Modeling and Model-Based Testing of Software Product Lines

Mahsa Varshosaz

Supervisors:
Mohammad Reza Mousavi
Gerardo Schneider
Wojciech Mostowski
Dedicated to my parents Nader and Shahla.
Abstract

Software product line (SPL) engineering has become common practice for mass production and customization of variability intensive systems. A software product line comprises a family of software systems which share a managed core set of artifacts and also have a set of well-defined variabilities. 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 quality assurance of software systems. In MBT, an abstract model, which captures the desired behavior of the system, is used to generate test cases. The test cases are executed against a real implementation of the system and the conformance between the implementation and the specification is checked by comparing the observed outputs with the ones prescribed by the model.

Software product lines have been applied in a number of domains with mission critical systems. MBT is one of the techniques that has been used for analysis 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 can be very costly and time consuming. To tackle this problem, 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. So far, several formalisms have been proposed for modeling SPLs. In this thesis, we conduct a study on such modeling techniques, focusing on four fundamental formalisms, namely featured transition systems, modal transition systems, product line calculus of communicating systems, and 1-selecting modal transition 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
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.

Moreover, we conduct a survey on sampling techniques, which have been proposed as a solution for handling the large number of products in analysis. In general, in product sampling a subset of products that collectively cover the behavior of the product line are selected. Performing tests on well selected sample set can reveal most of the faults in all products. We provide a classification for a catalog of studies on product sampling for software product lines. Additionally, we present a number of insights on the studied work as well as gaps for the future research.
Acknowledgments

This has been an amazing journey full of both ups and downs. This thesis would have not been possible without the help and support of many people.

Firstly, my sincere gratitude to my supervisor Prof. Mohammad Reza Mousavi for the continuous support of my research, for his patience and encouragement, and for believing in me in the first place. Working with you has been a great experience and I am grateful for everything that you have taught me. Thank you for your constant guidance and support.

Also, I would like to thank my co-supervisor Prof. Gerardo Schneider at Chalmers University for his immense and valuable support. Many thanks to my co-supervisor Wojciech Mostowski at Halmstad University for his help and support.

Besides my supervisors, I would like to thank Prof. Jiri Srba, Dr. Ferruccio Damiani, Dr. Alberto Lluch Lafuente, and Prof. David de Frutos Escrig for agreeing to be a part of the defence committee.

I would like to thank all the wonderful people with whom I have collaborated. Thank you for sharing your knowledge and experiences with me. Special thanks to Dr. Malte Lochau and Lars Luthmann for the excellent collaboration that we have had. Also, I would like to thank Prof. Ina Schaefer and Dr. Jan Tretmans for giving me the opportunity of having very interesting research visits to their groups.

I am thankful to all my friends from Halmstad University for their support and friendship. Thanks for all the great memories and experiences that we have shared and for being there for me in difficult times. Thanks to my lab mates for interesting discussions. Linda and Suleyman, I appreciate your persistence in keeping up the delightful fika tradition. Thanks to Deycy for all the nice chats and for showing me cheerful salsa music. Saeed, Anita, and Kevin thanks for your support and for sharing the joy of climbing with me.

I would like to thank my parents for supporting me throughout my studies and my life. Thank you for always being there for me and believing in me. Thanks to Parisa and Amin for inspiring me and for their immense support through this journey.
Finally, thanks to Raul for making the past two years truly incredible. Having your support and encouragement has been invaluable for me.
Contents

1 Introduction ........................................... 1
  1.1 Motivation ........................................ 1
  1.2 Problem Definition ................................ 3
  1.3 Approach .......................................... 3
    1.3.1 Modelling Software Product Lines .......... 4
    1.3.2 Model-Based Testing of SPLs ............... 5
  1.4 Contributions ..................................... 6

2 Background ........................................... 7
  2.1 Model-Based Testing .............................. 7
    2.1.1 Harmonised State Identification Method .... 7
    2.1.2 Input-Output Conformance Testing .......... 10
  2.2 Software Product Lines .......................... 11
    2.2.1 Feature Diagrams ............................ 12
    2.2.2 Featured Transition Systems ................. 13
    2.2.3 Product Line Calculus of Communicating Systems ... 14
    2.2.4 Modal Transition Systems ................... 14
    2.2.5 Model Based Testing of Software Product Lines .... 16

3 Summary of Contributions .......................... 19
  3.1 Paper I: Delta-Oriented FSM Based Testing ....... 19
  3.3 Paper III: Complete IOCO Test Cases: A Case Study .... 21
  3.4 Paper IV: Basic Behavioral Models for Software Product Lines: Revisited ........................................ 22
  3.5 Paper V: Comparative Expressiveness of Product Line Calculus of Communicating Systems and 1-Selecting Modal Transition Systems ........................................ 22
  3.6 Paper VI: A Classification of Product Sampling for Software Product Lines ........................................ 23
3.7 Paper VII: Modal Transition System Encoding of Featured Transition Systems ........................................ 24

4 Conclusion and Perspectives ........................................ 25

References ......................................................... 27

A Paper I ....................................................... 35
B Paper II ..................................................... 53
C Paper III ................................................... 75
D Paper IV .................................................... 85
E Paper V ..................................................... 103
F Paper VI .................................................. 119
G Paper VII ................................................. 135
List of Figures

2.1 Model based testing process. .......................... 8
2.2 Feature diagram for a vending machine product line. ........ 12
This thesis summarizes the following publications.

Paper I  
Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi.  

Paper II  
Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi.  

Paper III  

Paper IV  
Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi.  

Paper V  

Paper VI  

Chapter 1
Introduction

In this section, we give a brief introduction to the Software Product Line (SPL) engineering paradigm and the main challenges in testing and modelling SPLs.

1.1 Motivation

Software product line engineering has become common practice in developing variability intensive systems aiming for mass production and customisation. An SPL consists of a set of software systems, also called products. The products in an SPL are developed from a common set of core assets and by reusing software artifacts with the purpose of satisfying the specific needs of a particular market. Furthermore, there are well-defined variabilities among products. The commonalities and variabilities among the products in an SPL are usually specified using the concept of feature. A feature is described as a distinctive user-visible aspect or characteristic of the software [30]. In SPL engineering systematic reuse is enabled in different phases of software development by considering the well-defined commonalities and variabilities, which leads to reduced price and time to market [49].

Many industrial companies from a variety of domains have adopted SPL engineering in developing their software systems; examples of such companies include Boeing, Bosch, General Motors, Hewlett Packard, Philips, Siemens, and Toshiba [73]. As some of the software systems produced in this manner are safety critical, applying quality assurance techniques such as testing and verification is very important. In many cases SPLs have been used to build complicated systems such as automotive systems, power plants, medical devices, and televisions [73], where the product line contains 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 [57]. Given this potential combinatorial explosion, in many cases performing any type of analysis on the individual products of a product line using conventional techniques is challenging if not infeasible. Consequently, in the last two
decades, researchers have proposed many techniques which are tailored for the analysis of SPLs [15, 26, 21, 28, 42, 46, 56, 39]. 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. Among several analysis techniques, testing SPLs has been gaining increasing attention in both academia and industry [43, 62].

In general, (manual) testing can be a laborious and costly process. Hence, automatising different phases of testing, such as test case generation and execution, is of great importance to gain efficiency. Model-Based Testing (MBT) is a technique, which brings structure, rigour, and effectiveness into the testing process. The main advantage of this technique is test automation which allows for generating and executing test cases in a more efficient manner. In MBT, test cases are generated using a model of the system, called a test model. The test model describes the expected behaviour of the system. Once test cases are generated, by applying an algorithm on the test model, the test cases are executed on an operational version of the system, namely the system under test (SUT). Throughout the test case execution, it is checked that the behaviour of the implementation of the system is compliant to the expected behaviour. 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 [64].

Several modelling and model-based testing approaches for SPLs have been proposed so far [15, 26, 21, 28, 42, 46, 56, 39]. In modelling SPLs, there are two fundamental entities that need to be considered: the software platform (core), which is the basis for development of the products, and the variability or variation points, which are the places that products differ from the software platform. A major challenge in modelling SPLs is providing a modelling framework with the right level of abstraction to integrate variability information with behavioural specification, which is sufficiently expressive in order to check the behavioural conformance (i.e., testing the compliance between an implementation of a product and its specification). A commonly used solution for modelling SPLs is to annotate the existing modelling framework with the concept of variability. Examples of such modelling frameworks are Featured Transition Systems (FTSs) [15, 16], which are extensions of Labeled Transition Systems (LTSs). In FTSs, the transitions are annotated with propositional formulae which are satisfiable for a subset of products. These indicate the presence of transitions in individual products. Other alternatives include Modal Transition Systems (MTSs) [35], which are also an extension of LTSs. In an MTS transitions are classified into may and must transitions. 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. Fischbein et al. [25] has argued that these models are adequate for modelling SPLs, because in these models, may transitions represent op-
tional behaviour and must transitions are used for mandatory behaviour of
the products in an SPL. In the aforementioned models, the behaviour of all
the products in an SPL is represented in one abstract model and the be-
haviour of individual products can be derived from this model. However, there
are other approaches such as abstract delta modelling [14], where a part of
the behaviour of the products (usually the common behaviour) is described
as a core and the model for each product is obtained by application of a set
of delta modules (i.e., modules specifying changes) to the core model. (More
details about formalisms proposed for modelling SPLs are provided in Section
2.2.) There have been several techniques adapted for efficient model based
testing of SPLs [21, 28, 42, 46, 56, 39]. 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. Although, using many
MBT techniques proposed for SPLs lead to more efficient testing of SPLs and
hence having the possibility of performing more rigorous tests; in many cases,
applying such techniques is still difficult and costly. Product sampling is an-
other technique used for selecting a subset of products that collectively cover
the behaviour of the product line and hence performing tests on this subset of
products can reveal most of the faults in all products [40].

