Test Models and Algorithms for Model-Based Testing of Software Product Lines

Mahsa Varshosaz

Supervisors:
Mohammad Reza Mousavi
Gerardo Schneider
Abstract

Software product line (SPL) engineering has become common practice for mass production and customization of software. A software product line comprises a family of software systems which share a managed core set of artifacts. There are also a set of well-defined variabilities between the products of a product line. The main idea in SPL engineering is to enable systematic reuse in different phases of software development to reduce cost and time to release.

Model-Based Testing (MBT) is a technique that is widely used for checking the quality of software systems. In MBT, test cases are generated from an abstract model, which captures the desired behavior of the system. Then, the test cases are executed against a real implementation of the system and the compliance of the implementation to the specification is checked by comparing the observed outputs with the ones prescribed by the model.

Software product lines have been applied in many domains in which systems are mission critical and MBT is one of the techniques that is widely used for quality assurance of such systems. As the number of products can be potentially large in an SPL, using conventional approaches for MBT of the products of an SPL individually and as single systems can be very costly and time consuming. Hence, several approaches have been proposed in order to enable systematic reuse in different phases of the MBT process.

An efficient modeling technique is the first step towards an efficient MBT technique for SPLs. There have been several formalisms proposed for modeling SPLs. In this thesis, we conduct a study on such modeling techniques, focusing on three fundamental formalisms, namely featured transition systems, modal transition systems, and product line calculus of communicating systems. We compare the expressive power and the succinctness of these formalisms.

Furthermore, we investigate adapting existing MBT methods for efficient testing of SPLs. As a part of this line of our research, we adapt the test case generation algorithm of one of the well-known black-box testing approaches, namely, Harmonized State Identification (HSI) method by exploiting the idea of delta-oriented programming. We apply the adapted test case generation algorithm to a case study taken from industry and the results show up to 50
percent reduction of time in test case generation by using the delta-oriented HSI method.

In line with our research on investigating existing MBT techniques, we compare the relative efficiency and effectiveness of the test case generation algorithms of the well-known Input-Output Conformance (ioco) testing approach and the complete ioco which is another testing technique used for input output transition systems that guarantees fault coverage. The comparison is done using three case studies taken from the automotive and railway domains. The obtained results show that complete ioco is more efficient in detecting deep faults (i.e., the faults reached through longer traces) in large state spaces while ioco is more efficient in detecting shallow faults (i.e., the faults reached through shorter traces) in small state spaces.
Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Mohammad Reza Mousavi for the continuous support of my research, for his patience, motivation, and immense knowledge. Also, I would like to thank my co-supervisor Prof. Gerardo Schneider at Chalmers University for his immense and valuable support of my research.

Besides my advisors, I would like to thank the rest of my thesis committee: Dr. Maurice ter Beek, and Dr. Slawomir Nowazyck, for their insightful comments and encouragement.

I thank my fellow labmates and my friends in ITE department in Halmstad University for all the great time and interesting discussions.

Last but not the least, I would like to thank my family for supporting me throughout my research and my life.
## Contents

1 Introduction  
  1.1 Motivation .................................. 1  
  1.2 Problem Definition ............................ 3  
  1.3 Approach .................................. 3  
     1.3.1 Modeling Software Product Lines ............ 4  
     1.3.2 Model-Based Testing of SPLs ................. 5  
  1.4 Contributions ............................ 5  

2 Background and Related Work  
  2.1 Model Based Testing ........................ 7  
     2.1.1 Harmonized State Identification Method ........ 7  
     2.1.2 Input-Output Conformance Testing ............ 9  
  2.2 Software Product Lines ........................ 11  
     2.2.1 Feature Diagrams .......................... 11  
     2.2.2 Featured Transition Systems ................. 12  
     2.2.3 Product Line Calculus of Communicating Systems ... 13  
     2.2.4 Modal Transition Systems ................... 14  
     2.2.5 Model Based Testing of Software Product Lines .... 15  

3 Summary of Papers  
  3.1 Paper I: Delta-Oriented FSM Based Testing .......... 17  
  3.2 Paper II: Basic Behavioral Models for Software Product Lines:  
     Expressiveness and Testing Equivalences ............. 18  
  3.3 Paper III: Corrigendum to Basic Behavioral Models for Software  
     Product Lines: Expressiveness and Testing Equivalences .... 19  
  3.4 Paper IV: Complete IOCO Test Cases: A Case Study .... 19  

4 Conclusion and Perspectives  
References  
A Paper I
<table>
<thead>
<tr>
<th>Paper</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Paper II</td>
<td>49</td>
</tr>
<tr>
<td>C Paper III</td>
<td>71</td>
</tr>
<tr>
<td>D Paper IV</td>
<td>91</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Model Based Testing Process. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Feature diagram for a vending machine. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

vii
List of Publications

This thesis summarizes the following publications.


Chapter 1
Introduction

In this section, we provide an introduction to the software product line engineering paradigm and the main challenges in testing SPLs.

1.1 Motivation

Product lines (first invented in the automotive domain) have become a common engineering practice to enable mass production with reduced price and time to market [43]. The product line paradigm has also been applied in the practice of software development, in order to reuse software artifacts and cater for efficient usage of resources. In this context, a software product line is a set of software-intensive systems which are developed from a common set of core assets with a specific mission or with the purpose of satisfying the specific needs of a particular market. The set of software systems in an SPL share a common, managed set of features. Moreover, there are well-defined variabilities among products, which makes it possible to enable systematic re-use of artifacts, to reduce the cost and time to market, and to use standard tools and uniform interfaces. The commonalities and variabilities of various products in a product line are often factored out in terms of a feature. A feature is a distinctive user-visible aspect or characteristic of the software [27]. The products in a product line can be represented by subsets of features.

Software product line engineering has been adopted by many companies and corporations such as Boeing, Bosch, General Motors, Hewlett Packard, Philips, Siemens, and Toshiba, and in a variety of domains some of which are mission critical [60]; the application of techniques such as testing and verification to guarantee the quality of the software in SPLs has increasingly gained attention in industry and also has become the topic of interest in many research streams [37, 53].

Testing is usually a manual and laborious process which is time consuming and costly. Hence, using automated and systematic techniques to check the quality of software in an efficient manner is of great importance. Model-based
testing is an approach, which brings structure, rigour, and effectiveness into the testing process. The main virtue of this approach is test automation which allows for generating and executing test cases in an efficient way. In MBT, a model which represents the desired behaviour of a system is used to generate the set of test cases. Then, some properties of the system are checked by systematically performing experiments (executing test cases) on an executing version of the system, namely the system under test (SUT). There are different kinds of MBT methods which depend on the model being used, the level of formality involved, and also the extent of accessibility and observability of the SUT [55].

So far, there have been several attempts for modeling and model-based testing of software product lines [13, 23, 18, 25, 36, 40, 49, 34]. In an SPL, the software platform (core) is a fundamental entity and the basis for development of the products. Variability or variation points which are the places where a product differs from the software platform also become an essential property in specifying different artifacts in an SPL. A major challenge in modelling SPLs is to provide sufficiently expressive descriptions of the platforms and products in order to perform behavioral conformance checking (i.e., to test if an implementation of a product conforms to its specification). Most widely used solutions provided by engineering researchers for modelling SPLs were to annotate the existing modeling framework with the concept of variability. There are typical examples such as Featured Transition Systems (FTSs) [13, 14] which are extensions of Labeled Transition Systems (LTSs). In FTSs, the presence of transitions in individual products is indicated by annotating the transition with propositional expressions that are satisfiable by subsets of products. Other alternatives include Modal Transition Systems (MTSs) [30], which are also an extension of LTSs where the set of transitions is partitioned into may and must transitions. The may transitions may or may not be present in implementations of the model but the presence of must transitions, as their name suggests, is obligatory in any implementation of the model. It has been argued by Fischbein et al. [22] that these models are adequate for modeling SPLs, because in these models, may transitions represent optional behavior and must transitions are used for modeling the mandatory part of the behavior of the products in an SPL. Furthermore, there are other approaches such as abstract delta modeling [12], where the model for each product is obtained by application of a set of delta modules (i.e., modules specifying changes) to a core model which describes the common core behavior. However, these modeling frameworks just like UML class diagrams allow for static description of a platform and the derived products. (More details about formalisms proposed for modeling SPLs are provided in Section 2.2). Furthermore, several MBT techniques have been adapted for efficient testing of SPLs [18, 25, 36, 40, 49, 34]. The main idea in most of such works is to re-use the set of generated test artifacts between products of a product line by considering their commonalities and variabilities.
1.2 Problem Definition

As mentioned before, SPL engineering has been used to build product lines of systems such as automotive systems, power plants, medical devices, and televisions [60]. Many of the aforementioned product lines contain a large number of features. An example of a software product line taken from the open-source community is the Linux Kernel which has more than 11,000 features [50]. Considering the large number of features in a product line, the number of products in the product line can be exponentially large (in the number of features) as the products are the result of different combination of features.

Given this combinatorial explosion, in many cases performing any type of analysis on the products of a product line could be challenging compared to single systems. A naive way of analyzing an SPL is to apply the conventional and standard techniques which are used for single systems for each product in the SPL individually. Due to the potentially large number of products in an SPL, using such an approach can be very costly and time consuming in many cases. Consequently, in the last two decades, researchers have proposed many analysis techniques which are tailored for the analysis of SPLs [13, 23, 18, 25, 36, 40, 49, 34]. The common idea followed by these techniques is to systematically reduce the analysis effort by exploiting the knowledge about the features and the commonalities and the variabilities among the products of an SPL.

In this thesis, we focus on model-based testing of SPLs. As in MBT the set of test cases is generated from a model that captures the behavior of the system; then as a prerequisite of MBT of SPLs we need to construct an abstract model of the SPL. Hence, efficient modeling of an SPL is also a relevant problem in the context of the MBT of SPLs. In our research, we come up with solutions for the following research questions:

1. How do some of the formalisms used for modeling SPLs compare in terms of expressiveness and succinctness?
2. How to adapt a model-based test-case generation technique to enable reuse of testing artifacts among products of an SPL?

In the following section, we provide an overview of our work to provide solutions to the above questions.

1.3 Approach

This section is divided into two parts: first, we explain our work related to modeling SPLs and then, we explain our work regarding testing SPLs.
1.3.1 Modeling Software Product Lines

Model-based testing for SPLs, along with many other verification and quality-assurance techniques, requires a modeling technique that can efficiently capture the behavior of SPLs. Using models which are used for single systems make it impossible to perform any kind of analysis on the whole product line at once. Therefore, several formalisms have been proposed for modeling SPLs more efficiently and compactly. Some of the examples of such models are FTSs [13], MTSs [30] and their various extensions [21, 29, 6, 52], and Product Line Calculus of Communicating Systems (PL-CCSs) [23].

These formalisms can be compared from different perspectives such as succinctness, expressiveness, compositionality. Among such properties, expressiveness and succinctness, are very important as they, respectively, indicate the capability of a class of models in capturing and representing different behavior emerging from the combination of features in SPLs and the size of the model, which can affect the cost of any analysis that is based on the model.

In order to answer the first question mentioned in Section 1.2, we compared the expressiveness of three formalisms, namely FTSs, MTSs, and PL-CCSs (more detail about these models can be found in Section 2.2) in defining product behaviors (specified by LTSs). The product models are concrete implementations of the abstract model used for describing the behavior of the SPL and are generated by considering a refinement relation defined for the abstract model.

In a nutshell, we compare the expressiveness of the formalism $X$ with formalism $Y$ as follows:

- We seek an encoding from formalism $X$ to $Y$, denoted by $E: X \rightarrow Y$ that satisfies the following correctness criterion $\forall x \in X \quad [x] = [E(x)]$, where $[\cdot ]$ denotes the semantic function that represents the set of LTS implementations of the model.

- We say that formalism $Y$ is at-least as expressive as formalism $X$ if such an encoding exists. And we say $Y$ is less expressive than $X$ in case $X$ is at-least as expressive as $Y$ and such an encoding from $X$ to $Y$ (as mentioned above) does not exist.

Our results show that MTSs are less expressive than both FTSs and PL-CCSs. Also, we concluded that the PL-CCSs are as expressive as FTSs.

Succinctness is another property of these formalisms that is interesting to study as the size of the models can affect the applicability of the analysis techniques based on the model. In order to show that a class of models $X$ is exponentially more succinct compared to the class of models $Y$, we show that there exists $x \in X$ such that considering any alternative encoding $E$ from $X$ into $Y$ we have the size of the model resulted from encoding of $x$ is exponentially larger compared to the size of $x$, i.e., $|x| << |E(x)|$ and there exists an encoding
1.4. CONTRIBUTIONS

From $Y$ into $X$ we have the size of the model resulted from encoding of $y$ is in the polynomial order of the size of $y$, i.e., for all $y \in Y$, $|E'(y)| = o(k, |y|^r)$ where $k, r \in \mathbb{N}$ and $||$ is used to represent the size of the models. So far, we have studied the succinctness of two classes of models namely, PL-CCSs and FTSs.

1.3.2 Model-Based Testing of SPLs

As mentioned before, MBT of SPLs is more challenging compared to MBT of single systems due to the potentially large number of products. To adapt MBT to the SPL setting, we need to enable systematic reuse in different phases of MBT. Hence, the conventional MBT techniques used for single systems should be tailored for testing SPLs, aiming to reduce the testing effort.

An idea applied in approaches used for testing SPLs is to also generate the set of test artifacts for the platform and also generating feature test artifacts using the knowledge about the features and to combine these sets to generate the set of test artifacts for the products in the product line [18, 25, 49, 34]. Using such approaches the set of test artifacts related to the common behavior in the SPL are generated once, resulting in a reduction in testing effort.

Following this idea, and to answer the second research question mentioned in Section 1.2, we study some of the well-known MBT methods such as input-output conformance testing (ioco) method, W-method, and HSI-method. We have tailored the test case generation algorithm for the HSI-method building by generating the test cases for a core model and then adapting the set of generated test cases for different products in the SPL.

Furthermore, as part of our study on investigating existing MBT techniques, we compare the relative efficiency and effectiveness of the test case generation algorithms of the well-known ioco testing approach and the complete ioco which is another testing technique used for Input-Output Transition Systems (IOTSSs) that was developed to extend the test case generation by notion of fault coverage. The comparison is done using three case studies taken from the automotive and railway domains. A subsequent step for this line of our research is to adapt the complete ioco method for efficient testing of SPLs.

1.4 Contributions

The contributions of the work covered in this thesis can be summarised as follows:

- Providing structure in the body of knowledge about fundamental formalisms used for modeling SPLs,
- Studying the comparative expressiveness between three classes of models, namely featured transition systems, modal transition systems and product line calculus of communicating systems [7, 58].
• Adopting the HSI-method for efficient test case generation for software product lines building upon the idea of delta-oriented programming [59],

• Measuring the efficiency of the test case generation algorithm used in two well-known MBT methods, namely, complete ioco and ioco [39].
Chapter 2
Background and Related Work

In this section we explain some of the concepts that have been used in this thesis.

2.1 Model Based Testing

Software testing is a widely used technique for quality assurance of software systems. However, testing can be costly and time consuming and estimates show that this process can consume about 30-50 percent of the total software development cost [55]. The complexity of the testing techniques grows faster than the complexity of the systems being tested. Hence, systematic testing of software to reduce costs is important. Model based testing, is a rigorous technique that allows for automation in different phases of the testing process. In MBT, a set of test cases is generated from an abstract model that captures the interesting behavior of the system. This algorithmic generation of test cases allows for generating a large number of test cases in a shorter time. Then, the compliance of a real implementation of the system with the model is checked by executing the set of generated test cases against it and comparing the observed outputs with the expected ones. Figure 2.1 shows an overview of the MBT process. So far several MBT methods have been proposed. We have used and extended two well-known MBT techniques which are explained in the following.

2.1.1 Harmonized State Identification Method

One of the well-known black-box testing methods is the Harmonized State Identification (HSI) method [45]. In this method, the test models are Finite State Machines (FSMs). The model comprises a set of states and transitions and moves from one state to another by receiving an input and generating an output. The formal definition of an FSM is as follows.
Definition 2.1.1. A Finite State Machine (FSM) $M$ is a 6-tuple $(S, s_0, I, O, \mu, \lambda)$, where:

- $S$ is a finite set of states,
- $s_0 \in S$ is the initial state,
- $I$ and $O$ are, respectively, finite nonempty sets of input and output symbols,
- $\mu : S \times I \rightarrow S$ is the transition function,
- $\lambda : S \times I \rightarrow O$ is the output function.

Intuitively, whenever a machine receives input $a$ at state $s$, it deterministically traverses to state $\mu(s, a)$ and generates output $\lambda(s, a)$. A transition from state $s$ to state $s'$ with input $i$ and generated output $o$ is represented by quadruple $(s, i, o, s')$, or alternatively by $s \xrightarrow{I/o} s'$. For a sequence $x \in I^*$, we define $\mu(s, x)$ and $\lambda(s, x)$ in the standard manner to denote, respectively, the final state that the machine ends in and the sequence of generated outputs, after receiving the input symbols in $x$ one by one. Furthermore, we also informally recall that two states are $X$-equivalent ($X \subseteq I^*$) if and only if the two states produce the same output for every input sequence $\sigma \in X$ (see [10] for a formal definition). Lastly, two machines $M, M'$ are $X$-equivalent, denoted by $M \equiv_X M'$, if and only if for every state of $M$ there is an $X$-equivalent state of $M'$ and vice versa. Machine $M$ is said to conform to machine $M'$ if and only if they are $I^*$-equivalent.
2.1. MODEL BASED TESTING

In the HSI method, the main idea is to establish conformance between an FSM test model \( M \) and an unknown machine \( M' \), modeling the real implementation, by generating a finite test case from \( M \) and executing them against \( M' \). There are a set of assumptions that should hold for these machines: both machines are deterministic and minimal, all states in each of these machines are reachable from the machines’ initial states, both machines have reliable reset sequences, which take the respective machine from the current state to the initial state, and finally \( M' \) has at most as many states as \( M \). The HSI method consists of two phases. The first phase comprises checking the existence of states in the implementation that are \( I^* \)-equivalent to the ones in the test model. In the second phase, the output and the target of the transitions for the corresponding states are tested for conformance. In order to reach all of the states in the machine, the HSI method uses a set of input sequences, called the state cover set, and denoted by \( Q \), which can be defined as follows.

Definition 2.1.2. (State Cover Set) Consider an FSM \( M = (S, s_0, I, O, \mu, \lambda) \); a state cover set of \( M \) is a set of sequences such that:

\[
\forall s \in S \cdot \exists x \in Q \cdot \mu(s_0, x) = s
\]

Also the HSI method uses a separating family of sequences, which is denoted by \( Z \) and comprises sets of separating sequences for all states. A set of separating sequences identifies and tests the target states after running each element of the state cover set. The set of separating sequences can be defined as follows.

Definition 2.1.3. (Separating Sequences) Consider an FSM \( M = (S, s_0, I, O, \mu, \lambda) \); the set of separating sequences for a state \( s \in S \), is denoted by \( z_s \) and includes sequences that can distinguish \( s \) from all other states in \( S \), that is:

\[
\forall s, s' \in S \cdot s \neq s' \Rightarrow \exists x \in \text{Pref}(z_s) \cap \text{Pref}(z_{s'}) \cdot \lambda(s, x) \neq \lambda(s', x),
\]

where \( \text{Pref}(\cdot) \) denotes the set of prefixes of a set of sequences.

A separating family of sequences for an FSM, is a set comprising the separating sequences of all states. Hence, the set of test cases executed in the first phase are generated as: for each state \( s \in S \), let \( q_s \) and \( z_s \) denote, respectively, the sequence in the state cover set which leads to \( s \) and the set of separating sequences generated for \( s \). Then, the test cases generated in the first phase is given by \( \bigcup_{s \in S} r.q_s.z_s \), where \( r \) is the reset sequence. In the second phase of the HSI method, the output and the target state of the remaining transitions, not visited while traversing the state cover set, are checked.

2.1.2 Input-Output Conformance Testing

Input-output conformance testing, has been widely used as an MBT technique by many industrial cases. In this technique, LTSs and their extensions are used as test models.
Among the extensions of LTSs, IOTSs are used as test models by ioco. In IOTSs, the set of actions are separated into input and output actions. The formal definition of an IOTS is as follows.

Definition 2.1.4. An IOTS $M$ is a 5-tuple $(S, I, O, h, s_0)$, where:

- $S$ is a set of states,
- $I$ and $O$ are, respectively, disjoint sets of input and output actions,
- $h \subseteq S \times (I \cup O \cup \{\delta\}) \times S$ is the transition relation, with the symbol $\delta \not\in (I \cup O)$ denoting quiescence (lack of output),
- $s_0 \in S$ is the initial state.

An IOTS is called input-enabled if in each and every state inputs are enabled, possibly after some internal transitions (i.e., transitions with action labels which are not observable by the environment). In ioco testing theory, the conformance of an implementation of the system to a specification is checked using a set of test cases which are generated by following a recursive, non-deterministic algorithm [26, 5]. For each recursive step, it chooses among three possibilities: (i) ending the test case with the verdict pass; (ii) applying any input allowed by the specification which can be interrupted by an output arrival; or (iii) waiting for an output and checking it, or concluding the implementation is in quiescence. In [55], it is proven that this process guarantees to eventually fail all non-conforming implementations.

The ioco testing theory has been adapted for extensions of IOTSs. Complete ioco [38] is one of such techniques that is developed for testing Mealy IOTSs [48]. The behavior of Mealy IOTSs is similar to deterministic Mealy machines in that they only receive inputs in quiescent states, i.e., states where no outputs or internal transitions are enabled. Mealy IOTSs are considered more general than Mealy Machines, because they remove the one-to-one synchrony between inputs and outputs. This class of models is important since several results from IOTS and FSM testing theories, such as the use of fault domains, converge for this class of IOTSs.

In FSM-based testing, the concept of fault domain is used to guarantee the fault coverage of test suites [10]. In several FSM-based testing approaches, the problem of generating complete test suites has been addressed by considering assumptions about test models and possible implementation faults [11]. As there are no standard fault models for IOTSs, in ioco such a concept is not directly applicable. This problem is addressed by complete ioco. The test models in complete ioco should satisfy a set of properties, namely, the test models should be Mealy IOTS which contain no cycles labeled only with outputs, for each non-quiescent state, at most one transition must be labeled with an output, each state must be reachable from the initial state, any two distinct states must be distinguishable, and its transition relation must be a function.
2.2. SOFTWARE PRODUCT LINES

The test case generation algorithm in this method comprises the following steps:

1. Generating the transition cover set, which is a set comprising sequences that visit each and every quiescent state.

2. Generating the characterization set, which is a set containing input sequences that produce different outputs for each pair of quiescent states.

3. Concatenating the reset operation, comprising sequences from the transition cover and the characterization sets: The reliable reset operation, that moves the execution to its initial state, is concatenated along with sequences from the transition cover and the characterization sets; the resulting output produced by the specifications is recorded, which is to be compared with that of the implementation during test execution.

This process is deterministic and repeatable and the test suite generated by the algorithm detects all faults in the fault domain.

2.2 Software Product Lines

In this section we explain some of the concepts and constructs related to SPLs and their MBT.

2.2.1 Feature Diagrams

Software product line engineering enables systematic reuse in development of software-intensive systems. A product line is a family of software systems which are developed by sharing a common core set of artifacts and have well-defined variabilities. The commonalities and variabilities between products of a product line are described in terms of features. A feature is a distinctive user-visible aspect or characteristic of the software [27]. A product line contains a set of features and a product can be identified by a subset of features. There are different relationships among the features in an SPL. Some features may require or exclude others. Feature diagrams are a common means for the compact representation of the relationships between the features in an SPL. A feature diagram is a tree like structure in which each node represents a feature. Each feature may have a number of sub-features (represented as child nodes in tree), which can be optional or mandatory. The sibling sub-features can be in an alternative relation, which indicates that only one of the features can be included in a product and also they can be in an or relation, which indicates that one or more of the features can be present in a product that includes the parent feature. Also the features can be in an include or exclude relation. A simple example of a feature diagram is represented in Fig. 2.2. This is a feature diagram for a simple vending machine. There are different relations between features. For example features b, o, and c are mandatory and
others are optional, denoted, respectively, by filled and empty circles at the lower-end of the relations. Also features $c$ and $d$ have alternative relationship, which means a vending machine can either receive Euro or Dollars. Features $t$, $c$, and $p$ are in an or relation, which means a vending machine that belongs to this product line can provide one or more of these beverages. Furthermore, the feature $d$ excludes feature $c$, which means a product that accepts Dollars can not provide cappuccino.

![Feature diagram for a vending machine.](image)

As mentioned above, feature diagrams are structures that describe the relationships among the features of an SPL. In order to represent the behavior of the products including these features, different behavioral models can be used. To avoid modeling products of a product line individually, several conventional models have been extended to be used for efficient modeling of SPLs. In this thesis we focus on three fundamental models which have been used for modeling SPLs, namely FTSs, PL-CCSs and MTSs.

2.2.2 Featured Transition Systems

Featured Transition Systems (FTSs) [13] are extensions of LTSs proposed for modeling SPLs. Each FTS comprises a set of states and transitions. The transitions are labeled with pairs of actions and feature expressions. The feature expression specifies in which product models the transition is a valid transition. Hence, by exploiting feature expressions as annotations, the behaviour of all products can be compactly depicted in one model. We assume a propositional formula $\phi \in B(F)$ is called a feature expression, where $B(F)$ denotes the set of all propositional formulae obtained by considering the elements of the feature set $F$ as propositional variables and that $B = \{\top, \bot\}$. The formal definition of an FTS is as follows [13]:

Definition 2.2.1. A Featured Transition System (FTS) is a quintuple $(P, A, F, \rightarrow, A_p, p_{init})$, where:
- $P$ is a set of states,
2.2. SOFTWARE PRODUCT LINES

- $A$ is a set of actions,
- $F$ is a set of features,
- $\rightarrow \subseteq S \times B(F) \times A \times S$ is the transition relation satisfying the following condition:
  \[ \forall P, a, P', \phi, \phi' \ (\{P, \phi, a, P'\} \in \rightarrow \land \{P, \phi', a, P'\} \in \rightarrow) \Rightarrow \phi = \phi', \]
- $\Lambda \subseteq \{\lambda : F \rightarrow B\}$ is a set of product configurations,
- $p_{\text{init}} \subseteq P$ is the set of initial states.

Additionally, a model-checking algorithm for checking LTL formula against FTSs was given in [13]. Cordy et al. [15] extended the earlier work [14, 13] by combining non-Boolean features and multi-features in a high-level specification language called TVL$^*$. An algorithm for constructing an FTS from a behavioral specification written in TVL$^*$ was also given. In [51], a family-based approach for efficient model-checking of properties formulated by a logic which combines modalities with feature expressions over FTSs using mCRL2 [16] is proposed.

2.2.3 Product Line Calculus of Communicating Systems

In this section, we consider Product Line Calculus of Communicating Systems (PL-CCS) [23], which is an extension of Milner’s Calculus of Communicating Systems (CCS) [35]. In this formalism a new operator $\oplus$, called binary variant, is introduced to represent the alternative relation between multiple choices. The syntax of this process algebra is an extension of CCS syntax which is given in the following.

Considering $A = \Sigma \cup \bar{\Sigma} \cup \{\tau\}$ as the alphabet, where $\Sigma$ is a set of symbols and $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. The syntax of each term $e$ in PL-CCS, is defined by the following grammar: Nil | $\alpha$.e | $e + e'$ | $e \oplus_i e'$ | $e \parallel e'$ | $e[f]$ | $e \setminus L$, where Nil denotes the terminating process, $\alpha$. denotes the action prefixing for action $\alpha \in A$, $+$ and $\parallel$, respectively, denote non-deterministic choice and parallel composition, $e[f]$ denotes renaming by means of a function $f$ where $f : A \rightarrow A$, for each $L \subseteq A$. $e \setminus L$ denotes the restriction operator (blocking (co)actions in $L$), and finally $e \oplus_i e$ denotes a family of binary operators indexed with natural number $i$.

As mentioned above, the main difference between CCS and PL-CCS is the introduction of the binary variant operator $\oplus_i$. This operator is different from the ordinary alternative composition operator $+$ in CCS in that the binary variant choice is made once and for all. As an example, consider the process terms $P = a.P + b.P$ and $Q = a.Q \oplus_1 b.Q$: in the recursive process $P$, making the choice between $a$ and $b$ is repeated in each recursion, while process $Q$ makes a choice between $a$ and $b$ in the first recursion, and in all the following iterations the choice is respected. This means that process $Q$ behaves deterministically after the first iteration with respect to the choice between $a$ and $b$. For the
sake of simplicity in the formal development of the theory, Gruler assume that in every PL-CCS term, there is at most one appearance of the operator $\oplus_i$ for each index $i$.

According to [23], the validity of products can be asserted using model checking formulae specified in a multi-valued modal $\mu$-calculus [47].

As similar work done on using process algebra in the context of SPLs; in [33], Lochau et. al propose a delta oriented process calculus called DeltaCCS for modeling the behavior of systems with variability and an incremental model checking algorithm for such models against $\mu$-calculus properties. Also, in [56], variant process algebra is proposed, which is a calculus where the process terms are tagged by a set of variants where they are enabled.

### 2.2.4 Modal Transition Systems

Another extension of LTSs used for modeling SPLs are MTSs [30]. In this model, the transitions are divided into two sorts, namely, may and must transitions. May transitions may (or may not) be present in the implementation behavior, while must transitions are always present in the implementation models. It is required that all must transitions also have a corresponding may transition. According to [30], an MTS can be defined as follows:

**Definition 2.2.2.** A modal transition system is a quadruple $(P, A, \rightarrow_\diamond, \rightarrow_\Box)$, where:

- $P$ is a set of states or processes,
- $A$ is a set of actions,
- $\rightarrow_\diamond \subseteq S \times A \times S$ is the so-called may transition relation,
- $\rightarrow_\Box \subseteq \rightarrow_\diamond$ is the so-called must transition relation.

