a new output *error* to the set of output symbols and an additional state called s_{er} to the set of states. The output of the transitions in which the input is not labeled with its corresponding feature is *error* and goes to s_{er} state.

Figure 4.3 shows the F^2SM describing the beverage vending machine product line. For convenience, transitions with *error* output and into s_{er} are not shown. For example, the output transitions from s_0 with input from the set $I \times F - \{(V, coin), (F, sodareq)\}$ all go to s_{er} and produce the *error* output. As it can be seen, the priority of transition $S0 \xrightarrow{(F,sodareq)} S6$ is higher than $S0 \xrightarrow{(V,coin)} S1$, so in any product where the feature *F* is present the transition $S0 \xrightarrow{(V,coin)} S1$ is removed (note in passing that *V* is present in all products). The definition of this priority makes it possible to serve free soda in products that have the *F* feature. We know that implicitly there is a relation of need between the features of the child and the parent. Therefore, we also added the requirement relations between *N* and *L* features with their parent *T*, to the set of requirement relations.

As mentioned earlier, each configuration is a set of features. To obtain the characterization machine of a configuration, we remove those edges from the F^2SM of the product line which are labeled with features not present in that configuration. We also remove edges with lower priority if necessary. In this case, some states may no longer be available from the initial state, and the machine may not be minimal any longer. Therefore, by removing the unreachable states and then obtaining the equivalent minimized machine, we reach the finite state machine for the desired configuration. The finite state machine for configuration *C* is called $M_S[C]$.

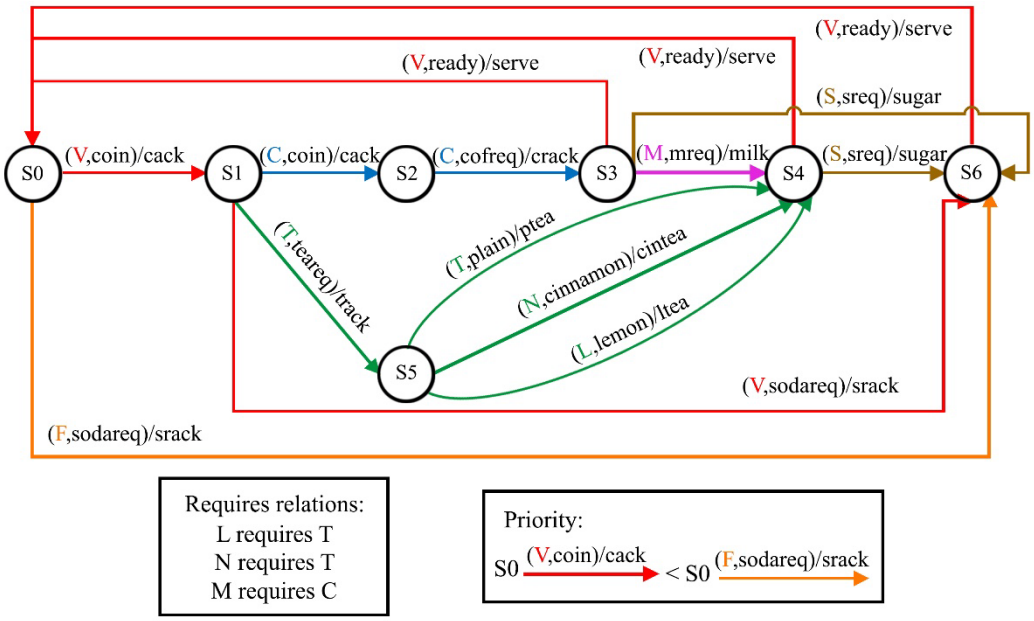


Figure 4.3 Finite state machine describing the beverage vending machine product line

4.3 Modified test method

The transition cover set and separating set are two basic sets used in the *W* test method. So, in order to change thi–s method for use in testing the software product line, we must change the algorithms for constructing these two sets. We will address this issue in the next two sections. In the next section, we first explain the algorithm for obtaining the separator set to be used in the test method. Then we prove the correctness of the proposed algorithm and finally apply it on the example mentioned in the previous section.

4.3.1 Set of separating sequence

In the process of obtaining the finite state machine of a product, some transitions are removed from the original machine. Sequences from the separating set *W* that contain at least one of these transitions are also removed from the set. The remaining sequences must form a separating set for the finite state machine of the product. For this purpose, the main separating set, which is obtained from the characterization machine of the entire product line, should be such that it has a

separating sequence for each of the two separate states in every possible configuration. The set W is formally defined below. Before that, two other concepts that are used in this chapter are also defined.

Definition 4.2 The set F for a sequence x contains the features that label exactly all the transitions of the sequence x.

Definition 4.3 If A is a set of sequences and C is a configuration, then the set $A \backslash C$ is obtained by removing the sequences of the set A that contain transitions labeled with features not present in C. In other words,

$$A \setminus C = \{x \in A \mid F_x \subseteq C\}$$

Definition 4.4 The set W is a separating set for Ms if, for every valid configuration C, $W \setminus C$ is a separating set for M_S [C].

For two distinct states s and t in S all the separating sequences with possible different feature sets must be included in W. Consequently, for every valid configuration C, and for any two distinct states s and t in $S_{MS[C]}$ (the set of machine states M_S [C]), there will be at least one separating sequence in $W \setminus C$. If in the process of constructing $W \setminus C$ all the separating sequences for s and t are removed and there is no separating sequence for these two states in $W \setminus C$, then these states are equivalent, and in the minimization stage of the machine constructing process M_S [C] must be merged into one state.

As we know mandatory features are part of any valid configuration. Therefore, a separating sequence of two states s and t, labeled only with mandatory features is sufficient to distinguish between these two states in any configuration. To build the set W for M_S we need to obtain the separating sequences for all pairs of distinct states in M_S. Algorithm 4.1 obtains the set of the shortest separating sequences for two states s and t and returns it in the *Sep* set. This algorithm is an extension of the algorithm for obtaining the shortest separating sequence for two states presented and proved elsewhere [8].

Algorithm 4.1 Function of constructing all separating sequences of two distinct states

SS_M (s, t: *state*):

1. Partition the set of states i.e., S_{MS}, into equivalence classes ρ
2. Let *Sep* set equal to the empty set
3. Find the smallest index i such that s and t belong to different blocks of ρi
4. If i is equal to 1, add each $a \in I$ for which it is λ (s, a) $\neq \lambda$ (t, a) to *Sep*
5. Otherwise, for each $a \in I$ where δ (s, a) and δ (t, a) belongs to different blocks of $\rho i-1$, add $a.SS_M$ (δ (t, a), δ (s, a)) to *Sep*
6. Return *Sep* as the result

Example 4.1 We partition the set of states of the *VM* machine in Figure 4.3 in the manner described in Section 2. Each state in $\rho 1$ is placed in a separate class. So, for distinct states, the smallest i index is equal to 1, and as a result, the shortest separating sequences are 1 in length. For s_0 and s_2 all inputs except (*V, coin*), (*F, sodareq*) and (*C, cofreq*) lead to the same *error* output. So, we have:

$$SS_{VM}(s_0, s_2) = \{(V, coin), (F, sodareq), (C, cofreq)\}$$

Similarly we have

$$SS_{VM}(s_2, s_5) = \{(C, cofreq), (T, plain), (N, cinnamon), (L, lemon)\}$$

By obtaining SS_M (s, t) we can find the separating sequences of these two states that should be included in the W set. We call the set of these separating sequences, which must be in W, $Q_{s,t}$. The process of constructing $Q_{s,t}$ is that we first set it equal to *SSM* (s, t) and then remove the unnecessary and redundant sequences from it. The formal definition of the set $Q_{s,t}$ and the method of obtaining it are given below.