1.2 Problem Definition

In this thesis, we focus on model-based testing of SPLs. As abstract models are
prerequisites of MBT techniques, efficient modelling is also a relevant problem
in the context of 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 modelling SPLs compare in
terms of expressiveness and succinctness?

2. How can we adapt a model-based test-case generation technique to enable
reuse of testing artifacts among products of an SPL?

3. What are the interesting characteristics of sampling techniques and how
using such techniques can impact MBT of SPLs?

In the following, we give a brief overview of our approach to answer the
above mentioned questions.

1.3 Approach

We divide this section into two parts: the first part is about modelling SPLs,
and the second part is related to the MBT of SPLs.
1.3.1 Modelling Software Product Lines

Models are prerequisites of model-based testing, as well as many other verification and quality-assurance techniques. Hence, considering SPLs, devising a modelling technique that can efficiently capture the behaviour of SPLs is necessary. By using conventional models, which are used for modelling single systems, for modelling SPLs it is impossible to perform any kind of analysis on the whole product line at once. Therefore, several formalisms have been proposed for more efficiently and compactly modelling SPLs. Some examples of such models are FTSs [15], MTSs [35] and their various extensions [24, 34, 7, 61], and Product Line Calculus of Communicating Systems (PL-CCSs) [26].

There are several aspects of these formalisms such as succinctness, expressiveness, and compositionality that can be considered in comparison and when making a choice for modelling. Expressiveness and succinctness, are two properties that are very important as they, respectively, indicate the capability of a class of models in capturing and representing different behaviour 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, first, 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 behaviours (specified by LTSs). The product models are concrete implementations of the abstract model used for describing the behaviour 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 $\forall x \in X \ [x] = [E(x)]$, where $[\ ]$ denotes the semantic function that represents the set of LTS implementations of the model. This criterion asserts that for all SPL models $x \in X$, there is an SPL model $E(x) \in Y$, such that their sets of implementations coincide.

- We say that formalism $Y$ is at-least as expressive as formalism $X$ if such an encoding exists. Moreover, 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 (in this part of our research we base our work on models with finite behaviour). The results of the works are presented as a hierarchy of models in terms of their expressive power. Later, to extend our hierarchy, we provide a comparison of the expressive power of an extension of MTSs, namely, 1-Selecting
Modal Transition Systems (MTSs), with PL-CCSs. Moreover, the theory of MTSs has been extensively studied \cite{31} and based on that, various tools have been developed to support their analysis \cite{2, 20, 72, 32, 5}. In addition, MTSs benefit from many desirable formal properties such as inherent notions of semantic refinement being compatible with parallel composition, thus enabling compositional reasoning. Hence, given the result that shows FTSs are more expressive than MTSs (considering finite behaviour), we provide a translation from the class of FTSs into the class of MTSs. We prove that this translation is semantic preserving. This way it is possible to use a more expressive model (i.e., FTSs) and through our translation also use the existing analysis tools for MTSs, which have more variety compared to FTSs.

The size of models can affect the applicability of the analysis techniques. Hence, succinctness is another property of these formalisms that is interesting to study. In order to show that a class of models $X$ is exponentially more succinct than another class of models $Y$, we show that there exists $x \in X$ such that given any alternative encoding $E$ from $X$ into $Y$, it holds the size of the model resulted from encoding of $x$ is exponentially larger than the size of $x$, i.e., $|x| < |E(x)|$. Furthermore, we prove that there exists an encoding $E'$ from $Y$ into $X$ such that the size of the model resulted from encoding of $y$ is in the linear order of the size of $y$, i.e., for all $y \in Y$, $|E'(y)| = O(|y|)$ and $| |$ is used to represent the size of the models. So far, we have studied the relative succinctness of two classes of models namely, PL-CCSs and FTSs since they have the same expressive power.

1.3.2 Model-Based Testing of SPLs

As it was pointed out before, there are many challenges in the MBT of SPLs compared to the MBT of single systems due to the potentially large number of products. To tackle these challenges, it is required to enable systematic reuse in different phases of MBT, such as test case generation and execution. Hence, conventional MBT techniques have to be tailored for efficiency and reducing testing effort.

The main idea applied in many approaches used for testing SPLs is to generate a set of test artifacts for the platform once, additionally to generate 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 \cite{21, 28, 56, 39}. Hence, by reusing the test artifacts generated for the platform the testing cost and effort can be reduced.

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

As a 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 (IOTs) that was developed to extend the test case generation by notion of fault coverage. Three case studies taken from the automotive and railway domains are considered in the comparison. A subsequent step for this line of our research is to adapt the complete ioco method for efficient testing of SPLs. In line with this part of our study, and to answer the third research question in Section 1.2, we have conducted a survey on sampling techniques that can be used for model-based testing of SPLs. In this part of our study, we provide a classification of some of the existing sampling techniques based on the input data that they use, the technique and the evaluations performed.

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 modelling SPLs [8],
- Studying the comparative expressiveness among four classes of models, namely featured transition systems, modal transition systems, product line calculus of communicating systems, and 1-selecting modal transition systems [8, 69, 71],
- Providing a semantic preserving translation from the class of FTSs into the class of MTSs, implementing an algorithm for translation and performing experiments on examples [70],
- Adopting the HSI-method for efficient test case generation for software product lines building upon the idea of delta-oriented programming [68],
- Measuring the efficiency of the test case generation algorithm used in two well-known MBT methods, namely, complete ioco and ioco [45],
- Providing a classification for sampling techniques used in combination with analysis techniques for SPLs [67].
Chapter 2
Background

In this section, we explain some of the concepts and constructs that are used in the rest of the thesis. The section is divided into two parts dedicated to model-based testing and software product lines. (The related work is briefly included in each part.)

2.1 Model-Based Testing

One of the commonly used techniques for quality assurance of software systems is testing. However, testing software systems in practice can be costly and time consuming. The studies have shown that testing can take around 30-50 percent of the software development time \[ \text{[64]} \]. As the complexity of the software systems grows; it is essential to have systematic testing techniques to reduce the time and cost of testing process. To this end, model-based testing techniques have been proposed that allow for enabling automation in different phases of testing. Using model-based testing techniques, an abstract model of the system, which represents the expected behaviour of the system under test, is used for generating test cases automatically. The algorithmic generation of test cases allows for generating a large number of test cases in a shorter time and thus testing software systems more rigorously. Then, by executing the generated set of test cases on a real implementation of the system, the compliance of the behaviour of the implementation to the specification of the system is checked. Figure 2.1 shows an overview of the MBT process. Several model-based testing techniques have been proposed so far, of which \[ \text{[43]} \] provides an overview. In this thesis we introduce two MBT techniques that have been used in the rest of the thesis.

2.1.1 Harmonised State Identification Method

Harmonised State Identification (HSI) method \[ \text{[51]} \] is a well-known black-box model-based testing technique. Using this method the compliance of the be-
Figure 2.1: Model based testing process.

behaviour of an implementation of the system with its specification can be established. In this approach, test models, which represent the expected behaviour of the system, are Finite State Machines (FSMs). The test cases are generated automatically using the test models. First, we give the formal definition of an FSM.

Definition 2.1.1. A finite state machine $M$ 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,
- $\mu : S \times I \to S$ is the transition function,
- $\lambda : S \times I \to O$ is the output function.

Based on the transition function, whenever a machine receives input $\alpha$ at state $s$, it deterministically traverses to state $\mu(s, \alpha)$ and generates output $\lambda(s, \alpha)$. Using a quadruple $(s, i, o, s')$, or alternatively by $s \xrightarrow{i/o} s'$, we represent a transition from state $s$ to state $s'$ with input $i$ and output $o$. We define $\mu(s, x)$ and $\lambda(s, x)$, where $x \in I^*$ is a sequence of inputs, in the standard manner to denote, respectively, the final state in which the machine ends 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, where $X \subseteq I^*$, if and only if the two states produce the same output for every input sequence $\sigma \in X$ (see [12] for a formal definition). Lastly we define that two machines $M$ and $M'$ are $X$-equivalent for $X \subseteq I^*$, 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 in the HSI method is to establish conformance between an FSM test model $M$ and an unknown machine $M'$ which describes the behaviour of an implementation of the system. The conformance is checked by generating a finite number of test cases from $M$ and executing them against $M'$. According to the HSI method, a set of assumptions should hold for these machines in order for the method to be sound and exhaustive: all states (in both of these machines) are reachable from the initial state of the respective machine, both machines are deterministic and minimal, both machines have reliable reset sequences that 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. In the first phase, for each state the existence of an $I^*$-equivalent corresponding state in the implementation is checked. The second phase consists of checking the conformance between the output and the target of the transitions for the corresponding states in the implementation and specification. There are two main sets of sequences which are used in these two phases. The first set is the state cover set, denoted by $Q$. This set is used to reach all of the states in the machine, and it is formally 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 $Q$ 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, denoted by $Z$. This set comprises sets of separating sequences for all states. A set of separating sequences for a state $s$ can distinguish $s$ from all other states. This set is used to identify and test the target states of transitions after reaching an state in the model by running elements 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. The set of test cases executed in the first phase
are generated by combining the sequences from these sets and the reset sequence, i.e.: for each state $s \in S$, let $q_s$ and $z_s$, respectively, denote the sequence in the state cover set which leads to $s$ and the set of separating sequences generated for $s$. The set of test cases in the first phase are $\bigcup_{s \in S} r.q_s.z_s$, where $r$ is the reset sequence. In the second phase of the HSI method, the output and the target state of the remaining transitions, which are not visited while traversing the state cover set, are checked.