Using MTSs we can describe the behavior of optional and mandatory features of a product line in terms of may and must transition relations, respectively.

In [4, 2, 3, 32, 28, 52], MTSs and their extensions have been exploited as formal models to perform rigorous analysis of SPLs. In [4, 2, 3], valid products are derived from a given MTS by model checking against formulae expressed in a deontic logic called Modal-Hennessy-Milner-Logic (MHML). Also, in [28, 32], interface theory and a testing theory based on MTSs for SPLs have been developed. Furthermore, in [52], MTSs are extended with variability constraints for modeling behavioral variability.

There have been several variants of MTSs introduced [6, 29, 21]. Fecher et al. in [21], compare two variants of MTSs, namely Disjunctive MTSs (DMTSs) [29], and 1MTSs [21] from the expressive point of view. DMTSs are variants of MTSs in which hyper transitions, transitions with multiple target states,
are featured. Similar to MTSs, two types of Hyper transitions are considered: must/may hyper transitions. In DMTSs, the must hyper transitions represent an OR relation between mandatory multiple choices and may hyper transition represents an OR relation between optional multiple choices. The 1-Selecting MTSs (1MTSs) are similar to DMTSs in that they also feature hyper transitions. The difference is in the interpretation of the choices. In 1MTSs, the must hyper transitions represent an XOR relation between mandatory multiple choices and may hyper transition represents an XOR relation between optional multiple choices. In [21] it is shown that the two formalisms have the same expressive power, i.e., they induce the same sets of LTSs as their implementations. Also, in [20], the refinement relation and the expressive power of some of the extensions of the MTSs are discussed.

2.2.5 Model Based Testing of Software Product Lines

Much research has been carried out on MBT of SPLs. The closest lines of research to that of our work, are the ones related to incremental FSM-based testing. There are some existing approaches for incremental testing of finite state machines [18, 25, 36, 40, 49]. In this line of research the goal is to modularize the test-case generation process and/or test-case execution process considering the changes such as adding, removing, or modifying transitions or states in test models. Focusing on re-generating or re-executing tests for those parts that are influenced by the changes should eventually lead to saving time and effort. In [18, 25], which are approaches built upon the delta-oriented modeling idea, it is assumed that the behavior of the core implementation is unchanged after application of each and every delta module, which specify a set of changes in the core module, and the emphasis is put on the effect of changes on the extended part of the implementation. The approach of [49] is also used in incremental MBT of FSMs. The approach of [49] aims at completing a given set of test cases, but does not per se address the changes in the test model.

In a recent survey, Oster et al. [37] observe that there is a considerable gap regarding testing in the current software engineering approaches to SPLs. Despite this gap, there is already some body of research on the theory and application of MBT for SPLs (see, e.g., [19, 37, 54] for recent surveys). Among these approaches, the closest to our research are those developed by Lochau et al. [34]. They propose a delta-oriented and state-machine-based testing methodology for SPLs and instantiate this methodology in a case study using IBM Rational Rhapsody and Automated Test-case Generator (ATG).

Furthermore, there have been several approaches regarding MBT of object-oriented programs by using sequence- or state-diagrams as test models (see, e.g., [8, 44, 57]). So far, in our research we have followed object-oriented principles such as encapsulation and data-hiding in our modeling framework proposed for delta-oriented FSM-based testing and organized our test models based on specification of class instantiations and their dependencies. In our
work on MBT of SPLs [59], we build upon earlier works such as [24, 57]; in particular, our test models are reminiscent of class state machines (CSMs) introduced in [24]. In this approach, the system under test need not be implemented as an object-oriented program; the abstract test-cases from our test-models can be used to test different types of implementation. This is achieved by means of adapters that turn the abstract test-cases into concrete test-cases for different programming languages and implementation platforms.
Chapter 3
Summary of Papers

3.1 Paper I: Delta-Oriented FSM Based Testing

In this work we adapt the well-known black-box MBT methods, namely the HSI-method [45], for efficient testing of SPLs. To this end, we build upon the idea of Delta-Oriented Programming (DOP). Delta-oriented programming is a framework for developing SPLs, in which a product line is specified in terms of a core module and a set of delta modules. The delta modules specify a set of changes in the core module. The code for each product is obtained by application of a set of delta modules to the core. DeltaJava [46], is a DOP framework for programming SPLs in Java. In this framework, the core consists of a set of Java classes and the delta modules specify a number of changes (additions, removals and modifications of member objects, methods, and classes) in the core module.

The HSI-method uses FSMs as test models. First, we apply the idea of delta-oriented modeling for the test models by considering an FSM to be the test model for the core product; the states of the FSM denote abstraction of state valuations and the transition of the FSM represent the method calls, their returned values and their effect on the state valuations. Then, we define a set of delta modules that describe the set of changes in the core. The model for the products is obtained by applying delta modules to the core FSM. In this paper, we focus on the incremental subset of DeltaJava. Hence, the core represents a minimal subset of features that is common among products and deltas add new behavior to the core or to the composition of the core with the other delta modules.

Brief contributions: I have been the main author of this paper; My contributions in this work can be summarised as follows:

- Adapting the HSI-method for efficient test case generation for SPLs,
- Providing the complexity analysis for the adapted approach,
• Application of the proposed method on a case study and analysis of the results.


In this paper, we provide the result of a comprehensive study on some of the fundamental formalisms which are extensions of behavioral models for SPLs that extend labeled transition systems in different ways [22, 4, 2, 3, 32, 28, 14, 13, 15, 23].

Such models can be considered as semantic models for family-based extensions of higher level models such as domain specific languages (DSLs), or those based on the Unified Modeling Languages (UML) state or sequence diagrams. Hence, putting structure in the body of knowledge about such fundamental models can help the designers of higher level languages to make the appropriate choices when defining the semantics of their languages.

The structure of this paper can be divided into the following two parts:

• We compare the expressive power of three fundamental formalisms, namely PL-CCSs, FTSs, and MTSs based on the set of concrete implementations of such models, i.e, LTSs.

• We investigate the extensional and intentional notions of testing equivalences (pre-orders) for each of them. In order to compare the expressive power of PL-CCSs with two other models we consider the PL-LTSs which are used as a semantic domain of PL-CCSs, in our comparison.

In this work, the defined extensional testing notions provided regarding the testing equivalences are novel. They are of course based on and some slight extensions of well-known notions of tests for LTSs (e.g., of [17, 42, 55] and particularly of [1]).

Our results, show that the MTSs are the least expressive of all three formalisms. PL-LTSs are less expressive than FTSs (this result is correct due to a restriction considered in the definition of the transition relation of PL-LTSs. Later, in paper III, we relax this restriction and prove that PL-LTSs are at least as expressive as FTSs) and more expressive compared to MTSs. Featured transition systems have the most expressive power among these formalisms.

Brief contributions: In this paper I have been the second author. My contributions in this work can be summarised as follows:

• Conducting a survey on formalisms used for modeling SPLs in order to obtain the set of models considered in the comparison,

• Providing the comparative expressiveness between FTSs and PL-LTSs.

This paper is a corrigendum to [7], where we relax a restrictive assumption that has been made about the transition rules of PL-LTSs.

As PL-LTSs are defined as the semantic domain for PL-CCS models [23]; in order to keep track of variant choices, a configuration vector is included in the state of PL-LTSs. In each PL-LTS, the size of the vector is equal to the number of the variant choices in the corresponding PL-CCS term. The elements of the configuration vector can denote choosing left, right or an undecided choice. Namely, in [7], we assumed that in each step, only one of the variant choices can be resolved. (Based on this assumption each transition can change only one of the elements of the configuration vector in the target state.) This turned out to be an overly restrictive assumption compared to the definition given for the PL-LTS transition rules in [35]. Considering this assumption, as a part of the results it was shown that PL-LTSs cannot capture some types of behavior such as three-way choices which can be captured by FTSs. Hence, it was concluded that the class of PL-LTSs is less expressive than the class of FTSs. In this paper, we relaxed this assumption and showed that the class of PL-LTSs is at least as expressive as the class of FTSs. Also, we showed that using FTSs for modeling SPLs leads to more compact models compared to using PL-CCS.

Brief contributions: I am the main author of this paper. My contributions in this work are summarised in the following:

- Providing the comparative expressiveness of FTSs and (relaxed notion of) PL-LTSs,
- Providing a comparison on the succinctness of the FTSs and PL-LTSs.

3.4 Paper IV: Complete IOCO Test Cases: A Case Study

In this paper, we compare the efficiency of the test case generation algorithm used by the traditional ioco approach and another method called complete ioco which also uses IOTSs as test models and applies fault domains to obtain complete test suites with guaranteed fault coverage.

In order to measure the efficiency of the test case generation algorithms, the following three case studies are considered:

- A model of turn indicator lights in Mercedes vehicles was presented in [41], which describes the functionality of left/right turn indication, emergency flashing, crash flashing, theft flashing and open/close flashing.
• The Body Comfort System [31], which is a case study taken from the automotive domain, describing the internal locks and signals of a vehicle model.

• The ETCS Ceiling Speed Monitor (CSM) [9], which is part of the European standard specification for train control systems.

We took the following steps to complete the measurements: We modelled the behavior of the above case studies to use them as test models for ioco and complete ioco. Then, a set of 20 faulty mutants for each specification model was produced. The test case generation processes using ioco and complete ioco were both run 50 times. The obtained results for the two methods were compared. Our results show that the complete ioco is more efficient in detecting deep faults in large state spaces while ioco is more efficient in detecting shallow faults in small state spaces.

Brief contributions: I am the third author of this paper. My contributions in this work can be summarised as follows:

• contributing in designing the experiments and generation of the test models from the specifications of the case studies, which are used for comparing the effectiveness of ioco and complete ioco test case generation algorithms.
Chapter 4
Conclusion and Perspectives

In this thesis we provided an overview of challenges for modeling and model-based testing of software product lines. We studied some of the fundamental models used for efficient modeling of SPLs. We provided the comparative expressiveness of three models, namely FTSs, MTSs and PL-CCSs. The comparative expressiveness was provided by means of describing an encoding from one class of models to the other and then showing that the corresponding models in two classes are implemented by the same set of labeled transition systems. Our results show that MTSs are the least expressive in the hierarchy. We also show that PL-LTSs are as expressive as FTSs. We also compared the succinctness of the PL-LTSs and FTSs. Our results show that modeling product lines using FTSs can lead to more compact models compared to using PL-LTSs. In our work we have only considered the finite behavior of the systems. Considering the infinite behavior is part of our future work. Completing the lattice of expressive power given in [7], by including other formalisms such as 1-selecting MTSs, disjunctive MTSs and parametric MTSs [6], is another line of our future work regarding modeling SPLs.

As another part of our work regarding MBT of SPLs, we adapted the test case generation algorithm for the HSI-method by exploiting the idea of delta-oriented modeling for efficient test case generation for SPLs. We presented the SPL behavior using a core test model which is an FSM and a set of delta modules that indicate changes in the core model. We considered DeltaJava to indicate the realistic type of changes that can happen in the core test model. We applied our approach to an industrial case study, which resulted in up to 50 percent reduction on time for generating test cases. As part of future work related to this part of our research we intend to extend our approach to cover the complete syntax of DeltaJava, in particular, consider modifying and removing methods. We also plan to implement our approach on top of one of the existing MBT tools.

As a part of study about existing MBT approaches, we also studied the efficiency of ioco which is a well-known MBT approach in comparison to com-
plete ioco. The comparison was run based on three case studies taken from industry. The results show that the complete ioco is more efficient in detecting deep faults in large state spaces while ioco is more efficient in detecting shallow faults in small state spaces. As a part of future work related to this line of our research, we plan to apply this comparison to different specifications and additionally investigate the prioritization of test cases in the execution of test suites in order to gain more insight about the performance of the complete ioco method.
References


REFERENCES


Appendix A

Paper I
Delta-Oriented FSM-Based Testing

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi

2015 International Conference on Formal Engineering Methods (ICFEM)
Paris, France, November 3-6, 2015
Delta-Oriented FSM-Based Testing

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi

Center for Research on Embedded Systems
Halmstad University, Sweden
{mahsa.varshosaz,harsh.beohar,m.r.mousavi}@hh.se

Abstract. We use the concept of delta-oriented programming to organize FSM-based test models in an incremental structure. We then exploit incremental FSM-based testing to make efficient use of this high-level structure in generating test cases. We show how our approach can lead to more efficient test-case generation, both by analyzing the complexity of the test-case generation algorithm and by applying the technique to a case study.

Keywords: Model-Based Testing, FSM-based Testing, HSI Method, Software Product Lines, Delta-Oriented Programming, DeltaJava

1 Introduction

Software product lines (SPLs) have become common practice thanks to their potential for mass production and customization of software. Testing software product lines, and in particular, their model-based testing are topics of increasing relevance in the research literature and also industrial practice [4, 10, 17]. In this paper, we propose the formal foundations of a delta-oriented framework for model-based testing, Delta-oriented programming (DOP) and in particular, DeltaJava [14], is a framework for SPLs, in which a product line is specified in terms of applications of a number of deltas (changes: additions, removals and modifications of member objects, methods, and classes) from a core product. The overall goal of the research commenced by this paper is to allow for efficient test-case generation and test-case execution for delta-oriented models and their corresponding programs. In this paper, we focus on test-case generation and show whether and how test-case generation for delta-oriented models can be made more efficient by benefiting from their incremental structure.

To this end, we use finite state machines (FSMs) as test models whose structure is based on DeltaJava; there is a test-model for the core product, which includes abstraction of state valuations as its states and the method calls, their return values and their effect on the abstract state as its transitions. Then, test models for different products are obtained by incrementally modifying the details of the core model (e.g., adding models for classes, member objects and

* The work of M.R. Mousavi has been partially supported by the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation in the context of the AUTO-CAAS H6G project.
methods). In this paper, we focus on the incremental subset of DeltaJava, in which the core represents a minimal set of features and the deltas incrementally add to the core or the composition of core with other deltas (but do not remove anything from them). We also adopt the well-known Harmonized State Identification (HSI) method [13] and adapt it to the delta-oriented structure of our test models.

The remainder of this paper is organized as follows. In Section 2, we review several pieces of related work and identify their similarities and differences with the present paper. In Section 3, we recall some preliminary notions regarding FSM-based testing and the syntax of delta-oriented models. We specify the syntactic structure of our running example in Section 4, which we use throughout the rest of the paper to illustrate various formal definitions. Subsequently, we define the semantic domain of our test models in Section 5 and show how the test models of various products can be obtained from the semantics of the core model by applying a delta composition operator. In Section 6, we show how test cases can be generated from the test models of various products and analyze the complexity of test-case generation. In Section 7, we provide some empirical results obtained from comparing the effectiveness of the application of the delta-oriented testing method with the HSI-method for a case study. We conclude the paper and present the directions of our ongoing research in Section 8.

2 Related Work

Incremental FSM-Based Testing The closest line of research to that of the present paper is incremental FSM-based testing, which is extensively researched in the past few years [3, 6, 9, 11, 15]. This line of research aims at modularizing the test-case generation and/or test-case execution process with respect to changes such as adding, removing, or modifying transitions or states in test models. Such a modularization should eventually lead to saving time and effort in re-generating or re-executing tests by focusing on those parts that are influenced by the change. The approaches of [3, 6] differ from our approach in that they assume that the behavior of the core implementation is unchanged after each and every delta and focus on the effect of changes on the extended part of the implementation; we have no assumption about the behavior of the implementation due to the application of a delta. Our focus in this paper is on test-model semantics and test-case generation rather than test-case selection and execution. The approach of [15] is different from ours in that it aims at completing a given set of test cases, but does not per se address the changes in the test model. Our approach is mostly based on [9, 11] and applies it at a higher level of abstraction to delta-oriented models inspired by the DeltaJava framework of [14].

Model-Based Testing of SPLs In a recent survey, Oster et al. [10] observe that there is a considerable gap regarding testing in the current software engineering approaches to SPLs. Despite this gap, there is already some body of research on the theory and application of model-based testing for SPLs (see, e.g., [4, 10,
Among these approaches, the closest to our approach are those developed by Malte Lochau, Ina Schaefer, et al. They propose a delta-oriented and state-machine-based testing methodology for SPLs and instantiate this methodology in a case study using IBM Rational Rhapsody and Automated Test-case Generator (ATG). Our approach follows the same structure and formalizes the part that has been implemented in IBM Rhapsody, by means of ideas from incremental FSM-based testing. This paves the way for further formal analyses of the technique proposed in [8], as well as further improvements by considering more relaxed fault models.

Object-Oriented Model-Based Testing There is a large body of literature regarding model-based testing of object-oriented programs by using sequence- or state-diagrams as test models (see, e.g., [1, 12, 18]). We follow object-oriented principles such as encapsulation and data-hiding in our modeling framework and organize our test models based on specification of class instantiations and dependencies. In this sense, our work builds upon earlier work in this direction such as [5, 18]; in particular, our test models are reminiscent of class state machines (CSMs) introduced in [5]. Our work differs from this line of work in two ways: firstly, our focus is on incremental changes in test models and not so much on testing object oriented programs. Secondly, in our approach the system under test need not be implemented as an object-oriented program; the abstract test-cases from our test-models can be used to test different types of implementation. This is achieved by means of adapters that turn the abstract test-cases into concrete test-cases for different programming languages and implementation platforms.

3 Preliminaries

3.1 FSM-Based Testing

In this section, we explain the basic concepts of FSM-based testing and delta-oriented modeling techniques used throughout the rest of the paper. We use the Harmonized State Identification (HSI) method [13] as the basis of our model-based testing technique. In the HSI method, test models are Finite State Machines (FSMs), specifying the desired behavior of systems. The formal definition of an FSM, borrowed from [2], is as follows.

**Definition 1.** (Finite State Machine) A Finite State Machine (FSM) $M$ is a 6-tuple $(S, s_0, I, O, \mu, \lambda)$, where $S$ is a finite set of states, $s_0 \in S$ is the initial state, $I$ and $O$ are, respectively, finite nonempty sets of input and output symbols, $\mu : S \times I \rightarrow S$ is the transition function and $\lambda : S \times I \rightarrow O$ is the output function.

Intuitively, whenever a machine receives input $a$ at state $s$, it deterministically traverses to state $\mu(s, a)$ and generates output $\lambda(s, a)$. A transition from state $s$ to state $s'$ with input $i$ and generated output $o$ is represented by quadruple $(s, i, o, s')$, or alternatively by $s \xrightarrow{i/o} s'$. For a sequence $x \in I^*$, we define $\mu(s, x)$
and $\lambda(s, x)$ in the standard manner to denote, respectively, the final state that the machine ends in and the sequence of generated outputs, after receiving the input symbols in $x$ one by one. Furthermore, we also informally recall that two states are $X$-equivalent ($X \subseteq I^*$) if and only if the two states produce the same output for every input sequence $\sigma \in X$ (see [2] for a formal definition). Lastly, two machines $M, M'$ are $X$-equivalent, denoted by $M \equiv_X M'$, if and only if for every state of $M$ there is an $X$-equivalent state of $M'$ and vice versa. Machine $M$ is said to conform to machine $M'$ if and only if they are $I^*$-equivalent.

The main idea of the HSI method is to establish conformance between an FSM test model $M$ and an unknown machine $M'$, modeling the implementation, by generating a finite test case from $M$ and applying it to $M'$. There are a set of assumptions that should hold for these machines, which are specified next.

**Definition 2. (HSI method assumptions)** The HSI method can be applied on machines $M$ and $M'$, which satisfy the following assumptions:

1. Both $M$ and $M'$ are deterministic, i.e., for each state and each input $i$, there is at most one outgoing transition labeled with $i$.
2. Both $M$ and $M'$ are minimal, i.e., there are no distinct $I^*$-equivalent states in either of them. Note that if $M$ is not minimal, an equivalent minimal machine can be generated using a minimization algorithm such as [7].
3. All states in $M$ are reachable from its initial state $s_0$.
4. Both machines $M$ and $M'$ have reliable reset sequences, which take the respective machine from the current state to the initial state.
5. $M'$ has at most as many states as $M$.

The HSI method consists of two phases. The first phase comprises checking the existence of states in the implementation that are $I^*$-equivalent to the ones in the test model. In the second phase, the output and the target of the transitions for the corresponding states are tested for conformance. In order to reach all the states in the machine, the HSI method uses a set of input sequences, state cover set, denoted by $Q$, which is defined below.

**Definition 3. (State Cover Set)** Consider an FSM $M = (S, s_0, I, O, \mu, \lambda)$; a state cover set of $M$, denoted by $Q$, is a set of sequences such that:

$$\forall s \in S \cdot \exists \pi \in Q \cdot \mu(s_0, x) = s$$

A state cover set of an FSM can be obtained by building a spanning tree such that, the nodes are states of the FSM and the edges are chosen from the set of transitions in the FSM. The set of sequences obtained as the state cover set are then the paths from the initial state to the nodes in the spanning tree.

As another ingredient of the first phase, i.e., checking the existence of test-model states in the implementation, the HSI method uses a separating family of sequences, which is denoted by $Z$ and comprises sets of separating sequences for all states. A set of separating sequences identifies and tests the target states after running each element of the state cover set. The separating set for a state is defined as follows.
Definition 4. (Separating Sequences) Consider an FSM $M = (S, s_0, I, O, \mu, \lambda)$; the set of separating sequences for a state $s \in S$, is denoted by $z_s$ and includes sequences that can distinguish $s$ from all other states in $S$, that is:

$\forall s, s' \in S : s \neq s' \Rightarrow \exists x \in \text{Pref}(z_s) \cap \text{Pref}(z_{s'}) : \lambda(s, x) \neq \lambda(s', x),$

where $\text{Pref}(\cdot)$ denotes the set of prefixes of a set of sequences.

A separating family of sequences for an FSM, is a set comprising the separating sequences of all states, that is $Z = \bigcup_{s \in S} \{z_s\}$.

Hence, the set of test cases executed in the first phase are generated as follows. For each state $s \in S$, let $q_s$ and $z_s$ denote, respectively, the sequence in the state cover set which leads to $s$ and the set of separating sequences generated for $s$. Then, the test cases generated in the first phase is given by $\bigcup_{s \in S} r \cdot q_s \cdot z_s$, where $r$ is the reset sequence of the FSM and for two sets $A$ and $B$ of sequences, $A \cdot B$ denotes the concatenation of two sets and is defined as $\{a \cdot b | a \in A \land b \in B\}$. This way, in addition to checking the existence of the states, the output and target state of the transitions which are included in the spanning tree are checked for conformance to the specification.

In the second phase of the HSI method, the output and the target state of the remaining transitions, not visited while traversing the state cover set, are checked using the following set of test cases. For each of the remaining transitions such as $s \xrightarrow{a} s'$, the set of all $r \cdot q_s \cdot i \cdot z_{s'}$ sequences is added to the set of test cases.

3.2 Delta-Oriented Syntactic Structure

Inspired by DeltaJava [14], our test-models for an SPL are structured into a core model and a set of delta models. The core model describes the correct behavior of a valid configuration in the SPL. The implementation of other products is obtained by applying delta models to the core model. The structure of our models is defined by the syntax of DeltaJava, which is described below.

A core model comprises a set of Java classes and a set of interfaces, that is:

```
core {Feature names} {Java classes and interfaces},
```

where feature names specify the set of features which are included in the configuration corresponding to the core model.

Delta models describe sets of changes to the core model. The structure of a delta in the DeltaJava language is given in Fig. 1. In this syntax, a delta model may add/remove fields, methods, or interfaces from classes in the core model. Also, it can modify the existing ones. A class can also be added or removed from a core model by applying a delta model. The keyword `after` can be used in order to specify the order of the application of a set of delta model to the core model. The `when` keyword is used to specify that this delta can be applied when a set of features are being included in the configuration. In the remainder of this paper, we only consider incremental delta-oriented models, i.e., those models that only
add model classes, methods or fields. In this paper, we focus on an incremental subset of the syntax, designated in blue, which assumes a minimal core and incremental additions by various deltas. Particularly, in Section 5, we provide a semantic domain in terms of FSMs for a subset of these syntactic structures, which covers adding classes, methods and fields to a core FSM model.

4 Running Example

In this section, we present the syntax of a DeltaJava example, which is used throughout the rest of this paper. The core model of this example consists of one class, named Bridge. This class has a field that represents the availability of the bridge and also a set of functions, which manipulate and report the value of this field. The syntax of the core model is given in Fig. 2.

Fig. 1. DeltaJava syntax.

```
delta (name) { after (delta names) }
when (application condition) {
  remove (class or interface name)
  adds class (name) { (standard Java class)
  adds interface (name) { (standard Java interface)
  modifies interface (name) {
    (remove | add | rename) method header clauses
  }
  modifies class (name) {
    (remove | add | rename) field clauses | (remove | add | rename) method clauses
  }
}
```

Fig. 2. Core- and delta models of the running example.
We consider two different delta models to be added to the core model given in Fig. 2. The first delta model consists of the addition of a class. The class controller controls the status of the lights in both side of the bridge in order to guarantee a mutually exclusive access to the bridge. This delta is added when the feature controller is included in a product. The second delta model is added to the core model when the pedestrian light feature is included in the product. This delta model consists of adding a field to the bridge class, which represents the status of the pedestrian light, as well as two methods, which can set and reset the value of the pedestrian light.

5 Delta-Oriented FSM Modeling

In this section, we define a semantic domain based on FSMs for the syntactic structure of DeltaJava models. We assume that the transitions in our test models concern the call / return behavior of a set of modules. The states in a test model concern a symbolic aggregation of concrete states, where each concrete state corresponds to a valuation of variables. The granularity of this abstraction is modeler’s choice, as long as it respects the HSI assumptions. Moreover, it is assumed that the set of fields used and manipulated by a method call, its possible return values and its effect on the value of these fields are known.

To start with, we define the following basic concepts for our semantic domain.

Definition 5. (Abstract Valuations) Assume a set \( V \) of variables and a set \( D \) of their possible values; for simplicity, we have left out typing information here and throughout the paper. Then \( \text{Val}^V \subseteq 2^{V \rightarrow D} \), is an abstract valuation (i.e., a set of valuations) of \( V \). The set of all such abstract valuations of \( V \) is denoted by \( \text{VAL}^V \). We remove the superscript of an abstract valuation, if the set of variables is clear. For an abstract valuation \( \text{Val}^V \subseteq 2^{V \rightarrow D} \) and for \( V' \subseteq V \), we write \( \text{Val}^V \downarrow V' \) to denote element-wise domain restriction of \( \text{Val} \) to \( V' \), that is leaving out the valuation of those variables not mentioned in \( V' \).

Definition 6. (Object Structure) We formalize the structure of an object \( \text{obj} \) of class \( c \), as a \( 3 \)-tuple \((\text{Id}, \text{Flds}, \text{Mtds})\), where \( \text{Id} \) is the object’s unique identifier and \( \text{Flds} \) and \( \text{Mtds} \), respectively, denote the set of fields and methods in the class \( c \). (To avoid name clashes, we assume that all members of \( \text{Flds} \) and \( \text{Mtds} \) are prefixed with \( \text{Id.} \)) A method is represented by a \( 5 \)-tuple \((\text{Id}, \text{Inprms}, \text{Outprm}, \text{Clds}, \text{UsedVars})\), where \( \text{Id} \), \( \text{Inprms} \) and \( \text{Outprm} \), respectively, denote the name of the method and the list of the input parameters and the output returned by the method; \( \text{Clds} \) denotes the set of methods that are called in the body of this method and \( \text{UsedVars} \) is the set of variables read from or written to in the method. Note that \( \text{UsedVars} \) may comprise both members of \( \text{Flds} \) and model variables. The latter are variables that the test modeler has added to the model to capture unspecified details, e.g., associations and dependencies, without cluttering the model.

In the rest of the paper, we recognize the components of the above-given tuples, by indexing the name of the intended component with the name of the
object or the method. For example, \( \text{Inprms}_m \) denotes the input parameters of the method \( m \). Next, we define the concept of post-condition for methods.

**Definition 7.** (Effect and Return Value Functions) The effect of calling a method \( m \) is defined by a function \( \text{Effect}_m : \text{VAL}^{\text{Inprms}_m \cup \text{UsedVars}_m} \rightarrow \text{VAL}^{\text{UsedVars}_m} \). Similarly, its set of admitted return values is defined by:

\[
\text{RetVal}_m : \text{VAL}^{\text{Inprms}_m \cup \text{UsedVars}_m} \rightarrow 2^D.
\]

### 5.1 Core Model Semantics

In this section, we define the semantic domain for core models. The behavior of a core model results from execution of the methods called in the objects instantiated from the core model classes. (A conscious choice is to be made by the modeler as to which methods from which abstract states are included in the model.) Hence, the finite state machine describing the behavior of a set of objects is defined as follows.