Definition 4.5 The set $Q_{s,t}$ for two states $s, t \in S$ is a set of input sequences such that for every valid configuration C such that $s, t \in S$ (M_S [C]), there exists an input sequence $x \in Q_{s,t}$ such that λ (s, x) $\neq \lambda$ (t, x) and $Fx \subseteq C$.

As mentioned earlier, if the two states s and t have a separating sequence with all transitions labeled with mandatory features, the presence of this separating sequence in W is sufficient to distinguish these two states in any configuration.

This case is handled in the first step of the $Q_{s,t}$ construction algorithm below. If there are multiple separating sequences labeled with the same feature set in *SSM* (*s, t*), only one of them is enough to be included in $Q_{s,t}$. Which of these sequences is better to include depends on the conditions explained below. The cost of the test depends on the number of test cases and their length. Therefore, in choosing separating sequences we should keep in mind the number and length of separating sequences in the *W* set should be minimized.

The set of separating sequences for pairs of states can have commonality. So in the construction of $Q_{s,t}$, when choosing between separating sequences *x* and *y* with the same features we choose the sequence that appears in the largest number of sets *Q* that have been computed up to that point. We thus try to minimize the number of members of the *W* set, which also affects the number of test cases. If the number of sets *Q* containing *x* and *y* are the same, the selection is made based on the length of the sequences. This means that the sequence with a shorter length remains in *Q* and causes the longer sequence to be removed. This way we try to keep the sum of the length of separating sequences in *W* to a minimum. In fact, the process of using the commonality of *Q* sets to select the desired separating sequence uses product commonality and similarities, which makes the test items reusable. Additionally the relation *requires* also needs to be considered. If the relation *f*1 *requires f*2 holds in a product line, then there is no valid configuration of this product line that contains *f*1 but does not contain *f*2. In other words, every valid configuration that contains *f*1 also contains *f*2. The *requires* relation can be used to reduce the number of separator sequences in the *Q* sets. Suppose that *x* and *y* are two separating sequences for *s* and *t*. We know that F_x and F_y are respectively the sets of features that the transitions of sequences *x* and *y* are labeled with. If any feature is present in F_y, or in F_x, or there is a feature in F_x that requires it, *x* can be removed from $Q_{s,t}$. This can be expressed formally as follows.

$$\forall f1 \in F_y - F_x, \exists f2 \in F_x \quad \textit{such that} \quad f2 \textit{ requires } f1 \rightarrow \textit{discard } x \textit{ from } Q_{s,t}$$

By subtracting F_x from F_y, we discard features of *y* that are also in *x* and examine the remaining features. Establishing the above condition leads to the fact that in any configuration where the sequence *x* is present, sequence *y* is also present. So the presence of *x* in the set of separating sequences creates redundancy. It should be mentioned that, if *x* includes other features in addition to

the same features as y, and these features are in the *requires* relation with the features of y, then there is no problem in removing it. Nevertheless, the existence of x implies the existence of y. However, the opposite is not true. This means that if y has more features in addition to the same features as x or that are in the *requires* relation, then we cannot remove the sequence of x from $Q_{s,t}$. Because sequence y does not exist in the configuration that does not contain these additional features, it should not cause the removal of x. All these points are taken into account in Algorithm 4.2.

Consider two transitions $tr1$ and $tr2$ with the same origin state, labeled with features f and g, respectively. Suppose $(tr1, tr2) \in Prio$. So in any configuration C such that $f, g \in C$, the transition $tr2$ is removed from the configuration due to its lower priority. Therefore, in Algorithm 4.2, if we pass through the transition $tr2$ by applying the sequence x to s or t, then x cannot cause the removal of another sequence, because it is not present in some configurations which also include F_x. Therefore, before deleting a sequence, we must always check if the sequence that caused it to be deleted has a lower priority (than another) If this is the case then we will prevent the removal of the sequence. There is one exception namely, when the sequence to be deleted also includes the same transition with low priority (i.e., $tr2$). In this case we can perform the removal. This condition must be also observed in Algorithm 4.3 which is used for the construction of the transition cover set.

The *HasLowerPriorityTransition* function, which is used in Algorithms 4.2 and 4.3 receives a state and a sequence and determines whether by applying that sequence to the given state the machine passes through a low priority transition.

Algorithm 4.2 Constructing the set of required separating sequences for s and t

Function GenerateQ (s, t: *state*) {
  Let $Qs,t = SS_M(s, t)$
  For every separating sequence $x \in Q_{s,t}$ {
    Let xLowPrio = HasLowerPriorityTransition $(s, x) \vee$
HasLowerPriorityTransition (t, x)
    If $F_x \subseteq M_F \wedge \neg$ *xLowPrio, then let* $Q_{s,t} = \{x\}$ *and return* $Q_{s,t}$
    If $\exists y \in Q_{s,t}$ *such that* $F_x = F_y$, *then* {
      Let yLowPrio = HasLowerPriorityTransition $(s, y) \vee$
HasLowerPriorityTransition (t, y)
      If $|\{Q_{s',t'} \mid s', t' \in S \wedge y \in Q_{s',t'}\}| > |\{Q_{s',t'} \mid s', t' \in S \wedge x \in Qs',t'\}|$ *then*

If ¬ *yLowPrio then Let* $Q_{s,t} = Q_{s,t} - \{x\}$
Else If ¬ *xLowPrio then Let* $Q_{s,t} = Q_{s,t} - \{y\}$
If $|\{Q_{s',t'} \mid s', t' \in S \wedge x \in Q_{s',t'}\}| > |\{Q_{s',t'} \mid s', t' \in S \wedge y \in Q_{s',t'}\}|$ *then*
If ¬ *xLowPrio then Let* $Q_{s,t} = Q_{s,t} - \{y\}$
Else If ¬ *yLowPrio then Let* $Q_{s,t} = Q_{s,t} - \{x\}$
} (*end of If*)
If $\exists y \in Q_{s,t}$ *such that* $\forall f \in F_y - F_x$, $\exists f' \in F_x$ *such that f' requires f then* {
Let yLowPrio = HasLowerPriorityTransition (*s, y*) ∨ *HasLowerPriorityTransition* (*t, y*)
If ¬ *yLowPrio then Let* $Q_{s,t} = Q_{s,t} - \{x\}$
} (*end of If*)
} (*end of For*)
return $Q_{s,t}$
}

For a better understanding of the above algorithm we use it on the beverage vending machine product line.

Example 4.2 Consider the two states s_2 and s_5 of the machine from Figure 4.3. In Example 4.1 we obtained the set of all the shortest separating sequences of these two states.

$$SSVM\ (s_2, s_5) = \{(C, cofreq), (T, plain), (N, cinnamon), (L, lemon)\}$$

Since *N requires T* and *L requires T*, according to Algorithm 4.2 (*N, cinnamon*) and (*L, lemon*) are removed from the set of necessary separating sequences, and as a result, we have:

$$Q_{s2,s5} = \{(C, cofreq), (T, plain)\}$$

For s_0 and s_2 on the other hand, we had $SSM\ (s_0, s_2) = \{(V, coin), (F, sodareq), (C, cofreq)\}$. In order to obtain $Q_{s0,s2}$ we note that the output transition from $s0$ with input (*V, coin*) has a lower priority than the transition with input (*F, sodareq*), and therefore, the separating sequence (*V, coin*), although labeled with the mandatory feature, cannot cause the removal of another sequence, so we have:

$$Q_{s0,s2} = \{(V, coin), (F, sodareq), (C, cofreq)\}$$

Input (*V, sodareq*) has an output other than *error* only when exiting state s_1, s_0 (*V, sodareq*) is a separating sequence for s_1 from any other state. Furthermore, since *F* (*V, sodareq*) = {*V*}, we have:

$$\forall\, t \in S_{VM}\, (t \neq s_1) \cdot Q_{s1,t} = \{(V,\ sodareq)\}$$

Now we have to check whether $Q_{s,t}$ constructed by Algorithm 4.2 conforms to the Definition 4.5. This is accomplished by the following lemma.