2.1.2 Input-Output Conformance Testing

Input-output conformance testing (ioco) is a well-known MBT technique, which has been applied to many industrial cases. This technique uses LTSs and their extensions as test models.

Input-Output Transition Systems (IOTSs) are used as test models by ioco. In IOTSs the set of actions are partitioned 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,
- $h : S \times (I \cup O \cup \{\delta\}) \times S$ is the transition relation, with the symbol $\delta \notin (I \cup O)$ 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 the ioco testing theory, the conformance of an implementation of the system $i$ to a specification $s$, denoted by $i \text{ ioco} s$, is checked using a set of test cases which are generated by following a recursive, non-deterministic algorithm [29, 6]. 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) receiving an output (or noticing lack there of) and comparing it with the expected outputs. The results of executing test cases on the implementation can be pass or fail. Given the generated test cases, a test suite (comprising a set of test cases) $T$ is considered sound iff no false errors are detected i.e., for any implementation of the system such as $i$ it holds $i \text{ ioco} s$ implies $i$ passes $T$. Furthermore, $T$ is considered exhaustive iff using $T$ all errors are detected i.e., for any implementation of the system such as $i$ it holds $i \text{ ioco} s$ if $i$ passes $T$. In [64], 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 [44] is one such technique that is developed for testing Mealy IOTSs [55]. The behaviour 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 [12]. 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 [13]. 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. Pavia et al. in [44] use a fault model for an IOTS with \( n \) stable states, where a stable state is a state with a nonempty set of outgoing transitions and the transitions only have input actions, as the set of all possible implementations of the IOTS with at most \( n \) stable states and same set of input actions. 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.

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 characterisation set, which is a set containing input sequences that produce different outputs for each pair of quiescent states.

3. Concatenating the reliable reset operation along with sequences from the transition cover set and the characterisation set; 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 that are developed by sharing a common core set of artifacts and have well-defined variabilities. The commonalities and variabilities among the products of a product line are described in terms of features. A feature is a distinctive user-visible aspect or characteristic of the software [30]. 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 [53] 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 denoted by xor, 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 have cross tree relations such as include or exclude with other features. 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 in this diagram. 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 edges. Also features c and d have alternative relationship, which means a vending machine can either receive Euros 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 p, which means a product that accepts Dollars cannot provide cappuccino.

![Feature Diagram for a Vending Machine Product Line](image-url)

Figure 2.2: Feature diagram for a vending machine product line.
As mentioned above, feature diagrams are structures that describe the relationships among the features of an SPL. In order to represent the behaviour of the products including these features, different behavioural models can be used. Several conventional models have been extended for efficient modelling of SPLs, which are used to avoid modelling products in the product line individually. In this thesis, we focus on four fundamental models which have been used for modelling SPLs, namely FTSs, PL-CCSs, MTSs, and 1MTSs.

### 2.2.2 Featured Transition Systems

Featured Transition Systems [15] are extensions of LTSs proposed for modelling SPLs. Each FTS comprises a set of states and transitions. The transitions are labeled with pairs of actions and feature expressions. A 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 captured in one model. A propositional formula $\phi \in \mathbb{B}(F)$ is called a feature expression, where $\mathbb{B}(F)$ denotes the set of all propositional formulae obtained by considering the elements of the feature set $F$ as propositional variables and we denote by $\mathbb{B}$ the set of Booleans $\{\top, \perp\}$. The formal definition of an FTS is as follows [15]:

**Definition 2.2.1.** A featured transition system is a quintuple $(P, A, F, \rightarrow, \Lambda, p_{\text{init}})$, where:

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

A variety of analysis techniques for FTSs have been proposed. A model-checking algorithm for checking properties against FTSs was given in [15]. Cordy et al. [17] extended the earlier work [16, 15] by combining non-Boolean features and multi-features in a high-level specification language called TVL$^\ast$. An algorithm for constructing an FTS from a behavioural specification written in TVL$^\ast$ was also given. In [60], a family-based approach is proposed for efficient model-checking of properties formulated by a logic which combines modalities with feature expressions over FTSs using mCRL2 [18].
2.2.3 Product Line Calculus of Communicating Systems

In this section, we consider product line calculus of communicating systems [26], which is an extension of Milner’s Calculus of Communicating Systems (CCS) [41]. In PL-CCS a new operator \( \oplus_i \), 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 atomic actions and \( \bar{\Sigma} = \{ \bar{\alpha} \mid \alpha \in \Sigma \} \). The syntax of each term \( e \) in PL-CCS, is defined by the following grammar:

\[
e, e_0 ::= \text{Nil} | e \cdot e | e + e_0 | e_i e_0 | e_k e_0 | e[f] | e \mid L,
\]

where \( \text{Nil} \) denotes the terminating process, \( \cdot \) denotes 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 \( L \subseteq A \), \( \mid L \) denotes the restriction operator (blocking (co)actions in \( L \)), and finally \( i \) 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 \( P = a \cdot P + b \cdot P \) and \( Q = a \cdot Q \oplus_1 b \cdot Q \); 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 made between \( a \) and \( b \). For the sake of simplicity in the formal development of the theory, Gruler et al. assume that in every PL-CCS term, there is at most one appearance of the operator \( \oplus_i \) for each index \( i \).

According to [26], the validity of products can be asserted using model checking formulae specified in a multi-valued modal \( \mu \)-calculus [54].

As similar work done on using process algebra in the context of SPLs; in [38], Lochau et al. propose a delta oriented process calculus called DeltaCCS for modelling the behaviour of systems with variability and an incremental model-checking algorithm for such models against \( \mu \)-calculus properties. Also, in [65], the variant process algebra is proposed, which is a calculus where 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 modelling SPLs are MTSs [35]. 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 behaviour, while must transitions are always present in the implementation
models. It is required that the set of must transitions is a set of may transition. According to [35], an MTS is defined as follows:

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

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

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

In [4, 2, 3, 37, 33, 61], 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 [33, 37], an interface theory and a testing theory based on MTSs for SPLs have been developed. Furthermore, MTSs are extended with variability constraints for modelling behavioural variability and in this line of work FTSs are encoded into MTSs by annotating the target MTSs with variability constraints when needed [59, 61, 58].

There have been several variants of MTSs introduced [7, 34, 24]. Fecher et al. [24], compare two variants of MTSs, namely Disjunctive MTSs (DMTSs) [34], and 1MTSs [24] from the expressiveness 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 and may hyper transitions. In DMTSs, must hyper transitions represent an OR relation between mandatory multiple choices and may hyper transitions represent 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, in comparison with DMTSs, 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 [24] it is shown that the two formalisms have the same expressive power, i.e., for each 1MTS there is a DMTS which induce the same set of LTSs as its implementations and vice versa. Also, in [23], the refinement relation and the expressive power of some of the extensions of the MTSs are discussed. In this thesis, we consider 1MTSs as one of the formalisms included in our expressiveness power comparisons. The formal definition of a 1MTS is as follows:
Definition 2.2.3. A 1-selecting modal transition system (1MTS), is a tuple $(\mathcal{S}, \mathcal{A}, \rightarrow, 
rightarrow, s_{\text{init}})$, where:

- $\mathcal{S}$ is a set of states or processes,
- $\mathcal{A}$ is a set of actions,
- $\rightarrow \subseteq \mathcal{S} \times (2^{\mathcal{A} \times \mathcal{S}} \setminus \emptyset)$ is the must hyper transition relation,
- $\nrightarrow \subseteq \mathcal{S} \times (2^{\mathcal{A} \times \mathcal{S}} \setminus \emptyset)$ is the may hyper transition relation,
- $s_{\text{init}} \subseteq \mathcal{S}$ is a non-empty set of initial states.

More details about expressive power of 1MTSs is provided in [71].

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 [21, 28, 42, 46, 56]. In this line of research the goal is to modularise 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 changes should eventually lead to saving time and effort.

In [21, 28], which are approaches building upon the delta-oriented modelling idea, it is assumed that the behaviour 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 [56] is also used in incremental MBT of FSMs. The approach of [56] aims at completing a given set of test cases, but does not per se address the changes in the test model.

In a survey, Oster et al. [43] 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., [22, 43, 63] for recent surveys). Among these approaches, the closest to our research are those developed by Lochau et al. [39]. 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., [9, 50, 66]). So far, in our research we have followed object-oriented principles such as encapsulation and data-hiding in our modelling framework proposed for delta-oriented FSM-based testing and organised our test models based on specification of class instantiations and their dependencies. class state
2.2. SOFTWARE PRODUCT LINES

machines (CSMs) introduced in [27]. 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 Contributions

3.1 Paper I: Delta-Oriented FSM Based Testing

In this paper, we provide an adaptation of a well-known black-box MBT method, namely, HSI-method [51], for efficient testing of SPLs. To enable reuse in the testing process, we modify the test case generation algorithm of the HSI-method. We adopt the idea of Delta-Oriented Programming (DOP), which is a framework for programming SPLs. In DOP, the products in an SPL are specified by means of a core module and a set of delta modules. The core module describes the commonalities among products. The delta modules describe a set of changes in the core module. Hence, a product is specified by applying a set of delta modules to the core. In this paper, we particularly consider DeltaJava [52], which is used for programming SPLs in Java. Using DeltaJava, the core module is developed as a set of Java classes. The set of changes described by each delta module can be of type adding, removing or modifying the methods and fields of the existing classes and also adding or removing a whole class to/from the core module.