**Definition 8.** (Object FSM) An FSM \( M(O) = (S, s_0, I, O, \mu, \lambda) \) is a semantic model for a set \( O \) of objects from the set \( C \) of classes, if it satisfies the following conditions:

1. \( S \subseteq \text{VAL}^V \) where \( V \subseteq \bigcup_{o \in O, m \in \text{Mtds}_o} \text{UsedVars}_m \) is a subset of model variables and fields in \( O \), this means that each state in \( S \) is an abstract valuation of a subset of model variables and fields.
2. \( I \subseteq \bigcup_{o \in O, m \in \text{Mtds}_o} \{ (\text{Id}_m) \times \text{VAL}^{\text{Inprms}_m} \} \); this means that each input in the input symbols set comprises a method name and a set of passed arguments.
3. \( O \subseteq D \) is the set of possible return values of the method calls in \( I \).
4. \( \mu : S \times I \rightarrow S \), is a transition function satisfying the following conditions:
   - \( (1) \forall o \in O, m \in \text{Mtds}_o, \text{val} \in \text{VAL}^{\text{Inprms}_m \cup \text{UsedVars}_m}, i, s, s' \in S : \mu(s, i) = s' \land i = (\text{Id}_m, \text{val}) \Rightarrow \text{Effect}_m(s \downarrow \text{UsedVars}_m, \text{val}) \subseteq s' \downarrow \text{UsedVars}_m, \)
   - \( (2) \forall s, x \in S : \exists i \in I : (s, x) = s \)
   - \( (3) \forall s, x \in S : \mu(s, r) = s_0 \)
5. \( \lambda : S \times I \rightarrow O \) is an output function satisfying the following condition:

\[
\forall o \in O, m \in \text{Mtds}_o, \text{val} \in \text{VAL}^{\text{Inprms}_m \cup \text{UsedVars}_m}, i \in I, s \in S : \lambda(s, i) = o \land i = (\text{Id}_m, \text{val}) \Rightarrow \text{RetVal}_m(s \downarrow \text{UsedVars}_m, \text{val}) = o.
\]

Our notion of abstract states are reminiscent of similar notions (based on the category-partition method) in the literature [19]. Regarding the transition function, condition (1) specifies that there can be a transition from one state to another, labeled with a method call as input, only if this method call maps one of the concrete evaluations of the used variables in the source to another concrete valuation in the target state. Condition (2) requires that all states included in the set of states are reachable from the initial state. Condition (3) postulates that the given FSM has a reset sequence \( r \). Regarding the output function, the
condition specifies that the output of the FSM for each given input is exactly the set of admitted outputs for the corresponding method.

A test model for core, defined below, is then an object FSM comprising a set of objects from the core model classes.

**Definition 9. (Test Model for Core)** A test model for core is a minimal object FSM \( M(\mathcal{O}) \) such that each object in \( \mathcal{O} \) is instantiated from a class in the core model.

For example, the FSM corresponding to the core model in the running example is demonstrated in Fig. 3 (a). This FSM is minimal and it satisfies the reachability condition. The reset sequence of this FSM is \( \text{SetAvl}() \).

\[
\begin{align*}
\text{SetAvl}() &/ \tau \\
\text{ResetAvl}() &/ \tau \\
\text{CheckAvl}() &/ f \\
\text{CheckLsig}() &/ f \\
\text{CheckRsig}() &/ t \\
\text{SetPassed}() &/ \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{GetReq}(1) &/ \tau \\
\text{GetReq}(0) &/ \tau \\
\text{SetPassed}() &/ \tau \\
\text{GetReq}({0},{1}) &/ \tau, \text{GetReq}({0},{1}) &/ \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{CheckLsig}() &/ f, \text{CheckRsig}() / f, \text{CheckLsig}() / t, \text{CheckRsig}() / t \\
\end{align*}
\]

Fig. 3. (a) FSM modeling the bridge class, (b) FSM modeling the controller class

### 5.2 Delta Application

In this section, we define the semantic domain for delta models and the application of a delta to a core model. As mentioned in Sect. 3.2, a delta comprises a set of operations applying changes to the core model. In order to give a practical definition to a delta model and the type of changes that it can make to the core model, we focus on adding a class, on one hand and adding a set of fields and methods, on the other hand. The reason we combine adding fields and methods in one step is that often adding new methods requires some additional fields. Moreover, in several cases the new abstract valuations (additional state-partitions) due to the additional fields can only preserve minimality, if new methods are also added to tell them apart. We leave the deltas concerning removals and modifications of methods and removal of fields for future work. Hence, for now we
are assuming that the core model comprises the least mandatory set of features and the model regarding each product is generated incrementally from the core model.

We proceed by defining the effect of applying a delta containing each of the above-mentioned changes on the core model’s FSM.

Adding a Class The test model for the added class has the structure and abides by the constraints of object FSMs given in Definition 8. Hence, we assume that the test model for the added class \( c \) is given as a minimal object FSM \( M_d(O_d) \) where \( O_d \) only contains objects of class \( c \) with a fresh identifier (not mentioned among the identifier of core objects and other deltas).

For example, the FSM describing the behavior of an object of the controller class is depicted in Fig. 3 (b). In this figure, \( \langle \text{CheckAvl} \rangle \) is an extra model variable included in the state, representing the returned value of \( \text{CheckAvl}() \) and cutting the dependency with the core model. The result of adding a class to the core model is defined as follows.

Assume that the test models for the core and the delta models are object FSMs \( M(O) = (S, s_0, I, O, \mu, \lambda) \) and \( M_d(O_d) = (S_d, s'_0, I_d, O_d, \mu_d, \lambda_d) \), respectively. In order to define the composition of the core and the delta, we first specify the possible connections between the model variables of delta and core. Assuming that \( V \) and \( V_d \), respectively, denote the variables in the domain of the states in \( S \) and \( S_d \), then, the (partial) composition function \( \gamma : V_d \rightarrow V \) specifies which (model) variables in \( V_d \) should match which variables in \( V \). Moreover, the methods of the delta class can initiate method calls to instances of the core class included in the delta class (if any). Here, for the sake of simplicity, we consider that each delta method can contain at most one method call to the core, but the generalization to a sequence of core method calls is straightforward. We assume that the set of methods in the core model and the set of methods in the delta model are denoted, respectively, by \( MTD \) and \( Mtds \).

**Definition 10.** The result of composing the above-given models \( M \) and \( M_d \) with regards to \( \gamma \) is an FSM \( M'(O') = (S', s'_0, I', \mu', \lambda') \), where:

\[
\begin{align*}
- S' &= \{ \text{val} \in VAL_{\cup \lambda \gamma} \mid \text{val} \downarrow V \in S \land \text{val} \downarrow V_d \in S' \land \forall v_{\gamma_e} \in V_d, \forall v \in V : \gamma(v_{\gamma_e}) = v \Rightarrow \text{val} \downarrow \{ v_{\gamma_e} \} = \text{val} \downarrow \{ v \} \}; \text{ for the composition to be well-defined, we assume } V \text{ and } V_d \text{ to be disjoint, } \\
- s'_0 &\text{ is the initial state such that } s'_0 \downarrow V = s_0 \text{ and } s'_0 \downarrow V_d = s'_0, \\
- I' &= I \cup I_d \\
- O' &= O \cup O_d \\
- \mu' : S' \times I' \rightarrow S', \text{ is the transition function. For each } i \in I', \text{ we distinguish the following three cases:} \\
- \bullet i \in I \text{ concerns a method call from the core; then, the following condition should be satisfied}
\end{align*}
\]

\[
\forall m \in MTD, s'_1, s'_2 \in S', \downarrow i = (Id_m, \text{val}) \Rightarrow (\mu'(s'_1, i) = s'_2) \Leftrightarrow \\
\exists_{s_2, v_{\gamma_e} \in S' \cdot \downarrow 1} \downarrow \text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \wedge \mu(s_1, i) = s_2
\]
• \( i \in I_d \) concerns a method call from delta that does not have any nested call to the core; then, the following condition should be satisfied

\[
\forall m \in M_d, s, s' \in S \cdot i = (Id_m, \text{val}) \Rightarrow \left\{ \begin{array}{l}
\mu'(s'_i, i) = s'_2 \\
\exists s' \in S \cdot s'_1 \downarrow \text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \\
\text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \mu(s'_1, i) = s'_2
\end{array} \right.
\]

• \( i \in I_d \) concerns a method call from delta that has a nested method call \( n_i \) to the core; then the following condition should hold:

\[
\forall m \in M_d, n \in M_T, s, s' \in S \cdot i = (Id_m, \text{val}) \land n_i = (Id_n, \text{val}_n) \Rightarrow \\
\mu'(s'_i, i) = s'_2 \iff \exists s' \in S \cdot s'_1 \downarrow \text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \\
\text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \mu(s'_1, i) = s'_2 \land \\
\exists s, i \in S' \cdot s_1 \downarrow \text{UsedVars}_n = s'_1 \downarrow \text{UsedVars}_n \land \\
\text{UsedVars}_n = s'_1 \downarrow \text{UsedVars}_n \land \mu(s_1, n_i) = s'_2
\]

– \( X' : S' \times I' \rightarrow O' \) is the output function; for each \( i \in I' \), we distinguish the following two cases:

• either \( i \in I \), then the following condition should hold:

\[
\forall m \in M_T, s_0 \in O' \cdot i = (Id_m, \text{val}) \Rightarrow \\
\mu(s'_i, i) = o \iff \\
\exists s \in S \cdot s_1 \downarrow \text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \\
\text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \mu(s_1, i) = o
\]

• or \( i \in I_d \), then the following condition should hold:

\[
\forall m \in M_d, s_0 \in O' \cdot i = (Id_m, \text{val}) \Rightarrow \\
\mu(s'_i, i) = o \iff \\
\exists s \in S \cdot s_1 \downarrow \text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \\
\text{UsedVars}_m = s'_1 \downarrow \text{UsedVars}_m \land \mu(s_1, i) = o
\]

In the definition of transition function, a case distinction is made based on whether the method calls (in the delta model) have a nested method call or not. In the former case the valuations of the variables belonging to both core and delta models can change in the target state while in the latter case only the valuation of the variables belonging to the delta model can change. In the definition of output function these two cases are defined as one since the effect of the output of the inner method calls, if any, of a method call in the delta model is captured by the corresponding model variables which are included in the states of the delta model.

Fig. 4. (a) demonstrates the FSM resulting from the addition of the controller class to the bridge class. Note that the \( \gamma \) function is defined to match the valuation of the model variable \( x_{\text{CheckedAPI}} \) in the delta with the variable \( \text{Avl} \) in the core.

**Theorem 1.** Based on the assumptions made about the core model and the delta model, the resulting FSM of Definition 10 satisfies the assumptions (1)-(4) of Definition 2.

Note that the last constraint of Definition 10 is implementation-dependent and hence, it can only prove without sufficient assumptions on the implementation. This is out of the scope of the present paper.
Adding Fields and Methods

In this section, we discuss the effect of adding a set of fields and methods to the core module.

Let $X$ and $E$, respectively, denote the set of fields and methods added by a delta. Also, assume that $V$ denotes the variables in the domain of the states in the core model, and that a method can comprise method calls. The addition of $X$ and $E$ to the core FSM results in another FSM in which the abstract states and transitions accommodate $X$ and $E$. The formal definition of the application function has a similar structure to the case of adding a class.

**Theorem 2.** Assumptions (1)-(4) of Definition 2 are preserved under the addition a set of fields and methods to a core FSM model.

As an example, Fig. 4. (b) demonstrates the FSM resulting from the addition of the delta $D_{PedLight}$ to the core model. This delta adds a new field, namely, $Psig$, and two methods, namely, $SetPsig$ and $ResetPsig$, to the class $Bridge$.

6 Delta-Oriented Testing

In this section, we explain the incremental test-case generation method. In the remainder of this section, we assume that the core model is an object FSM such as $M(O) = (S, s_0, I, O, µ, λ)$ and the set of all methods of the classes in this core model are denoted by $MTD$. The state cover set and the separating family of sequences computed for $M$ are, respectively, denoted by $Q$ and $Z$. We assume that $q_s ∈ Q$ denotes a sequence in the state cover set that ends in state $s$ and $z_s$ denotes the set of sequences which separate $s$ from other states. For example,
the state cover set and the separating family of sequences for the core model represented in Fig. 3 are, respectively, $Q = \{\varepsilon, \text{ResetAvl}()\}$ and $Z = \{z_0, z_1\} = \{{\text{CheckAvl}}\}, \{{\text{CheckAvl}}\}$.

6.1 Test-Case Generation for Class Addition

Let $M_d(Q_d) = (S_d, s_0^d, I_d, O_d, \mu_d, \lambda_d)\text{ be the FSM that is composed with core model, with regards to the composition function } \gamma, \text{ as a result of adding the new class to the core module. We assume that the state cover set and the family of separating sequences for this FSM are, respectively, denoted by } Q_d \text{ and } Z_d. \text{ The resulting object FSM is } M'(O') = (S', s_0', I', O', \mu', \lambda'), \text{ as defined in Sect. 5.2, and the set of test cases for this FSM are computed as follows.}

In order to compute the new state cover set, denoted by $Q'$, we need to build the spanning tree of $M'$. Assuming that $P_0(S_d, E_d)$ is the spanning tree built for $M_d$, where $S_d$ denotes the set of vertices and $E_d \subseteq S_d \times I_d \times S_d$, denotes the set of edges in this tree, and $P(S, E)$ is the spanning tree built for $M$, where $S$ and $E \subseteq S \times I \times S$, are, respectively, the set of vertices and edges in this tree. Moreover, we assume that $V$ and $V_d$, respectively, denote the set of variables included in $S$ and $S_d$. The spanning tree for $M'$, denoted by $P'(S', E')$, where $E' \subseteq S' \times I' \times S'$, is built using $P$ and $P_d$ as follows. Note that each state $s' \in S'$ can be represented by $(s, s_d)$, where $s \in S$ and $s_d \in S_d$, that is $s' \downarrow V = s \downarrow V$ and $s' \downarrow V_d = s_d \downarrow V_d$. Starting from the root of the tree, the $(s_0, s_0^d)$, for each state such as $(s, s_d)$, we add the following child nodes:

1. $(s', s_d)$, where for some $i \in I$, we have $(s, i, s') \in E$
2. $(s, s_d^i)$, where for some $i \in I'$ which is corresponding to a method call that does not contain any nested method calls, we have $(s_d, i, s_d^i) \in E_d$
3. $(s', s_d^j)$, where for some $i \in I'$ that contains a method call denoted by $j \in I$, we have $(s_d, i, s_d^j) \in E$ and $\mu(s, j) = s'$.

Assuming that $|S| = n, |S_d| = m$ and $|S'| = n'$, then the worst-case complexity of computing the spanning tree is $O(n'(m + n))$. The state cover set is computed by traversing the resulting spanning tree.

The family of separating sequences $Z'$ is defined as $\bigcup_{s' \in S'} \{z_{s'}\}$, where for each state $s' = (s, s_d) \in S'$, we have that $z_{s'} = z_s \cup z_{s_d}$.

For example, the state cover set and the family of separating sequences for the FSM corresponding to the controller class in Fig. 4. (b) are as follows: $Q_4 = \{\varepsilon, \text{GetReq}(\theta), \text{GetReq}(T)\}, Z_4 = \bigcup_{s \in \text{core}} (z_s = \{\text{CheckLsig}, \text{CheckRsig}\})$.

Hence, the state cover set and the family of separating sequences for the FSM resulted adding the class are: $Q' = Q = \{\varepsilon, \text{ResetAvl}()\}, Z' = \bigcup_{s \in \text{core}} (z_s = \{\text{CheckAvl}, \text{CheckLsig}, \text{CheckRsig}\})$, respectively.

A special case of adding a class is when there are no nested method calls. In such a case the state cover set is equal to the state cover set of the core model that is $Q = Q'$. The computation of separating sequences remains intact with respect to the general case.
Complexity Analysis: The difference of complexity of the delta-oriented testing approach compared to the HSI method, in this case, is in the computation of the family of separating sequences. As explained above, in this case the delta-oriented approach obtains the family of the separating sequences for the new FSM, just using $Z_d$ and $Z$. Hence, defining $m = |S_d|$, and $q = |I_d|$, the complexity of computing $Z'$, using the delta-oriented approach, is $O(qm^2) + f_u$, where $f_u$ is the complexity of computing the union of two sets. Assuming that the delta has $n'$ states where $n' \leq m \cdot n$, and $p = |I'|$, the complexity of computing the family of separating sequences, using the HSI method, for this FSM is $O(pm^2)$. It should be noticed that this computation is done for each product in a product line separately, where the number of the products can increase exponentially in terms of the features. Practically, in a product line we have $m \ll n$, hence $O(qm^2) + f_u \ll O(pm^2)$. In other words, there can be a substantial gain in calculating the separating sequences using the delta-oriented approach.

7 Empirical Results

In order to check the efficiency of the proposed algorithm, we applied our method to a software system from the health-care domain. In order not to reveal the structure of the commercial system, we dispense with the details that are not necessary for understanding the experimental results. The core logic of this system includes six classes and its main functionality is to detect devices in the surroundings and control users’ access to them. Each user can create and complete a set of tasks after accessing a device. We considered the proportion of time required to generate test cases for 4 different models in two cases: using the delta-oriented approach, and using the plain monolithic HSI method. (In this work, we only consider the reduction in the test-case-generation time; we leave the study of the test-case-execution time as future work).

In order to compute the test-case generation time, we performed the algorithms in both methods in a step-by-step manner and manually, while counting the basic computation steps in these algorithms. Because these basic steps are common to both methods and consume a constant amount of time, we could hence come up with a precise comparison of the time required for test-case generation.

First, we considered a core FSM with 11 abstract states and 74 transitions. This core model included a set of objects, which model a group of users, devices and tasks created by users. Then, we applied a delta which comprised the addition of a method to a class in order to enable modification of a field in the core model. The result of applying this delta is another FSM with the same number of states and 85 transitions. Using the delta-oriented approach for generating test cases resulted in a 50-percent reduction in test-case generation time. This difference is due to that the spanning tree and the family of separating sequences are computed anew in the HSI method, while the delta oriented approach reuses the sequences computed for the core model.
We also applied a delta concerning the addition of an object of a task to the core model which resulted in another FSM with 16 states and 89 transitions. In this case, applying the delta-oriented approach resulted in a 40-percent reduction in test-case generation time. (For more detailed data, we refer to Fig. 5.)

Subsequently, we considered another core model including 21 abstract states and 167 transitions. We applied a delta comprising the addition of the same method as above to the core model, which resulted in the same number of states and 188 transitions. Applying the delta-oriented approach results in a 50-percent reduction in the test-case generation time.

The last delta in this software product line comprised the addition of an object of a device to the last core model, with 37 states and 215 transitions. The reduction in the test-case generation time in this latter case is 30 percent.

The results show that in cases that we can reuse the separating sequences and the state cover set of the core model, such as the addition of a set of methods that do not change the number of states, the delta-oriented approach can be very efficient. The above-mentioned results are summarized in Fig. 5.

8 Conclusions and Future Work

In this paper, we introduced test models and test-case generation methods for delta-oriented FSM-based testing, based on the DeltaJava syntax. Our test-case generation method is a lifting of the incremental test-case generation for the HSI method, using a higher level of abstraction suitable for our DeltaJava-based models. We showed, both using complexity analysis and by application to a case study, that the delta-oriented approach can increase the efficiency of test-case generation.

We are studying realistic, yet more relaxed fault models (than those underlying the HSI method). Such a fault model can capture the possible mutual effects of different behavior in deltas and core. Then, we will identify parts of test cases that need not be re-executed and also independent pieces of behavior that can be reduced, e.g., using partial-order reduction [16]. Moreover, we intend to extend our approach to the full syntax of DeltaJava and in particular, consider modifying and removing methods, building upon the results of [9, 11]. Finally, we plan to implement our approach in a programming environment and organize more extensive experiments with our industrial partner.

<table>
<thead>
<tr>
<th>Core Model</th>
<th>After Applying Delta</th>
<th>HSI Test Case Generation steps</th>
<th>Delta-Oriented Test Case Generation Steps</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>74</td>
<td>11</td>
<td>85</td>
<td>194</td>
</tr>
<tr>
<td>11</td>
<td>74</td>
<td>16</td>
<td>89</td>
<td>213</td>
</tr>
<tr>
<td>21</td>
<td>187</td>
<td>21</td>
<td>188</td>
<td>419</td>
</tr>
<tr>
<td>21</td>
<td>187</td>
<td>37</td>
<td>215</td>
<td>731</td>
</tr>
</tbody>
</table>

Fig. 5. Results obtained from test-case generation for the case study
References

Appendix B

Paper II
Basic Behavioral Models for Software Product Lines: Expressiveness and Testing Equivalences

Harsh Beohar, Mahsa Varshosaz and Mohammad Reza Mousavi

2016 Science of Computer Programming Journal
Basic behavioral models for software product lines: Expressiveness and testing pre-orders

Harsh Beohar, Mahsa Varshosaz, Mohammad Reza Mousavi * ,1

Centre for Research on Embedded Systems (CERES), School of IT, Halmstad University, Sweden

ARTICLE INFO

Article history:
Received 10 October 2014
Received in revised form 11 June 2015
Accepted 12 June 2015
Available online 9 July 2015

Keywords:
Software product lines
Formal specification
Behavioral specification
Labeled transition systems
Featured transition systems
Modal transition systems
Calculus of communicating systems (CCS)
Product line CCS (PL-CS)

ABSTRACT

In order to provide a rigorous foundation for Software Product Lines (SPLs), several fundamental approaches have been proposed to their formal behavioral modeling. In this paper, we provide a structured overview of those formalisms based on labeled transition systems and compare their expressiveness in terms of the set of products they can specify. Moreover, we define the notion of tests for each of these formalisms and show that our notions of testing precisely capture product derivation, i.e., all valid products will pass the set of test cases of the product line and each invalid product fails at least one test case of the product line.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

1.1. Motivation

Software product lines (SPLs) are becoming increasingly popular as efficient means for mass production and mass customization of software. Hence, establishing formal foundations for specification and verification of SPLs can benefit a large community and can have substantial impact. In the last few years, many researchers have spent substantial effort in extending various formalisms and their associated reasoning techniques to the SPL settings, of which [1–5] provide a comprehensive overview.

In this paper, we put some structure to the body of knowledge regarding some of the most fundamental extensions of behavioral models for SPLs, namely, those based on labeled transition systems, such as those proposed or studied in [6–19]. These basic models can serve as semantic models for extensions of higher level models such as domain specific languages (DSLs), or those based on the Unified Modeling Language (UML) state or sequence diagrams. Hence, bringing more structure into the body of knowledge about these fundamental computational models can help the language designers of higher level languages to make the right choice when defining the semantics of their language.

* Corresponding author.
E-mail addresses: harsh.beohar@hh.se (H. Beohar), mahsa.varshosaz@hh.se (M. Varshosaz), m.mousavi@hh.se (M.R. Mousavi).
1 The work of M.R. Mousavi has been partially supported by the Swedish Research Council (Vetenskapsrådet) award number: 621-2014-5057 (Effective Model-Based Testing of Concurrent Systems) and the Swedish Knowledge Foundation (Stiftelsen för Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAS H & G project with award number 20140312°.
The structure proposed in this paper is twofold: first we compare the expressive power of these fundamental models and second, we explore the extensional and intensional notions of testing equivalence (pre-orders) for each of them. Some of the expressiveness results reported in the present paper are hinted at in the literature, but to our knowledge, have never been formalized and proven before. Regarding the testing equivalences, the extensional notions of testing defined in this paper – for the models proposed in [14,15,17] – are novel. They are of course based on and slight extensions of well-known notions of tests for labeled transition systems (e.g., of [20–22] and particularly that of [23]).

1.2. Running example

In order to illustrate the different approaches, we use the following simple example originally due to Asirelli et al. [7] and further elaborated by Classen [2].

Example 1. We model a product line for vending machines, which accept one-Euro coins (€1) exclusively for the European market and one-Dollar coins (€1) exclusively for the American market. A user has a choice of adding sugar or no sugar, after which she is allowed to choose a beverage among coffee, tea, or cappuccino. Furthermore, the following constraints hold on each product:

1. Coffee must be offered by each and every variant of this product line.
2. Cappuccino is served only by the European machines and whenever cappuccino is served, a ring-tone must ring.
3. Tea is an optional feature for both markets.

1.3. Contributions

The objects of study in this paper are three popular models of computation that are used in the literature to model SPLs. Namely, we study modal transition systems [24], product line labeled transition systems [17], and featured transition systems [15]. The contributions of this paper are as follows:

- Firstly, we formally show that the class of modal transition systems are strictly less expressive than the class of product line labeled transition systems, which are in-turn strictly less expressive than the class of featured transition systems.
- Secondly, we show how the test expressions of Abramsky [23] can be used to characterize product derivation for each of the above models of software product line.

1.4. Paper structure

The rest of this paper is organized as follows. In Section 2, we present an overview of the product-line formalisms studied in this paper and recall or define their intuitive notion of derived products. In Section 3, we compare the expressiveness of formalisms by comparing their set of definable products. In Section 4, we define the extensional notions of test for the formalisms and prove that they coincide with their intensional counter-parts. The paper is concluded in Section 5 with a summary of the results and some directions of our ongoing research.

2. Fundamental behavioral models of SPLs

2.1. Overview

Conventional formal models such as labeled transition systems (LTSs) can be used to specify the behavior of systems at a high level of abstraction. Namely, LTSs specify how a system execution evolves on an abstract machine in terms of transitions that are labeled with the information that is received/made available through each execution step from/to the outside world. However, in order to formally specify a software product line, one needs specific (semantic) notions to refer to variation points, distinguish different features and refer to their possible interactions. Below, we give an overview of several alternatives proposed as fundamental behavioral models of software product lines.

As the first alternative, Fischbein et al. [6] argue that modal transition systems (MTSs) [24] are adequate extensions of LTSs to model a software product line: MTSs partition the transitions into may and must transitions and hence, each MTS has an associated set of possible implementations. Subsequently, several researchers [7–9,12,13] adopted modal transition system as a formal model to perform rigorous analysis of software product lines. Several pieces of work [7–9] addressed the issue of deriving valid products from a given MTS by model checking against formulae expressed in a deontic logic called Modal-Hennessy–Milner-Logic (MHML). Others [13,12] developed an interface theory and a testing theory for software product lines.

Classen et al. [14,15] took a different route by annotating LTSs with features of a feature diagram. Intuitively, a feature diagram specifies, by means of a graphical notation, the set of valid products by specifying constraints on the presence of features. The result of annotating LTSs with features is called a featured transition system (FTS). Furthermore, an LTL-model checking algorithm in the context of featured transition system was given in [15]. Cordy et al. [16] extended the earlier
work [14,15] by incorporating non-boolean features and multi-features in a high-level specification language called TVL*.

Also, an algorithm was given to construct an FTS from a behavioral specification written in TVL*.

As another alternative, Gruler et al. [17] extended Milner’s CCS [25] into a process calculus called PL-CCS. The extension involves introducing the “binary variant” \(\odot\) operator to represent the alternative features of a product line. Like MTSs, the validity of products is asserted by model checking formulae specified in a multi-valued modal mu-calculus, originally due to [26]. The semantics of the logic specifies, for each PL-CCS process, the set of configurations that satisfy the logical formula.

In addition, there are also other alternatives for specifying the behavior of SPLs that fall beyond the scope of this article. For example, there are proposals based on re-using existing process algebras with data (see, e.g., [27–29]) or extending existing Petri Nets with features (see, e.g., [30,31]). In order to perform a formal comparison of expressiveness, we confine ourselves to models that are based on LTSs (e.g., those models that specify a set of products captured by an LTS) and hence, the other approaches mentioned in this paragraph are not considered any further. Also, there are extensions of higher-level formalisms such as UML that are not considered in this paper, since they either lack formal semantics or their semantics can be expressed in the more fundamental formalisms such as those studied in this paper.

In Fig. 1, we summarize the different extensions of LTSs for the behavioral modeling of SPLs. In this figure, the solid arrows show the possibility of transforming a model from one formalism into another. In [12], the authors gave a semantics for a restrictive notion of FTS in terms of an MTS. One of the two main contributions of this paper is to complete this picture, and hence generalize the result of [12], by presenting (or showing the impossibility of) encodings among PL-CCS, MTSs and FTSs. This is represented by the dashed arrow in Fig. 1. In the remainder of this section, MTSs are surveyed in Section 2.2 and FTSs are reviewed in Section 2.3. In Section 2.4, we survey process-algebraic approaches to SPL specification.

### 2.2. Modal transition systems

#### 2.2.1. Specifying SPLs

Modal transition systems extend labeled transition systems by distinguishing two different sorts of transitions, namely, may and must transitions. May transitions, as their name suggests, may (or may not) be present in the implementation behavior, while must transitions are always present. As a sanity condition, it is required that all must transitions also have a corresponding may transition. The following definition formalizes these concepts.

**Definition 1.** A modal transition system (MTS) [24] is a quadruple \((P,A,\rightarrow,M)\), where \(P\) is a set of states or processes, \(A\) is a set of actions, \(\rightarrow\subseteq P \times A \times P\) is the so-called may transition relation, and \(\rightarrow\subseteq M\) is the so-called must transition relation.

An MTS can only describe the behavior of optional and mandatory features of a product line in terms of may and must transition relations, respectively. An MTS specifies a unique LTS when \(\rightarrow\subseteq M\), i.e., when all transitions are must transitions, and vice versa, an LTS can be interpreted as an MTS by interpreting ordinary transitions as must transitions. Throughout the rest of this section, we fix the letters \(P, P', Q, Q'\) to denote the states of an MTS, whereas, \(p, p', q, q'\) are used to denote the states of an LTS.

**Example 2.** Consider the informal description given in Example 1. The MTS shown in Fig. 2(a) (due to [7]), formally specifies this product line. In this MTS, solid arrows denote must transitions and dashed arrows denote may transitions. (For must transitions we dispense with drawing the corresponding may transitions and tacitly assume their presence.)

#### 2.2.2. Deriving products

A key notion within the theory of MTS is modal refinement [24], which allows for deriving products (MTSs with fewer may and more must transitions) from product lines, or testing conformance of products to product lines. Informally, if \(P\) refines \(Q\), then all must transitions of \(P\) are simulated by \(Q\), while all may transitions of \(Q\) are simulated by \(P\). Another intuition shared by modal refinement relation is that some of the may transitions of \(P\) can be either transformed into must transitions of \(Q\) or blocked by \(Q\), whenever \(P\) refines \(Q\).
Definition 2. A binary relation $R \subseteq P \times P$ is a modal refinement [24] relation if and only if the following transfer properties are satisfied.

1. $\forall P, Q, Q' \in P, Q \subseteq P \Rightarrow \exists Q' \in P, Q \subseteq Q' \subseteq P$.
2. $\forall P, Q, Q' \in P, Q \subseteq P \Rightarrow \exists Q' \in P, Q \subseteq Q' \subseteq P$.

A modal specification $P$ refines a modal specification $Q$, denoted $P \leq Q$, if there exists a modal refinement relation $R$ such that $P \subseteq Q$. The set of products implementing a modal specification $P$ is denoted as $P = \{P \mid P \leq P\}$.