Lemma 4.1 For each valid configuration in the software product line, and for any distinct states *s* and *t* in the finite state machine describing that configuration, there is an input sequence in $Q_{s,t}$ as computed by Algorithm 4.2 that is a separating sequence for those two states, and all its features exist in the configuration. We formally have:

$$\forall C \in V\ C,\ \forall s,t \in Min\ (M_s[C]) \Rightarrow \exists x \in Q_{s,t} \cdot x \in SS_M\ (s,\ t) \wedge F_x \subseteq C$$

Proof. The proof is by contradiction. Suppose there is a configuration for which there is no sequences in $Q_{s,t}$ that has the two condition $x \in SS_M(s,\ t)$ and $F_x \subseteq C$. That is,

$$\exists C \in V\ C,\ \exists s,\ t \in Min\ (M_s[C])\ \nexists\ x \in Q_{s,t} \cdot x \in SS_M(s,\ t) \wedge F_x \subseteq C$$

We know that both *s* and *t* in the finite state machine of any configuration *C* have a separating sequence with its features available in that configuration (otherwise the two states would be equivalent). This means that

$$\forall C \in V\ C,\ \forall s,\ t \in Min\ (M_s[C]) \Rightarrow \exists x \in SS_M\ (s,\ t) \cdot F_x \subseteq C$$

From here, several situations arise:

- If the features that the transitions of sequence *x* are labeled with are all mandatory features, and the sequence *x* starting from *s* and *t,* does not include transitions with low priority, then in Algorithm 4.2 we will have $Q_{s,t} = \{x\}$. On the other hand, for every valid configuration *C* we have $M_F \subseteq C$. So $F_x \subseteq C$, and as a result, a contradiction.
- If there is a sequence $y \in Q_{s,t}$ with $F_x = F_y$, the sequence *x* in Algorithm 4.2 can be removed from $Q_{s,t}$. We know that $F_x \subseteq C$ and $F_x = F_y$, therefore we have $F_y \subseteq C$. So there is a sequence *y* in $Q_{s,t}$ whose feature set is a

subset of C. Also, since $Q_{s,t} = SS_M(s, t)$ is placed first, then all the existing sequences in $Q_{s,t}$ are separating sequences of s and t. As a result, the assumption is once again violated.

- Sequence x may be omitted in Algorithm 4.2, if there is sequence y in $Q_{s,t}$ such that for every f in $F_y - F_x$, there is an f' in F_x that the relation f' *requires* f is established. According to the definition of this relation, if f' is in C, f will also be a member of C. Therefore the features of y that are not in F_x are also members of C. So, all the features of sequence y are in configuration C and as a result, again a contradiction. ■

In the last two cases in the proof we determined that even if x is removed from $Q_{s,t}$, there is still a sequence in it that matches the intended conditions. By obtaining the set Q for all the distinct states in the F^2SM describing the software product line, we can construct the set of separating sequences W for each configuration. The algorithm to obtain this set is given in Section 4.3.3 (Algorithm 4.6). In the next section we will discuss how to construct the transition cover set.

4.3.2 Transition cover set

In this section we obtain the transition cover set (abbreviated P) for the model describing the entire product line, MS. Having P, one can obtain the transition cover set for each product. As mentioned before, for each $s \in S$ and each $a \in I$ in M_S there is a transition from s with input symbol a. Therefore, Definition 2.4 of Chapter 2 can be rewritten as follows: The set of input sequences P is a transition cover set for the finite state machine M, if for each transition $s \xrightarrow{a}$ there exist sequences $x, y \in P$ such that $x = y.a$ and $\delta(s_0, y) = s$.

We know that the transition cover set is closed under prefix. In the following we can see that the proposed algorithm for constructing the set P preserves this property. The formal definition of the transition cover set for the product line is given below.

Definition 4.6 The set of input sequences P is a transition cover set for MS if, for every valid configuration C, $P \setminus C$ is a transition cover set for the finite state machine describing C, i.e., $M_S[C]$.

To obtain the set P we need to find the necessary sequences to cover each

transition. For each configuration and for each transition in which that configuration appears we want to have at least one sequence in *P* that covers that transition (that is, starts from the initial state and ends with that transition). In order to build these sequences we first obtain sequences with different sets of features that end at the origin of each transition. Algorithm 4.3 describes the method of obtaining these sequences for each state. After executing this algorithm, for each state *S* we have a set *needPath(S)* of sequences with different features, each of which constituting a path from the initial state to state *S*.

In order for the prefix property to be established in the set *P*, it is necessary that the union of these sets on all states of the machine have the prefix property. That is, if the sequence *x* is in *needPath(s)*, then every prefix of *x* must be in $\cup_{t \in s}\, needPath(t)$. In order to establish this property we traverse the finite state machine in a breadth-first order.

Algorithm 4.3 Building *needPath* for all finite state machine states

```
Function GenerateneedPaths(){
let L = 0 ∧ ReachedStates = ∅ ∧ needPath(s0) = {ϵ}
do the steps below, until needPath(s) does not change for any s:
   if L equals 0, then for each outgoing transition from s0: tr do
      if the output symbol of tr not equals error then {
         let ns = destination state of tr
         add transition symbol of tr to needPath(ns)
         add ns to ReachedStates
      }
   if L is greater than 0, then
      for each state st, in ReachedStates and for each outgoing transition from st: tr
         let tsym = transition symbol of tr
         let ns = destination state of tr
         if the output symbol of tr not equals error then
            for each path p of length L in needPath(st), do
               if p has not passed the state ns, then add p.tsym to needPath(ns)
   }
   for each state s in ReachedStates
      let selectedPath(s) = ApplyConstraints(needPath(s))
      for each path p in needPath(s) − selectedPath(s) do {
   remove p from needPath(s)
```

```
    for each path p' in needPath(t) · (t ≠ s) do
        if p is a prefix of p' then remove p' from needPath(t)
    }
}
Function ApplyConstraints(needPath(s)) {
    Let selectedPath(st) = ∅
    Foreach path p in needPath(st) do
        If Fp ⊆ MF ∧ ¬ HasLowerPriorityTransition(s0, p) then
            Let selectedPath(st) = {p} and break the loop
        If ∃p' ∈ needPath(st) such that Fp = Fp' then {
            If |p| ≤ |p'| then
                If ¬ HasLowerPriorityTransition(s0, p) then add p to selectedPath(st)
                Else If ¬ HasLowerPriorityTransition(s0, p') then add p' to selectedPath(s
                    Else Let selectedPath(st) = selectedPath(st) ∪ {p, p'}
            Else
                If ¬ HasLowerPriorityTransition(s0, p') then add p' to selectedPath(st)
                Else If ¬ HasLowerPriorityTransition(s0, p) then add p to selectedPath(st
                    Else Let selectedPath(st) = selectedPath(st) ∪ {p, p'}
        }
        If ∃p' ∈ needPath(st) such that ∀f1 ∈ Fp' − Fp, ∃f2 ∈ Fp such that f2 requires f
        ∧ ¬ HasLowerPriorityTransition(s0, p') then add p' to selectedPath(st)
    return selectedPath(st)
}
```