In order to adapt the HSI method for SPLs, first, we use delta oriented modelling technique to specify the behaviour of products. The HSI method uses FSMs as test models. Hence, in our work, the core module is an FSM. In this FSM, the states are abstract representations of variable valuations and the transitions represent method calls, their effect on the state valuations and their returned value. The delta modules describe a set of possible changes on the structure of the core FSM. The considered change types are mainly inspired by the type of changes covered in syntax of DeltaJava and their interpretation in the modelling level. In this paper, we only consider the addition part in the syntax. Hence, the core model represents a minimal subset of features that is common among products and the delta modules describe adding new behaviour to the core by adding states and transitions to the core FSM or composing the core FSM with other FSMs that represent the behaviour of the objects of a class.
Contribution statement: I have been the main author of this paper; My contributions in this work can be summarised as follows:

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


In this paper, we present the results of a comprehensive study on a set of formalisms that are used for modelling SPLs. The studied formalisms [25, 4, 2, 3, 37, 33, 16, 15, 17, 26], are mostly extending conventional behavioral models to enable modelling variability.

The models considered in the study can be used as a semantic domain for higher level models such as domain specific languages or extensions of Unified Modelling Languages (UML) state or sequence diagrams that are used for modelling SPLs. The purpose of this work has been to put an structure on the body of the knowledge about such fundamental models. The results can help the designers of higher level languages to make appropriate choices when defining the semantics.

In general, the results of this paper can be divided into the following two parts:

- We provide a comparison between 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. (The considered models have finite behaviour which has been assumed implicitly in encoding of models.)

- 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-CCS, in our comparison. 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 [19, 48, 64] and particularly of [1]).
According to the results, for finite behaviour the MTSs are the least expressive of all three formalisms. PL-LTSs are less expressive than FTSs (this result is subject to a restriction considered in the definition of the transition relation of PL-LTSs. Later, in paper IV, we show that by relaxing this assumption PL-LTSs are at least as expressive as FTSs) and more expressive than MTSs.

Contribution statement: The summary of my contributions in this work are as follows:

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

- Providing the encoding from FTSs into PL-LTSs, and the proof regarding the comparison of the expressive power.

3.3 Paper III: 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 [47], which describes the functionality of left/right turn indication, emergency flashing, crash flashing, theft flashing and open/close flashing,

- The Body Comfort System [36], 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) [11], which is part of the European standard specification for train control systems.

We took the following steps to complete the measurements: We modelled the behaviour 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 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.

Contribution statement: I am the third author of this paper. My contributions in this work can be summarised as follows:
• contributing in designing the experiments and modelling.

3.4 Paper IV: Basic Behavioral Models for Software Product Lines: Revisited

In paper II, we provided a comparison of the expressive power of three fundamental models, namely, PL-CCSs, MTSs, and FTSs. As a part of the results it was shown that PL-LTSs (which are considered as the semantic domain for PL-CCSs) are less expressive than FTSs. In this paper, it was assumed that in each step only one variant choice can be resolved. This turns out to be a more restrictive definition than the transition rules of PL-LTSs provided by Gruler et al. in [26]. In this paper, we relax this assumption. Then we provide a new encoding from FTSs into PL-LTSs by considering the more liberal definition of the PL-LTS and prove that PL-LTSs are at lease as expressive as FTSs. Furthermore, we show that the proofs showing that FTSs are at least as expressive as PL-LTSs remain sound and hence we conclude that FTSs and PL-LTSs are equally expressive.

Additionally, we provide a comparative succinctness analysis of the size of PL-LTSs and FTSs. We show that for each PL-LTS an encoding into FTSs exists that results in an FTS with a linear size (in terms of the size of the original PL-LTS). Then, we show that there are FTSs which are encoded into PL-LTSs (by considering any sound encoding) with exponential size in terms of the original FTS. Hence, we conclude that FTSs are exponentially more succinct than PL-LTSs. (In this paper models with finite behaviour are considered.)

Contribution statement: 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.

3.5 Paper V: Comparative Expressiveness of Product Line Calculus of Communicating Systems and 1-Selecting Modal Transition Systems

In this paper we provide a comparison of the expressive power of PL-CCSs and 1MTSs. In paper II, we showed that PL-CCSs are more expressive than MTSs. 1MTSs are extensions of MTSs with (may/must) hyper transitions. The hyper transitions make it possible for modelling a set of alternative behaviour. As a part of the results of this work we point out a few issues regarding the product derivation using the refinement relation of the 1MTSs, e.g., in modelling persistent choice. Then, we propose a new refinement relation for 1MTSs which
addresses these issues. We prove that the new refinement relation still enjoys
the same intuitive properties as original refinement relation of 1MTSs.

Next, to compare the expressive power of PL-CCSs and 1MTSs, we show
that an encoding from PL-LTSs (the models used as the semantic domain for
PL-CCSs) to 1MTSs exist. The main idea behind the encoding is to translate
a set of transitions in the PL-LTSs corresponding to a variant choice to hyper
transitions in the resulting 1MTS. Then we prove that the LTSs implementing
the PL-LTS are also valid implementations of the resulting 1MTS and vice
versa.

Contribution statement: I am the main author of this paper. My contribu-
tions in this work are summarised in the following:

- Investigating the existing issues with the modelling behaviour of SPLs
  using 1MTSs,
- Providing a new refinement relation to address the issues,
- Proving the properties of the new refinement relation,
- Providing the translation from PL-LTSs into MTSs, and
- Proving that the translation is semantic preserving.

3.6 Paper VI: A Classification of Product Sampling for
Software Product Lines

In this paper, we provide a classification for a catalog of studies on product
sampling for software product lines. In this work, first we conduct a search on
the existing work on product sampling. We put a structure on a part of the
extensive work on product sampling by providing detailed insights that can
be used for the purpose of teaching and applying these techniques in practice.
We provide three main characteristics for sampling techniques: the input data
used in the sampling process, the technique used for sampling and also the
performed evaluations. We provide more sub-characteristics and provide more
details. We identify the significant results, as well as gaps for future research.

Contribution statement: I am the main author of this paper. My contribu-
tions in this work are summarised in the following:

- Contributing to the search and literature review on sampling,
- Providing the initial classification and contributing to the modification,
- Providing the running example to explain the main concepts in the paper.
CHAPTER 3. SUMMARY OF CONTRIBUTIONS

3.7 Paper VII: Modal Transition System Encoding of Featured Transition Systems

In Paper II, it was shown that FTSs are more expressive than MTSs (considering finite behaviour). There are specific types of behaviour, e.g., persistence choice and alternative behaviour, which cannot be modelled using MTSs. There has been extensive studies on the theory of MTSs [31], and various tools have been developed for performing analysis on MTSs [10, 20, 5]. Considering this and other advantages of modelling the behaviour using MTSs such as compositionality, in this work we study the connections between FTSs, which are more recent, and MTSs. The main goal of this work is to provide a semantic preserving translation from FTSs into MTSs. This way it is possible to benefit from modelling using a more expressive power while using existing tools for analysis of MTSs. As (for finite behaviour) FTSs are more expressive than MTSs, the result of translating an FTS can be potentially a set of MTSs. As a part of the results we show that the translation is semantic preserving (which means that the set of implementing LTSs of an FTS are equal to the set implementing the resulting set of MTSs). Furthermore, we provide an algorithm for the proposed translation and an implementation. The algorithm is applied to a number of examples.

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

• Providing the translation from FTSs into MTSs,

• Proving that the translation is semantic preserving.
Chapter 4
Conclusion and Perspectives

In this thesis, we provided an overview of challenges for modelling and model-based testing of software product lines. We studied some of the formalisms used for efficient modelling of SPLs. We provided the comparison on the expressive power between four fundamental models, namely FTSs, MTSs, PL-CCSs, and 1MTSs. 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. Moreover, we compared the succinctness of the PL-LTSs and FTSs. Our results show that modelling product lines using FTSs can lead to having exponentially more compact models compared to using PL-LTSs. Furthermore, we pointed out some limitations in modelling the behaviour of SPLs using 1MTSs. We provided a new refinement relation for 1MTSs to address these problems and we proved some of the properties of the proposed refinement relation. Moreover, we show that 1MTSs are at least as expressive as PL-LTSs. In a main part of our work, we have only considered the finite behaviour of the systems and have defined the encodings based on that. Considering the infinite behaviour is a part of our future work. We conjecture that for the same results some of the encodings may lead to systems with finite branching and infinite states. But more investigation is required for providing the proofs. Completing the lattice of expressive power given in [8], by including other formalisms such as disjunctive MTSs and parametric MTSs [7], is another line of our future work regarding modelling SPLs.

As another part of our research on MBT of SPLs, we adapted the test case generation algorithm for the HSI-method by exploiting the idea of delta-oriented modelling for efficient test case generation. The SPL behaviour is presented 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 a line 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 complete 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 priorisation of test cases in the execution of test suites in order to gain more insight about the performance of the complete ioco method.