Example 3. Consider the product line MTS specified in Example 2. The LTS shown in Fig. 2(b) specifies a product which serves only coffee and is customized for the American market. It is not hard to see that the LTS shown in Fig. 2(b) refines the MTS of Fig. 2(a) by transforming certain may transitions into must transitions, or prohibiting them.

2.3. Featured transition systems

2.3.1. Specifying structural aspects

In [14], the authors pointed out that the derived products from an MTS may be invalid (and counter-intuitive) due to the inherent lack of expressiveness in MTSs for specifying feature constraints. In common practice, feature diagrams [32] have been used to model such constraints using a graphical notation. A feature diagram represents all the products of an SPL in terms of features that are arranged hierarchically. Usually, feature diagrams are represented by a directed acyclic graph, of which each node is a feature. There are different kinds of edges between a parent (feature) and its children (sub-features), namely, the ones representing the mandatory sub-features, and the others representing the optional sub-features. Furthermore, a feature diagram also specifies three additional constraints over features that may span over different levels of abstraction:

1. Alternative relationship, i.e., the designated sub-features can never be simultaneously present in any product.
2. Exclude relationship, i.e., different features at different levels of hierarchy can never be simultaneously present in any product.
3. Require relationship, i.e., if a feature is present in a product, the related feature should also be present in the same product.

For more information and a formal treatment of the syntax and the semantics of feature diagrams, we refer to [32].

Example 4. Consider the feature diagram shown in Fig. 3, which formalizes the features and feature constraints of Example 1 [7]. In this diagram every machine must consists of the features coin (o), and beverage (b) and may comprise an
optional feature ring-tone ($r$). The coin feature is further decomposed into two alternative features euro ($e$) and dollar ($d$).

Furthermore, Fig. 3 also specifies that cappuccino ($p$) requires ring-tone ($r$) denoted by a uni-directional dashed line and cappuccino is absent in the machine that takes dollars represented by bidirectional dashed line.

A feature diagram only specifies the structural aspects of variability in an SPL. To formally analyze the behavior of an SPL in [15], the transitions of a labeled transition system are annotated with logical constraints on the presence or absence of features; the features used in such logical constraints are assumed to be already specified in a feature diagram.

Let $\mathbb{B} = \{\top, \bot\}$ be the set of Boolean constants and let $\mathbb{B}(F)$ be the set of all propositional formulae generated by interpreting the elements of the set $F$ as propositional variables. For instance, in the context of Example 4, the formula $e \land \neg d$ asserts the presence of euro coin and the absence of dollar coin payment features. We let $\phi, \phi'$ range over the set $\mathbb{B}(F)$.

**Definition 3.** A featured transition system (FTS) is a quintuple $(P, A, F, \rightarrow, \Lambda)$, where

1. $P$ is the set of states,
2. $A$ is the set of actions,
3. $F$ is a set of features,
4. $\rightarrow \subseteq S \times A \times \mathbb{B}(F) \times S$ is the transition relation satisfying the following condition:
   $$\forall P, a, P', \phi, \phi' \quad ((P, a, P') \in \rightarrow \land (P, a, \phi', P') \in \rightarrow) \implies \phi = \phi',$$
5. $\Lambda \subseteq \{\lambda : F \rightarrow \mathbb{B}\}$ is a set of product configurations.

Just like in the case of MTSs, we reserve the symbols $P, P', Q, Q'$ to denote the states of an FTSs. Furthermore, we write $P \xrightarrow{\Delta} Q'$ to denote an element $(P, a, \phi, Q') \in \rightarrow$.

**Example 5.** Consider the MTS in Fig. 2(a); we obtain an FTS by discarding the distinction between may and must transitions and instead, annotating every transition with a formula over features given in Fig. 3. In the following table, we give the propositional formula associated with every transition of the vending machine example.

<table>
<thead>
<tr>
<th>Transitions</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 \xrightarrow{td} s_2$</td>
<td>$e$</td>
</tr>
<tr>
<td>$s_1 \xrightarrow{td} s_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>$s_2 \xrightarrow{coffee} s_3$</td>
<td>$c$</td>
</tr>
<tr>
<td>$s_2 \xrightarrow{tea} s_6$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transitions</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2 \xrightarrow{\text{cappuccino}} s_7$</td>
<td>$p$</td>
</tr>
<tr>
<td>$s_{12} \xrightarrow{\text{ring a tone}} s_{13}$</td>
<td>$p \Rightarrow r$</td>
</tr>
<tr>
<td>Remaining transitions</td>
<td>$m$</td>
</tr>
</tbody>
</table>

Lastly, the set of product configurations of the vending machine is the following set of 10 products specified by the feature diagram of Example 4 [9]:

$$\Lambda = \{[m, o, b, c, e], [m, o, b, c, e, r], [m, o, b, c, e, t], [m, o, b, c, e, t, r], [m, o, b, c, e, r], [m, o, b, c, d], [m, o, b, c, d, r], [m, o, b, c, d, t], [m, o, b, c, t, r], [m, o, b, c, e, p, r, t]\}.$$
Definition 4. Given a feature specification $P$ (i.e., a state in an FTS), a set of selected features $\lambda \in \Lambda$ induces a state $\Delta_\lambda(P)$ in an LTS defined by the following operational rule:

$$\lambda \vdash \phi \quad P \xrightarrow{a\phi} Q \quad \Delta_\lambda(P) \xrightarrow{a} \Delta_\lambda(Q).$$

It was argued in [14,15] that FTSs are better suited to model a software product line than a modal transition system. The crucial difference between the two is that all the may transitions in an MTS are independently optional, while in an FTS, one can make a finer distinction among them by annotating them with more complex boolean formulae pertaining to different types of feature constraints. In other words, the choice among transitions in an FTS depends on the product configuration, whereas the choice among may transitions in an MTS is nondeterministic [15].

Example 6. Consider the FTS given in Example 5 and the LTS given in Fig. 2(b) with the addition of transition $1 \xrightarrow{\Delta_0} 2$. The latter is not a valid product of the former and cannot be derived from Definition 4. Note that there exists no valid set of features or product configuration $\lambda \in \Lambda$ such that $\lambda(e) = \lambda(d) = \top$. This is due to the semantics of the feature diagram depicted in Fig. 3, which specifies that e, d are alternative features. As a result, the transition relation defined in Definition 4 cannot have a choice between the actions $1e, 1d$.

Although the above example suggests that the class of FTSs is expressive enough to specify the different inter-feature relationships, the notion of deriving valid products (Definition 4) by an LTS is syntactical in nature (e.g., compared to the notion of deriving valid products by an MTS). This syntax-driven notion of valid product derivation is too rigid for any semantic analysis such as testing. In particular, we note that Definition 4 is not even closed under strong bisimulation (see [33] for a formal definition). Next, we present a notion of deriving valid products from an LTS which generalizes Definition 4.

Definition 5. Given an FTS $(P, A, F, \rightarrow, \Lambda)$, an LTS $(P, A, \rightarrow)$, and a product feature $\lambda \in \Lambda$. A family of binary relations $R_\lambda \subseteq P \times P$ (parameterized by product configurations) are called product-derivation relations if and only if the following transfer properties are satisfied.

1. $\forall_{P, Q, p, q} (P R_\lambda p \wedge P \xrightarrow{a\phi} Q \wedge \lambda \vdash \phi) \Rightarrow \exists_{q} P \xrightarrow{a} q \wedge Q R_\lambda q$;
2. $\forall_{P, Q, p, q} (P R_\lambda p \wedge p \xrightarrow{\pm\phi} q) \Rightarrow \exists_{q} P \xrightarrow{\pm\phi} Q \wedge \lambda \vdash \phi \wedge Q R_\lambda q$.

A state $p \in P$ in an LTS derives the product feature $\lambda$ from an FTS-specification $P \in P$, denoted by $P \vdash_\lambda p$, if there exists an $R_\lambda$ product-derivation relation such that $P R_\lambda p$.

We end this section on FTSs by highlighting two intuitive properties of the product derivation relation.

Lemma 1. For any given feature specification $P$ and the derived product $\Delta_\lambda(P)$, we have $P \vdash_\lambda \Delta_\lambda(P)$.

Lemma 2. Given any feature specification $P$ and a derived product $p$ with $P \vdash_\lambda p$, for some product configuration $\lambda$. If $q$ is strongly bisimilar (in the sense of Park [33]) to $p$, then $P \vdash_\lambda q$.

2.4. Product line process algebra

Gulak et al. [17] extended Milner’s Calculus of Communicating Systems (CCS) [25] into PL-CCS by introducing the “binary variant” operator $\otimes$ to represent the alternative relationship in feature diagrams.

Definition 6. Let $A = \Sigma \cup \Sigma^* \cup \{\epsilon\}$ be the alphabet, where $\Sigma = \{\bar{a} | a \in \Sigma\}$. The syntax of PL-CCS terms $e$ is defined by the grammar $\text{Nil} | a.e | e + e' | e @ e' | e e' | e f | \epsilon \Lambda$, where $\text{Nil}$ is the deadlocking process, for each $a \in A$, $\alpha_a$ denotes action prefixing, $\alpha_+$ denotes non-deterministic choice, $\alpha_\otimes$ denotes binary variant, $\alpha_\parallel$ denotes parallel composition, for each $f : A \rightarrow A$, $\lambda[f]$ denotes renaming by means of $f$, and for each $L \subseteq A, \lambda \notin L$ denotes the restriction operator (blocking coactions in $L$). In addition, one may define recursive processes by means of process identifiers and equations.

At the first sight, the variant operator $\otimes$ is reminiscent of the ordinary alternative composition operator $+$ from CCS; however, they are substantially different, as the binary variant operator remembers the chosen alternative. For example, consider process terms $s = a.b.s + c.s$ and $t = a.(b.t @ c.t)$. Intuitively, the recursive process $s$ keeps on making a choice between $b$ and $c$ upon performing $a$; whereas in $t$ the choice is made at the first iteration after performing the action $a$ and it is recorded for and respected in all future iterations, i.e., the process behaves deterministically once the choice between “features” $b$ and $c$ is made once and for all.

As syntactic sugar, a unary operator $(\cdot)$, called the optional operator, was also introduced to represent the optional features of a feature diagram. It can be defined in PL-CCS as $(P) = P \otimes \text{Nil}$.
Example 7. The following process definition specifies the vending machine product line in PL-CCS.

\[
\begin{align*}
S_1 &= \text{1e.s2} \uplus \text{1d.s2} \\
S_2 &= \text{cappuccino.s10} + \text{coffee.s10} \\
S_3 &= \text{coffee.s3} + (\text{tea.s6} + \text{cappuccino.s7}) \\
S_4 &= \text{coffee.s10} + (\text{tea.s9} + \text{cappuccino.s9}) \\
S_5 &= \text{pour sugar.s10} \\
S_6 &= \text{pour sugar.s9} \\
S_7 &= \text{pour sugar.s8}
\end{align*}
\]

The semantics of a PL-CCS term is defined in terms of product line labeled transition systems [17], recalled below, using a structural operational semantics. Roughly, the states and the transition relations of a product line labeled transition system are enriched with configuration vectors (i.e., functions of type \([L, R, ?]^I\) with \(I\) being an index set) that records the selection made in past about the alternative features.

Definition 7. Let \((L, R, ?)^I\) denote the set of all total functions from an index set \(I\) to the set \((L, R, ?)\). A product line labeled transition system (PL-LTS) is a quadruple \((\mathbb{P}, \times (L, R, ?)^I, A, I, \rightarrow)\) consisting of a set of states \(\mathbb{P} \times (L, R, ?)^I\), a set of actions \(A\), and a transition relation \(\rightarrow \subseteq (\mathbb{P} \times (L, R, ?)^I) \times (A \times (L, R, ?)^I) \times (\mathbb{P} \times (L, R, ?)^I)\) satisfying the following restrictions:

1. \(\forall P, v, Q, v' (P, v) \xrightarrow{a, i} (Q, v') \implies v' = v_{\nu}\).
2. \(\forall P, v, Q, v', i (P, v) \xrightarrow{a, i} (Q, v') \wedge v' \neq v \implies \exists i \left( v'(i) \neq v(i) \land \forall_{j \neq i} v'(j) = v(j) \right)\).
3. \(\forall P, v, Q, v', i (P, v) \xrightarrow{a, i} (Q, v') \land (v' \neq v) \implies v'(i) = v(i)\).

Notice that, as a consequence of item 2 in Definition 7, for any transition \((P, v) \xrightarrow{a, i} (Q, v')\), if \(v' \neq v\), we can find a unique \(i \in I\) such that \(v'(i) \neq v(i) \land \forall_{j \neq i} v'(j) = v(j)\). Furthermore, it should also be noted that Conditions 1, 2, and 3 follow from the operational rules given by Gruler et al. to a PL-CCS term in [17].

Now in order to define when an LTS is a valid product of a given PL-LTS, we need the following notion of ordering on configurations and configuration vectors.

Definition 8. The ordering relation \(\sqsubseteq\) on the set \((L, R, ?)\) is defined in the following way:

\[
\sqsubseteq = (\{?, \?\}, (L, L), (R, R), (\?, L), (\?, R)).
\]

We lift this ordering relation to the level of configuration vectors by letting \(v \sqsubseteq v' \iff \forall_{i \in I} v(i) \sqsubseteq v'(i)\), for any \(v, v' \in (L, R, ?)^I\).

We end this section on PL-CCS by proposing a definition of product-derivation relations in a similar vein to Definition 5.

Definition 9. Let \((\mathbb{P}, \times (L, R, ?)^I, A, \rightarrow)\) be a PL-LTS and let \((P, A, \rightarrow)\) be an LTS. A family of binary relations \(\mathcal{R}_\theta \subseteq (\mathbb{P}, \times (L, R, ?)^I) \times \mathbb{P}\) (parameterized by every product configuration \(\theta \in (L, R, ?)^I\)) is a family of product-derivation relations if and only if the following transfer properties are satisfied:

1. \(\forall P, Q, v, v', p ((P, v) \mathcal{R}_{\theta_p} (Q, v') \wedge v' \sqsubseteq \theta) \implies \exists q \mathcal{R}_{\theta_q} (Q, v')\).
2. \(\forall P, Q, v, v', q ((P, v) \mathcal{R}_{\theta_p} (Q, v') \wedge v' \sqsubseteq \theta) \implies \exists q \mathcal{R}_{\theta_q} (Q, v')\).

A state \(p \in \mathbb{P}\) in an LTS is (the initial state of) a product of a PL-LTS \((P, v)\) with respect to a configuration vector \(\theta\), denoted by \((P, v) \mathcal{R}_\theta p\), if \(v \sqsubseteq \theta\) and there exists an \(\mathcal{R}_\theta\) product-derivation relation such that \(P \mathcal{R}_\theta p\).

3. Expressiveness results

The goal of this section is to formally compare the expressiveness of the three product line formalisms as outlined in the previous section. Before we do so, let us bring all the three formalisms under one single definition of a product line structure. Intuitively, a product line structure consists of a product line specification and a semantic function \(\|\) that maps a specification into a set of implementations (valid products) modeled as LTSs.

Definition 10. Let \((P, A, \rightarrow)\) be an LTS. A product line structure is a tuple \(\mathcal{M} = (\mathcal{M}, \| \), \ where \(\mathcal{M}\) is the class of all intended product line models or specifications (in our case: MTSSs, FTSs, and PL-LTSs) and \(\| : \mathcal{M} \to 2^\mathbb{P}\) is the semantic function mapping a product line specification to a set of LTSs.
Definition 11. An encoding \( E : \mathcal{M} \rightarrow \mathcal{M}' \) from a product line structure \( \mathcal{M} = (\mathcal{M}^1, \mathcal{M}^2) \) into a product line structure \( \mathcal{M}' = (\mathcal{M}'^1, \mathcal{M}'^2) \) is a function \( E : \mathcal{M} \rightarrow \mathcal{M}' \) satisfying the correctness criterion \( [\not\in = E \circ [\not\in ] ] \).

We say that the product line structure \( \mathcal{M}' \) is at-least as expressive as \( \mathcal{M} \) if and only if there exists an encoding \( E : \mathcal{M} \rightarrow \mathcal{M}' \).

Furthermore, we say that the product line structure \( \mathcal{M}' \) is less expressive than \( \mathcal{M} \), if and only if \( \mathcal{M} \) is at-least as expressive as \( \mathcal{M}' \), and \( \mathcal{M}' \) is not at-least as expressive as \( \mathcal{M} \), i.e., there does not exist any encoding \( E : \mathcal{M} \rightarrow \mathcal{M}' \).

In the remainder of this section, we explore the expressiveness among the classes of MTSs, FTSs, and PL-LTSs. In order to do this, we start with relating the two less expressive models, i.e., MTSs and PL-LTSs, to each other, and then move up in the lattice of expressiveness. Note that the “at-least as expressive” relation is transitive (by the composition of the encoding functions) and hence, we can use the transitivity to relate the least (i.e., MTSs) and the greatest (i.e., FTSs) points in the lattice, once we relate the middle-point (i.e., PL-LTSs) to each of them.

The following two theorems relate the expressiveness of PL-LTSs and MTSs.

**Theorem 1.** The class of PL-LTSs is at-least as expressive as the class of MTSs.

**Proof.** Consider the MTS \( (P, A, \rightarrow, \rightarrow, \emptyset) \) and some \( P \in P \). Let \( \rightarrow \subseteq P \times A^* \times P \) be the reachability relation defined as follows:

\[
P \xrightarrow{p} P \xrightarrow{p'} P 
\]

Let \( \text{tr}(P) \) be the set of traces generated by \( P \), i.e., \( \text{tr}(P) = \{ p \mid P \xrightarrow{p} Q \} \). For state \( P \), we define a family of transition relations parameterized by the traces of \( P \) as follows:

\[
\frac{Q \xrightarrow{\sigma, Q \rightarrow Q'} s \in \text{tr}(P) \quad \nu \xrightarrow{\nu} s \in \text{tr}(P) \quad (Q, a, Q') \rightarrow R}{(Q, \nu) \xrightarrow{\theta} (Q') \nu'}
\]

where \( Q \xrightarrow{\sigma, Q \rightarrow Q'} s \in \text{tr}(P) \) and the expression \( \nu \xrightarrow{\sigma, Q \rightarrow Q'} (Q, a, Q') \rightarrow R \) for \( X \in \{ L, R \} \) is defined in the following way:

\[
v'(s)(Q) = \begin{cases} X & \text{if } s = s' \land Q = Q' \land a = a \land Q'' = Q'' \\ \text{otherwise} \end{cases}
\]

We fix \( E(P) = (P, \nu_0) \), where \( \nu_0(s)(Q) = \{ s' \} \) for \( s, s' \in \text{tr}(P) \). Furthermore, we let the symbols \( \theta, \nu' \) range over the total configuration vectors, i.e., the functions of type \( \text{tr}(P) \rightarrow \bigcup_{s \in \text{tr}(P)} (L, R) \). Now we are in the position to show that \( [P] = [E(P)] \), where \( [P] = \{ p \mid P \xrightarrow{p} p \} \) and \( [E(P)] = \{ p \mid E(P) \xrightarrow{p} p \} \). We divide the proof into two obligations: \( [E(P)] \subseteq [P] \) and \( [P] \subseteq [E(P)] \), which we prove below.

**Proof of \( [E(P)] \subseteq [P] \).** Let \( P \) be a state in an LTS such that \( P \xrightarrow{p} p \), for some \( \theta \). Define a relation \( Q \xrightarrow{R} R' \) if \( (Q, a, Q') \rightarrow R \) for \( a \in (L, R) \) is defined in the following way:

1. Let \( Q \xrightarrow{p} Q' \) and \( Q \xrightarrow{R} \). Then, \( (Q, a, Q') \rightarrow \theta(a)(Q, a, Q') \). Note that \( \theta(a)(Q, a, Q') \) can be either \( L \) or \( R \). Then, we find \( v'' \xrightarrow{\theta} \). Now from the transfer property of \( \nu \), we find a \( q' \) such that \( q = q' \land (Q, v') \rightarrow \). Hence, \( Q \xrightarrow{R} \).

2. Let \( \nu \xrightarrow{\theta} \) and \( Q \xrightarrow{R} \). Trivial.

**Proof of \( [P] \subseteq [E(P)] \).** Let \( P \) be some state in an LTS such that \( P \xrightarrow{p} P \). Let \( \text{tr}(P) \) denote the set of maximal traces from the state \( P \). (Note that a maximal trace is either an infinite trace or a finite trace that leads to a deadlock state.) For any maximal trace \( s \in \text{tr}(P) \), we know that there is a unique execution \( e' \) starting from \( [p] \), such that dom(\( e' \)) = \( s \), where \( [p] \) is the transition system modulo strong bisimulation defined in the standard way.

\[\text{Definition: An execution } e' \text{ starting from } [p], \text{ is a function } e' : [s']^* \rightarrow 2^P \text{ for any } s \in \text{tr}(P) \text{ such that } e(s) = [p], \text{ and } e(s') \rightarrow e(s') = q \iff q \in e(s') \land q' \in e(s') \land q \rightarrow q', \text{ for every } s' \leq s. \text{ Here } s \text{ is the prefix relation between any two words.}\]
Theorem

Therefore, transfer at Theorem 1

Next, we define a relation $R_{\theta_p}$ as follows:

Next, we show that $R_{\theta_p}$ is a product derivation relation.

1. Let $(Q, v) \overset{\Delta}{\rightarrow} (Q', v')$, $v' \subseteq \theta_p$, and $(Q, v)R_{\theta_p}q$. Then, from the construction of $R_{\theta_p}$ we have $v \subseteq \theta_p \land s \in tr_m(p) \land P \overset{\Delta}{\rightarrow} Q \land Q \subseteq e_t(s) \land q \in e_t(s')$, for some $s \in tr_m(p)$, $s' \subseteq s$.
   (a) Let $Q \overset{\Delta}{\rightarrow} q'$. Then, $Q \leq q$ (because $Q \leq e_t(s') \land q \in e_t(s')$) and from the transfer property of modal refinement we find $q \Rightarrow q' \land Q' \leq q'$, for some $Q'$. Thus, there is a maximal trace $s \in tr_m(p)$ such that $s \alpha \subseteq s$, $q \in e_t(s')$, and $q' \in e_t(s')$. Hence, due to the construction of $R_{\theta_p}$, we get $(Q', v')R_{\theta_p}q'$.
   (b) Let $Q \overset{\Delta}{\rightarrow} (Q, q')$. Then, $v' \models v \models [[(Q, a, Q')] \rightarrow L)$. But $v' \subseteq \theta_p$, so $\theta_p(s)(Q, \overset{\Delta}{\rightarrow} Q') = L$. Thus, by the definition of $\theta_p$, we find $q' \Rightarrow q' \land Q' \leq q'$, for some $Q'$. Furthermore, since $q$ is reachable from $p$ and $q \overset{\Delta}{\rightarrow} q'$, there is a maximal trace $s \in tr_m(p)$ such that $e_t(s') = q$ and $e_t(s') = q'$ (for some $s' \subseteq s$). So let $v' \models v \models (Q, q') \rightarrow L$ and thus $v' \subseteq \theta_p$. Moreover, we find $(Q, v) \overset{\Delta}{\rightarrow} (Q', v')$. Thus, by the construction of $R_{\theta_p}$, we have $(Q', v')R_{\theta_p}q'$.

Theorem 2. The class of MTs is less expressive than the class of PL-LTs.

Proof. Due to Theorem 1, we know that PL-LTs is at least as expressive as MTs. It hence remains to prove that MTs is not at least as expressive as PL-LTs, which we show by means of the example depicted in Fig. 4.

We prove by contradiction that the PL-LTS depicted in Fig. 4 (left), where $P = a.Nil \| b.Nil$ cannot be encoded using any sound encoding (satisfying Definition 11) to an MT. To show this, observe the transition systems of the derived LTSs $p$ and $q$ drawn in Fig. 4 under $\theta = L$ and $\theta' = K$.

Suppose there is an encoding $E$ satisfying Definition 11. Clearly, $(P, 7) \Rightarrow p$ and $(P, 7) \Rightarrow q$. Then by correctness of $E$ we have $E((P, 7) \Rightarrow p)$ and $E((P, 7) \Rightarrow q)$, and $E((P, 7) \Rightarrow p)$ and $E((P, 7) \Rightarrow q)$, respectively. So let $r$ be a state $r$ of the following form: $r \overset{a}{\rightarrow} r'$, and $r \overset{a}{\rightarrow} r''$ (for some $r', r''$) such that $E((P, 7) \Rightarrow r)$. And by correctness of $E$ we get $(P, 7) \Rightarrow p$ or $(P, 7) \Rightarrow q$. However, $(P, 7) \not\Rightarrow p$ and $(P, 7) \not\Rightarrow q$.

Now that we have related the expressiveness of PL-LTSs and MTs, we move to the other end of the spectrum, namely to the comparison of PL-LTSs and FTSS, which is achieved by means of the following two theorems.

Theorem 3. The class of FTSSs is at-least as expressive as the class of PL-LTSs.

Proof. Let $(P \times \{L, R, \top\}, A, \rightarrow)$ be a PL-LTS. The corresponding FTSS is denoted by $(P \times \{L, R, \top\}, A, \rightarrow^t, \Lambda)$, where:

- $F = \bigcup_{i \in I} \{L_i, R_i\}$.
- $\Lambda = \bigcup_{i \in I} \{\Lambda_i \cup \{L_i, R_i\}\}$.
- The transition relation $\rightarrow^t$ is defined in the following way:
  
  \[
  (P, v) \overset{a}{\rightarrow^t} (Q, v) \quad \text{and} \quad (P, v) \overset{a}{\rightarrow^t} (Q, v') \quad \phi = v'((i)) \quad \Sigma(i, v, v')
  \]

Where $\Sigma(i, v, v') \iff v'((i)) \neq v((i)) \land \forall_{j \neq i} v((j)) = v((j))$. 

\[
\begin{align*}
(P, v) \overset{a}{\rightarrow^t} (Q, v) & \quad (P, v) \overset{a}{\rightarrow^t} (Q, v',) \\
(P, v) \overset{a}{\rightarrow^t} (Q, v) & \quad (P, v) \overset{a}{\rightarrow^t} (Q, v')
\end{align*}
\]
For any \((P, v) \in \mathcal{P} \times \{L, R\}^l\), we fix \(E(P, v) = (P, v)\). Let \([P, v] = \{p \mid \exists q \in \mathcal{L}(P, v) \cap p\}\) and \([P, v]' = \{p \mid \exists q \in \mathcal{L}(P, v) \cap p\}\). In the next step, we need to show that \([P, v] = [P, v]'\).

For any \((P, v) \in \mathcal{P} \times \{L, R\}^l\), we define \(\lambda_{i_0} \in \Lambda\) as follows:

\[
\lambda_{i_0}(L) = \top \iff \theta(i_0) = L \quad \text{and} \quad \lambda_{i_0}(R) = \bot \iff \theta(i_0) = R.
\]

Furthermore, consider the following relation \(R_{\lambda_{i_0}}\) such that

\[
(Q, v') \lambda_{i_0} q \iff (Q, v')_\rho q.
\]

It is straightforward to show that \(R_{\lambda_{i_0}}\) is a product derivation relation.

\[
([P, v] \subseteq [P, v]') \iff \lambda \in \Lambda. \quad \text{Then} \quad (P, v) \lambda \rho p \text{ for some } \theta \in \{L, R\}.
\]

Let \(\theta_{i_0}(i) = L \iff \lambda(L) = \top \quad \text{and} \quad \theta_{i_0}(i) = R \iff \lambda(R) = \bot\). Define a relation \(R_{\theta_{i_0}}\) such that \((Q, v') R_{\theta_{i_0}} q \iff (Q, v') \lambda_{i_0} q\). It is straightforward to verify that \(R_{\theta_{i_0}}\) is a product derivation relation for PL-LTS.

\[\blacksquare\]

**Theorem 4.** The class of PL-LTSs is less expressive than the class of FTSs.

**Proof.** Due to Theorem 3, we know that FTSs are at least as expressive as PL-LTSs. It remains to show that PL-LTSs are not at least as expressive as FTSs.

Consider the FTS as shown in Fig. 5, where \(f_a, f_b, f_c\) are three distinct features and the set of valid products \(\Lambda\) is defined as the smallest set of functions satisfying the following constraint:

\[
(f_a \implies (\neg f_b \land \neg f_c)) \land (f_b \implies (\neg f_a \land \neg f_c)) \land (f_c \implies (\neg f_a \land \neg f_b)).
\]

Through a proof by contradiction, we show that there is no encoding \(E\) that can transform the FTS \(P\) in the correct way. Suppose otherwise there is an encoding \(E\) of \(P\) into an PL-LTS whose configuration vectors are of type \([L, R, \_]\) (for some index set \(I\)) such that \([P]\) = \([E(P)]\) where \([P]\) = \([p\mid \exists q \in \Lambda \cap p\]\) and \([P]\) = \([Q, v]\) for some state \(Q\) and configuration vector \((Q, v) \subseteq \{L, R, \_\}\) in an PL-LTS, and \([E(P)]\) = \([Q, v]\) for some state \(Q\) and configuration vector \((Q, v) \subseteq \{L, R, \_\}\).

Clearly, the transition systems \((p_x, p_y), \{x, \{p_x, x, p'_y\}\}\) (for \(x \in \{a, b, c\}\)) are three valid products of the given FTS \(P\), i.e., \((p_x, p_y, p_z) \subseteq \{P\}\). So from the correctness requirement of \(E\) we have \([p_x, p_y, p_z] \subseteq \{E(P)\}\). Let \(i_0, i_1, i_2\) be the corresponding total configuration vectors that derives the products \(p_x, p_y, p_z\), respectively. Thus, for every \(x \in \{a, b, c\}\) we have \(v_x \subseteq \theta_{i_0}\). Furthermore, from the transfer property of product derivation we find \((Q, v_x) \lambda_{i_0} (Q, v_x)\), such that \(v_x \subseteq \theta_{i_0}\) for \(x \in \{a, b, c\}\). Clearly, \(v_x \neq v_y\) for \(x \in \{a, b, c\}\) because otherwise we can derive a transition system which contains choices of \(a, b, c\) or \(a, c\). For instance, if \(v_x = \theta_{i_0}\) then the transfer property of product derivation ensures that \(v_x = \theta_{i_0}\).