Variable L indicates the level to which the machine has been traversed. To build the *needPath* sets, we start from level zero that is, state s_0, and for each output edge from this state $s_0 \xrightarrow{(f,a)} t$ that does not have an output *error* $\xrightarrow{(f,a)}$ we add (*f, a*) to *needPath*(*t*). We also add state *t* to the set *ReachedStates*. In fact, at each stage, we add new seen states to this set. Then we increment *L* and go to the next level. Now let's assume that we have proceed to level *k* that is, $L = k$. For every state $s \in$ *ReachedStates* and its every output transition $s \xrightarrow{(f,a)} t$ that has no *error* output $\xrightarrow{(f,a)}$ we consider every path *p* in *needPath*(s). If the destination state of this route is at level *k* of the machine (or in other words, *p* has passed *k* transition) and also *P* has not passed state *t*, we add the path *p.(f, a)* to the paths necessary to reach *t*, i.e. *needPath(t)*. In fact, each time we add the paths of length *k + 1* that do

not have loops and are the continuation of the paths we found on the previous level, corresponding to their destination. This way we preserve the prefix property in the union of the *needPath* sets. Additionally, if *ReachedStates* does not contain state *t*, we include this state in that set as well. At the end of the loop, after the *needPath* sets have been updated, they are also checked by the *ApplyConstraints* function in a way similar to the three conditions mentioned in Algorithm 4.2, and unnecessary paths are removed. It should be noted that by deleting a path, all the paths in each *needPath* that are the suffix of this path should also be deleted. The algorithm terminates as soon as no *needPath* set is updated. In other words, the execution of the loop continues as long as *needPath(s)* changes for at least one $s \in S$.

Recall that the *ApplyConstraints* function removes unnecessary paths and returns the rest in the *selectedPath* set. When choosing between two paths p and p' with the same set of attributes in *needPath(s)*, we use the length criterion. That is, if the path p is shorter and does not include any transition with low priority, it can cause the deletion of p' and so only p is added to the set of *selectedPath(s)*.

Lemma 4.2 For each valid configuration in the software product line, and for each state s in the finite state machine describing that configuration, there is a path in *needPath*(s) that contains all the feature of the configuration. Formally,

$$\forall C \in VC, \forall s \in Min\,(M_s[C]) \Rightarrow \exists p \in needPath(s) \cdot F_p \subseteq C$$

Proof. The proof is once again by contradiction: Suppose there is a configuration for which there is no path in *needPath*(s) that satisfies condition $Fp \subseteq C$:

$$\exists C \in VC, \exists s \in Min\,(M_s[C]) \nexists p \in needPath(s) \cdot F_p \subseteq C$$

We then are in one of the following situations:

- If all the transitions of the path p are labeled with mandatory features and also the path p starting from the initial state does not include transitions with low priority, then in Algorithm 4.3 we will have *needPath*(s) = {p}. On the other hand, for every valid configuration C we have $MF \subseteq C$ *and so* $F_p \subseteq C$. The assumption is thus violated.
- If a path p' of length shorter than p and $F_p = F_{p'}$ in *needPath(s)* exists and p' does not include a low-priority transition, then p is deleted based on the

existence of p'. But the presence of p' with $F_{p'} \subseteq C$ in *needPath(s)* violates the assumption.

- Path p may be omitted in Algorithm 4.3, if there exists a path p' in *needPath(s)* such that for every f' in $F_{p'} - F_p$, there is an f in F_p such that the relation *f requires f'* holds. According to this relation, if f is in C, f' will also be a member of C. Therefore the features of p' that are not in F_p are also members of C. That is, all features of path p' are in configuration C and as a result the assumption is violated. ■

Example 4.3 We want to construct *needPath(s)* for each state $s \in S_{VM}$. At the beginning of Algorithm 4.3, the value of variable L is zero, and with one execution of the loop, we have:

$$needPath(s_1) = \{(V, coin)\},\ needPath(s_6) = \{(F, sodareq)\}$$

In the next execution of the loop, (*V, coin*).(*V, sodareq*) is added to *needPath*(*s*6), and although both transitions of this sequence are labeled with the mandatory feature *V*, it cannot eliminate the sequence *(F, sodareq)*, because it includes the transition with low priority $s_0 \xrightarrow{(V,coin)} s_1$. We also have:

$$needPath(s_2) = \{(V, coin).(C, coin)\}\ ,\ needPath(s_5) = \{(V, coin).(T, teareq)\}$$

At $L = 2$ in the execution of the function *ApplyConstraints* the sequences (*V, coin*).(*T, teareq*).(*N, cinnamon*) and (*V, coin*).(*T, teareq*).(*L, lemon*), which were added to *needPath*(*s*4) at the beginning of the loop are removed due to the existence of the sequence (*V, coin*).(*T, teareq*).(*T, plain*) and the relations *N requires T* and *L requires T*. It is worth noting that because all three sequences contain the low-priority transition $s_0 \xrightarrow{(V,coin)} s_1$, deletion is possible.

Finally, at *L=5*, the *needPath* sets are completed and fixed for all states and the algorithm terminates. The sets *needPath* for states s_1, s_2 and s_5 remain unchanged from earlier. For the other states we have:

$$needPath(s_3) = \{(V, coin).(C, coin).(C, cofreq)\}$$
$$needPath(s_4) = \{(V, coin).(C, coin).(C, cofreq).(M, mreq),\ (V, coin).(T, teareq).(T, plain)\}$$
$$needPath(s_6) = \{(V, coin).(V, sodareq),\ (F, sodareq)\}$$

After executing Algorithm 4.3 and constructing the sets *needPath*(*s*) for each $s \in S$ we can obtain the transition cover set for the finite state machine M_S that describes the specifications of the product line. Algorithm 4.4 shows how to build this set.

Algorithm 4.4 Building the transition cover set for the product line specification machine

Function GeneratePSet() {
  Let PSet = ∅
  For each state $s \in S$ *do*
    For each path $p \in$ *needPath(s) do*
      If $\exists p' \in$ *needPath(s) such that* $F_{p'} \subseteq F_p$ *then remove p from needPath(s)*
  For each transition tr : $s \xrightarrow{(f,a)} t$ *in* M_S *do*
    If the output symbol of tr not equals error then
      Add needPath(s).(f, a) to PSet
Add the empty string ϵ*, to PSet*
}

At the beginning of the algorithm for each state *s* the paths inside *needPath(s)* are checked and if possible a number of unnecessary paths are removed. Suppose the paths *p* and *p′* are two paths from *s0* to *s*, which were placed in *needPath(s)* by Algorithm 4.3. Path *p* is considered unnecessary and can be omitted if $F_{p'} \subseteq F_p$. Indeed, in this case for every configuration *C* where the path *p* is in M_S [*C*], *p′* will also exist in that machine. We know that $F_{p'} \subseteq F_p$, so whenever $F_p \subseteq C$ then it will also be the case that $F_{p'} \subseteq C$. Overall, the membership of *p* in *needPath(s)* is not necessary. We must note that the reverse is not true. That is, there can be a configuration whose machine contains *p′* but does not contain *p*. So, the presence of *p′* in *needPath(s)* is required.