Moreover, we conducted a survey on product sampling techniques. Such techniques are used for efficient analysis of SPLs as only a subset of products are selected for analysis. we provide a classification for a catalog of studies on product sampling mainly based on characteristics related to the input-data, the technique, and the evaluation of sampling. We provide a number of insights as well as gaps for the future research.
References


REFERENCES


REFERENCES


REFERENCES 31


REFERENCES


Appendix A

Paper I
Delta-Oriented FSM-Based Testing

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi

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],
17] for recent surveys). Among these approaches, the closest to our approach are those developed by Malte Lochau, Ina Schaefer, et al. [8]. 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, $\mu: S \times I \to S$ is the transition function and $\lambda: S \times I \to O$ 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 (\( X \subseteq \mathcal{P} \)) 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:

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

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:

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

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 $S \times q_s \times z_s$, where $r$ is the reset sequence of the FSM and for two sets $A$ and $B$ of sequences, $A \cup B$ denotes the concatenation of two sets and is defined as $\{ \alpha \beta | \alpha \in A \wedge \beta \in B \}$. 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 $s \xrightarrow{\alpha} s'$, the set of all $r \cup q_s \cup i \cup z_{s'}$ 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:

$$\text{core} \langle \text{Feature names} \rangle \{ \langle \text{Java classes and interfaces} \rangle \},$$

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. 2. Core- and delta models of the running example.](image-url)
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 \( M \) of objects \( \mathcal{O} = (S, s_0, I, O, \mu, \lambda) \) is a semantic model for a set \( \mathcal{O} \) of objects from the set \( C \) of classes, if it satisfies the following conditions:

1. \( S \subseteq \text{VAL}^V \), where \( V \subseteq \bigcup \mathcal{O}. m \in \text{Mtds}_m \). \( \text{UsedVars}_m \) is a subset of model variables and fields in \( \mathcal{O} \); this means that each state in \( S \) is an abstract valuation of a subset of model variables and fields.
2. \( I \subseteq \bigcup \mathcal{O}. m \in \text{Mtds}_m \). \( \{\text{Id}_m\} \times \text{VAL}^{\text{Inprms}_m} \); 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:
   - \( (1) \forall o \in \mathcal{O}, m \in \text{Mtds}_m, \text{val} \in \text{VAL}^{\text{Inprms}_m}, i \in I, s, s' \in S \cdot \mu(s, i) = s' \land i = (\text{Id}_m, \text{val}) \Rightarrow \text{Effect}_m(s \downarrow \text{UsedVars}_m, \text{val}) \subseteq s' \downarrow \text{UsedVars}_m. \)
   - \( (2) \forall s \in S. \exists i \in I. (s_0, x) = s \) \( \Rightarrow \exists i \in I. \forall s \in S \cdot \mu(s, i) = s_0 \)
   - \( (3) \forall s \in S. \exists i \in I. \forall s \in S \cdot \mu(s, i) = s_0 \Rightarrow \text{RetVal}_m(s \downarrow \text{UsedVars}_m, \text{val}) = o \).

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 ex-
ample is demonstrated in Fig. 3 (a). This FSM is minimal and it satisfies the
reachability condition. The reset sequence of this FSM is $\text{SetAvl}()$.

![Diagram](image.png)

**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 applica-
tion 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. More-
over, 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 modi-
fications 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 the core 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, x,CheckAvl is an extra model variable included in the state, representing the returned value of 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, µ, λ) and M_d(O_d) = (S_d, s_d, I_d, O_d, µ_d, λ_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 γ : V_d → 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 methods of 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 γ is an FSM M’(O’) = (S’, s_0’, I’, O’, µ’, λ’), where:

- S’ = \{ val ∈ VAL \cup \{0\} \mid va! ∈ S \land va! ∈ S_0 \land \forall v_d \in V_d. \forall v \in V. \gamma(v_d) = v \Rightarrow va! \{v\} = va! \{v_d\} \}; for the composition to be well-defined, we assume V and V_d to be disjoint,
- s_0’ is the initial state such that s_0’ ↓ V = s_0 and s_0’ ↓ V_d = s_d,
- I’ = I \cup I_d
- O’ = O \cup O_d
- µ’ : S’ × I’ → S’, is the transition function. For each i ∈ I’, we distinguish the following three cases:
  - i ∈ I concerns a method call from the core; then, the following condition should be satisfied
    \[ \forall m ∈ MTD. \exists s_1, s_2 ∈ S. s_1 ↓ _{UsedVars_m} s_2 \]
• $i \in L_d$ concerns a method call from delta that does not have any nested call to the core; then, the following condition should be satisfied

$$\forall m \in Mtd, n_k', s \in S' \cdot i = (I_m, \nu) \Rightarrow \left\{ \mu'(s', i) = s'_{1'}, \exists s_{1'}, s \subseteq S_m \cdot s'_{1'} \downarrow \text{UsedVars}_m = s'_{1'} \downarrow \text{UsedVars}_m \land s_{1'} \downarrow \text{UsedVars}_m \land \mu_d(s_{1'}', i) = s_{1''} \right\}$$

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

$$\forall m \in Mtd, n \in MTD, n_k', s \in S' \cdot i = (I_m, \nu) \land n_i = (I_{n_i}, \nu_{n_i}) \Rightarrow \left\{ \mu'(s', i) = s'_{1'}, \exists s_{1'}, s \subseteq S_m \cdot s'_{1'} \downarrow \text{UsedVars}_m = s'_{1'} \downarrow \text{UsedVars}_m \land s_{1'} \downarrow \text{UsedVars}_m = s'_{1'} \downarrow \text{UsedVars}_m \land s_{1'} \downarrow \text{UsedVars}_m \land \mu_d(n_{1}', i) = s_{1''} \right\}$$

- $\lambda': S' \times I' \to O'$ 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:

$$\forall m \in MTD, n \in O', s \in S' \cdot i = (I_m, \nu) \Rightarrow \left\{ \lambda'(s', i) = o \Leftrightarrow \exists s \subseteq S_m \cdot s \downarrow \text{UsedVars}_m = s \downarrow \text{UsedVars}_m \land s \downarrow \text{UsedVars}_m \land \lambda(s, i) = o \right\}$$

  • or $i \in L_d$, then the following condition should hold:

$$\forall m \in Mtd, n \in O', s \in S' \cdot i = (I_m, \nu) \Rightarrow \left\{ \lambda'(s', i) = o \Leftrightarrow \exists s \subseteq S_m \cdot s \downarrow \text{UsedVars}_m = s' \downarrow \text{UsedVars}_m \land \lambda_d(s_{1'}, i) = o \right\}$$

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 $x_{CheckAvl}$ in the delta with the variable $Avl$ 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 proven 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 $DPedLight$, 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 $M(O) = (S, s_0, I, O, \mu, \lambda)$ 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 $q_s \in Q$ 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 = \{ e, \text{ResetAvl()} \} \text{ and } Z = \{ z_{avl}, z_{reset} \} = \{ \{ \text{CheckAvl()} \}, \{ \text{CheckAvl()} \} \}.

6.1 Test-Case Generation for Class Addition

Let 
\[ M_d(Q_d) = (S_d, s_d^0, I_d, O_d, \mu_d, \lambda_d), \]
be the FSM that is composed with core model, with regards to the composition function \( \gamma \), 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 \( Q_d \) and \( Z_d \). The resulting object FSM is 
\[ M'(O') = (S', s'_0, I', O', \mu', \lambda'), \]
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 \( P_d(S_d, E_d) \) 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, that is \( (s_0, s_0^0) \), 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. \((s, s_d')\), where for some \( i \in I' \) which is corresponding to a method call that does not contain any nested method calls, we have \((s_d, i, s_d') \in E_d \)
3. \((s', s_d')\), where for some \( i \in I' \) that contains a method call denoted by \( j \in I \), we have \((s_d, i, s_d') \in E \) 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: \( Q_2 = \{ \{ e, \text{GetReq}(0), \text{GetReq}(1) \} \} \) and \( Z_2 = \bigcup_{s \in S} \{ z_s = \{ \text{CheckLsig()}, \text{CheckRsig()} \} \} \).

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

<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>50</td>
<td>388</td>
<td>194</td>
<td>388</td>
<td>50</td>
</tr>
<tr>
<td>40</td>
<td>389</td>
<td>253</td>
<td>421</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>74</td>
<td>419</td>
<td>624</td>
<td>50</td>
</tr>
<tr>
<td>30</td>
<td>167</td>
<td>751</td>
<td>1049</td>
<td>30</td>
</tr>
</tbody>
</table>

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

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.
References

Appendix B

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

Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi

Basic behavioral models for software product lines: Expressiveness and testing pre-orders
Harsh Beohar, Mahsa Varshosaz, Mohammad Reza Mousavi *
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-CCS)

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 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 language to make the right choice when defining the semantics of their language.

* Corresponding author.
E-mail addresses: harsh.beohar@hhb.se (H. Beohar), mahsa.varshosaz@hhb.se (M. Varshosaz), m.reza.mousavi@hhb.se (M.R. Mousavi).

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 HUG project with award number 20140312*.

http://dx.doi.org/10.1016/j.scico.2015.06.005
0167-6423/© 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/).
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 \((\text{e})\) exclusively for the European market and one-Dollar coins \((\text{d})\) 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 is 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*.

As another alternative, Gruier et al. [17] extended Milner’s CCS [25] into a process calculus called PL-CCS. The extension involves introducing the “binary variant” ⊩ 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 the current paper. For example, there are proposals based on re-using existing process algebras with data (see, e.g., [27–29]) or extending 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 \((P, A, →, →_⊂)\), where \(P\) is a set of states or processes, \(A\) is a set of actions, \(→_⊂ \subseteq P \times A \times P\) is the so-called may transition relation, and \(→ \subseteq →_⊂\) 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 LTS specifies a unique LTS when \(→_⊂ = →\), 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. $\forall P, P', Q \subseteq P, A \quad (P R Q \wedge P \not\xrightarrow{\star} P') \implies \exists Q' \subseteq P \quad Q \not\xrightarrow{\star} Q' \wedge P'RQ'$.
2. $\forall P, Q, Q' \subseteq P, A \quad (P R Q \wedge Q \rightarrow Q') \implies \exists P' \subseteq P \quad P \not\rightarrow P' \wedge P'RQ'$.