Next, we show that \(i_0 \neq i_1 \land i_2 \neq i_3 \land i_3 \neq i_4\) then we can derive a product which has a choice between \(a, c\). Since \(i_0, i_2\) are the only elements whose values are changed by \(v_x, v_y, v_z\), so from Condition (7) we have \(\tau_i(i_4) = \tau_j(i_2) = 7\). Define a function \(\theta_{i_1}(i)\) as follows:

\[
\theta_{i_1}(i) = \begin{cases} 
\theta(i) & \text{if } i \neq i_0 \\
\theta(i) & \text{if } i = i_0, \\
L & \text{if } i = i_2 \land \theta(i_2) = R, \\
L & \text{if } i = i_0 \land \theta(i_0) = R.
\end{cases}
\]

Next, we show that \(v_x \subseteq \theta_{i_1}\). If \(i \neq i_2 \land i \neq i_3\) then clearly \(v_x \subseteq \theta_{i_1}\) because \(\theta_{i_1}(i) = \theta(i)\). If \(i = i_2 \lor i = i_3\) then \(v_x \subseteq \theta_{i_1}\). Thus, \(v_x \subseteq \theta_{i_1}\).

As a result, we can derive a product that contains a choice between \(a, c\) by using \(\theta_{i_1}\); however, such a product is clearly not a valid product of the given FTS \(P\) as it violates the condition \(\Lambda\).

On the other hand, if \(i_0 = i_2\), then we show that using \(\theta_{i_0}\) we can derive a product which has a choice between either \(a, b\) or \(b, c\). It suffices to show that \(v_x(i_2) \subseteq \theta_{i_0}(i_2)\) or \(v_x(i_2) \subseteq \theta_{i_0}(i_2)\) because for every \(i \neq i_2\) we have \(v_x(i_2) \subseteq v_x(i_2)\). We claim that \(v_x(i_2) \neq \theta_{i_0}\). Suppose otherwise \(\theta_{i_0}(i_2) = 7\). Then, since \(i_0 = i_2\) we find \(v_x \subseteq \theta_{i_0}\). As a result, from the transfer property of \(\tau_i\), there must be a \(q\) such that \(p \rightarrow q\) otherwise \(E(P) \not\models \tau_i\). Hence, we can derive a product that contains a choice between either \(a, b\) or \(b, c\). However, such a product is clearly not a valid product of the given FTS \(P\) as it violates the condition \(\Lambda\). Likewise, if \(i_0 = i_3\).
We can hence summarize the results of this section by the following diagram:

\[
\text{MTSs} \longrightarrow \text{PL-LTSs} \longrightarrow \text{FTSs},
\]

where the arrow \(\longrightarrow\) indicates the "less-expressive-than" relation, i.e., the existence of an encoding from one product line structure into another and the lack of encoding in the other directions. In other words, the class of MTSs (FTSs) is the least (most) expressive product line structure considered in this paper. The fact that MTSs are less expressive than FTSs follows from the transitivity of the "less-expressive-than" relation; to emphasize this fact, we give the evidence of the lack of encoding from FTSs to MTSs in the following example.

**Example 8.** Consider the FTS drawn (left) in Fig. 6 with the set of features \(F = \{f, f'\}\) and the set of valid product configuration \(\Lambda = \{\lambda, \lambda'\}\) with \(\lambda(f) = \top, \lambda(f') = \bot\) and \(\lambda'(f) = \top, \lambda'(f') = \bot\). The transition systems of the derived processes \(P\) and \(Q\) under \(\lambda\) and \(\lambda'\), respectively, are drawn in Fig. 6. Now by contradiction we show that there is no encoding \(E\) satisfying Definition 11.

Suppose there is an encoding \(E\) satisfying Definition 11. Clearly, \(P \vdash_{\lambda} P\) and \(P \vdash_{\lambda'} Q\). Then by correctness of \(E\) we have \(E(P) \preceq P\) and \(E(P) \preceq Q\). Thus, we can derive the following transitions (for some modal states \(P_a, P_b\)) from the transfer property of modal refinement:

\[ E(P) \models_{\lambda} P_a \quad E(P) \models_{\lambda'} P_b. \]

Therefore, there exists a state \(r\) of the following form: \(r \xrightarrow{a} r'\) and \(r \xrightarrow{b} r''\) (for some \(r', r''\)) such that \(E(P) \preceq r\). And by correctness of \(E\) we get \(P \vdash_{\lambda} r\) or \(P \vdash_{\lambda'} r\). However, \(P \not\vdash_{\lambda} r\) and \(P \not\vdash_{\lambda'} r\).

4. Testing pre-orders for SPLs

In Section 2, we reviewed three different notions of product derivations based on a particular product line structure. These notions are intensional in nature, i.e., they require the products to be modeled completely as LTSs, and moreover, their models must be available in their entirety during testing. This assumption is rather unrealistic for practical systems. In practice, one needs an extensional notion of testing that can be used to generate a test-suite from a product-line specification (e.g., an MTS) in an offline or on-the-fly manner, in order to test a black-box implementation. Based on the foundational studies carried out in [22,21,20], such notions have been developed and extensively studied for various LTS-based formalisms [22,34]; however, we are not aware of any such notion for MTSs, PL-LTSs, and FTSs (the only exceptions being our recent work [35,36], as well as the recent work by Devroey et al. [37,38]). In the remainder of this section, we adopt the testing framework of [23] and adapt its notion of test to MTSs, PL-LTSs, and FTSs to characterize the corresponding product derivation relation for the respective product-line structure.

The notions elaborated in this section lay the theoretical connection between the intensional (trace-based comparison) and the extensional (test-case execution) notions of conformance. In order to turn this theory into a practical testing scheme, some degrees of unboundedness have to be tamed: firstly, a fault-model (regarding the implementation) [39], a notion of coverage [37], or a test-selection algorithm [40] has to be adopted to choose a finite set of test cases. Moreover, some assumptions about valid products and the interaction of their features (combined with the aforementioned methods or a bound on the maximum length of test-cases) can be used to select a finite set of incremental test-suites for various products [35,36].

4.1. Modal refinement as a testing pre-order

Consider a set of test expressions \(T\), ranged over by \(t\), generated by the following grammar [23]:

\[
t ::= \text{SUCC} \mid \text{FAIL} \mid \text{at} \mid \hat{t} \mid t_1 \land t_2 \mid t_1 \lor t_2 \mid \forall \xi \mid \exists \xi.
\]

Throughout this section, we assume that the product line structure under investigation is image finite.
Intuitively, SUCC and FAIL denote the successful and the failed tests, respectively, i.e., for every MTS the test SUCC (FAIL) will always pass (fail). The expression $a(t)$ tests the existence of a must-transition labeled $a$ and then examines the sub-test $t$. Furthermore, if an MTS refuses to perform the must-transition $a$, the verdict for this test is fail. The expression $a(t)$ tests the existence of a may-transition labeled $a$ and then examines the sub-test $t$. Furthermore, if an MTS refuses to perform the may-transition $a$, then the verdict for this test is success (pass). The tests of the form $t_1 \land t_2$ and $t_1 \lor t_2$ represent testing different copies of a machine using the sub-tests and subsequently, combining the results [20]. The tests of the form $\forall t$ and $\exists t$ represent global testing by, respectively, quantifying universally and existentially over runs of sub-test $t$.

Given a modal specification $P$ and an implementation $p$ (modeled as an LTS), the main idea is to assert indirectly whether the implementation $p$ is a valid product of the specification $P$, i.e., whether they are related by a modal refinement relation. Throughout this section, we use $P$ to denote a state of a modal specification and $p$ to denote the state of an LTS implementation. For this purpose, we need a concept of interaction between a test case and a state in an MTS (and an LTS). To this end, we recall the notion of experiment expression $E$ [23], generated by the following grammars:

\[
E ::= \top \mid \bot \mid (t \parallel P) \mid E_1 \land E_2 \mid E_1 \lor E_2 \mid \forall E \mid \exists E.
\]

\[
E ::= \top \mid \bot \mid (t \parallel P) \mid E_1 \land E_2 \mid E_1 \lor E_2 \mid \forall E \mid \exists E.
\]

Fig. 7 provides the operational interpretation of experiment expressions over an MTS. Note that only the rules of the expressions $a(t \parallel P)$ and $a \parallel P$ (i.e., rules 3–6) are modified with respect to the original rules presented in [23], while the rest of the operational rules are quoted verbatim for the sake of completeness. In particular, we define a transition relation $\rightarrow$ between any two experiment expressions as the smallest relation satisfying the rules of Fig. 7. Note that we do not need a separate set of rules to specify the experiment expressions interacting with an LTS, because they can also be derived from the rules of Fig. 7 by considering the transitions of LTS as both may and must transitions.

Once we have a transition system whose states are experiment expressions interacting with either a specification or an implementation, we can use this structure to define the set of results of evaluating a test on a process. The outcome of a single test is either successful or unsuccessful, which can be modeled as a two-point domain $\mathcal{O} = \bot \sqcup \top$. However, due to nondeterminism, sets of outcomes are required to get the results of all possible runs (cf. [23]). These outcomes are modeled using a set of truth values (or more precisely, using the Plotkin powerdomain $P(\mathcal{O}) = \{\bot\} \subseteq \{\bot, \top\} \subseteq (\top)$). The semantics of the operators $\land, \lor, \forall, \exists$ over the Plotkin powerdomain $P(\mathcal{O})$ can be found in [23]. In addition, given an MTS $(P, A, \rightarrow, \rightarrow, \rightarrow)$, we define the function $O : T \times P \rightarrow P(\mathcal{O})$ in the following way:

\[
O(t, P) = (T \mid (t \parallel P) \rightarrow \top) \cup (\bot \mid (t \parallel P) \rightarrow \bot).
\]
Lemma 3. Let \( P \) be a modal specification. Then, for any test expression \( t \) we have
\[
O(\forall t, P) = \forall O(t, P) \quad \text{and} \quad O(\exists t, P) = \exists O(t, P).
\]

Before we turn our attention to the characterization of modal refinement as a testing pre-order, we first give a semantic preserving transformation \( \llbracket \cdot \rrbracket_\Delta \) (Lemma 4) that transforms an HML formulae (interpreted over MTSs due to [41]) into the set of tests \( T \). We will use this transformation in the proof of Theorem 5 to establish a link between testing pre-order \( \preceq \) and the modal refinement relation \( \preceq \).

Consider the set of all HML formulae \( \Phi \) generated by the following grammar:
\[
\psi ::= \bot | T | (\alpha)\psi | [\alpha]\psi | \psi \land \psi' | \psi \lor \psi'.
\]
The semantics of \( \bot, T, \land, \lor \) is standard, while the nonstandard semantics of \( (\alpha)\psi, [\alpha]\psi \) is given as follows [41]:

1. \( P \models (\alpha)\psi \iff \exists P. P \overset{\alpha}{\rightarrow} P' \land P' \models \psi \).
2. \( P \models [\alpha]\psi \iff \forall P. P \overset{\alpha}{\rightarrow} P' \Rightarrow P' \models \psi \).

Following [23], we give a transformation \( \llbracket \cdot \rrbracket_\Delta : \Phi \rightarrow T \) of HML formulae to the set of test expressions.
\[
\llbracket T \rrbracket_\Delta = \text{SUCC}, \quad \llbracket \bot \rrbracket_\Delta = \text{FAIL},
\]
\[
\llbracket \psi \land \psi' \rrbracket_\Delta = \llbracket \psi \rrbracket_\Delta \land \llbracket \psi' \rrbracket_\Delta, \quad \llbracket \psi \lor \psi' \rrbracket_\Delta = \llbracket \psi \rrbracket_\Delta \lor \llbracket \psi' \rrbracket_\Delta,
\]
\[
\llbracket (\alpha)\psi \rrbracket_\Delta = \forall \llbracket \psi \rrbracket_\Delta, \quad \llbracket [\alpha]\psi \rrbracket_\Delta = \exists \llbracket \psi \rrbracket_\Delta.
\]
By setting the above technical machinery, we now prove that an MTS satisfies a HML formula \( \psi \) if and only if it passes the test \( \llbracket \psi \rrbracket_\Delta \).

Lemma 4. Let \( P \) and \( p \) be a state in an MTS and an LTS, respectively. Then, for any \( \psi \in \Phi \) we have
\[
P \models \psi \iff O(\llbracket \psi \rrbracket_\Delta, P) = \{T\} \quad \text{and} \quad p \models \psi \iff O(\llbracket \psi \rrbracket_\Delta, p) = \{T\}.
\]

Proof. The proof is by induction on \( \psi \); the cases for \( \bot, T, \land, \lor \) are straightforward.

1. Let \( \psi = (\alpha)\psi' \) and \( P \models \psi \). Then,
\[
\forall P. P \overset{\alpha}{\rightarrow} P' \Rightarrow P' \models \psi'.
\]
\[
\exists P. P \overset{\alpha}{\rightarrow} P' \land P' \models \psi'.
\]
(Induction hypothesis)
(Truth table [23]: \( \exists \{\bot, T\} = \{T\} \) and \( \exists \{\top\} = \{T\} \))

2. Let \( \psi = [\alpha]\psi' \) and \( P \models \psi \). Then,
\[
\exists P. P \overset{\alpha}{\rightarrow} P' \land P' \models \psi'.
\]
(Induction hypothesis)
(Truth table [23]: \( \exists \{\bot, T\} = \{T\} \) and \( \exists \{\top\} = \{T\} \))

\( \square \)
The following theorem is the first attempt towards characterization; namely, it shows that if the observations obtained from test-cases on an implementation are all allowed by the product line, then the implementation is a valid product (a modal refinement) of the product line.

**Theorem 5.** Let $P$ and $p$ be states in an MTS and an LTS, respectively. Then,

$$\forall t \in T \ O(t, P) \sqsubseteq O(t, p) \Rightarrow P \leq p.$$  

**Proof.** Suppose $\forall t \in T \ O(t, P) \subseteq O(t, p)$. In lieu of the modal characterisation given by Boudol and Larsen [41, Theorem 31], we show that $p \models \psi$ whenever $P \models \psi$, for any $\psi \in \Phi$. Suppose $P \models \psi$. Then, from Lemma 4 we know that $O([\psi]_A, P) = \{\top\}$. Since the element $\{\top\}$ is the maximum in the Plotkin powerdomain $\mathcal{P}[O]$ and $O([\psi]_A, P) \subseteq O([\psi]_A, p)$ we know that $O([\psi]_A, p) = \{\top\}$. From Lemma 4, we conclude that $p \models \psi$. \qed

To see why the converse of Theorem 5 does not hold, consider the states $P$ and $p$ given in Fig. 8, where dashed transitions denote may transitions and solid transitions denote must transitions. The dotted lines show the witnessing refinement relation between $P$ and $p$; thus, $P \not\leq p$. Consider the test $t = \text{addSucc}$. Clearly, $O(t, P) = \{\bot, \top\}$ and $O(t, p) = \{\bot\}$. However, $\{\bot, \top\} \not\subseteq \{\bot\}$.

Thus, in order to obtain a full characterisation of modal refinement, we need to restrict ourselves to the set of test expressions $T' \subseteq T$ generated by the grammar given below.

**Corollary 1.** Let $P$ and $p$ be states in an MTS and an LTS, respectively. Let $T' \subseteq T$ be the set of tests generated by the following grammar:

$$t ::= \text{SUCC} \mid \text{FAIL} \mid \exists t_1 \mid \forall t_1 \mid t_1 \land t_2 \mid t_1 \lor t_2.$$  

Then, $\forall t \in T' \ O(t, P) \subseteq O(t, p) \Leftrightarrow P \leq p$.

It follows also from Corollary 1 that if an LTS is not a valid product of an MTS, i.e., $P \not\leq p$, then it is sufficient to find a test $t \in T$ such that $O(t, P) \not\subseteq O(t, p)$. Moreover, for such an invalid product there always exists a test-case, which tells us why the product is invalid.

**Example 9.** Recall the MTS given in Fig. 2(a) and represent it by $P$. Consider an LTS given in Fig. 9 and represent it by $p$.

Observe that by adopting the test $t = \text{addSucc} \text{pour sugar pour coffee}$ we can show that $p$ is not a valid product of $P$ because $O(t, P) = \{\top\}$ and $O(t, p) = \{\bot\}$. Thus $\{\top\} \not\subseteq \{\bot\}$; hence, $P \not\leq p$.

4.2. **Testing pre-orders for FTSs and PL-LTSs**

Similar to the case of MTSs, we are not aware of any extensional notion of testing for FTSs and PL-LTSs (the product-derivation relation of the latter is similar to the product-derivation relation of FTSs). To fill in this gap, we modify the testing framework of MTS (given in the previous section) and show how to characterize our notion of product derivation of an FTS (PL-LTS) by a testing equivalence.

Recall the set of tests $T$ generated from the grammar given in the previous subsection. We give now an interpretation of a test $t \in T$ over an FTS $(P, A, F, \rightarrow, \Lambda)$. It should not be surprising that only the semantics of the tests of the form $\text{addSucc}$
and $\Delta t$ needs to be modified, while the semantics of the remaining operators (from Fig. 7) remains unchanged. Formally, we first define a family of transition relations, parameterized by product configurations, by modifying rules 3–6 in the following way.

$$
\begin{align*}
\frac{P \xrightarrow{a} P'}{\Delta t \parallel P \rightarrow \tau \parallel P'} \quad (3') \\
\frac{\exists Q.a \ P \xrightarrow{a} Q \quad \lambda \models \phi}{\Delta t \parallel P \rightarrow \bot} \quad (4')
\end{align*}
$$

Second, we define the family of observation functions (parameterized by the product configurations) which essentially evaluates an expression in a product modelled as an LTS.

$$
O_{\lambda}(t, P) = \{T \mid t \parallel P \rightarrow_{\lambda} T\} \cup \{\bot \mid t \parallel P \rightarrow_{\lambda} \bot\}.
$$

In a similar vein, we also give an interpretation of a test $t \in T$ over a PL-LTS $T \times [L.R.?]^T.A., \rightarrow$), where only rules 3–6 are modified and the remaining operational rules of the operators (except $\Delta t$, $\Delta t$) are unchanged.

$$
\begin{align*}
\frac{(P, v) \xrightarrow{a} (P', v')} {\Delta t \parallel (P, v) \rightarrow_{\sigma} t \parallel (P', v')} \quad (3'') \\
\frac{\exists Q.a \ (P, v) \xrightarrow{a} (Q, v') \quad v' \subseteq \theta}{\Delta t \parallel (P, v) \rightarrow_{\sigma} \bot} \quad (4'')
\end{align*}
$$

Lastly, we define a function $O_{\lambda} : T \times [L.R.?]^T \rightarrow \mathcal{P}(\emptyset)$, parametrized by configuration vectors, as follows:

$$
O_{\lambda}(t, P) = \{T \mid t \parallel (P, v) \rightarrow_{\sigma} T\} \cup \{\bot \mid t \parallel (P, v) \rightarrow_{\sigma} \bot\}.
$$

Just like in the case of MTTS, we have the following lemma.

**Lemma 5.** Let $P$ be a state in an LTS and let $\lambda$ be a product configuration. Then, for any test expression $t$ we have

$$
O_{\lambda}(\forall t, P) = \forall O_{\lambda}(t, P) \quad \text{and} \quad O_{\lambda}(\exists t, P) = \exists O_{\lambda}(t, P)
$$

Next, we give the main result of this subsection; namely that our notion of test-cases is both sound and complete for the generalized notion product derivation.

**Theorem 6.** Let $P$ be an LTS specification, $p$ be state in an LTS, and $\lambda$ be a product. Then,

$$
P \vdash_{\lambda} p \Leftrightarrow \forall t \in T \ O_{\lambda}(t, P) = O(t, p).
$$

**Proof.** (⇒) Suppose otherwise, $P \not\vdash_{\lambda} p$ and for all tests $t$, we have $O_{\lambda}(t, P) = O(t, p)$. Then we distinguish the following cases:

1. Either, there exists $a, Q$ such that $P \xrightarrow{a} Q, \lambda \models \phi$, and for all $q$, if $p \xrightarrow{a} q$ then $Q \not\vdash_{\lambda} q$. Let $p(a) = \{q \mid p \xrightarrow{a} q\}$. Due to image finiteness assumption we know that the set $p(a)$ is finite. We identify the following cases:
   (a) Suppose $p(a) = \emptyset$. Then,
   $$
   \begin{align*}
   T &\in O_{\lambda}(aSUCC, P) \\
   \Rightarrow &\exists O_{\lambda}(aSUCC, P) = \{T\} \quad (\text{Truth table of } \exists)
   \Rightarrow &\exists O_{\lambda}(aSUCC, P) = \{T\} \quad (\text{Lemma 5: } O_{\lambda}(\exists t, s) = \exists O_{\lambda}(t, s)).
   \end{align*}
   $$
   But, $O(aSUCC, p) = \{\bot\}$; thus,
   $$
   O(aSUCC, p) = \exists O(aSUCC, p) = O(aSUCC, p) = \{\bot\}.
   $$
   Hence, a contradiction follows.

   (b) Suppose $p(a) = \{q_1, \ldots, q_n\}$. Then, by induction hypothesis there exists sub-tests $t_1, \ldots, t_n$ such that $O_{\lambda}(t_i, Q) \neq O(t_i, Q)$. Hence, a contradiction follows.

2. Otherwise, $p(a) = \{q_1, \ldots, q_n\}$. Then, by induction hypothesis there exists sub-tests $t_1, \ldots, t_n$ such that $O_{\lambda}(t_i, Q) \neq O(t_i, Q)$.
2. Or, $O_\lambda(t_1, Q) = \{\top\} \land O(t_i, q_i) = \{\bot\}$. Let $t' = t_1 \land \cdots \land t_n$. Consequently,

\[
\top \in O_\lambda(\lambda t', P) \quad (\therefore O_\lambda(t_i, Q) = \{\top\})
\]

\[
\Rightarrow \exists O_\lambda(\lambda t', P) = \{\top\} \quad \text{(Truth table of } \exists \text{)}
\]

\[
\Rightarrow O_\lambda(\exists \lambda t', P) = \{\top\} \quad \text{(Lemma 5: } O_\lambda(\exists \phi, P) = \exists O_\lambda(t, P))
\]

But,

\[
O(\lambda t', p) = \{\bot\} \quad (\therefore O(t_i, q_i) = \{\bot\})
\]

\[
\Rightarrow \exists O(\lambda t', p) = \{\bot\} \quad \text{(Truth table: } \exists \{\bot\} = \{\bot\})
\]

\[
\Rightarrow O(\exists \lambda t', p) = \{\bot\} \quad \text{(Lemma 5: } O_\lambda(\forall t, Q) = \forall O_\lambda(t, Q))
\]

Hence, a contradiction.

ii. Or, $O_\lambda(t_1, Q) = \{\bot\} \land O(t_i, q_i) = \{\top\}$. Let $t' = t_1 \lor \cdots \lor t_n$. Consequently,

\[
\bot \in O_\lambda(\lambda t', Q) \quad (\therefore O_\lambda(t_i, Q) = \{\bot\})
\]

\[
\Rightarrow \forall O_\lambda(\lambda t', Q) = \{\bot\} \quad \text{(Truth table of } \forall \text{)}
\]

\[
\Rightarrow O_\lambda(\forall \lambda t', Q) = \{\bot\} \quad \text{(Lemma 5: } O_\lambda(\forall t, Q) = \forall O_\lambda(t, Q)).
\]

Furthermore,

\[
O(\lambda t', p) = \{\top\} \quad \text{(since } O(t_i', q_i) = \{\top\})
\]

\[
\Rightarrow \forall O(\lambda t', p) = \{\top\} \quad \text{(Truth table: } \forall \{\top\} = \{\top\})
\]

\[
\Rightarrow O(\forall \lambda t', p) = \{\top\}
\]

Hence, a contradiction.

2. Or, there exists $a, q$ such that $p \models a, q$ and for all $Q, \phi$, if $P \models a, Q \land \lambda \models \phi$, then $Q \not\models a, q$. Due to image finiteness assumption, we know that the set $P(a) = \{Q \mid \exists \phi \models P \models a, Q \land \lambda \models \phi\}$ is finite.

(a) Suppose $P(a) = \emptyset$. Then,

\[
O_\lambda(\text{FAIL}, P) = \{\top\} \quad (\therefore P(a) = \emptyset)
\]

\[
\Rightarrow \forall O_\lambda(\text{FAIL}, P) = \{\top\} \quad \text{(Truth table of } \forall \text{)}
\]

\[
\Rightarrow O_\lambda(\forall \text{FAIL}, P) = \{\top\} \quad \text{(Lemma 5: } O_\lambda(\forall t, P) = O_\lambda(\forall t, P))
\]

But, $O(\text{FAIL}, p) = \{\bot\}$ (since $p \not\models a, q$, $O(\text{FAIL}, q) = \{\bot\}$). Thus, $O(\forall \text{, } p) = \forall O(\text{FAIL}, p) = \{\bot\}$, which is a contradiction.

(b) Suppose $P(a) = \{Q_1, \cdots, Q_n\}$. Then, by induction hypothesis there exists sub-tests $t_1, \cdots, t_n$ such that $O_\lambda(t_1, Q_i) \neq O(t_1, Q_i)$.

i. Either $O_\lambda(t_1, Q_i) = \{\top\}$ and $O(t_i, q_i) = \{\bot\}$. Let $t' = t_1 \lor \cdots \lor t_n$. Then, from truth table of $\lor$ we have $O_\lambda(t', Q_i) = \{\top\}$. Consequently,

\[
O_\lambda(\lambda t', P) = \{\top\} \quad (\therefore O_\lambda(t', Q_i) = \{\top\})
\]

\[
\Rightarrow \forall O_\lambda(\lambda t', P) = \{\top\} \quad \text{(Truth table of } \forall \text{)}
\]

\[
\Rightarrow O_\lambda(\forall \lambda t', P) = \{\top\} \quad \text{(Lemma 5: } O_\lambda(\forall t, P) = O_\lambda(\forall t, P))
\]

Furthermore, $O(t', q) = O(t_1, q) \lor \cdots \lor O(t_n, q) = \{\bot\}$. Thus,

\[
\bot \in O(\lambda t', p)
\]

\[
\Rightarrow \forall O(\lambda t', p) = \{\bot\} \quad \text{(Truth table of } \forall \text{)}
\]

\[
\Rightarrow O(\forall \lambda t', p) = \{\bot\} \quad \text{(Truth table of } \forall \text{)}
\]

\[
\Rightarrow O(\forall t, p) = O(\forall t, p) \quad \text{(Lemma 5: } O_\lambda(\forall t, P) = O_\lambda(\forall t, P))
\]

Hence, a contradiction.

ii. Or $O_\lambda(t_1, Q_i) = \{\bot\}$ and $O(t_i, q_i') = \{\top\}$. Let $t' = t_1 \land \cdots \land t_n$. Then, $O_\lambda(t', Q_i) = \{\bot\}$. Consequently,

\[
O_\lambda(\lambda t', P) = \{\bot\} \quad \text{(since } O_\lambda(t', Q_i) = \{\bot\})
\]

\[
\Rightarrow \exists O_\lambda(\lambda t', P) = \{\bot\} \quad \text{(Lemma 5: } O_\lambda(\exists \phi, P) = \exists O_\lambda(t, P))
\]
Furthermore, \( O(t', q) = O(t_1, q) \land \cdots \land O(t_n, q) = \{\top\} \). Thus, 
\[
\top \in O(\tilde{a'}, p) \quad \text{(since } O(t', q) = \{\top\})
\]
\[
\Rightarrow \exists \tilde{a}(\tilde{a'}, p) = \{\top\} \quad \text{(Truth table of } \exists \)
\]
\[
\Rightarrow O(\tilde{a'}, p) = \{\top\} \quad \text{(} \exists \top(, p) = O(\exists t, p) [23]\).
\]

Hence, a contradiction follows.

(\(\Rightarrow\)) Suppose \( P \vdash_t p \). We show by induction on \( t \) that \( O_j(t, P) = O(t, p) \). The cases when \( t = \text{Succ}, \text{FAIL}, \top_1 \lor t_2, t_1 \land t_2, \top' \), \( \exists t' \) are straightforward.

The interesting cases are the following:

1. Let \( t = \bar{a'} \).
   (a) Let \( \top \in O_j(\bar{a'}, P) \). Then,
   \[
P \not\vdash \phi \wedge \lambda \models \top \in O_j(t', Q), \quad \text{for some } \phi, Q
   \]
   \[
   \Rightarrow \top \models \phi, Q
   \]
   \[
   \Rightarrow \top \in O_j(t', Q) \Rightarrow \top \in O(t', q)
   \]
   (Induction hypothesis)

   \[
   \Rightarrow \top \in O(\bar{a'}, p).
   \]

   (b) Let \( \bot \in O_j(\bar{a'}, P) \). Then we have the following cases:
   i. Either \( P \not\vdash \phi \wedge \lambda \models \bot \in O_j(t', Q) \), for some \( \phi, Q \). Similar to Case (a).
   ii. Or, \( \exists \phi, P \not\vdash Q \wedge \lambda \models \phi \). Then, \( O_j(\bar{a'}, P) = \{\bot\} \). Suppose otherwise, \( \top \in O(\bar{a'}, p) \). Then, \( \exists \top \top \models \phi, Q \). But, \( P \vdash \top \). Thus, \( \exists \phi, P \not\vdash Q \wedge \lambda \models \phi \), which is a contradiction.
   (c) Let \( \bot \in O(\bar{a'}, p) \). Similar to Case (a).
   (d) Let \( \bot \in O(\bar{a'}, p) \). Similar to Case (b).

2. Let \( t = \bar{b} \).

Furthermore, it follows from Lemma 1 that the notion of test cases remains sound and complete for the traditional notion of product derivation.