After making the necessary changes in the *needPath* sets the transition cover set can be obtained. For each transition *tr*: $s \xrightarrow{(f,a)} t$ we add the necessary paths to reach *s* with the input symbol of the transition and add it to the transition cover set. In fact, *needPath*(*s*).(*f, a*) is the set of necessary paths, to cover the transition *tr* in the entire product line. Since for every valid configuration *C* of the product line there is at least one path to *s* in *needPath*(*s*) that appears in the specification machine of this configuration, then we can be sure that in the set *P* for every

configuration and for each transition there is at least one sequence that represents the path that covers that transition.

Example 4.4 For the transition $s_6 \xrightarrow{(V,ready)} s_0$ and given *needPath*(s_6), the sequences (*F, sodareq*).(*V, ready*) and (*V, coin*).(*V, sodareq*).(*V, ready*) are added to the set *P* and cover this transition. Similarly, the necessary sequences to cover $s_4 \xrightarrow{(s,sreq)} s_6$ transition are:

(*V, coin*).(*C, coin*).(*C, cofreq*).(*M, mreq*).(*S, sreq*) and (*V, coin*).(*T, teareq*).(*T, plain*).(*S, sreq*)

Having the *needPath* sets from Example 4.3, we can also obtain the sequences needed to cover other transitions from the machine of Figure 4.3 and form the transition set for the beverage vending machine product line model.

4.3.3 Testing the software product line

Having a set of separating sequences and a set of transitions, it is now possible to test the software product line. We will reuse the obtained sets in the test of all products. We know that in order to create test sequences for each product we must have the sets of separating and transition cover sequences. We want to extract these sets from the sets of separating and transition cover sequences that we built for the entire product line. We denote the sets of separating and transition cover sequences for configuration *C* by *W* [*C*] and *P* [*C*], respectively.

4.3.3.1 Extracting the transition cover set for a product

Algorithm 4.5 extracts the transition cover set for each valid configuration of the product line from the set *P* that we constructed in Section 4.3.2. We defined a set named *ConfigP* and set it equal to *P* at the beginning. To obtain the transition cover set for configuration *C* we need to have the transitions in the configuration. We define a set called *Transitions*, which is initially equal to the set of all transitions in the machine describing the entire product line i.e., *Ms*, and then we remove transitions from it during several steps. As we saw earlier, a priority may be defined between transitions that exit the same state but are labeled with different attributes. Depending on the configuration features, transitions with lower priority may be omitted. If the features tagged with two lower and higher

priority transitions are both present in *C*, the lower priority transition will not appear in the configuration, so, we remove these transitions from the *Transitions* set. Also, sequences that contain these transitions should not appear in the transition cover set, so we remove them from *ConfigP*.

The next step in arriving at the transition cover set for configuration *C* is to discard sequences from the set *P* that contain at least one transition labeled with a feature other than the features of *C*; these sequences should be removed from *ConfigP*. We then remove the transitions in *Transitions* which are not covered by any sequence in *ConfigP*. The remaining transitions in *Transitions* form the set of configuration transitions *C*.

We constructed the set *P* for the entire product line in such a way that for each transition it contains sequences that cover that transition in different configurations (which contain that transition). So for each transition there may be more than one sequence in *P*. Also, due to the existence of commonality between configurations, in extracting the transition cover set for a configuration, there may also be transitions that are covered by more than one sequence in *ConfigP*. We are ready to remove these extra sequences. As mentioned earlier, the shorter the length of the test sequences, the lower the cost of the test. So we use the length criterion to remove the extra sequences. Algorithm 2.2 in Chapter 2, which builds the transition set in the *W* test method, also uses the idea of the shortest length. This algorithm constructs the test tree by traversing breadth-first the finite state machine. Then it obtains the machine transition set by extracting partial paths from this tree. Due to the breadth-first traversal, the sequence chosen to cover each transition is the shortest possible sequence. Here, we also choose the shortest sequence among the sequences covering a transition. We also choose the sequence with shorter length among the sequences covering a transition. The loop at the end of Algorithm 4.5 is responsible for this task. After executing this algorithm, *P*[*C*] will contain the necessary sequences to cover the configuration transitions of *C*.

Algorithm 4.5 Extraction of transition cover set for configuration *C*

Function DeriveConfigPSet() {
  Let ConfigP = P
  Let Transitions = the set of all transitions in M_s
  Foreach pair of transitions $(s \xrightarrow{(f,a)} t, s \xrightarrow{(g,b)} t') \in Prio$ *do*
    If $f \in C$ *and* $g \in C$ *then* {

Remove sequences containing the transition $s \xrightarrow{(g,b)} t'$ *from ConfigP*
Remove $s \xrightarrow{(g,b)} t'$ *from Transitions*
}

Foreach sequence sc in ConfigP do
If sc contains a transition tr : $s \xrightarrow{(f,a)} t$ *such that* $f \notin C$ *then*
Remove sc from ConfigP

Foreach transition $(tr : s \xrightarrow{(f,a)} t) \in$ *Transitions do*
If $\nexists sc \in ConfigP$ *such that* $sc = x.(f, a) \wedge \delta(s0, x) = s$ *then*
Remove tr from Transitions
Let ConfigTrans = Transitions

Let $P[C] = \emptyset$
While Transitions $\neq \emptyset$ *do {*
Let tr be a transition from Transitions
Let $sc \in$ *ConfigP be the shortest sequence, covering tr*
Remove all of the transitions of sc from Transitions
Add all of the prefixes of sc to $P[C]$
}
}

Example 4.5 In the implementation of Algorithm 4.5 for a configuration of the beverage vending machine product line with features *V*, *C*, *T*, *M*, and *S* the transition $s_4 \xrightarrow{(S,sreq)} s_6$ is not removed from the set of transitions. Among the covering sequences of this transition obtained in Example 4.4, the sequence with a shorter length, namely (*V, coin*).(*T, teareq*).(*T, plain*).(*S, sreq*), is selected and placed in the covering set of this configuration along with its prefixes.

Before arriving at the correct idea of choosing the sequence to cover each transition and removing redundant sequences (in the last loop of the algorithm), we tried other ideas that were not successful because they did not keep the total length of test sequences minimal. For instance, the two criteria by which sequences were selected to be included in $P[C]$ could have been (a) the greatest commonality between the transitions of the sequence and the uncovered

transitions, and (b) the greatest commonality between the set of prefixes of the sequence and the set of P [C] that has been built up to that point. We know that by selecting any sequence from *ConfigP* and adding it to P[C] its prefixes should also be added to this set. The second criterion aims to add fewer sequences to the coverage set each time. In fact, by using the combination of these two criteria, we tried to choose a sequence that includes a larger number of uncovered sequences and also minimizes the increases in the number of sequences inside P [C]. For this purpose, we used the product of these two criteria as the selection factor for the next sequence. However, the problem of this approach comes from the fact that the second criterion contradicts the first criterion. When we want to select a sequence that has more prefixes already in P [C], then most transitions in the sequence have already been covered.

4.3.3.2 Extracting the set of separating sequences for a product

After obtaining the transition cover set for the configuration, we proceed to extract the set of separating sequences or W [C]. We know that in the process of building a finite state machine for describing the characteristics of a product, some transitions are omitted and the resulting machine may no longer be minimal. More precisely, a number of states may no longer be accessible from the initial state, and some states may become equivalent to each other. In order to construct W [C], we must first determine all the states accessible from the starting state $s0$. To obtain these states we use the transitions C from the *ConfigTrans* set. Recall that this set is initialized in Algorithm 4.5. The source state of each of the transitions in *ConfigTrans* is an accessible state out of s_0. If the transition destination state is not any transition source state, it will be equivalent to the *ser* state. Reachable states are placed in a set called *ReachableStates*.