A modal specification $P$ refines a modal specification $Q$, denoted $P \leq Q$, if there exists a modal refinement relation $R$ such that $P R Q$. 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, 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:

\[
\forall P, a, P', \phi, \phi' \in \mathbb{B}(F) \iff (P, a, P', \phi, \phi') \in \rightarrow \implies \phi = \phi',
\]
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 $P \xrightarrow{a \phi} Q'$ 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>$s_1 \xrightarrow{\text{dollar}} s_2$</td>
<td>e</td>
</tr>
<tr>
<td>$s_1 \xrightarrow{\text{cappuccino}} s_2$</td>
<td>d</td>
</tr>
<tr>
<td>$s_2 \xrightarrow{\text{coffee}} s_5$</td>
<td>c</td>
</tr>
<tr>
<td>$s_2 \xrightarrow{\text{tea}} s_6$</td>
<td>t</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]:

\[
\Lambda = \{[m, o, b, c, e], [m, o, b, c, e, t], [m, o, b, c, e, r], [m, o, b, c, e, d], [m, o, b, c, d, r], [m, o, b, c, d], [m, o, b, c, d, t], [m, o, b, c, d, t, r], [m, o, b, c, e, p, r, t] \}
\]

### 2.3.2. Deriving valid products

In [15] a class of operators $\Delta_\lambda (\_)$, each parameterized by product configurations $\lambda \in \Lambda$, is introduced to project an FTS into various FTSs, thereby obtaining different products from a product line. Roughly, the operation $\Delta_\lambda (\_)$ prunes away those transitions from a product line whose feature constraints are not satisfied by the product configuration $\lambda$.  

![Feature diagram of the vending machine](image-url)
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:

$$\lambda \models \phi \quad P \overset{\alpha}{\rightarrow} Q \quad \Delta_\lambda(P) \overset{\alpha}{\rightarrow} \Delta_\lambda(Q) .$$

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 $1 \overset{\alpha_6}{\rightarrow} 2$. 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 can never 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 FTS 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 FTS which generalizes Definition 4.

Definition 5. Given an FTS $(P, A, F, \rightarrow, \Lambda)$, an LTS $(P, A, \rightarrow)$, and a product $\lambda \in \Lambda$. 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. $\forall \sigma, \rho, p, q \in \Sigma \cup \{\perp\}$, $P \overset{\rho}{\rightarrow} p \land \lambda \models \phi$ \Rightarrow $\exists q \in \Sigma \cup \{\perp\}$, $q \models \phi$.
2. $\forall \rho, p, q \in \Sigma \cup \{\perp\}$, $P \overset{\rho}{\rightarrow} p \land (q, q) \models \rho$.

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

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 $P \triangleright_\lambda \Delta_\lambda(P)$.

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

2.4. Product line process algebras

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

Definition 6. Let $A = \Sigma \cup \Sigma \cup \{\perp\}$ be the alphabet, where $\Sigma = \{a \mid a \in \Sigma\}$. The syntax of PL-CCS terms $e$ is defined by the grammar $\text{Nil} \mid a.e \mid e + e' \mid e @ e' \mid e \| e' \mid e.f \mid e.L$, where $\text{Nil}$ is the deadlocking process, for each $a \in A$, $a.$ denotes action prefixing, $+ . +$ denotes non-deterministic choice, $@$. denotes binary variant, $\|$. denotes parallel composition, for each $f : A \rightarrow A$, $\langle f \rangle$ denotes renaming by means of $f$, and for each $L \subseteq A \setminus L$ denotes the restriction operator (blocking $\langle L \rangle$ actions) in $L$. In addition, one may define recursive processes by means of process identifiers and equations.

At the first sight, the variant operator $\odot$ 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 $(\_)$, called the optional operator, was also introduced to represent the optional features of a feature diagram. It can be defined in PL-CCS as $\langle P \rangle = P \odot \text{Nil}$.
Example 7. The following process definition specifies the vending machine product line in PL-CCS.

\[\begin{align*}
S_1 &= \text{1e} . \text{s}_2 \oplus \text{1d} . \text{s}_2 \\
S_2 &= \text{sugar} . \text{s}_3 \oplus \text{no sugar} . \text{s}_4 \\
S_3 &= \text{coffee} . \text{s}_5 + \text{tea} . \text{s}_6 + \text{cappuccino} . \text{s}_7 \\
S_4 &= \text{coffee} . \text{s}_{10} + \text{tea} . \text{s}_9 + \text{sugar} . \text{s}_11 \\
S_5 &= \text{pour sugar} . \text{s}_{10} \\
S_6 &= \text{pour coffee} . \text{s}_{11} \\
S_7 &= \text{pour sugar} . \text{s}_8 \\
S_8 &= \text{pour coffee} . \text{s}_9 \\
S_9 &= \text{pour tea} . \text{s}_12 \\
S_{10} &= \text{pour coffee} . \text{s}_{12} \\
S_{11} &= \text{ring tone} . \text{s}_{13} + \text{skip} . \text{s}_{13} \\
S_{12} &= \text{take cup} . \text{s}_1.
\end{align*}\]

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 \([L, R, ?]\) with \(I\) being an index set) that records the selection made in past about the alternative features.

Definition 7. Let \((L, R, ?)\) 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 \((\mathcal{P} \times \{L, R, ?\}, A, I, \rightarrow)\) consisting of a set of states \(\mathcal{P} \times \{L, R, ?\}\), a set of actions \(A\), and a transition relation \(\rightarrow \subseteq (\mathcal{P} \times \{L, R, ?\}) \times (A \times \{L, R, ?\}) \times (\mathcal{P} \times \{L, R, ?\})\) satisfying the following restrictions:

1. \(\forall_{p, a, q, v, v'}(p, v) \xrightarrow{a,v} (q, v') \Rightarrow v' = v\).
2. \(\forall_{p, a, q, v, v'}(p, v) \xrightarrow{a,v} (q, v') \land v' \neq v \Rightarrow \exists \_ (v'(i) \neq v(i) \land \forall_{j \neq i} v'(j) = v(j)).\)
3. \(\forall_{p, a, q, v, v'}(p, v) \xrightarrow{a,v} (q, v') \land v(i) \neq v(i) \Rightarrow v'(i) = v(i).\)

Notice that, as a consequence of item 2 in Definition 7, for any transition \((p, v) \xrightarrow{a,v} (q, v')\), if \(v' \neq v\), we can find a unique \(i \in I\) such that \(v'(i) \neq v(i)\) and \(\forall_{j \neq i} v'(j) = v(j)\). 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 \(v, v' \in (L, R, ?)\).

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

1. \(\forall_{p, q, a, v, v'}((p, v) \mathcal{R}_\theta p \land (P, v) \xrightarrow{a,v} (Q, v') \land v' \sqsubseteq \theta) \Rightarrow \exists q \_ p \xrightarrow{a,v} (Q, v') \mathcal{R}_\theta q,\)
2. \(\forall_{p, a, q, p}((p, v) \mathcal{R}_\theta p \land (P, v) \xrightarrow{a,v} (Q, v') \land v' \sqsubseteq \theta) \Rightarrow \exists q \_ (P, v) \xrightarrow{a,v} (Q, v') \mathcal{R}_\theta q,\)

A state \(p \in \mathcal{P}\) 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 \((P, v) \uparrow_{\mathcal{R}_\theta} p\), 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 \(\mathbb{M}\) 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 \(\mathbb{M} = (\mathbb{M}, [\;])\), where \(\mathbb{M}\) is the class of all intended product line models or specifications (in our case: MTSS, FTSS, and PL-LTSs) and \([\;] : \mathbb{M} \to 2^P\) is the semantic function mapping a product line specification to a set of LTSs.
Definition 11. An encoding \( E : \mathcal{M} \to \mathcal{M}' \) from a product line structure \( \mathcal{M} = (0 \mathcal{E}, \emptyset, [ ]) \) into a product line structure \( \mathcal{M}' = (\mathcal{M}', [ ]) \) is a function \( E : \mathcal{M} \to \mathcal{M}' \) satisfying the correctness criterion \([ ] = E \circ [ ] \).

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 \( E : \mathcal{M} \to \mathcal{M}' \).

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 \( E : \mathcal{M} \to \mathcal{M}' \).

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

The following two theorems relate the expressiveness of PL-FTSSs and MTSSs.

Theorem 1. The class of PL-FTSSs is at-least as expressive as the class of MTSSs.

Proof. Consider the MTSS \( (P, A, \to, \longrightarrow) \) and some \( P \in P \). Let \( \to \subseteq P \times A^* \times P \) be the reachability relation defined as follows: \( P \to P' \) if and only if \( \forall q \in P \) \( \exists p \in A^* \) \( \exists q' \in P' \) such that \( \forall q \in P \). Let \( \text{tr}(P) \) be the set of traces generated by \( P \), i.e., \( \text{tr}(P) = \{ s \mid \exists q \in P \to q \}. \) For state \( P \), we define a family of transition relations parameterized by the traces of \( P \) as follows: \( \frac{\Delta}{\Delta} \text{tr}(P) \). We drop the subscript \( P \) from the family of transition relations whenever it is clear from the context.