**Theorem 7.** Let \((P, v)\) be a state in a PL-LTS, \( p \) be state in an LTS, and \( \theta \) be a configuration vector. Then,
\[
(P, v) \vdash \theta p \Leftrightarrow \forall t \in \mathcal{T} \quad O_{\theta}(t, P, v) = O(t, p).
\]

**Proof.** Similar to the proof of Theorem 6. \(\square\)

5. Conclusions

In this paper, we studied three fundamental behavioral models for software product lines, namely, modal transition systems, featured transition systems, and product-line labeled transition systems. In particular, we studied the expressiveness of these models by comparing their sets of definable products, which are assumed to be expressible as labeled transition systems. We have shown that modal transition systems are the least expressive of all three, featured transition systems are the most expressive, and product-line labeled transition systems are strictly in between the two. Then we moved to define extensional notions of product derivation and adapted the notion of tests by Abramsky to this end. We proved that the intensional notions of product derivation coincide with the extensional notions defined in this paper for each and every formalism.

Compositionality (pre-congruence) is well-studied for modal refinement in the context of MTSs [24]. However, this problem is understudied for FTSS and this is a high priority item in our future-research agenda. We envisage that using the divide and congruence approach of [42,43] could provide a solution in this regard (see [44] for our initial attempt in this direction). Another important topic in this area is defining a closed and finite notion of test-cases that can detect all faults, given a fault model (e.g., similar to the W-Method in FSM-based testing [39]). A third area of research, which builds upon the previously-mentioned topic, is to define an incremental procedure for testing different products of a product line.

A few other proposals for transition-system-based specifications of SPLs have been proposed that deserve further investigation. In [10,11], (Generalized) Extended Modal Transition Systems (GEMTSs) have been introduced in order to specify SPLs. These are variants of disjunctive normal forms [45]. We conjecture that this formalism is strictly in between MTSs and FTSS in terms of expressiveness. Also, in [19], a multi-modal semantics for “Variant Process Algebra” has been introduced, which we conjecture, is as expressive as featured transition systems. We leave proving these conjectures, as well as devising the appropriate extensional notion of testing for GEMTSs for future work.


Appendix C

Paper III
Corrigendum to Basic Behavioral Models for Software Product Lines: Expressiveness and Testing Equivalences

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi

Submitted to Science of Computer Programming Journal
Corrigendum to Basic Behavioral Models for Software Product Lines: Expressiveness and Testing Equivalences

Mahsa Varshosaz1, Harsh Beoharb, Mohammad Reza Mousavi1

1Center for Research on Embedded Systems (CERES), Halmstad University, Sweden
bUniversity of Duisburg-Essen, Germany

Abstract

In “Basic Behavioral Models of Software Product Lines”, we established an expressiveness hierarchy and studied the notions of refinement and testing for 3 fundamental behavioral models for software product lines. These models were modal transition systems, product line labeled transitions systems, and featured transition systems. It turns out that our definition of product line labeled transition systems is more restrictive than the one introduced by Gruler, Leucker, and Scheidemann. Adopting the original and more liberal notion will change the expressive results as we demonstrate in this corrigendum. Namely, we show that the original notion of product line labeled transition systems is as expressive as featured transition systems. As an additional result, we show that there are featured transition systems for which the size of the corresponding product line labeled transition system, resulted from any sound encoding, is exponentially larger than the size of the original model. Furthermore, we show that each product line labeled transition system can be encoded into an FTS, such that the size of the featured transition system is polynomial in terms of the size of the corresponding model. To summarise, the featured transition systems are as expressive as but exponentially more succinct than product line labeled transition systems.

© 2014 Published by Elsevier Ltd.

Keywords:

1. Introduction

Software Product Line (SPL) engineering is a software development technique enabling mass production and customisation. Using this technique, a family of software systems is developed efficiently based on a common core and by benefiting from systematic reuse of artifacts among products.

There are several quality assurance techniques such as model-based testing and model checking that require a model describing the behavior of the system. Hence, several behavioral models have been proposed to represent the behavior of the products in an SPL compactly and efficiently; examples of such models are featured transition systems (FTSs) [1], product line calculus of communicating systems (PL-CCSs) [2], and modal transition systems (MTSs) [3] and different extensions of MTSs [4–6].

These formalisms are comparable in terms of the types of behavior that they can capture and also in terms of their underlying formal model, i.e., Labeled Transition Systems (LTSs). In [7], we studied the comparative expressiveness...
of three of the above formalisms, namely PL-CCSs, FTSs and MTSs, where as a part of the results we concluded that the class of product line labeled transition systems (PL-LTSs) (used as the underlying semantic domain for PL-CCS models) is less expressive as the class of FTSs. This paper is a corrigendum to [7], where we relax a restrictive assumption that has been made about the transition rules of PL-LTSs (see Definition 7 in [7]).

Featured transition systems [1] are introduced as an extension of the labeled transition systems where the transitions are additionally labeled with feature expressions. Each feature expression is a propositional formula in which the variables represent the features of a product family. Feature expressions indicate the presence/absence of a transition in the model of each product (for more details see Section 2.2). Using FTSs, the behavior of all products is represented in a whole model and different types of analysis can be performed for all products at once using this model.

Product line calculus of communicating systems [2], is an extension of the Milner’s Calculus of Communicating Systems (CCS) [8]. Using PL-CCS, it is possible to model alternative behavior. The syntax of PL-CCS is an extension of the syntax of CCS by variant operator, which represents an alternative choice between its operands. A choice can be resolved once and for all. This means, in case of recursion that if, a variant choice is resolved in the first iteration, then it remains the same in the future iterations. In [2], PL-LTSs are defined as the semantic domain for PL-CCS models. In order to keep track of variant choices, a configuration vector is included in the state of PL-LTSs. In each PL-LTS, the size of the vector is equal to the number of the variant choices in the corresponding PL-CCS term. The elements of the configuration vector can denote a choice that is either undecided or decided in favour of the left-hand side or right-hand side variant. In [7], we assumed that in each step, only one of the variant choices can be resolved. (Based on this assumption each transition can change only one of the elements of the configuration vector in the target state.) This turns out to be an overly restrictive assumption compared to the definition given for the PL-LTS transition rules in [8]. Considering this assumption, as a part of the results it was shown that PL-LTSs cannot capture some types of behavior such as three-way choices which can be captured by FTSs. Hence, it was concluded that the class of PL-LTSs is less expressive as the class of FTSs.

In this paper, we relax the above-mentioned restriction and adapt the result to the original and more allowing definition of PL-LTSs [2]. We revisit the comparative expressiveness of the FTSs and PL-CCSs with respect to the products that they specify. We describe an encoding of FTSs into PL-LTSs and there by proving that for each FTS, the set of LTSs that implement an FTS are also valid implementations for the PL-LTS resulted from the encoding and vice versa. The results show that the class of PL-LTSs is at least as expressive as the class of FTSs. Given that the class of FTSs is at least as expressive as the class of PL-LTSs [7]; we conclude that the class of PL-LTSs is as expressive as the class of FTSs. We also provide a complexity analysis of the size of the PL-LTSs resulted from any sound encoding in terms of the number of the states of the corresponding FTS. The results of the succinctness analysis shows that FTSs are more compact formalisms compared to PL-LTSs to describe SPLs.

The rest of this paper is organized as follows. In Section 2, we review the basic definitions regarding feature transition systems and the product line calculus of communicating systems. In Section 3, we provide an encoding from FTSs into PL-LTSs. In Section 4, we show that the class of underlying semantic model of PL-CCS is at least as expressive as the class of FTSs. In Section 5, we provide the complexity analysis for the models resulted from encoding of FTSs. In Section 6, we provide an overview of related work and in Section 7, we conclude the paper and present the directions of our ongoing and future work.

2. Preliminaries

In this section, we provide the definition of constructs and concepts that are used throughout the paper.

2.1. Feature Diagram

In software product line engineering, the commonalities and variabilities among products are described using features. A feature is defined as “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system” [9]. Each product in a software product line is defined by a subset of features selected from the whole set of features of the SPL. There are different relations between the features in an SPL. Feature models [10] are a common means to compactly represent the set of products of an SPL in terms of features.

A feature model is a hierarchical structure consisting of nodes and edges between them. Each node in a feature model represents a feature in the SPL. The structure of a feature model is tree like. Each node can have a set of child
nodes. The features in an SPL can be optional, or mandatory. The mandatory features are present in all products of the SPL, while the optional features may be present in a subset of the products. A group of sibling features (nodes) can have the alternative relation, which means only one of the features in the group can be included in a product in case that the parent feature is selected. Also, a group of sibling features can have the or relation, which means one or more features in the group can be included in a product if the parent feature is selected. There are also cross tree relations such as requires (resp. excludes), where the inclusion of a feature results in inclusion (resp. exclusion) of other features. Each feature model can be represented by a propositional logic formula in which propositional variables represent the features in the SPL [11].

Example 1. An example of a feature model is depicted in Fig. 1. The feature model corresponds to a vending machine product line (the vending machine in this example is a simplified version of the one given in [7]).

In this feature model features such as coin (o), beverage (b) are mandatory and features tea (t) and cappuccino (p) are optional. The single letters given under each feature are used later to represent the features in the propositional formulae. The dashed two headed arrow represents the excludes relation between the cappuccino (p) and the 1d (d) features.

We assume that $\mathbb{B} = \{T, \bot\}$ is the set of Boolean constants and $\mathbb{B}(F)$ denotes the set of all propositional formulae generated by considering the elements of the feature set $F$ as propositional variables. (Each propositional formula $\phi \in \mathbb{B}(F)$ is called a feature expression.)

2.2. Featured Transition System

As mentioned before, in FTSs, the behaviour of all products can be compactly depicted in one model by exploiting feature expressions as annotations. We give the formal definition of an FTS based on [1] as follows:

Definition 1. A Featured Transition System (FTS) is a quintuple $(P, A, F, \rightarrow, \Lambda, \text{p_{init}})$, where

1. $P$ is a set of states,
2. $A$ is a set of actions,
3. $F$ is a set of features,
4. $\rightarrow \subseteq S \times \mathbb{B}(F) \times A \times S$ is the transition relation satisfying the following condition:

$$\forall_{P, a, P', \phi, \phi'} \left( (P, \phi, a, P') \in \rightarrow \land (P, \phi', a, P') \in \rightarrow \right) \implies \phi = \phi',$$

5. $\Lambda \subseteq \{ A : F \rightarrow \mathbb{B}\}$ is a set of product configurations,
6. \( p_{\text{init}} \subseteq \mathcal{P} \) is the initial state.

**Example 2.** Consider the FTS given in Fig. 2. This FTS describes the behaviour of the products in the vending machine product line. (In this paper, we consider the finite behaviour of systems. Hence, Fig. 2 represents a part of the finite behaviour of the vending machine product line.)

The set of product configurations for this FTS is as follows:

\[
\{(m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c), (m, o, b, d, c)\}
\]

Considering two feature expressions such as \( \phi_1 \) and \( \phi_2 \) in \( \mathcal{B}(F) \), we say \( \phi_1 \) refines \( \phi_2 \), denoted by \( \phi_1 \leq \phi_2 \), if \( \phi_1 \rightarrow \phi_2 \). We say two feature expressions \( \phi_1 \) and \( \phi_2 \) are exclusive, denoted by \( \phi_1 \Rightarrow \phi_2 \) if \( \neg(\phi_1 \land \phi_2) \) holds.

Another formal structure that is used in this paper (to describe the behaviour of each product) is LTS. An LTS is defined as follows.

**Definition 2 (LTS).** A labeled transition system is a tuple \((S, A, \rightarrow, s_{\text{init}})\), where \( S \) is a set of states, \( A \) is a set of actions, \( \rightarrow \subseteq S \times A \times S \) is the transition relation, and \( s_{\text{init}} \) is the initial state.

Consider LTS \((S, A, \rightarrow, s_{\text{init}})\) and \( s_{\text{init}} = s_0\); an initial finite path in this LTS is a sequence such as \( \rho = s_0 a_1 s_1 a_2 \cdots a_n s_n \), where \( \forall_{0 \leq k < n} : s_i \xrightarrow{a_i} s_{i+1} \). By \( \rho(k) \), we denote the \( k \)th state in path \( \rho \). We denote the set of all initial finite paths in LTS \( T \) by \( \text{Paths}(T) \). For a path \( \rho \), \( \text{Trace}(\rho) \) denotes the sequence of actions in the path. For example \( \text{Trace}(s_0 a_1 s_1 a_2 \cdots s_n) = a_1 \cdots a_n \).

**2.2.1. Deriving Valid Products**

The refinement relation formalises the notion of product derivation as follows [7]:

**Definition 3.** Given an FTS \( fts = (P, A, F, \rightarrow, \Lambda) \), and LTS \( l = (S, A, \rightarrow, s_{\text{init}}) \), and a product \( \Lambda \in \Lambda \). A family of binary relations \( \mathcal{R}_l \subseteq P \times S \) (parameterized by product configurations) are called product-derivation relations if and only if the following transfer properties are satisfied.
1. \( \forall_{P,Q,a,s} (P \rightarrow_{\alpha} Q \land \Delta \models \phi) \Rightarrow \exists_{t} (s \rightarrow_{a} t \land Q \rightarrow_{\Delta} t) \)

2. \( \forall_{P,a,s} (P \rightarrow_{a} s \land s \rightarrow_{\alpha} t) \Rightarrow \exists_{Q,\Delta} (P \rightarrow_{a} Q \land \Delta \models \phi \land Q \rightarrow_{\Delta} t) \)

A state \( s \in S \) derives the product \( \lambda \) from an FTS-specification \( P \in \mathcal{P} \), denoted by \( P \rightarrow_{\mathcal{P}} s \), if there exists a product-derivation relation \( R \) such that \( P \rightarrow_{\mathcal{P}} R s \).

We say \( l \) is a valid implementation of \( \mathcal{F} \), denoted by \( \mathcal{F} \models l \), if and only if there exists a product configuration \( \lambda \in \Lambda \) such that \( p_{\text{init}} \rightarrow_{\lambda} s_{\text{init}} \).

**Example 3.** As an example, Fig. 3 depicts an LTS which implements the FTS in Fig. 2 and describes the behaviour of a product in the vending machine product line serving coffee and tea with and without sugar.

![Figure 3. an LTS implementing the FTS in Fig. 2.](image)

2.3. Product Line Process Algebras

PL-CCS is an extension of Milner’s Calculus of Communicating Systems (CCS) [8] in which a new operator \( \oplus \), called binary variant, is introduced to represent the alternative relation between features. The syntax of this process algebra is given in the following definition [7].

**Definition 4 (PL-CCS).** Assuming the alphabet \( A = \Sigma \cup \Delta \cup \{t\} \), where \( \Sigma \) is a set of symbols and \( \Delta = \{\bar{a} | a \in \Sigma\} \). The syntax of PL-CCS terms \( e \) is denied by the following grammar:

\[ \text{Nil} | \alpha.e | e + e' | e \oplus e' | e \parallel e' | e[f] | e[L] \]

where Nil denotes the terminating process, \( \alpha \) denotes the action prefixing for action \( \alpha \in A \), \( + \) and \( \parallel \) respectively denote non-deterministic choice and parallel composition, \( [f] \) denotes renaming by means of a function \( f \) where \( f : A \rightarrow A \) for each \( L \subseteq A \), \( L \) denotes the restriction operator (blocking (co)actions in \( L \)), and finally \( \oplus \) denotes a family of binary operators indexed with natural number \( i \).
The difference between the introduced binary variant operator \( \oplus \) and the ordinary alternative composition operator + in CCS is that the binary variant choice is made once and for all. As an example, consider the process terms \( P = b.P + c.P \) and \( Q = b.Q \oplus c.Q \); recursive process \( P \) keeps making choices between \( b \) and \( c \) in each recursion, while process \( Q \) makes a choice between \( b \) and \( c \) in the first recursion, and in all the following iterations the choice respected. This means that process \( P \) behaves deterministically after the first iteration with respect to the choice between \( b \) and \( c \). For the sake of simplicity in the formal development of the theory, Guerler et al. assume that in every PL-CCS term, there is at most one appearance of the operator \( \oplus \) for each and every index \( i \). We use the same assumption throughout the rest of the paper, as well.

The semantics of a PL-CCS term is defined based on product line labeled transition systems [2], using a structural operational semantics. (We refer to [2] for the formal semantics of PL-CCS). Each state in a product line labeled transition system \((\Pi, L, R, ?, I)\) is as follows:

**Definition 5** (PL-LTS). Let \((\Pi, L, R, ?, I)\) denote the set of all total functions from an index set \( I \) to the set \([L, R, ?]\). A product line labeled transition system (PL-LTS) is a quintuple \((\Pi \times [L, R, ?]^I), A, I, \rightarrow_{PRT}, \text{init}\) consisting of a set of states \(\Pi \times [L, R, ?]^I\), a set of actions \(A\), and a transition relation \(\rightarrow_{PRT}\) \subseteq \((\Pi \times [L, R, ?]^I) \times (A \times [L, R, ?]^I) \times (\Pi \times [L, R, ?]^I)\), and an initial state \(\text{init}\), satisfying the following restrictions:

1. \( \forall P, v, v' \in [L, R, ?]^I \) \( (P, v) \rightarrow a^{v'} (P, v') \) \( \Rightarrow v' = v''. \)

2. \( \forall P, v, v' \in [L, R, ?]^I \) \( (P, v) \rightarrow a^{v'} (P, v') \) \( \wedge v(i) \neq \# \) \( \Rightarrow v'(i) = v(i). \)

3. \( \forall P, v, v_0, v', v_0, v, v_0, v', v_0, v_0 \in [L, R, ?]^I \) \( (P_0, v_0) \rightarrow a^{v_0} (P, v) \) \( \wedge v_0(i) = v_1(i) \wedge v_0(i) \neq v'_0(i) \) \( \Rightarrow (P_0, v_0) \rightarrow a^{v_0} (P, v') \) \( \wedge v_0(i) = v_1(i) \wedge v_0(i) \neq v'_0(i). \)

The first condition indicates that each transition in the model is labeled with the configuration vector in the target state of the transition. The second condition shows that after making a variant choice which leads to assigning the value of an element in the configuration vector to \( L \) or \( R \), that value remains the same in the following steps. The third condition indicates that the same choice can not be resolved in multiple states in the model. (This follows from the definition of the semantics for PL-CCS terms in [2], where each variant operator is labeled with a unique index.)

Assuming that in the above defined PL-LTS, \( P_{\text{init}} = (P_0, v_0); \) an initial finite path in this PL-LTS is a sequence such as \( (P_0, v_0) \rightarrow a^{v_0} (P_1, v_1) \rightarrow a^{v_1} (P_2, v_2) \cdots \rightarrow a^{v_n} (P_{n+1}, v_{n+1}) \). We denote the set of all such paths for a PL-LTS \( PRT \) by \( \text{Paths}(PRT) \). We define the following relations between configuration vectors in a PL-LTS which are used in the rest of the paper.

**Definition 6** (Configuration Ordering). The preorder \( \sqsubseteq \) on the set \([L, R, ?]^I\) is defined as:

\[ \sqsubseteq = (\{?, ?, (L, L), (R, R), (L, R), (R, L), (?,?), (?,?)\}). \]

We lift this ordering relation to the level of configuration vectors by defining \( v \sqsubseteq v' \) \( \iff \forall i \in [L, R, ?]^I \) \( v(i) \sqsubseteq v'(i) \), for any \( v, v' \in [L, R, ?]^I \).

Using this relation we can specify if a configuration vector is more refined compared to the other (i.e. has less undecided choices).

**Definition 7** (Configuration conflict). The relation \( \bowtie \) on the set \([L, R, ?]^I\) is defined as:

\[ \bowtie = [(L, R), (R, L)]. \]

We lift this relation to the level of configuration vectors by defining \( v \bowtie v' \) \( \iff \exists i \in [L, R, ?]^I \) \( v(i) \bowtie v'(i) \), for any \( v, v' \in [L, R, ?]^I \).
Using this relation we can specify if there is a conflict between two configuration vectors (i.e. there is at least one element which is decided in both configuration vectors and the decision is not the same).

In order to define the set of LTS implementations of a PL-LTS, the refinement relation for PL-LTSs is given as follows.

**Definition 8** (Refinement for PL-LTSs). Let \(\text{plt} = ([P \times \{L, R, ?\}]^1, A, \rightarrow)\) be a PL-LTS and let \(l = (S, A, \rightarrow, s_{init})\) be an LTS. A family of binary relations \(R_{\theta} \subseteq ([P \times \{L, R, ?\}]^1) \times [P\) (parameterized by every product configuration \(\theta \in \{L, R\}^1\)) is a family of product-derivation relations if and only if the following transfer properties are satisfied:

1. \(\forall p, a, p', s \in \text{plt} \quad (s, p, a, p') R_{\theta} \quad \Rightarrow \quad \exists t : s \rightarrow t \land (s', p, a, p') R_{\theta} t\),

2. \(\forall p, a, p', s \in \text{plt} \quad (s, p, a, p') R_{\theta} \quad \Rightarrow \quad \exists t : s \rightarrow t \land (s', p, a, p') R_{\theta} t\).

A state \(s \in \text{plt}\) in an LTS is (the initial state of) a product of a PL-LTS \((P, v)\) with respect to a configuration vector \(\theta\), denoted by \((P, v) \rightarrow_{\theta} s\), if \(v \subseteq \theta\) and there exists a product-derivation relation \(R_{\theta}\) such that \((P, v) R_{\theta} s\).

We say \(l\) is a valid implementation of the PL-LTS \(\text{plt}\), denoted by \(\text{plt} \models l\) if and only if there exists a configuration vector \(\theta \in \{L, R, ?\}^1\) such that \(p_{init} \models_{\theta} s_{init}\).

### 2.4. Encoding

In order to compare the expressiveness between different modelling formalisms of SPLs, we give two following definitions, respectively, for product line structure and encoding.

**Definition 9** (Product Line Structure). Assuming that LTS denotes the class of all LTSs. A product line structure is a tuple \(M = ([M], [\cdot])\), where \([\cdot]\) is the class of the intended product line models (in this paper FTSs and PL-LTSs) and \([\cdot] : M \rightarrow \text{LTS}\) is the semantic function mapping a product line model to a set of product LTSs that can be derived from the product line model.

Consider the tuple \((\text{FTS}, [\cdot])\), which is a product line structure defined for the class of FTSs. For an arbitrary LTS \(\text{fts}\) and arbitrary LTS \(l\), it holds \(l \in [\text{fts}] \Leftrightarrow \text{fts} \models l\) (see Definition 3). Similarly, considering the tuple \((\text{PL-LTS}, [\cdot])\), which is the product line structure defined for the class of PL-LTSs. For an arbitrary PL-LTS \(\text{plt}\) and arbitrary LTS \(l\) it holds \(l \in [\text{plt}] \Leftrightarrow \text{plt} \models l\) (see definition of \(\text{plt} \models l\) in Section 2.3).

**Definition 10** (Encoding). An encoding from a product line structure \(M = ([M], [\cdot])\) into \(M' = ([M'], [\cdot'])\), is defined as a function \(E : M \rightarrow M'\) satisfying the following correctness criterion: \(\cdot = \cdot' \circ E\).

We say a product line structure \(M'\) is at least as expressive as \(M\) if and only if there exists an encoding \(E : M \rightarrow M'\).

### 3. FTS to PL-LTS Encoding

In this section, we provide an encoding from FTSs to PL-LTSs and thereby show that PL-LTSs are at least as expressive as FTSs.

**Definition 11** (FTS to PL-LTS Encoding). Consider a feature transition system such as \(\text{fts} = ([P, A, F, \rightarrow, \Lambda, p_{init})\). The PL-LTS resulting from the encoding, denoted by \(E(\text{fts})\) is a quintuple \((\bar{P}, A, I, \rightarrow, \bar{p}_{init})\), where:

- \(\bar{P} \cup \{p_0\} \subseteq [P \times \{L, R, ?\}]^1\), is the set of states, where \(p_0\) is a state not belonging to \(P\),
- \(A\) is the set of actions,
- \(I = |\Lambda| - 1\) is the index set,
- \(\bar{p}_{init} = (p_0, \{?\}^1)\) is the initial state,
Theorem 2. The class of PL-LTSs is at least as expressive as the class of FTSs.

Consider an arbitrary bijective function $X : \Lambda \to \{0, \ldots, |\Lambda| - 1\}$, where $|\Lambda| 

Theorem 1. Each PL-LTS resulted from encoding of a FTS, using the encoding given in Definition 11, satisfies the conditions in Definition 5.

Proof. Consider an arbitrary FTS $fts = (\vec{P}, A, F, \rightarrow, L, p_{init})$ and the PL-LTS resulted from the encoding, $E(fts) = (\vec{P}, A, L, \rightarrow, p_{init})$. The first condition in Definition 5, as is follows: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \implies v' = v''$. It is trivial to see that the first condition in Definition 5 holds, due to the construction of the transition relation in Definition 11.

Next, we consider the second condition in Definition 5, that is: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land v(i) \neq ? \implies v'(i) = v(i)$.

According to Definition 11, the transitions in $E(fts)$ are defined using the following two rules:

1. $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \in p_{init} \land \lambda \models \phi \implies (p_{init}, ?)^{\vec{A}} \xrightarrow{a_{v_i}} (Q,v')$

2. $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \notin p_{init} \land \lambda \models \phi \implies (P,v) \xrightarrow{a_{v_i}} (Q,v')$

If the transition is due to item 1., then $v(i) \neq ?$ cannot hold and hence this condition holds trivially. If the transition is due to rule 2., the configuration vector in the target state of a transition is the same as the configuration vector in the source state of the transition. Hence, the second condition in Definition 5 is satisfied.

Finally, we consider the third condition in Definition 5, that is: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land (P_{init}, v_{init}) \rightarrow (Q, v'_i) \land v(i) = ? \land v'_i(i) \neq ? \implies v'(i) = ? \implies (P_{init}, v_{init}) = (P, v)$. According to Definition 11, only the transitions emanating from the initial state of $E(fts)$ have source and target states with different configuration vectors. Since, $E(fts)$ has a single initial state, the third condition in Definition 11 is preserved by $E(fts)$.

4. Comparative Expressiveness

In this section, first we prove that the class of PL-LTSs is at least as expressive as the class of FTSs. Then given the results from [7], which shows that the class of FTSs is at least as expressive as the class of PL-LTSs, we conclude that the class of PL-LTSs is as expressive as the class of FTSs.

Theorem 2. The class of PL-LTSs is at least as expressive as the class of FTSs.

Consider an arbitrary bijective function $X : \Lambda \to \{0, \ldots, |\Lambda| - 1\}$, where $|\Lambda| - 1$. For each product configuration $\lambda \in \Lambda$, we define $v_\lambda$ to be the configuration vector such that $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (j < X(\lambda) \implies v_\lambda(j) = R) \land (j = X(\lambda) \implies v_\lambda(j) = L) \land (X(\lambda) < j \implies v_\lambda(j) = ?)$. Then, the transition relation is the smallest set satisfying the following two conditions:

$\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \in p_{init} \land \lambda \models \phi \implies (p_{init}, ?)^{\vec{A}} \xrightarrow{a_{v_i}} (Q,v')$

$\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \notin p_{init} \land \lambda \models \phi \implies (P,v) \xrightarrow{a_{v_i}} (Q,v')$

Example 4. An example of encoding an FTS into a PL-LTS is depicted in Fig. 4. In this figure, part (a) represents an FTS resulted from removing feature tea from the FTS in Fig. 2 and part (b) represents the PL-LTS resulted form the encoding.

As it can be seen the encoding results in a blow up in the size of the model. In the remainder of the paper, we show that for some FTSs, such exponential blow up in the size after encoding, regardless of the applied encoding, is unavoidable. Next, we show that the conditions in Definition 5 are satisfied by the PL-LTSs resulted after the encoding.

Theorem 1. Each PL-LTS resulted from encoding of a FTS, using the encoding given in Definition 11, satisfies the conditions in Definition 5.

Proof. Consider an arbitrary FTS $fts = (\vec{P}, A, F, \rightarrow, L, p_{init})$ and the PL-LTS resulted from the encoding, $E(fts) = (\vec{P}, A, L, \rightarrow, p_{init})$. The first condition in Definition 5, as is follows: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \implies v' = v''$. It is trivial to see that the first condition in Definition 5 holds, due to the construction of the transition relation in Definition 11.

Next, we consider the second condition in Definition 5, that is: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land v(i) \neq ? \implies v'(i) = v(i)$.

According to Definition 11, the transitions in $E(fts)$ are defined using the following two rules:

1. $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \in p_{init} \land \lambda \models \phi \implies (p_{init}, ?)^{\vec{A}} \xrightarrow{a_{v_i}} (Q,v')$

2. $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land P \notin p_{init} \land \lambda \models \phi \implies (P,v) \xrightarrow{a_{v_i}} (Q,v')$

If the transition is due to item 1., then $v(i) \neq ?$ cannot hold and hence this condition holds trivially. If the transition is due to rule 2., the configuration vector in the target state of a transition is the same as the configuration vector in the source state of the transition. Hence, the second condition in Definition 5 is satisfied.

Finally, we consider the third condition in Definition 5, that is: $\forall_{\vec{P},\vec{A},L,\rightarrow, p_{init}} (P,v) \xrightarrow{a_{v_i}} (Q,v') \land (P_{init}, v_{init}) \rightarrow (Q, v'_i) \land v(i) = ? \land v'_i(i) \neq ? \implies v'(i) = ? \implies (P_{init}, v_{init}) = (P, v)$. According to Definition 11, only the transitions emanating from the initial state of $E(fts)$ have source and target states with different configuration vectors. Since, $E(fts)$ has a single initial state, the third condition in Definition 11 is preserved by $E(fts)$.
This means the proof of the theorem can be reduced to proving the PL-LTS resulted from applying the encoding given in Definition 11 to Proof.