To recognize the equivalence of two states we use the set of their separating sequences. Two states $s, t \in S$ are equivalent in a configuration C if there is no separating sequence in $Q_{s,t}$ whose property set is a subset of C. In other words, these two states do not have a separating sequence in the description machine of the specification C, so they are equivalent. We keep in the *ReachableStates* set only one (arbitrary) representative of each equivalence class thus obtained.

First, we set $W[C]$ to $\emptyset$. Then, for states $s, t \in$ *ReachableStates* we select the separating sequence x so that $F_x \subseteq C$ from $Q_{s,t}$ and add it to $W[C]$. We must note that there is a possibility of intersection between separating sets. As a result, there

may be s', $t' \in ReachableStates$ such that $x \in Q_{s',t'}$. Therefore, there is no need to further examine these pairs of states to choose the separating sequence. Algorithm 4.6 below shows the process of constructing the set $W[C]$.

Algorith 4.6 Extracting the set of separating sequences for configuration C

Function GenerateConfigW() {
  Let ReachableStates = ∅
  For each transition $(tr : s \xrightarrow{(f,a)} t) \in ConfigTrans$ *do*
    Let ReachableStates = ReachableStates ∪ {s}
  For each pair of states s, t ∈ *ReachableStates do*
    If ∄$x \in Q_{s,t}$ *such that* $F_x \subseteq C$ *then*
      Let ReachableStates = ReachableStates − {s}

  Let $W[C]$ = ∅
  Let markedPairs = ∅
  For each pair of states s, t ∈ *ReachableStates do*
    If $(s, t) \notin markedPairs$ *then* {
      $W[C] = W[C] \cup \{x\} \cdot x \in Q_{s,t} \wedge F_x \subseteq C$
      Let markedPairs = markedPairs ∪ {(s, t)}
      If $x \in Q_{s',t'} \cdot s', t' \in ReachableStates$ *then*
        Let markedPairs = markedPairs ∪ {(s', t')}
    }
    For each state s ∈ *ReachableStates do*
      If ∄$x \in W[C]$ *such that the output sequence* λ (s, x) *contains no error, then*
        Let $W[C] = W[C] \cup \{(f, a)\}$ *such that* $(f, a) \in I$ *and* λ $(s, (f, a)) \neq error$
}

It is possible that all separating sequences chosen to distinguish state s from other states produce output sequence *error*. In this case, s cannot be distinguished from *ser*, and it is necessary to add a separating sequence to $W[C]$ to distinguish these two states. For this purpose it is enough to choose a sequence which consists of only one output transition from state s which has no error output. This review and, if necessary, applying the required changes, is done at the end of Algorithm 4.6.

Example 4.6 Consider configuration $C = \{V, F\}$ of the beverage vending machine product line. Algorithm 4.2 will determine that that $Q_{s0,s6} = \{(V, ready)\}$. Now we execute Algorithm 4.6. According to the transitions of this configuration, s_0 and $s6$ are known as accessible states from the initial state. Then $(V, ready)$ is placed in $W[C]$. In the last loop of the algorithm, in order to distinguish s_0 from *ser*, the sequence $(F, sodareq)$ is also added to $W[C]$.

Chapter 5

Evaluation

We will now compare our test method described in the previous chapter with the original test method that is, the *W* method. For this purpose, we implemented both methods in Java and conducted experiments to evaluate the proposed method.

5.1 Creating behavioral models of production lines

To compare the original and the new test methods, we first produced a number of F^2SM as models of product lines. The input parameters for the generator include the number of states, the number of input and output symbols, the initial number of transitions and the number of features. The result of running the program is a file that stores the generated machine information in text form. According to the initial number of transitions, we randomly select one source state and one destination state each time. We randomly select the input and output symbols as well as the feature that the transition is labeled with. As mentioned earlier, the resulting finite state machine F^2SM must be deterministic. We also know that the transition input consists of a feature along with an input symbol. Thus, when choosing the feature and input symbol for a transition we must pay attention that there is no output transition from intended source state with that feature and input symbol. For this purpose, it is necessary to add it to the list of available transitions every time a transition is added to the machine, so that we can avoid nondeterminism. It should be mentioned that if two transitions with the same input symbol and different characteristics are generated for the same state, a priority must be defined between those two transitions, again to avoid nondeterminism. One of the factors in the evaluation of some results is the number of transitions labeled with mandatory features. For this reason, the possibility of determining the number of these transitions has been provided in the product line model building program. It is also possible to determine the maximum number of priorities defined between pairs of transitions.

5.2 Validation of the model

After creating the F^2SM machine according to the above description we need to check whether the machine meets the required conditions for testing (as mentioned in Section 2). If this is not the case, we modify the machine. First, all states should be reachable from the initial state. By performing a breadth-first traversal on the machine we can find the unreachable states. For each such a state we create a path of arbitrary length from the initial state to that state. Another requirement is that the machine must be minimal. To check for minimality we first create the ρ_0, ρ_1, ... sets, as described in Section 2. If the number of equivalence classes thus obtained is equal to the number of machine states then all classes are singletons and thus the machine is minimal. Otherwise, for each two states in the same class we must add two transitions with different outputs to the machine, so that their origin is the two states mentioned and their destination is two states of different classes. As mentioned earlier, when adding a new transition, we must pay attention to keep the machine deterministic. We now have a minimal and deterministic finite state machine.

Finally, by determining the required features and defining requirement and exclusion relations between some features we obtain the model of a hypothetical production line. We know that there is an implicit relation between each feature of a child and its parent. In order to apply this concept of the feature graph, in the implementation of the methods, it is necessary to put each pair of features of the child and the parent in the requirement relationship. Obviously this is not necessary in cases where the parent feature is identical to the root feature, since the root feature is present in all products. By having a set of features, their types, and the relationships between them, we can easily obtain the set of valid configurations.

5.3 Implementation of test methods

Now that we generated the production line model, it is time to apply our test method. To create the separating sets we first implement Algorithm 4.1 presented in Section 4, which obtains all separating sequences of two separate states. Then we implement Algorithm 4.2, which determines the set of separating sequences needed for all pairs of distinct states. We execute the implemented algorithm on the finite state machine created earlier. Then, for each valid configuration we

execute Algorithm 4.6 and obtain the set of separating sequences for that configuration.

We implement Algorithms 4.3 and 4.4 to produce the transition cover set for the entire production line. By executing these algorithms on the machine describing the entire production line we obtain a set from which the transition cover set for each valid configuration can be extracted. For this purpose we implement Algorithm 4.5 and execute it for each configuration. With the transition cover sets and separating sequences the set of test sequences for each configuration can be obtained by concatenating the two sets above.

To apply the original method, i.e., the *W* test method to the software production line, we need to implement this method separately for each product in the production line. First, we need to obtain the model describing the specification of each product from the model of the whole production line. To achieve the finite state machine of a product we remove those transitions from the finite state machine of the product line that are labeled with features not present in that product. We also remove transitions with lower priority Removing these transitions may make some states unreachable from the initial state. In this case, with a breadth-first traversal we can identify reachable states and thus remove the remaining states. We then compute the equivalence classes of the states (as in Section 2) and merge all the members in each class.

To implement the *W* test method, we need to generate the transition cover sets and the separating sequences. Algorithm 2.2 should be implemented first to obtain the transition cover set. Then, by extracting partial paths from the test tree we form the transition set. Using the separating sequences and the transition cover sequences we obtain the set of test sequences for the product.