Next, we construct a PL-FTSS whose configuration vectors are functions of type

\[
\text{tr}(P) \to \bigcup_{v \in \text{tr}(P)} \{ L, R, ? \}
\]

The transition relation between the states of PL-FTSSs is defined as the smallest relation satisfying:

\[
\begin{align*}
\Delta &\subseteq Q \quad s \in \text{tr}(P) \quad v' = v \restriction ((Q, a, Q') \to L) \\
\Delta &\subseteq Q \quad s \in \text{tr}(P) \quad v' = v \restriction ((Q, a, Q') \to R) \\
\Delta &\subseteq Q \quad s \in \text{tr}(P) \quad v' = v \restriction ((Q, a, Q') \to ?)
\end{align*}
\]

where \( Q \to Q \subseteq Q \) \iff \( Q \to Q \) and the expression \( v' = v \restriction ((Q, a, Q') \to X) \) for \( X \subseteq \{ L, R \} \) is defined in the following way:

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

We fix \( E(P) = (P, v_0) \), where \( v_0(s)(X) = 0 \) for \( s \in \text{tr}(P) \). Furthermore, we let the symbols \( \theta, \theta' \) range over the total configuration vectors, i.e., the functions of type \( \text{tr}(P) \to \{ L, R \} \). Now we are in the position to show that \( \left[ P \right] = \left[ E(P) \right] \), where \( [P] = \{ \theta \mid \theta \in \text{tr}(P) \} \) and \( \left[ E(P) \right] = \{ \theta \mid \exists q \in E(P) \to q \}. \) We divide the proof obligation into two obligations: \( \left[ E(P) \right] \subseteq \left[ P \right] \) and \( \left[ P \right] \subseteq \left[ E(P) \right] \), which we prove below.

\( \left[ E(P) \right] \subseteq \left[ P \right] \): Let \( p \) be a state in an LTS such that \( p \to p \), for some \( \theta \). Define a relation \( Q \to Q \equiv \exists v_0, v \restriction ((Q, a, Q') \to \theta(\bar{Q})) \) for \( a \in \text{tr}(P) \).

1. Let \( \Delta \subseteq Q \) and \( Q \to Q \). Then, \( (Q, 0 \to Q) \wedge v \restriction \theta \wedge P \to Q \wedge P \to q \) for some \( v, s \). Clearly, \( s \in \text{tr}(P) \). Let \( v' = v \restriction ((Q, a, Q') \to (Q, a, Q')) \). Note that \( \theta(\bar{Q}) \to Q \) can be either \( L \) or \( R \). Then, we find \( v' \restriction \theta \). Now from the transfer property of \( \to \) we find a \( q' \) such that \( q \to q' \land (Q, v') \to q' \). Hence, \( Q \to Q \).

2. Let \( Q \to Q \) and \( Q \to Q \). Trivial.

\( \left[ P \right] \subseteq \left[ E(P) \right] \): Let \( p \) be some state in an LTS such that \( p \to p \). 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 every maximal trace \( s \in \text{tr}(p) \), we know that there is a unique execution \( e \) starting from \([p]_s \) such that \( dom(e) = s \), where \([p]_s \) is the transition system modulo strong bisimulation defined in the standard way. Therefore, for any maximal trace...
Proof. Let $(Q, v) \triangleright \phi$ and $(Q, v') \triangleright \phi$. Then, from the construction of $R_{\nu}$, we have $v \equiv \theta_p \wedge s \in Z(p)$ and $P \triangleright Q \wedge Q \not\leq e_i(s') \wedge q \in e_i(s')$, for some $s \in Z(p)$, $s' \leq s$. Thus, by the construction of $R_{\nu}$, we have $(Q', v') \triangleright Q$. Furthermore, since $q$ is reachable from $p$ and $q \triangleright q'$, there is a maximal trace $\tau \in Z(p)$ such that $e_i(\tau) = q$ and $e_i(\tau) = q'$ (for some $s' \leq s$). So let $v' = v \upharpoonright (Q, a, Q') \to L$ and thus $v' \equiv \theta_p$. Moreover, we find $(Q, v) \overrightarrow{\nu} (Q', v')$. Thus, by the construction of $R_{\nu}$, we have $(Q', v') \triangleright Q$. □

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

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 $P = a.Nil \oplus b.Nil$ cannot be encoded using any sound encoding (satisfying Definition 11) to an MTs. To show this, observe the transition systems of the derived LTSs $p$ and $q$ drawn in Fig. 4 under $\sigma = L$ and $\sigma' = K$.

Suppose there is an encoding $E$ satisfying Definition 11. Clearly, $(P, 7) \models p$ and $(P, 7) \models q$. Then by correctness of $E$ we have $E((P, 7) \equiv p \wedge E((P, 7) \equiv q$. Thus, we can derive the following transitions (for some modal states $P_a$, $P_b$) from the transfer property of modal refinement:

$$E((P, 7) \overrightarrow{\sigma} P_a \wedge E((P, 7) \overrightarrow{\sigma} P_a).$$

Therefore, there exists a state $r$ of the following form: $r \overrightarrow{\sigma} r'$ and $r \overrightarrow{\sigma} r''$ (for some $r', r''$) such that $E((P, 7) \leq r$. And by correctness of $E$ we get $(P, 7) \models r$ or $(P, 7) \models r$. However, $(P, 7) \models r$ and $(P, 7) \models r$. □

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 $(P \times \{-1, 1\})$, $A, \to$ be a PL-LTS. The corresponding FTSS is denoted by $(P \times \{-1, 1\}$, $A, F, \to, \Lambda)$, where:

- $F = \bigcup_{L \subseteq \{1, 1\}} 0, 1, \Lambda = \bigcup_{L \subseteq \{1, 1\}} 0, 1$.
- The transition relation $\to$ is defined in the following way:

$$\begin{align*}
(P, v) \overset{a, v'}{\to} (Q, v) & \quad (P, v) \overset{a, v'}{\to} (Q, v') \quad \phi = \psi(i) \quad \Sigma(i, v, v') \\
(P, v) \overset{a, \psi}{\to} (Q, v) & \quad (P, v) \overset{a, \psi}{\to} (Q, v') \quad \psi(i) = \psi(j).
\end{align*}$$

where $\Sigma(i, v, v') \iff \psi(i) \neq \psi(j) \wedge \forall_{f \neq \psi} \psi(j) = \psi(j)$. Fig. 4. A PL-LTS (left) that cannot be encoded as an MTs.
For any \((P, v) \in \mathcal{P} \times \{L, R, ?\}^l\), we fix \(E(P, v) = (P, v)\). Let \(\[(P, v)\] = \{p \mid \exists_{\theta \in \Lambda} (P, v) \vdash_\theta p\}\) and \(\{P, v\}' = \{p \mid \exists_{\theta \in \Lambda} (P, v) \vdash_\theta p\}\). In the next step, we need to show that \(\[(P, v)\] = \{P, v\}'\).

\(\{[(P, v) \subseteq \{P, v\}'\}: \) Let \(p \in \{P, v\}'.\) Then \((P, v) \vdash_\theta p\), for some \(\theta \in \{L, R\}\). Define a configuration \(\lambda_\theta \in \Lambda\) as follows:

\[
\lambda_\theta(i) = T \iff \theta(i) = L \quad \text{and} \quad \lambda_\theta(R) = T \iff \theta(i) = R.
\]

Furthermore, consider the following relation \(R_{\lambda_\theta}\) such that \((Q, v') \in R_{\lambda_\theta} q \iff (Q, v') \vdash_\theta q\). It is straightforward to show that \(R_{\lambda_\theta}\) is a product derivation relation.

\(\{(P, v) \subseteq \{P, v\}\}\}: \) Let \(p \in \{P, v\}'.\) Then \(P \vdash_\theta p\) for some \(\theta \in \{L, R\}\), be a configuration vector defined as \(\theta(i) = L \iff \lambda(L) = T \quad \text{and} \quad \theta(i) = R \iff \lambda(R) = T\). Define a relation \(R_{\theta_0}\) such that \((Q, v') \in R_{\theta_0} q \iff (Q, v') \vdash_\theta q\). It is straightforward to verify that \(R_{\theta_0}\) is a product derivation relation for PL-LTS.

**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 \(\{L, R, ?\}^l\) (for some index set \(I\)) such that \(\[P\] = [E(P)\']\), where \(\[P\] = \{p \mid \exists_{\theta \in \Lambda} P \vdash_\theta p\}\), \(E(P) = (Q, v_0)\) (for some state \(Q\) and configuration vector \(v_0 \in \{L, R, ?\}^l\) in an PL-LTS), and \(\{E(P)\]' = \{p \mid \exists_{\theta \in \Lambda} (Q, v) \vdash_\theta p\}\).

Clearly, the transition systems \((P_x, p, v_0), \{x \mid \{P_x, x, p\}'\}\) for \(x \in \{a, b, c\}\) are three valid products of the given FTS \(P\), i.e., \(\{P_a, P_b, P_c\} \subseteq \{P\}\). So from the correctness requirement of \(E\) we have \(\{P_a, P_b, P_c\} \subseteq \{E(P)\}'\). Let \(\theta_0, \theta_1, \theta_2\) be the corresponding total configuration vectors that derives the products \(P_a, P_b, P_c\), respectively. Thus, for every \(x \in \{a, b, c\}\) we have \(v_0 \in \theta_0.\) Furthermore, from the transfer property of product derivation we find \((Q, v_0) \xrightarrow{\lambda_\theta} (Q, v)\) such that \(v_0 \in \theta_0\) for \(x \in \{a, b, c\}\). Clearly, \(v_0 \neq v_0\) for any \(x \in \{a, b, c\}\) because otherwise we can derive a transition system which contains \(\exists_0 \exists_1 a \rightarrow q\) because \(v_0 = v_0 \circ \theta_0\). Therefore, let \(\theta_a, \theta_b, \theta_c \in I\) be the unique elements such that for every \(x \in \{a, b, c\}\) we have \(v_0(x) \neq v_0(i) \land v_0(x) = v_0(i)\) (recall Condition (7(2))).