We show

Consider

\[ s \]

Assume an arbitrary LTS \( h \),

\[ \text{fts} \]

It suffices to show that each LTS \( h \) is also a valid implementation of \( E(\text{fts}) \), the PL-LTS resulted from applying the encoding given in Definition 11 to \( \text{fts} \), and vice versa, i.e., \( \text{fts} \rightarrow l \Leftrightarrow E(\text{fts}) \rightarrow l \).

This means the proof of the theorem can be reduced to proving \( \llbracket \text{fts} \rrbracket = \llbracket E(\text{fts}) \rrbracket \) (see Definition 9).

Consider \( \text{fts} = \langle \mathcal{P}, \mathcal{A}, F_\rightarrow, \mathcal{A}, p_{\text{init}} \rangle \) and \( E(\text{fts}) = \langle \mathcal{P}, \mathcal{A}, l \rightarrow, p_{\text{init}} \rangle \); we separate the bi-implication in the proof obligation into the following two implications:

We show \( \llbracket \text{fts} \rrbracket = \llbracket E(\text{fts}) \rrbracket \) by proving the following two proof obligations: \( \llbracket \text{fts} \rrbracket \subseteq \llbracket E(\text{fts}) \rrbracket \) and \( \llbracket E(\text{fts}) \rrbracket \subseteq \llbracket \text{fts} \rrbracket \).

- \( \llbracket \text{fts} \rrbracket \subseteq \llbracket E(\text{fts}) \rrbracket \): In order to prove \( \llbracket \text{fts} \rrbracket \subseteq \llbracket E(\text{fts}) \rrbracket \), we show that \( \forall l \in \text{LTS} : l \in \llbracket \text{fts} \rrbracket \Rightarrow l \in \llbracket E(\text{fts}) \rrbracket \).

Assume an arbitrary LTS \( l = (\mathcal{S}, \mathcal{A}, \rightarrow, s_{\text{init}}) \) s.t. \( l \in \llbracket \text{fts} \rrbracket \), which means that \( \text{fts} \rightarrow l \) (see Section 2.4). We prove
Consider an arbitrary pair of states in \( P \) and \( s \in S \), \((P, v) \vdash_{\theta} s \) holds if a product-derivation relation such as \( R_0 \) exists such that \((P, v)R_0 s \) and \( R_0 \) satisfies the following properties:

1. \( \forall P, Q, s, t, a \in \text{fts} \cdot (P, s) \overset{a}{\rightarrow} Q \land \lambda \models \phi \Rightarrow \exists \theta \cdot s \overset{a}{\rightarrow} t \land (Q, v') \vdash_{\theta} t \),

2. \( \forall P, Q, s, t, a \in \text{fts} \cdot (P, s) \overset{a}{\rightarrow} Q \land \lambda \models \phi \land Q, v' \vdash_{\theta} t \),

Hence, the next step we prove that such a relation exists and that the initial states are related by it.

Based on Definition 3, the assumption \( \text{fts} \vdash_{\theta} \) implies that for some \( \lambda \in \Lambda \) a product-derivation relation \( R_1 \subseteq P \times S \) exists which satisfies the following properties:

1. \( \forall P, s, Q, t, \lambda \in \Lambda \cdot \exists \theta \cdot (P, s) \overset{a}{\rightarrow} Q \land \lambda \models \phi \land Q, v' \vdash_{\theta} t \),

2. \( \forall P, s, Q, t, \lambda \in \Lambda \cdot \exists \theta \cdot (P, s) \overset{a}{\rightarrow} Q \land \lambda \models \phi \land Q, v' \vdash_{\theta} t \),

We define a family of binary relations \( R_0 \) (parameterised by configuration vectors in \( \Theta \)) such that:

\[
\forall P \in P, s \in S \cdot \exists \lambda \in \Lambda \cdot ((P, s) \vdash_{p_{\text{mut}}} \land (P, s) \vdash_{p_{\text{mut}}} s \vdash_{\theta} = v_j) \land \\
((P, s) \vdash_{p_{\text{mut}}} \land (P, s) \vdash_{p_{\text{mut}}} s \vdash_{\theta} = v_j)
\]

Assume that LTS \( l \) derives product \( \lambda \) from FTS \( fts \); Let \( \theta = v_j \) and \( R_0 \) be a member of the above defined family of relations. Next, we prove that \( R_0 \) satisfies the properties of a product-derivation relation (statements (1) and (2)).

Consider an arbitrary pair of states in \( R_0 \), say \((P, v)R_0 s \) based on the definition given above for \( R_0 \), it holds \( P, s \) where we distinguish the two following cases: \((P, v) = p_{\text{mut}} \) and \((P, v) \neq p_{\text{mut}}\).

- First, we prove that statement (1) is satisfied by \( R_0 \).

\[ * (P, v) = p_{\text{mut}}. \]

Thus \( v = (\lnot v_j) \) and \( P = p_0 \) (see Definition 11). Consider an arbitrary transition emanating from \((p_0, (\lnot v_j))\) of the form \( (p_0, (\lnot v_j)) \overset{a}{\rightarrow} (Q, v') \); based on Definition 11, such a transition is resulted from encoding of one of the outgoing transitions from \( P \), i.e.:

\[
\forall a \in A \cdot (p_0, (\lnot v_j)) \overset{a}{\rightarrow} (Q, v') \Rightarrow \exists \phi \cdot P \overset{\phi}{\rightarrow} Q \land \lambda \models \phi \land P \in p_{\text{mut}}
\]  

(1.i)

Considering property (a) satisfied by the relation \( R_0 \), \((P, v)R_0 s \) implies that the following statement holds:

\[
\forall a \in A \cdot (P, v) \overset{a}{\rightarrow} Q, s \vdash_{\theta} \phi \Rightarrow \exists \theta - s \overset{a}{\rightarrow} t \land Q, v_j \vdash_{\theta} t
\]  

(1.ii)

Based on the definition of \( R_0 \), \( Q, v_j \vdash_{\theta} t \land v_j \vdash_{\theta} v_j \) holds only in case that \( x \neq \lambda \). Given that \( \theta = v_j \), based on the definition of the relation \( \subseteq \) (see Definition 6) it holds \( v_j \subseteq \theta \) (notice that for any \( x \in \Lambda \) such that \( x \neq \lambda \), based on the definition of \( v_j \) it holds \( v_i \vdash_{v_j} v_i \) and hence, \( v_j \subseteq \theta \). Thus, from (1.i) and (1.ii), the following statement is derived:

\[
\forall a \in A \cdot (p_0, (\lnot v_j)) \overset{a}{\rightarrow} (Q, v_j) \
\]

(1.iii)
Next, we prove that statement (2) is satisfied by which means Considering (1) Furthermore, consider property (b) satisfied by relation of the form \( (P, v_i) \rightarrow (Q, v_i) \) is \( v_j \) and such transition is resulted from encoding of one of the outgoing transitions from \( P \) in the FTS \( \beta_i \), i.e.:

\[
\forall_{\Delta P} \cdot (P, v_j) \xrightarrow{\Delta P} (Q, v_j) \iff \exists \phi \cdot P \xrightarrow{\phi} Q \land \lambda \models \phi \land P \notin p_{init} \tag{1.ii}
\]

Considering property (a) satisfied by the relation \( \mathcal{R}_i \), \( \mathcal{P} \mathcal{R}_i \mathcal{s} \) implies that the following statement holds:

\[
\forall_{\Delta P} \cdot (P, v_j) \xrightarrow{\Delta P} (Q, v_j) \land \forall t \Rightarrow \exists t \cdot s \xrightarrow{t} t \land Q \mathcal{R}_i t \tag{1.iii}
\]

Based on the definition of \( \mathcal{R}_0 \), \( Q \mathcal{R}_t \mathcal{t} \Rightarrow (Q, v_j) \mathcal{R}_0 t \land \theta = v_j \). Using the same reasoning as in previous case, from (1.iii) and (1.ii), the following statement is derived:

\[
\forall_{\Delta P} \cdot (P, v_j) \xrightarrow{\Delta P} (Q, v_j) \land v_j \subseteq \theta \Rightarrow \exists t \cdot s \xrightarrow{t} t \land (Q, v_j) \mathcal{R}_0 t \tag{1.iv}
\]

Considering (1.iii) and (1.ii), the following statement holds:

\[
\forall_{\Delta P, \Delta Q} \cdot ((P, v) \mathcal{R}_0 s \land (P, v) \xrightarrow{\Delta P} (Q, v') \land v' \subseteq \theta) \Rightarrow \exists t \cdot s \xrightarrow{t} t \land (Q, v') \mathcal{R}_0 t,
\]

which means \( \mathcal{R}_0 \) satisfies statement (1).

Next, we prove that statement (2) is satisfied by \( \mathcal{R}_0 \).

* \((P, v) \notin p_{init}\).

Thus, \( v = \{?\} \) and \( P = p_0 \) (see Definition 11):

Consider an arbitrary transition emanating from \((p_0, \{?\})\) of the form \((p_0, \{?\}) \xrightarrow{\Delta P} (Q, v_j)\); based on Definition 11, such a transition is resulted from encoding of an outgoing transition from \( P \), i.e.:

\[
\forall_{\Delta Q} \cdot (p_0, \{?\}) \xrightarrow{\Delta P} (Q, v_j) \iff \exists \phi \cdot P \xrightarrow{\phi} Q \land \lambda \models \phi \land P \in p_{init} \tag{2.i}
\]

Considering property (b) satisfied by the relation \( \mathcal{R}_i \), \( \mathcal{P} \mathcal{R}_i \mathcal{s} \) implies that the following statement holds:

\[
\forall_{\Delta t} \cdot (s \xrightarrow{t} t) \Rightarrow \exists \Delta Q \cdot P \xrightarrow{\Delta Q} Q \land \lambda \models \phi \land Q \mathcal{R}_i t \tag{2.ii}
\]

Based on the definition of \( \mathcal{R}_0 \), \( Q \mathcal{R}_t \mathcal{t} \Rightarrow (Q, v_j) \mathcal{R}_0 t \land \theta = v_j \). Hence, \((Q, v_j) \mathcal{R}_0 t \) holds only in case that \( \theta = v_j \). Given that \( \theta = v_j \), based on the definition of the relation \( \mathcal{R} \) (see Definition 6) it holds \( v_j \subseteq \theta \) and for all \( \lambda' \in \mathcal{C} \) such that \( \lambda \neq \lambda' \) it holds \( v_j \not\subseteq \theta \). Considering (2.i) and (2.ii), the following statement holds:

\[
\forall_{\Delta t} \cdot s \xrightarrow{t} t \Rightarrow \exists \Delta Q \cdot q \cdot (p_0, \{?\}) \xrightarrow{\Delta Q} (Q, v_j) \land v_j \subseteq \theta \land (Q, v_j) \mathcal{R}_0 t \tag{2.iii}
\]

* \((P, v) \notin p_{init}\).

Thus, (based on the definition of \( \mathcal{R}_0 \)) \( v = v_j \). Consider an arbitrary transition emanating from \((P, v_j)\) of the form \((P, v_j) \xrightarrow{\Delta P} (Q, v_j)\), (based on the Definition 11, the configuration vector in the target state of the outgoing transitions from \((P, v_j)\) is \( v_j \); such a transition is resulted from encoding of an outgoing transition from \( P \), i.e.:

\[
\forall_{\Delta Q} \cdot (P, v_j) \xrightarrow{\Delta Q} (Q, v_j) \iff \exists \phi \cdot P \xrightarrow{\phi} Q \land \lambda \models \phi \land P \notin p_{init} \tag{2.iv}
\]

Furthermore, consider property (b) satisfied by relation \( \mathcal{R}_i \); \( \mathcal{P} \mathcal{R}_i \mathcal{s} \) implies that the following statement holds:
Next, we define a family of binary relations with the following properties.

\[ \forall s, t \cdot (s \xrightarrow{a} t) \Rightarrow \exists Q, \phi \cdot P \xrightarrow{a} Q \land \phi \land Q \mathcal{R}_I t \]  

(2.v) Given the definition of \( \mathcal{R}_0, Q, \mathcal{R}_I, t \Rightarrow (Q, \nu_I) \mathcal{R}_I t \land \theta = \nu_J. \) Since, \( \theta = \nu_J \) and based on Definition 6 it holds \( \nu_J \subseteq \theta. \) Considering (2.iv) and (2.v), the following statement holds:

\[ \forall a, t \cdot s \xrightarrow{a} t \Rightarrow \exists a, Q \cdot (P, s) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta \land (Q, \nu_J) \mathcal{R}_I t \]  

(2.vi) Considering (2.iii) and (2.vi), the following statement holds:

\[ \forall_{P, R, s} \cdot ((P, v) \mathcal{R}_0 s \land s \xrightarrow{a} t) \Rightarrow \exists Q, \phi \cdot (P, v) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta \land (Q, \nu_J) \mathcal{R}_0 t \]

which means that \( \mathcal{R}_0 \) satisfies the second property of a product-derivation relation that is statement (2).

Based on the assumption \( fts \rhd l, \) it holds \( p_{\text{init}} \rhd s_{\text{init}}. \) As shown above, \( \mathcal{R}_0 \) satisfies the properties of a product-derivation relation given in Definition 8. Hence, based on the definition of \( \mathcal{R}_0 \) it holds that \( p_{\text{init}} \rhd s_{\text{init}} \Rightarrow (p_0, \{v\}) \rhd t_{\text{init}}. \) Thus, \( p_{\text{init}} \rhd s_{\text{init}} \). This means \( E(fts) < l \) and subsequently \( l \in \llbracket E(fts) \rrbracket \).

Hence, we conclude that \( \llbracket fts \rrbracket \subseteq \llbracket E(fts) \rrbracket \).

- \( \llbracket E(fts) \rrbracket \subseteq \llbracket fts \rrbracket. \)

Proof. In order to prove \( \llbracket E(fts) \rrbracket \subseteq \llbracket fts \rrbracket, \) we show that \( \forall l \in \text{LTS} \cdot l \in \llbracket E(fts) \rrbracket \Rightarrow l \in \llbracket fts \rrbracket. \)

Consider an arbitrary LTS \( l = (S, A, \rightarrow, s_{\text{init}}), s.t., l \in \llbracket E(fts) \rrbracket \) and subsequently \( E(fts) < l \) (see Definition 10).

We prove \( fts \rhd l \) and hence, \( l \in \llbracket fts \rrbracket. \)

To prove \( fts \rhd l, \) it suffices to show that for some product configuration \( \lambda \in \Lambda \) the following statement holds:

\( s_{\text{init}} \rhd_{PL-LTS} \lambda, \) (see Section 2.2).

Next, we show that the above statement is satisfied by \( l \) and \( fts. \) According to Section 2.2, \( P \rhd s \) holds if a product-derivation relation such as \( \mathcal{R}_I \) exists such that \( P \mathcal{R}_I s \) and \( \mathcal{R}_I \) satisfies the following properties:

1. \( \forall_{P, R, s} \cdot ((P, v) \mathcal{R}_0 s \land (P, v) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta) \Rightarrow \exists \theta, t \cdot s \xrightarrow{a} t \land (Q, \nu_J) \mathcal{R}_I t. \)
2. \( \forall_{P, R, s} \cdot ((P, v) \mathcal{R}_0 s \land s \xrightarrow{a} t) \Rightarrow \exists Q, \phi \cdot (P, v) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta \land (Q, \nu_J) \mathcal{R}_0 t. \)

Hence, in the next step we prove the existence of a relation between states of \( l \) and \( fts \) that satisfies the above properties.

Based on Definition 8, the assumption \( E(fts) < l \) implies that for some \( \theta \in \Theta, \) a product-derivation relation \( \mathcal{R}_I \subseteq \mathcal{E} \times \mathcal{S} \) exists, which satisfies the following properties:

1. \( \forall_{P, R, s} \cdot ((P, v) \mathcal{R}_0 s \land (P, v) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta) \Rightarrow \exists \theta, t \cdot s \xrightarrow{a} t \land (Q, \nu_J) \mathcal{R}_I t. \)
2. \( \forall_{P, R, s} \cdot ((P, v) \mathcal{R}_0 s \land s \xrightarrow{a} t) \Rightarrow \exists Q, \phi \cdot (P, v) \xrightarrow{a} (Q, \nu_J) \land \nu_J \subseteq \theta \land (Q, \nu_J) \mathcal{R}_0 t. \)

Next, we define a family of binary relations \( \mathcal{R}_I \) (parameterised by product configurations in \( \Lambda \)) such that:

\[ \forall_{P, R, s} \cdot ((P_0, \{v\}) \mathcal{R}_0 s \Rightarrow ((P, v) \mathcal{R}_0 s \land (P_0, \{v\}) \Rightarrow \mathcal{R}_I s \land s \xrightarrow{a} t) \land (P, v) \mathcal{R}_I s \land s \xrightarrow{a} t) \]

Assume that LTS \( l \) is derived from PL-LTS \( E(fts) \) with regards to the product configuration vector \( \theta = \nu_J. \) Let \( \mathcal{R}_I \) be a member of the above defined family. Next, we prove that \( \mathcal{R}_I \) satisfies the properties of a product-derivation relation (statements (1) and (2)).
For the sake of clarity we distinguish the following two cases:

First, we consider the case where $P \in p_{\text{out}}$:

Based on Definition 11, each transition emanating from $P$ such as $P \xrightarrow{\delta_{in}} Q$, is encoded as a transition in PL-LTS $E(\mathit{fts})$, i.e.:

$$\forall_{e,Q,t} \cdot P \xrightarrow{\delta_{in}} Q \land P \in p_{\text{out}} \land x' \models \phi \Leftrightarrow (p_{\text{in}}, \{\gamma\}) \xrightarrow{\mathit{a}, \gamma_{x'}} (Q, \nu_{x}) \quad (1.i)$$

Considering property (a) satisfied by relation $R_{\mathit{f}}, (p_{\text{in}}, \{\gamma\}) R_{\mathit{f}} s$ implies that the following statement holds:

$$\forall_{e,Q,t} \cdot ((p_{\text{in}}, \{\gamma\}) \xrightarrow{\mathit{a}, \gamma_{x'}}, \nu_{x'} \subseteq \emptyset) \Rightarrow \exists_{e} \cdot s \xrightarrow{a} t \land (Q, \nu_{x}) R_{\mathit{f}} t \quad (1.ii)$$

Based on the definition of $R_{\mathit{f}}, (Q, \nu_{x}) R_{\mathit{f}} t \Leftrightarrow Q R_{\mathit{f}} t \land \nu_{x} = \emptyset$. Hence, $Q R_{\mathit{f}} t$ only holds in case that $x' = \emptyset$. Thus, from (1.ii) and (1.iii), the following statement is derived:

$$\forall_{e,Q,t} \cdot (P \xrightarrow{\delta_{in}} Q \land \lambda \models \phi) \Rightarrow \exists_{e} \cdot s \xrightarrow{a} t \land Q R_{\mathit{f}} t \quad (1.iii)$$

Next, we assume $P \notin p_{\text{out}}$:

Based on Definition 11, each transition emanating from $P$ such as $P \xrightarrow{\delta_{in}} Q$, is encoded as a transition in PL-LTS $E(\mathit{fts})$, i.e.:

$$\forall_{e,Q,t} \cdot P \xrightarrow{\delta_{in}} Q \land P \notin p_{\text{out}} \land \lambda \models \phi \Leftrightarrow (p_{\text{in}}, \{\gamma\}) \xrightarrow{\mathit{a}, \gamma_{x'}} (Q, \nu_{x}) \quad (1.iv)$$

Considering property (a) satisfied by the relation $R_{\mathit{f}}, (P, \nu_{x}) R_{\mathit{f}} s$ implies that the following statement holds:

$$\forall_{e,Q,t} \cdot ((P, \nu_{x}) R_{\mathit{f}} s \land (P, \nu_{x}) \xrightarrow{\mathit{a}, \gamma_{x'}} (Q, \nu_{x}) \land \nu_{x} \subseteq \emptyset) \Rightarrow \exists_{e} \cdot s \xrightarrow{a} t \land (Q, \nu_{x}) R_{\mathit{f}} t \quad (1.v)$$

Based on the definition of $R_{\mathit{f}}, (Q, \nu_{x}) R_{\mathit{f}} t \Leftrightarrow Q R_{\mathit{f}} t \land \nu_{x} = \emptyset$. Thus, from (1.iv) and (1.v), the following statement is derived:

$$\forall_{e,Q,t} \cdot (P R_{\mathit{f}} s \land P \xrightarrow{\delta_{in}} Q \land \lambda \models \phi) \Rightarrow \exists_{e} \cdot s \xrightarrow{a} t \land (Q, \nu_{x}) R_{\mathit{f}} t \quad (1.vi)$$

Considering (1.iii) and (1.iii), the following statement holds:

$$\forall_{e,Q,t} \cdot (P R_{\mathit{f}} s \land P \xrightarrow{\delta_{in}} Q \land \lambda \models \phi) \Rightarrow \exists_{e} \cdot s \xrightarrow{a} t \land Q R_{\mathit{f}} t,$$

which means the relation $R_{\mathit{f}}$ satisfies statement (1).

Next, we prove that $R_{\mathit{f}}$ satisfies statement (2).

Considering an arbitrary pair of states in $R_{\mathit{f}}$, such as $P \parallel R_{\mathit{f}} s$. Based on the definition given for $R_{\mathit{f}}$, it holds $(p_{\text{in}}, \nu_{x}) R_{\mathit{f}} s$, where $\nu = \nu_{x}$ and $\nu' = \emptyset$. If $(P, \nu) R_{\mathit{f}} s$ where $\nu = \nu_{x}$ and $\nu = \nu_{x} \cup \nu_{x}'$ if $P \notin p_{\text{out}}$. For the sake of clarity we distinguish the following two cases: $P \in p_{\text{out}}$ and $P \notin p_{\text{out}}$.

First, we consider the case where $P \in p_{\text{out}}$:

Based on the Definition 11, each transition emanating $P$, such as $P \xrightarrow{\delta_{in}} Q$ is encoded as an outgoing transition from $(p_{\text{in}}, \{\gamma\})$, i.e.:

$$\forall_{e,Q,t} \cdot P \xrightarrow{\delta_{in}} Q \land P \in p_{\text{out}} \land x' \models \phi \Leftrightarrow (p_{\text{in}}, \{\gamma\}) \xrightarrow{\mathit{a}, \gamma_{x'}} (Q, \nu_{x}) \quad (2.i)$$

Considering property (b) of relation $R_{\mathit{f}}, (p_{\text{in}}, \{\gamma\}) R_{\mathit{f}} s$ implies that the following statement holds:
\[ \forall a_x, \exists s \overset{a}{\rightarrow} t \implies \exists Q, v_x, \exists (p_0, [l]) \overset{a_x}{\rightarrow} (Q, v_x) \land v_x \sqsubseteq \theta \land (Q, v_x) R_\theta t \]  

Based on the definition of \( R_{\theta} \), \((Q, v_x) R_\theta t \iff Q R_{\theta} t \land v_x = \theta \). Hence, \( Q R_{\theta} t \) holds only in case that \( \lambda = \theta \). Thus, from (2.i) and (2.ii), the following statement is derived:

\[ \forall a_x, \exists s \overset{a}{\rightarrow} t \implies \exists Q, \lambda \in \phi \land Q R_{\theta} t \]  

* Next, we assume \( P \notin p_{\text{init}} \).

Based on the Definition 11, each transition emanating \( P \), such as \( P \overset{a_x}{\rightarrow} Q \) is encoded as an outgoing transition from \( P \), i.e.:

\[ \forall a_x, Q, \exists s \overset{a}{\rightarrow} t \implies \exists Q, \lambda \in \phi \land (P, v_x) \overset{a_x}{\rightarrow} (Q, v_x) \]  

Considering property (b) satisfied by the relation \( R_{\phi} \), \((P, v_x) R_{\phi} s \) implies that the following statement holds:

\[ \forall a_x, \exists s \overset{a}{\rightarrow} t \implies \exists Q, v_x, (P, v_x) \overset{a_x}{\rightarrow} (Q, v_x) \land v_x \sqsubseteq \theta \land (Q, v_x) R_\theta t \]  

Based on the definition of \( R_{\phi} \), \((Q, v_x) R_\theta t \iff Q R_{\theta} t \land v_x = \theta \). Thus, from (2.iv) and (2.v), the following statement is derived:

\[ \forall a_x, \exists s \overset{a}{\rightarrow} t \implies \exists Q, \lambda \in \phi \land Q R_{\theta} t \]  

Considering two derived statements (2.iii) and (2.vi), the following statement holds:

\[ \forall P_{a_x}, \exists s \overset{a}{\rightarrow} t \implies \exists Q, \lambda \in \phi \land Q R_{\theta} t \]

Hence, we conclude that \( R_{\phi} \) satisfies the second property of product derivation relation that is statement (2).

The assumption \( E(fts) < \ell \) implies that \( p_{\text{init}} \not\in s_{\text{init}} \). Based on the above proof, \( R_{\phi} \) satisfies the properties of a product derivation relation given in Definition 3. Hence, based on the definition of \( R_{\phi} \) it holds that \((p_0, [l]) \not\in s_{\text{init}} \implies p_{\text{init}} \not\in s_{\text{init}} \). This means \( fts \not\in \ell \) and subsequently \( \ell \in fts \).

Hence, we conclude that \( \|E(fts)\|' \subseteq fts \).

\[ [Q.E.D] \] As it is shown above \( fts \subseteq \|E(fts)\|' \) and \( E(fts) \subseteq \|fts\| \). Thus, \( \|fts\| = \|E(fts)\|' \), which means the class of PL-LTSs is at least as expressive as the family of FTSs.

5. Succinctness Analysis

In this section, we provide an analysis of the succinctness (number of states and the configuration vector size included in the states) of PL-LTSs resulted from encoding of an FTS. Intuitively, we prove that for some FTSs the size of the PL-LTS which is resulted from any sound encoding, is exponential in terms of the number of states of the FTS. In the rest of this section, we assume that \( \Xi \) denotes any sound encoding from the class of FTSs FTS into the class of PL-LTSs PL-LTSs. We consider the FTS \( \Xi(fts) \) represented in Fig. 5. In this FTS, in each state such as \( s_l \) there is a variant choice between features \( f_x \) and \( f_y \). We assume \( E(fts) = (\hat{P}, A, I, \rightarrow, p_{\text{init}}) \) is the PL-LTS resulted from encoding of \( fts \) using an arbitrary encoding \( E \in E \). The FTS \( fts \) has \( 2^n \) non-trace equivalent LTS implementations each of which has exactly one path. We assume \( imp \) denotes the set of all such implementing LTSs.

First, we prove the following statement which is used to compute the least possible size of the configuration vector in the states of \( E(fts) \), i.e., \( \Xi(fts) \). Consider two distinct LTS implementations derived from the PL-LTS \( E(fts) \); at least one state in \( E(fts) \) exists such that each of the considered LTSs implements a distinct outgoing transition from that state. This in turn means that for each two distinct valid products (implementations) of the above mentioned
model, there should be at least one configuration vector corresponding to each of these products in the PL-LTS which is refined by that product’s vector such that these configuration vectors are conflicting in at least one bit. Thus, we formulate the following lemma.

**Lemma 1.** Assuming two LTSs, \( l_1, l_2 \in \text{Imp} \), such as \( l_1 = (S_1, A, \rightarrow_1, s^1_0) \) and \( l_2 = (S_2, A, \rightarrow_2, s^2_0) \), where \( \overrightarrow{\text{pref}} \! \neg \! \rightarrow_0 \! \rightarrow_1 s^1_0 \) and \( \overrightarrow{\text{pref}} \! \neg \! \rightarrow_0 \! \rightarrow_1 s^2_0 \). It holds that:

\[
\begin{align*}
\exists_{\overrightarrow{\text{pref}}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \\
\exists_{\overrightarrow{\text{pref}}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2)
\end{align*}
\]

**Proof.** Assuming that \( \text{Paths}(l_1) = \rho_1, \text{Paths}(l_2) = \rho_2, \Sigma = \text{pref}(\text{Trace}(\rho_1)) \cup \text{pref}(\text{Trace}(\rho_2)) \), where \( \text{pref} \) denotes the set of the finite prefixes of a sequence. We consider \( \sigma \in \Sigma \) such that \( \exists_{\text{pref}} \Leftrightarrow \vert \sigma \vert < \vert [\sigma] \vert \). Assume \( [\sigma] = k \); let \( \rho_1(k) = s^1_k \) and \( \rho_2(k) = s^2_k \), given that \( l_1, l_2 \in \text{Imp} \) are distinct it holds that: \( s^1_k \xrightarrow{\alpha} s^1_{k+1} \) and \( s^2_k \xrightarrow{\beta} s^2_{k+1} \), where \( \alpha \neq \beta \). Based on the condition (2) in Definition 8, it holds:

\[
\begin{align*}
\exists_{\text{pref}} & (P_1, v_1) \xrightarrow{\alpha} (Q_1, v'_1) \wedge v'_1 \not\subseteq \theta_1 \\
\exists_{\text{pref}} & (P_2, v_2) \xrightarrow{\beta} (Q_2, v'_2) \wedge v'_2 \not\subseteq \theta_2,
\end{align*}
\]

Given that \( l_1, l_2 \in \text{Imp} \) it holds \( [\text{Out}(s^1_k)] = 1 \) and \( [\text{Out}(s^2_k)] = 1 \); hence \( v'_1 \not\subseteq \theta_2 \) and \( v'_2 \not\subseteq \theta_1 \). Thus, it can be concluded that:\

\[
\begin{align*}
\exists_{\text{pref}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \\
\exists_{\text{pref}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2)
\end{align*}
\]

Next, we provide a lower bound for the size of the configuration vector in the states of the PL-LTSs resulted from encoding the FTSs represented in Fig. 5.

**Lemma 2.** Let \( E \in \mathcal{E} \) be an arbitrary encoding. The size of the configuration vector included in the states of \( E(\text{ft}_{\text{ts}}) \) (i.e., \( \Omega(| | |) \)) is at least \( n \).