5.4 Evaluation of the test method

To evaluate our test method we compare this method with the original method. For this purpose, we created several F^2SM, each describing a hypothetical production line in the manner described above. Then, we applied the proposed test method and the *W* method to each and obtained the test sequences for each configuration of production line. We considered the improvement percentage of the total length of the test sequences of all the products of each product line, as well as the extraction time of these sequences as comparison criteria of the two methods. Therefore, we considered the above two criteria in each of the two test

methods for each model of the production line. To make the results more accurate we defined several F^2SM with constant specification factors and calculated the average criteria. We considered the number of states, the number of input and output symbols, and the number of features as constant factors. Other characteristics are different for each machine, including the total number of transitions, the number of transitions labeled with mandatory features, the number of pairs of features in a requirement relationship and the number of defined priorities. Between 20 and 30 machines are defined in each category. Table 5.1 shows the results of applying the two test methods to the machines of each category as an average.

Table 5.1 Comparison of two test methods with the average percentage improvement of the total length of test sequences and test time.

Number of states	**Number of features**	**Average percentage of total length improvement**	**Average percentage of time improvement**
20	10	-2.1%	27.5%
25	12	-1.25%	44.05%
30	15	-2.7%	47.93%
40	17	2.25%	52.17%
50	20	2.5%	61.3%

We can see that the average percentage of improvement in the total length of the test sequences in each category, both negative and positive, is very small. This indicates that the test sequences change very little in the proposed method compared to the original method. We know that in extracting the transition cover sets and separating sequences for a product in the proposed test method a number of sequences are removed from the P and W sets produced for the entire product line. The method of removal is such that it tries to keep the mentioned set minimal. However, the sets generated by the two test methods for the same product are not necessarily identical. This is because the transition cover sets and separating sequences for one machine are not unique. In Algorithm 2.2 the order in which the nodes are traversed from left to right or right to left results in a different test tree and thus different transition cover sets. From the way the W set is generated, as explained in Chapter 2, it can also be concluded that the order of selecting pairs of states to obtain their separating sequence, as well as the separating sequence chosen at each step for that state pair, affects the generated

set. We then conclude that the existence of small differences in the total length of the test sequences, is logical and a sign of the applicability of our test method.

By comparing the average percentage of test sequence production time improvement in different categories in Table 5.1 we find that with the increase in the number of features or in other words the development of the product line, the percentage of time improvement also increases. With the increase in the number of features, the number of valid products also increases, and so the amount of repetitive work performed by the *W* method for the common parts of the products also increases. As a result, the difference in the production time of the test sequences in the two methods increases and a higher percentage of improvement is achieved.

It should be noted that the percentage of time improvement for small production lines (modeled with F^2SM with 10 states and 5 features) is negative (-15% on average). When applying the proposed test method to larger production lines, the amount of work needed to extract test cases for each product is amortized over the number of products and improves the time compared to the original method. Since product lines in the industry and the real world are large, this issue is not considered a problem for the proposed test method and does not limit its applicability.

5.4.1 The relationship between production line extension and percentage of test time improvement

In order to show the effect of expanding the product line, or in other words increasing the number of features, in improving the test time we designed an experiment in which the classification of machines is done based on the number of features. In this experiment we have three categories of machines. In each category the number of states is equal to 25, the number of input and output symbols is equal to 30 and 40, respectively, and the number of transitions labeled with mandatory features is 20. There are 20 finite state machines in each category. The result of applying the two test methods to these machines is shown in Table 5.2.

Table 5.2 Changing the comparison criteria with the increasing number of attributes.

Number of features	Average percentage of length improvement	Average percentage of time improvement
10	-0.6%	41%
12	-2.6%	47%
15	1.2%	54%

We note that the percentage of test case generation time improvement increases with the increase in the number of features. It is worth noting that the change in the total length of test cases is still insignificant. This experiment shows the extensibility of our test method.

5.4.2 The influence of the commonality of the products on the proposed test method

The more commonality of products in the production line, the less work is required to obtain the test sequences of all products in our test method. This happens because our method uses the sequences in the core of the product line that are common to all products when selecting the sequences to form the P and W sets if possible., As a result, the comparisons and the next necessary actions to select the appropriate sequence are not performed, whereas the amount of product commonality has no impact in the test with the W method. We develop an experiment to illustrate this. We generate a number of F^2SM in which the number of states, the number of input and output symbols, and the number of attributes are all the same namely, 25, 30, 40, and 12, respectively. In all machines, the number of pairs of features in the requirement relationship is equal to 2 and the maximum number of priorities defined between pairs of transitions is equal to 3. However, the number of transitions labeled with mandatory features can be different in each machine. For each machine, the ratio between the number of these transitions and the total number of transitions was calculated as a percentage. For machines with the same percentage of mandatory transitions, the average percentage of improvement in test case production time for all products is shown in the graph in Figure 5.1. The percentage of the number of transitions labeled with mandatory features actually indicates the amount of commonality of the products of the product line. As can be seen, increased commonality results in improved time to produce test cases using our method compared to the W method.

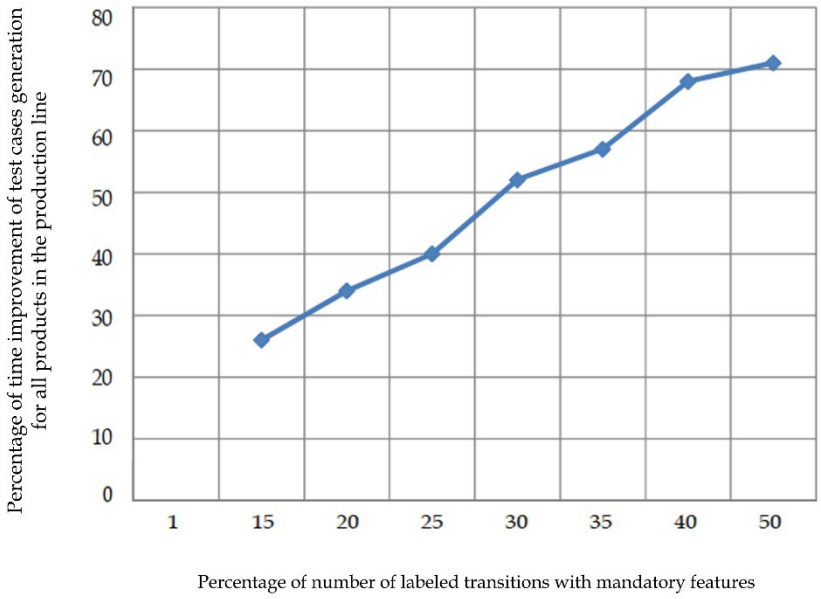


Figure 5.1 Increase in time improvement with the increasing commonality of product line products

Chapter 6

Conclusions

In this thesis we developed a model-based test for reactive software product lines. To perform a model-based test it is necessary to have a model of the expected behavior of the system under test. For this purpose we first introduced the concept of variability into the finite state machine and obtained an extension of this machine to model the system characteristics. To do that we labeled each transition of the machine with a feature of the product line. We defined priority between some transitions if necessary, according to the product line feature model. To obtain the finite state machine describing each product we removed the transitions from the machine describing the entire product line which are labeled with features not present in that product, as well as the transitions with low priority. As a consequence a number of states may become unavailable from the start state, so we removed them and obtained the equivalent minimum machine.

We then modified the *W* test method, which is an efficient method for testing finite state machines, such that we can use it for software product line testing. For this purpose we provided new algorithms for building the sets of covering transitions and separating sequences.

Existing model-based testing methods of the software product line have not used so far formal models to describe system specifications. The model-based testing presented in this thesis models the system specifications with formal methods. In the proposed test method all required test items are produced from the model describing the entire product line, and then they are customized to test each product. According to the results of the tests performed to evaluate our method the total length of the test cases of the product line is almost the same in both the proposed and original methods, but the production time of the test cases has been significantly reduced. Therefore, we conclude that the proposed test method is successful in reducing the cost of the test. We believe that this is due to the high degree in which our method reuses data used in constructing the tests.

6.1 Future work

One of the other methods of testing finite state machines is the *Wp* method. This method is slightly different from *W* test method and it reduces the length of the test cases. We believe that the *Wp* test method can also be changed to test software product lines, and thus a yet more efficient test method can be obtained.

The amount of researche on component-based software engineering is increasing. Shortening the development process and reducing costs are important goals in this field. One idea supporting this goal is to reuse components in different applications after they are developed and tested. A large reactive system can thus be obtained from the combination of smaller components. Given that each component is described by a finite state machine, a reactive system can be modeled as the combination of its components. Such a setup suggests that a theory of compositional model-based testing of finite state machines (that is, a method of obtaining global test cases based on the component tests) is worth exploring.

Bibliography

[1] ISO/IEC 26550:2015 - Software and systems engineering — Reference model for product line engineering and management

[2] G. Bockle K. Pohl and F. van der Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag Berlin Heidelberg, 2005.

[3] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In Software Product Line Conference (SPLC ’08), 2008.

[4] S. Khurshid E. Uzuncaova and D. Batory. Incremental test generation for software product lines. IEEE Transactions on Software Engineering, 36(3):309–322, 2010.

[5] E. Engstrom and P. Runeson. Software product line testing - a systematic mapping study. Information and Software Technology, 53:2–13, 2011.

[6] K. Pohl A. Reuys, E. Kamsties and S. Reis. Model-based system testing of software product families. In 17th International Conference on Advanced Information Systems Engineering, pages 519–534, 2005.

[7] J. Tretmans. Model based testing with labelled transition systems. Formal Methods and Testing, 4949:1–38, 2008.

[8] J.-P. Katoen M. Leucker and A. Pretschnerl M. Broy, B. Jonsson. Model-Based Testing of Reactive Systems: Advanced Lectures. Springer-Verlag Berlin Heidelberg, 2005.

[9] T.S. Chow. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering, 4(3):178–187, 1978.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. PROCEEDINGS OF THE IEEE, 84(8):1090–1123, 1996.

[11] A. Gill. Introduction to The Theory of Finite State Machines. McGraw-Hill Book Company, 1962.

[12] F. Belina and D. Hogrefe. The ccitt specification and description language sdl. Computer Networks and ISDN Systems, 16(4):311–341, 1989.

[13] M. A. Babar L. Chen and N. Ali. Variability management in software product lines: A systematic review. The 13th International Software Product Line Conference (SPLC 2009), (13):81–90, 2009.

[14] J. A. Hess W. E. Novak K. C. Kang, S. G. Cohen and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. In Carnegie-Mellon University Software Engineering Institue, 1990.

[15] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley, 2004.

[16] G. Halmans S. B. Gnter, S. Buhne and K. Pohl. Modeling dependencies between variation points in use case diagrams. In In Proceedings of 9th Intl. Workshop on Requirements Engineering - Foundations for Software Quality, pages 59–69, 2003.

[17] S. Segura D. Benavides and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: A literature review. Information Systems, 35(6):615–636, 2010.

[18] S. Helsen K. Czarnecki and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice, 10:7–29, 2005.

[19] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technical University of Ilmenau, 1998.

[20] D. Batory. Feature models, grammars, and propositional formulas. Software Product Lines (SPLC 2005), 3714:7–20, 2005.

[21] Mohd-Shafie, M.L., Kadir, W.M.N.W., Lichter, H. et al. Model-based test

case generation and prioritization: a systematic literature review. Softw Syst Model 21, 717–753, 2022.

[22] D. Harel and M. Politi. Modeling Reactive Systems with Statecharts: The statemate Approach. McGraw-Hill, 1998.

[23] U. Nyman K. G. Larsen and A. Wasowski. Modal i/o automata for interface and product line theories. In Proceedings of the 16th European Conference on Programming, pages 64–79, 2007.

[24] U. Nyman K. G. Larsen and A. Wasowski. Modeling software product lines using colorblind transition systems. International Journal of Software Tools for Technology Transfer, 9(5):471–487, 2007.

[25] P. Schobbens A. Legay A. Classen, P. Heymans and J.F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pages 335–344, 2010.

[26] A. Fantechi and S. Gnesi. A behavioral model for product families. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the foundation of software engineering, pages 521–524, 2007.

[27] D. Clarke and J. Proenca. Feature petri nets. In FMSPLE'10, 2010.

[28] M. Leucker A. Gruler and K. Scheidemann. Modeling and model checking software product lines. In FMOODS '08: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems, pages 113–131, 2008.

[29] M. Leucker A. Gruler and K. Scheidemann. Calculating and modeling common parts of software product lines. In Software Product Line Conference, pages 203–212, 2008.

[30] R. Milner. Communication and concurrency. Prentice-Hall, 1989.

[31] H. Gomaa and E. M. Olimpiew. Model-based test design for software

product lines. In SPLiT 2008- Fifth International Workshop on Software Product Line Testing, 2008.

[32] M. Kim S. Kang, J. Lee and W. Lee. Towards a formal framework for product line test development. In Proceedings of the 7th IEEE International Conference on Computer and Information Technology, pages 921–926, 2007.

[33] S. Mishra. Specification based software product line testing: a case study. In Proceedings of the Concurrency: Specification and Programming Workshop, pages 243–254, 2006.

[34] M. Roggenbach T. Kahsai and B. H. Schlinglof. Specification-based testing for software product lines. In Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), pages 10–14, 2008.

[35] D. Jackson. Software Abstractions: Logic, Language and Analysis. The MIT Press, 2006.

[36] F. Roessler J.J. Li, B. Geppert and D.M. Weiss. Reuse execution traces to reduce testing of product lines. In Proceedings of the International Workshop on Software Product Line Testing, 2007.

[37] S. Reis E. Kamsties, K. Pohl and A. Reuys. Testing variabilities in use case models. In Software Product-Family Engineering: 5th International Workshop, 2003.

[38] F. Robler B. Geppert, J. Li and D. M. Weiss. Towards generating acceptance tests for product lines. In 8th International Conference on Software Reuse (ICSR ’04), 2004.

[39] E. M. Olimpiew and H. Gomaa. Model-based testing for applications derived from software product lines. SIGSOFT Softw. Eng. Notes, 30(4):1–7, 2005.

[40] J. McGregor. Testing a software product line. Technical report, CMU/SEI, 2001.

[41] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In 2nd Intl. Conference on UML, 1999.

[42] M. Vieira J. Hartmann and A. Ruder. Uml-based approach for validating product lines. In Intl. Workshop on Software Product Line Testing (SPLiT), pages 58–64, 2004.

[43] A. Bertolino and S. Gnesi. Pluto: A test methodology for product families. In 5th Intl. Workshop on Product Family Engineering (PFE-5), 2003.

[44] R. M. Keller. Formal verification of parallel programs. Commun. ACM, 19(7):371–384, 1976.