**Proof.** Consider two LTSs \( l_1, l_2 \in \text{Imp} \), such as \( l_1 = (S_1, A, \rightarrow_1, s^1_0) \) and \( l_2 = (S_2, A, \rightarrow_2, s^2_0) \), where \( \overrightarrow{\text{pref}} \! \neg \! \rightarrow_0 \! \rightarrow_1 s^1_0 \) and \( \overrightarrow{\text{pref}} \! \neg \! \rightarrow_0 \! \rightarrow_1 s^2_0 \). According to Lemma 1, it holds that:

\[
\begin{align*}
\exists_{\overrightarrow{\text{pref}}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \\
\exists_{\overrightarrow{\text{pref}}} & (P, v) \xrightarrow{\alpha} (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2)
\end{align*}
\]

which means for any two arbitrary LTSs \( l_1, l_2 \in \text{Imp} \) it holds that: \( \exists_{\overrightarrow{\text{pref}}} \! \neg \! \rightarrow_0 \! \rightarrow_1 \text{last}(\rho) = (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \) and \( \exists_{\overrightarrow{\text{pref}}} \! \neg \! \rightarrow_0 \! \rightarrow_1 \text{last}(\rho) = (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \). Thus, \( \exists_{\overrightarrow{\text{pref}}} \! \neg \! \rightarrow_0 \! \rightarrow_1 \text{last}(\rho) = (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \). Recall that \( \equiv \{ (L, R), (R, L) \} \). This means \( \forall_{\theta_1, \theta_2, \exists_{\overrightarrow{\text{pref}}} \! \neg \! \rightarrow_0 \! \rightarrow_1 \text{last}(\rho) = (Q, v') \wedge (v' \not\subseteq \theta_1 \wedge v' \not\subseteq \theta_2) \). Hence, for each two products selected from \( \text{Imp} \), there are two states in the PL-LTS that the configuration vectors in these states are conflicting. As there are \( |\text{Imp}| = 2^n \), the minimum size of the configuration vector included in the state of the PL-LTS is \( \log(2^n) = n \).

Next, we prove the following theorem regarding the complexity of the PL-LTSs resulted from encoding of FTSs.
Lemma 3. Considering the class of all possible encodings from FTSs into PL-LTSs, denoted by E. There exists an FTS such that the size of the encoded PL-LTS (the number of states) is exponential in the number of the states in that FTS, regardless of which encoding is selected.

Proof. Let $E \in E$ be an arbitrary encoding and $E(fts) = (\bar{P}, A, I, \rightarrow, \bar{p}_{init})$ be the PL-LTS resulted from the encoding. Consider two distinct LTSs, $l_1, l_2 \in Imp$, such as $l_1 = (S_1, A, \rightarrow_1, s_0^1)$ and $l_2 = (S_2, A, \rightarrow_2, s_0^2)$, where $3_{\bar{p}_{init}} \cdot p \cdot \bar{v}_0 \cdot s_0^1$ and $3_{\bar{p}_{init}} \cdot p' \cdot \bar{v}_0 \cdot s_0^2$; according to Lemma 1, it holds that:

$$\exists_{\bar{P}, Q_v, s'}, \ (P, v) \xrightarrow{s'} (Q, v') \land (v' \subseteq \theta_1 \land v' \subseteq \theta_2) \land$$

$$\exists_{\bar{P}, Q_v, s'} \ (P, v) \xrightarrow{s'} (Q, v') \land (v' \subseteq \theta_1 \land v' \subseteq \theta_2).$$

Hence, for each two arbitrary LTSs $l_1, l_2$ it holds that $\exists_{\bar{Q_v}, v'}, \ (Q, v') \rightarrow_1 v' \Rightarrow v' \rightarrow_2$. As there are $|Imp| = 2^n$ it holds that the size of the set of states in $E(fts)$ is at least $2^n$.

Hence, we conclude that the total number of the states in $E(fts)$ is exponential in terms of the number of states in $fts$. \hfill \square

Next, we prove that an encoding from PL-LTSs into FTSs exists such that for any PL-LTS the size of the PL-LTS is polynomial in terms of the size of the FTS.

Theorem 3. An encoding $E \in E$ from PL-LTSs into FTSs exists such that for any PL-LTS $P$, the size of the model resulting from encoding of $P$ is polynomial in terms of the size of $P$, i.e., $|E(P)| = O(k|P'|)$ where $k, r \in \mathbb{N}$ and $|I|$ is used to represent the size of the models.

Proof. Let $(P \times \{L, R\}, A, \rightarrow, P_{init})$ be an arbitrary PL-LTS. We consider the encoding given in [7]. The corresponding FTS is denoted by $(\bar{P} \times \{L, R\}, A, F, \rightarrow, F_{init})$, where:

- $F = \bigcup_{i \in I} \{L_i, R_i\}$.
- $\Lambda = \bigcup_{i \in I} \{L_i \land R_i\}$.
- The transition relation $\rightarrow$ is defined in the following way:

$$\begin{align*}
(P, v) \xrightarrow{a} (Q, v) & \rightarrow \ (P, v) \xrightarrow{a'} (Q, v') \land \phi = \bigwedge_{i \in \Xi(i, v, v')} \\
(P, v) \xrightarrow{a} (Q, v) & \rightarrow \ (P, v) \xrightarrow{a'} (Q, v')
\end{align*}$$

where $\Xi(i, v, v') \iff v'(i) \neq v(i)$.

- $F_{init} = P_{init}$.

For any $(P, v) \in P \times \{L, R\}$, we fix $E(P, v) = (P, v)$. Considering the transition relation above, the result of encoding each transition $(P, v) \xrightarrow{a} (Q, v')$, for either $v = v'$ or $v \neq v'$, is one transition in the FTS. Hence, it is trivial to see that the size of the FTS resulted from the encoding remains polynomial (to be more precise it holds r=1 in the above complexity formula) in terms of the size of the original PL-LTS. \hfill \square

6. Related Work

In this section, we divide the related work into two main parts. First, we mention some work on modeling software product lines. In general, the conventional formal models such as labeled transition systems, which are also used for modeling single systems can be exploited for modeling the behavior of products in a software product line. Using such approach, the behavior of each product is modeled individually, which is not efficient and in case of product lines with a large number of products, it could be very costly.

In order to model software product lines more efficiently, specific semantic notions are required which can refer to variation points in the behavior of products and capture their behavioral differences.
There have been several attempts to extend the existing formalisms to model software product lines. We give an overview of some of these alternatives proposed as fundamental models of software product lines in the following.

The first alternative are modal transition systems [3], an extension of LTSs in which the set of transitions are partitioned into may and must transitions. Fischbein et al. [12] argues that these models are adequate for modeling software product lines. Since, in these models the may transitions represent optional behavior and most transitions are used for modeling mandatory part of the behavior, each MTS can have several implementations which represent the behavior of different products.

Furthermore, several researchers [13–17] have exploited modal transition systems as formal models to perform rigorous analysis of software product lines. There has been some work such as [13–15], which address the deriving valid products from a given MTS by model checking against formulae expressed in a deontic logic called Modal-Hennessy-Milner-Logic (MHML). Also, there are some works which have developed an interface theory and a testing theory for SPLs [16, 17].

In [1, 18], Classen et. al. enriches the notion of LTSs by annotating the transitions with features. The annotated model is called featured transition system (FTS), which comprises the behavior of all products in a software product line. The model of each product can be derived from the FTS of the corresponding product line. The presence or absence of the transitions in the model of each products is indicated by the feature that is labeled on the transition. If the feature is included in the product then the transition is included in the model of the product line. Also, a model checking algorithm for checking LTL formula against featured transition system was given in [1]. Cordy et al. [19] extended the earlier work [1, 18] by combining non-boolean features and multi-features in a high-level specification language called TVL+. An algorithm for constructing an FTS from a behavioral specification written in TVL+ was also given.

The next alternative is PL-CCS [2], which is an extension of Milner’s CCS [8], introduced by Gruler et al. [2]. In this extension of CCS a new operator, the “binary variant” denoted by $\oplus$, is introduced using which the alternative choices can be modelled. Same as MTSs, the validity of products can be asserted using model checking formulae specified in a multi-valued modal $\mu$-calculus [20].

The expressiveness of above mentioned fundamental models have been compared in [7]. In this work, the concept of encoding between these formalisms is defined and the comparison of the expressiveness is built upon the existence of the encodings. In this paper, we fuse the given definition for encoding and follow similar approach for comparing the expressiveness of the formalisms.

Also in [4], two variants of modal transition systems, namely disjunctive modal transition systems (DMTSs) [21], and 1MTSs [4] are compared from the expressive point of view. DMTSs are variants of MTSs in which hyper transitions, transitions with multiple target states, are featured. Two types of Hyper transitions are considered: mustmay hyper transitions. In DMTSs, the must hyper transitions represent an OR relation between mandatory multiple choices and may hyper transition represents an OR relation between optional multiple choices. The 1MTSs are similar to DMTSs in that they also feature hyper transitions. The difference is in the interpretation of the choices. In 1MTSs, the must hyper transitions represent an XOR relation between mandatory multiple choices and may hyper transition represents an XOR relation between optional multiple choices. In [4] it is shown that the two formalisms have the same expressive power, i.e., the induce the same sets of labeled transition systems as their implementations.

7. Conclusion

In this paper, we compared the expressiveness of the product line calculus of communicating systems and featured transition systems. In this work we used a more allowing definition for PL-LTSs (which are considered as the semantics domain for PL-CCS terms) in comparison to our previous work [7]. To this end, we described an encoding from the class of feature transition systems into the class of product line labeled transition systems. Then, we provided a proof to show that the set of labeled transition systems that implement an FTS, are also valid implementations of the PL-LTS resulted from encoding of the FTS and vice versa. As a result, we conclude that the class of PL-LTSs is at least as expressive as the class of FTSs. Given this result and considering that the class of FTSs is also at least as expressive as the class of PL-LTSs [7]; we concluded that the class of PL-LTSs is as expressive as the class of FTSs. We also provided a succinctness analysis of the models resulted from encoding. The results show that the size of the PL-LTS resulted from encoding an FTS is exponential in terms of the number of the states of the FTS.
Both in the present paper and in [7], we have only considered models with finite behavior; considering infinite behavior in our study of comparative expressiveness is among the future work that we aim to pursue. Completing the lattice of expressive power given in [7], by including the results of comparing the expressiveness of other formalisms, such as 1MTSs [4], DMTSs [21] and PMTSs [6] with formalisms included in the lattice, is another avenue for our future work.

8. References


Appendix D

Paper IV
Complete IOCO Test Cases: A Case Study

Sofia Costa Paiva, Adenilso Simao, Mahsa Varshosaz, and Mohammad Reza Mousavi

2016 Workshop of Automated Testing (A-TEST)
Seattle, USA, November 18, 2016
Complete IOCO Test Cases: A Case Study

Sofia Costa Paiva  
Instituto de Ciências Matemáticas e de  
Computação  
Universidade de São Paulo  
São Carlos, Brazil

Adenilso Simao  
Instituto de Ciências Matemáticas e de  
Computação  
Universidade de São Paulo  
São Carlos, Brazil

Mahsa Varshosaz  
Centre for Research on Embedded Systems  
School of Information Technology  
Halmstad University, Sweden

Mohammad Reza Mousavi  
Centre for Research on Embedded Systems  
School of Information Technology  
Halmstad University, Sweden

ABSTRACT

Input/Output Transition Systems (IOTSs) have been widely used as test models in model-based testing. Traditionally, input output conformance testing (IOCO) has been used to generate random test cases from IOTSs. A recent test case generation method for IOTSs, called Complete IOCO, applies fault models to obtain complete test suites with guaranteed fault coverage for IOTSs. This paper measures the efficiency of Complete IOCO in comparison with the traditional IOCO test case generation implemented in the JTorX tool. To this end, we use a case study involving five specifications produced by Faulty mutations of the specifications were produced in order to compare the efficiency of both test generation methods in killing them. The results indicate that Complete IOCO is more efficient in detecting deep faults in large state spaces while IOCO is more efficient in detecting shallow faults in small state spaces.

CCS Concepts

• Software and its engineering → Software testing and debugging; Software verification and validation;  
  Empirical software validation;

Keywords

Conformance testing, Input output conformance (IOCO), Complete input output conformance, Mealy input output transition systems, fault models

1. INTRODUCTION

Model-Based Testing (MBT) overcomes some of the challenges in software testing by automatically generating test cases from behavioral models such as Finite State Machine (FSM) and Input/Output Transition System (IOTS) [4, 8]. IOTSs have been widely used both in the research community and in industry as test models. IOTSs are more expressive than FSMs, especially when dealing with nondeterminism. They also provide a richer notion of conformance [8]. Contrary to FSMs, IOTSs impose no restriction on the sequence of inputs and outputs and can reach a state in which no output action is produced [17].

MBT for IOTSs was proposed by Tretmans [17], who established the Input/Output Conformance (IOCO) testing theory. This theory checks if an implementation conforms to a given specification by checking the inclusion of the implementation outputs in those of the specification. This check is only performed after executing the specification traces, allowing for the possibility of specifying partial test models. Tretmans also proposed a widely used algorithm for test case generation from IOTSs. This algorithm produces a test suite in a nondeterministic way, meaning that the proven completeness result is more of theoretical importance than of practical value. In IOCO, the interaction between the tester and the system under test is synchronous. However, in practice, many interactions are based on asynchronous communication or exchange of messages through buffers and can be modeled as queues.

In [13], the W-method from FSMs [5] has been adapted for a class of IOTSs, named Mealy IOTSs [13]. This class requires quiescence (i.e., absence of outputs) to be reached before the inputs are provided; therefore, problems related to the communication between testers and implementations can be eliminated. This method, called Complete IOCO, generates complete test suites for a specification IOTS with respect to a fault domain that contains all implementation IOTSs with at most as many states as the specification. The notion of test completeness, called n-completeness, has been reformulated from the corresponding FSM methods [15] to the IOTS model.

The aim of this paper is to measure the efficiency of Complete IOCO [13], an offline and deterministic test generation method, in comparison with the nondeterministic and online method of IOCO [17] as implemented in the JTorX tool [1]. To this end, we use the well-known ETCS Ceiling Speed Monitor benchmark from the railway domain [3, 2], as well
an example of Mealy IOTS, that shows input-completeness is enabled only in quiescent states [16]. Figure 1b presents the use of fault domains, converge on this class of IOTSs. several results from IOTS and FSM testing theories, such as the impossibility of applying any input allowed by the specification which can be interrupted by an output arrival; or (ii) waiting for an output and checking it, or concluding the implementation is in quiescence. It is proven in [17, 18] that this process is exhaustive, i.e., it is guaranteed to fail all non-conforming implementations; however, this exhaustiveness result does not define any upper bound on the recursive application of the process: exhaustiveness in IOCO is hence, a theoretical rather than a practical issue, since it does not come up with a finite test suite.

2.2 Test Case Generation in IOCO
Input/Output Conformance (ioco) testing theory [17] formally checks if an implementation conforms to a given specification. The test hypothesis assumes that implementations can be modeled by an input-complete IOTS, allowing the formalization of conformance notion. Given two IOTSs $S$ and $I$, representing respectively the specification and a given implementation, we write $I$ ioco $S$ if, for each trace $\alpha \in tr(I)$, we have out(I-after-\(\alpha\)) $\subseteq$ out(S-after-\(\alpha\)).

Tretmans [17] proposed one of the most widely used algorithms for test case generation from IOTSs [8, 17, 20, 19, 12]. It is a recursive and non-deterministic algorithm [9, 1]. For each recursive step, it chooses among three possibilities: (i) ending the test case with the verdict pass; (ii) applying any input allowed by the specification which can be interrupted by an output arrival; or (iii) waiting for an output and checking it, or concluding the implementation is in quiescence. It is proven in [17, 18] that this process is exhaustive, i.e., it is guaranteed to fail all non-conforming implementations; however, this exhaustiveness result does not define any upper bound on the recursive application of the process: exhaustiveness in IOCO is hence, a theoretical rather than a practical issue, since it does not come up with a finite test suite.

2.3 Test Case Generation in Complete IOCO
Fault domain is a concept used in FSM-based testing to guarantee the fault coverage of test suites [4, 10]. FSM-based methods address the problem of generating complete test suites, which build upon certain assumptions about test models and possible implementation faults [5, 6]. IOCO does not apply this concept, because there are no standard fault models for IOTSs as in FSM-based testing [8]. Hierons [7] demonstrated that implementation relations for asynchronous communications are undecidable, leading to several consequences such as the impossibility of applying fault domains. However, Hierons showed that implementation relations are decidable for some classes of IOTSs, such as Alternating IOTSs. Simao [16] proposed a generalization of Alternating IOTSs, called Mealy IOTSs, which pave the way for defining a general fault model for IOTSs.

Paiva and Simão [13] proposed a reformulation of the W-
method for FSMs [5] to IOTSs at generating complete test suites with complete fault coverage for a given fault domain and is targeted at the class of Mealy IOTS. Adopting this class of IOTSs as test models implies that one can avoid the distortion caused by asynchronous channels in testing, since in Mealy IOTSs an input is provided only if all outputs have been observed and quiescence is reached (i.e., all communication channels are known to be empty). The fault domain defined for this method contains all implementation IOTSs with at most as many stable states as the specification, covering output and transfer faults.

In order to define Complete IOCO for Mealy IOTSs, Mealy IOTS specifications should satisfy the following properties:

- **non-oscillating**: the Mealy IOTS contains no cycle labeled only with outputs;
- **observable**: its transition relation must be a function;
- **output-deterministic**: for each non-stable state, at most one transition must be labeled with an output;
- **minimal**: any two distinct states must be distinguishable;
- **initially-connected**: each state must be reachable from the initial state.

Complete IOCO generates test cases for every possible transition fault in the specification. To this end, it uses the transition cover set and the characterization set, briefly introduced below. The sequences comprising these sets then generate complete test suites in a bounded number of steps. Complete IOCO consists of three major steps:

1. Generation of transition cover set (also called test tree [5]) using breadth first search: this set comprises sequences that visit each and every stable state.
2. Generation of characterization set: This set contains input sequences that produce different outputs for each pair of stable states.
3. Concatenation of reset operation, with sequences from the transition cover and the characterization sets: The reliable reset operation, that moves the execution to its initial state, is concatenated along with sequences from the transition cover and the characterization sets; the resulting outputs produced by the specification are recorded, which is compared with that of the implementation during test execution.

Complete IOCO for Mealy IOTSs is deterministic and the process is repeatable, in contrast to IOCO. The test suite generated by the algorithm detects all faults in the fault domain. A case study, presented in [13] illustrated the feasibility of the method. However, more empirical studies with real specifications are needed to evaluate and measure the efficiency of this testing method.

### 3. SPECIFICATIONS

We have used the specifications of the following Cyber-Physical Systems for our study:

- Ceiling Speed Monitoring with Service Brake Intervention (SBI) and Emergency Brake Intervention (EBI) [3, 2],

#### 3.1 Ceiling Speed Monitor

The ETCS Ceiling Speed Monitor (CSM) [3, 2] is part of the European standard specification for train control systems. In this specification, two configurations of a train are possible: a train must have an Emergency Brake (EB) feature. However, a train may also have a Service Brake (SB) feature. The idea is that a train without the service brake feature must use the emergency brake feature to decrease the speed regardless of the situation, whereas the train with the service brake feature must use the emergency brake feature only in an emergency situation [3].

If the CSM detects an over-speeding threshold, then the ServiceBrake is triggered, if a Service Brake is available. Otherwise, the Emergency Brake is triggered. From SB, it is possible to return to Normal if the speed decreases after the intervention. When the train continues its acceleration, the Emergency Brake is triggered.

We have separated these two possible configurations in two different IOTSs - SBI (with Service Brake Intervention) and EBI (with Emergency Brake Intervention). The discrete inputs represent the conditions that trigger the action, defined in [2]. The outputs are the results provided by the specification. If a train is in a normal status and detects an overspeeding threshold, then the status changes to Warning, and if the speed continues increasing, then the emergency/service brake is fired. The conditions that trigger actions according to [2] are presented in Table 1.

Figures 2 and 3 show the IOTS specifications of SBI and EBI, respectively.

#### 3.2 Turn Indicator Lights

A model of turn indicator lights in Mercedes vehicles was presented in [14], which covers the functionality of left/right turn indication, emergency flashing, crash flashing, theft flashing and open/close flashing. The behavior model that comprises these functionalities is shown in Figure 4. The inputs in this model denote both discrete inputs (by pushing the turn indicator levers) as well as timing triggers.
of this system implement reactive control tasks interacting with each other and with the environment via input signals provided by sensors and output signals emitted to actuators.

We have used the specification of two components of this system: Standard Exterior Mirror Component (EM) and Standard Alarm System Component (AS). The AS Component controls the activation/deactivation of the alarm system as well as the triggering of the alarm and the EM Component controls the mirror movement [11]. The behavior of these components are represented in the IOTSs in Figure 5 and 6, respectively.

The initial state of AS model (AS_active_on/off) activates the alarm system and disables the monitoring. The alarm system can be deactivated (as_deactivated) and re-activated again (as_activated). The alarm monitoring of the alarm system is enabled (as_active_on) if the car is locked by using the car key (key_pos_unlock) when the system is active (as_active_off). If an alarm is detected (as_alarm_detected) and the alarm monitoring is enabled (AS_on), then the alarm is triggered (as_alarm_on). The triggered alarm is stopped (as_alarm_off), if the car is unlocked (key_pos_unlock), or the alarm time elapses (time_alarm_elapsed) sending a silent alarm (alarm_was_detected) [11].

The EM model specifies the behavior of the exterior mirror position adjustment. The upper, upper left, upper right, lower, lower left, lower right, left, right, and pending position of the mirror is represented by the corresponding states EM_top, EM_top_left, EM_top_right, EM_bottom, EM_bottom_left, EM_bottom_right, EM_left, EM_left_right, and EM_pending. The pending position (EM_pending) is the initial window position from the initial state. The exterior mirror moves down (em_mov_down), up (em_mov_up), right (em_mov_right), or left (em_mov_left), based on the corresponding movement command (em_mov_down, em_mov_up, em_mov_right, and em_mov_left, respectively). The mirror stops moving in the corresponding direction if the mirror reaches (em_pos_top, em_pos_bottom, em_pos_left, em_pos_right) one of its end positions. Based on its current position, the mirror is able to move into the prior directions until a new end position is reached [11].

4. CASE STUDY

In order to evaluate the effectiveness and the efficiency of Complete IOCO, we conducted a case study with specification models specified in the previous section. We use the JTorx implementation [1] of IOCO as a reference for our comparison with Complete IOCO. We note that Msaly IOTSs were expressive enough to capture all specification

<table>
<thead>
<tr>
<th>#</th>
<th>Conditions for EBI</th>
<th>Conditions for SBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>( V_{CUI} \leq V_{MRSP} )</td>
<td>( V_{CUI} \leq V_{MRSP} )</td>
</tr>
<tr>
<td>c1</td>
<td>( V_{CUI} &gt; V_{MRSP} )</td>
<td>( V_{CUI} &gt; V_{MRSP} )</td>
</tr>
<tr>
<td>c2</td>
<td>( V_{CUI} \leq V_{MRSP} + dV_{MRSP} )</td>
<td>( V_{CUI} \leq V_{MRSP} + dV_{MRSP} )</td>
</tr>
<tr>
<td>c3</td>
<td>( V_{CUI} &gt; V_{MRSP} + dV_{MRSP} )</td>
<td>( V_{CUI} &gt; V_{MRSP} + dV_{MRSP} )</td>
</tr>
<tr>
<td>c4</td>
<td>( V_{CUI} &lt; V_{MRSP} )</td>
<td>( V_{CUI} &lt; V_{MRSP} )</td>
</tr>
<tr>
<td>c5</td>
<td>( V_{CUI} &gt; V_{MRSP} )</td>
<td>( V_{CUI} &gt; V_{MRSP} )</td>
</tr>
<tr>
<td>c6</td>
<td>( V_{CUI} \leq V_{MRSP} ) ( \land ) ( V_{CUI} = 0 )</td>
<td>( V_{CUI} \leq V_{MRSP} ) ( \land ) ( V_{CUI} = 0 )</td>
</tr>
<tr>
<td>c7</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The system specifies the following behavior: upon reception of a turn indication message (\( ?c1 \)) with positive on-duration, for the indicator lamp, the output power is set to 100%. The lamp should be automatically switched off when the on-duration elapses and a timer is set for that. If the lamp is switched on and a new command arrives, the on-duration timer is set again and the lamp remains in its active state. A new turn indication message (\( ?c2 \)) can switch off the lamp interrupting the on-status.

### 3.3 Body Comfort Systems

The Body Comfort System [11] is a case study from the automotive domain, describing the internal locks and signals of a vehicle model. The different software components of this system implement reactive control tasks interacting...
The following steps summarize the methodology used in this study:

- **Preparation of faulty versions of specifications**: A set of 20 faulty mutants for each specification model was produced. We seeded one fault in each mutant, which could either be a transfer fault or an output fault.

To obtain the faulty versions of specifications, the following methodology was adopted: from the initial state of the unfolded specification tree, for each level of the tree, a random transition was selected to be seeded with a transfer fault (change the target state). In the same way, a random transition was selected to be seeded with an output fault (change the input/output label). Then, the output and transfer faults are equally distributed.

- **Test suite generation with IOCO (JTorX)**: Each mutant and IOTS specification were represented in the GRAPHML format. We ran 50 times the specification against each mutant in JTorX until the mutant is killed. We have limited the upper-bound of each execution in 60 seconds. We registered the total number of steps until killing the mutant, i.e., the number of (input or output) actions executed until the fault is detected.

- **Test suite generation with Complete IOCO**: We have produced a test suite for each specification model using our prototype tool for Complete IOCO for Mealy IOTSs based on the algorithm of [13]. For each specification, we executed 50 times the test suite against each mutant version and observed if the mutant is killed in the end. It turned out that all mutants could be killed (due to the completeness of the method) within the time limit of 60 seconds. In each execution, the sequence of test cases execution was randomly selected.

- **Analysis of results**: All mutants were killed by both methods; hence, we focused on their comparative efficiency. We measured 2 data points to compare the efficiency of the methods in finding faults: depth level of the fault in each mutant and the average number of steps until the mutant is killed.
5. ANALYSIS OF THE RESULTS

5.1 Results

Figures 7, 8, 9, 10 and 11 show the obtained results for each specification model. The horizontal (x) axis indicates each mutant in increasing order of fault depth (regarding the unfolded specification tree level) and the vertical (y) axis indicates the average number of steps until the mutant be killed. All mutants were killed by JTorX and Complete IOCO and the upper-bound limit was not reached, hence, it is not possible to compare the relative effectiveness and we focus on efficiency in the remainder of this section.

The results indicate that mutants with faults in larger state spaces and in a deeper level of the state space can be detected by Complete more efficiently. A deeper level indicates large traces, and a large state space should produce a number of traces. Otherwise, for smaller specifications, i.e., specifications with a short state space and short traces, IOCO (JTorX) outperforms Complete IOCO. SBI and EBI models have a large state space, hence, Complete IOCO is more efficient than JTorX, as seen in Figures 7 and 8. TIL model may be seeded with deep faults, but its state space is relatively small. Thus, it is more efficient for JTorX to detect the faults in this model, as seen in Figure 9. Likewise, in the EM model, although the state space is large, the faults are always at the shallower depth, i.e., the traces are short. Hence, IOCO (JTorX) was more efficient to detect the faults in this model, as seen in Figure 10. AS model is deeper than EM model, but it has a few number of traces and the results indicate that JTorX is more efficient to detect faults in this model. Furthermore, the order of test cases execution is a bias in detect faults.

This results point out W-method more efficient in detect faults in more deeper levels and in models that has a number of traces. Therefore, for larger and deeper specifications (large traces) W-method can obtain good results and guaranteed fault coverage. At the same way, JTorX can be more efficient to detect faults in plain levels and in models that has a few number of traces, because it is easier to traverse all traces. Thus, Complete IOCO, a deterministic and offline method, can be more efficient than the traditional IOCO method (online and nondeterministic) in some situations.

5.2 Threats to validity

Our results are naturally dependent on the choice of our case study. By varying among different sorts of examples, we tried to mitigate this threat. We intend to study a larger set of examples in the future to further address this issue.

We only considered the depth of faults and the size (the branching degree) of the specification as the relevant parameters in our research thesis. We can think of alternative ways of characterizing faults and compare the two methods based on these alternative classifications.

IOCO uses a random seed to steer the test-case generation, while the sequence of test cases in complete IOCO is typically fixed in the algorithm. Our results, hence, may be sensitive to the fixed order implemented in our prototype for Complete IOCO. Randomizing this order can mitigate this threat to the validity of our results.

6. CONCLUSIONS AND FUTURE WORK
In this paper, we compared the efficiency of the Complete IOCO and the IOCO test case generation methods in detecting faults. We considered specification models inspired by industrial cases to obtain realistic results. Faulty mutants of the specifications were produced in order to compare the efficiency of the two test generation methods. Complete IOCO is a deterministic and repeatable test generation method, in contrast to JTorX that implements the 1coo theory, which is non-deterministic.

The results point out that both methods revealed all faults seeded in our mutants. The results indicate that Complete IOCO is more efficient in detecting deeper faults in large state spaces, since this kind of fault is difficult to reach with the nondeterministic algorithm of JTorX.

As future work, we plan to apply this study with different kind of specifications, i.e., specifications with different characteristics regarding to traces number and size. Moreover, we intend to investigate the prioritization of test cases in the execution of test suites in order to gain more insight about the performance of Complete IOCO.

Acknowledgments

The research of S. C. Paiva and A. Simao have been supported by Sao Paulo Research Foundation (Grants 2012/ 09650-5 and 2012/02232-3), Coordination for the Improvement of Higher Education Personnel and National Counsel of Technological and Scientific Development. The work of Mohammad Reza Mousavi has been partially supported by the Swedish Knowledge Foundation (Stiftelsen for Kunskapsutveckling) in the context of the AUTO-GoG project (number: 20140312).

7. REFERENCES


Bibliography in this case