Next, we show that if \(a \neq i, b \neq i, c \neq i\) then we can derive a product which has a choice between a, c. Since \(i_a, i_b\) are the only elements whose values are changed by \(\theta_0, \theta_0\), so from Condition (7(3)) we have \(v_0(i_a) = v_0(i_b) = 7\). Define a function \(\theta'_0\) as follows:

\[
\theta'_0(i) = \begin{cases} 
\theta(i) & \text{if } i \neq i_a \land i \neq i_b, \\
\theta(i) & \text{if } i = i_a, \\
\theta(i) & \text{if } i = i_b \land \theta(i_b) = 0, \\
\theta(i) & \text{if } i = i_b \land \theta(i_b) = 1, \\
\theta(i) & \text{if } i = i_c \land \theta(i_b) = 1, \\
\theta(i) & \text{if } i = i_c \land \theta(i_c) = 0.
\end{cases}
\]

Next, we show that \(v_0 \subseteq \theta'_0\). If \(i \neq i_a \land i \neq i_b\) then clearly \(v_0(i) \subseteq \theta'_0(i)\) because \(\theta'_0(i) = \theta(i)\). If \(i = i_a\) or \(i = i_b\) then \(v_0(i) = v_0(i) = 7\). Thus, \(v_0(i) \subseteq \theta'_0(i)\). Next, we show that \(v_0 \subseteq \theta'_0\) if \(i \neq i_a \land i \neq i_b\) then \(v_0(i) = v_0(i) = 7\). Thus, \(v_0(i) \subseteq \theta'_0(i)\). Lastly, if \(i = i_a\) then \(v_0(i) \subseteq \theta'_0(i)\) because \(\theta'_0(i) = \theta(i)\). Thus, \(v_0 \subseteq \theta'_0\).

As a result, we can derive a product that contains a choice between a, c by using \(\theta'_0\); 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 \(i_a = i_c\) then we can derive a product which has a choice between either a, b or a, c. It suffices to show that \(v_0(i_a) \subseteq \theta_3(i_a)\) or \(v_0(i_c) \subseteq \theta_b(i_a)\) because for every \(i \neq i_b\) we have \(v_0(i) = v_0(i) = v_0(i) = \theta_3(i)\). We claim that \(v_0(i_a) \neq 7\). Suppose otherwise \(v_0(i_a) = 7\). Then, since \(i_a = i_c\) we find \(v_0(i_c) \subseteq \theta_3(i)\). As a result, from the transfer property of \(v_0\), there must be a \(p\) such that \(\exists_0 \exists_1 a \rightarrow q\) otherwise \(E(P), v_0(p)\). Thus, \(v_0(i_b) \neq 7\). Likewise, we can prove that \(v_0(i_c) \neq 7\). Thus, \(v_0(i_a), v_0(i_c) \in \{L, R\}\). And since \(\theta_3\) is a function whose co-domain is the set \(\{L, R\}\) we have either \(v_0(i_a) \subseteq \theta_3(i_a)\) or \(v_0(i_c) \subseteq \theta_3(i_c)\). Hence, we can derive a product that contains a choice between either a, b or b, c by using \(\theta'_3\); however, such a product is clearly not a valid product of the given FTS \(P\) as it violates the condition \(\Lambda\). Likewise, if \(i_b = i_c\).
(or $i_a = i_b$) then we can show that using $\theta_a$ ($\theta_b$) we can derive a product which has a choice between either $a, b$ ($a, c$) or $a, c$ ($b, c$). □

We can hence summarize the results of this section by the following diagram:

$$\text{MTSs} \rightarrow \text{PL-LTS} \rightarrow \text{FTSs},$$

where the arrow $\rightarrow$ 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 $\lambda(f) = T, \lambda(f') = 1$ and $\lambda'(f) = \lambda'(f') = T$. 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, $P \models_\lambda P$ and $P \not\models_\lambda Q$. Then by correctness of $E$ we have $E(P) \leq P$ and $E(Q) \leq Q$. Thus, we can derive the following transitions (for some modal states $P_a, P_b$) from the transfer property of modal refinement:

$$E(P) \xrightarrow{a \cdot_\lambda} P_a \quad E(P) \xrightarrow{b \cdot_\lambda} P_b.$$  

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 $E(P) \leq r$. And by correctness of $E$ we get $P \models_\lambda r$ or $P \models_\lambda r'$. However, $P \models_\lambda r$ and $P \not\models_\lambda r$.

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]:

$$t ::= \text{Succ} \mid \text{Fail} \mid a \cdot t \mid a \cdot \neg t \mid t_1 \land t_2 \mid t_1 \lor t_2 \mid \forall t \mid \exists t.$$

a. Throughout this section, we assume that the product line structure under investigation is image finite.
 Fig. 7. Operational interpretation of test experiments.

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 at experiments 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 at 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:

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

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

Fig. 7 provides the operational interpretation of experiment expressions over an MTS. Note that only the rules of the expressions at \( P \) and at \( P \) (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 \( \mathbb{O} = \{\bot, \top\} \). 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 \( P[\mathbb{O}] = \{\bot\} \subseteq \{\bot, \top\} \subseteq \{\top\} \) ). The semantics of the operators \( \land, \lor, \forall, \exists \) over the Plotkin powerdomain \( P[\mathbb{O}] \) can be found in [23]. In addition, given an MTS \( (P, A, \rightarrow, \mathbb{O}, \cdot) \), we define the function \( O : T \times P \rightarrow P[\mathbb{O}] \) in the following way:

\[
O(t, P) = (\top \mid (t \parallel P) \rightarrow \top) \cup (\bot \mid (t \parallel P) \rightarrow \bot).
\]

\[
O(t, P) = (\top \mid (t \parallel P) \rightarrow \top) \cup (\bot \mid (t \parallel P) \rightarrow \bot).
\]
where $\rightarrow$ is the reflexive and transitive closure of the transition relation $\rightarrow$ defined in Fig. 7. Similar to the operational semantics, we re-use the same notation to denote the result of executing a test expression of a test on an LTS, i.e., for an LTS $(P, A, \rightarrow)$, we write $O(t, p)$ to denote the results of executing test $t$ on $p$.

The following property is immediate from the rules of Fig. 7 (in particular, rules 19 and 23) on experiment expressions.

**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 $\llbracket \cdot \rrbracket_A$ (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 $\subseteq$ and the modal refinement relation $\preceq$.

Consider the set of all HML formulae $\Phi$ generated by the following grammar:

$$\phi ::= \bot \mid T \mid (a)\phi \mid [\phi \land \phi'] \mid [\phi \lor \phi'] \mid [\phi].$$

The semantics of $\bot$, $T$, $\land$, $\lor$ is standard, while the nonstandard semantics of $(a)\phi, [\phi]$ is given as follows [41]:

1. $P \models (a)\phi \iff \exists P. P \models_{\phi} P' \land P' \models \phi$.
2. $P \models [\phi] \iff \forall P. P \models_{\phi} P' \rightarrow P' \models \phi$.

Following [23], we give a transformation $\llbracket \cdot \rrbracket_A : \Phi \rightarrow T$ of HML formulae to the set of test expressions.

\[
\begin{align*}
\llbracket \bot \rrbracket_A &= \text{SUCC}, \\
\llbracket T \rrbracket_A &= \text{FAIL}, \\
\llbracket \phi \land \phi' \rrbracket_A &= \llbracket \phi \rrbracket_A \land \llbracket \phi' \rrbracket_A, \\
\llbracket \phi \lor \phi' \rrbracket_A &= \llbracket \phi \rrbracket_A \lor \llbracket \phi' \rrbracket_A, \\
\llbracket (a)\phi \rrbracket_A &= \forall a \llbracket \phi \rrbracket_A, \\
\llbracket [\phi] \rrbracket_A &= \exists \llbracket \phi \rrbracket_A.
\end{align*}
\]

By setting the above technical machinery, we now prove that an MTS satisfies a HML formula $\phi$ if and only if it passes the test $\llbracket \phi \rrbracket_A$.

**Lemma 4.** Let $P$ and $p$ be a state in an MTS and an LTS, respectively. Then, for any $\phi \in \Phi$ we have

$$P \models \phi \iff O([\phi]_A, p) = [T] \quad \text{and} \quad p \models \phi \iff O([\phi]_A, p) = [T].$$

**Proof.** The proof is by induction on $\phi$; the cases for $\bot$, $T$, $\land$, $\lor$ are straightforward.

1. Let $\phi = (a)\psi$ and $P \models \phi$. Then,

\[
\begin{align*}
\forall P. P \models_{\phi} P' \rightarrow P' \models \psi' &\iff \forall P. P \models_{(a)\psi} P' \rightarrow O([\psi']_A, P') = [T] \quad \text{(Induction hypothesis)} \\
\iff O([\psi]_A, P) = [T] &\iff \forall O([\psi]_A, P) = [T] \quad \text{(Rule 5 and Definition of O)} \\
\iff \forall O([\psi]_A, P) = [T], &\iff O([\forall a \phi]_A, P) = [T] \quad \text{(Lemma 3 : O(\forall t, p) = O(t, p))} \\
\iff O([\phi]_A, P) = [T].
\end{align*}
\]

2. Let $\phi = [\phi]$ and $P \models \phi$. Then,

\[
\begin{align*}
\exists P. P \models_{\phi} P' \land P' \models \psi' &\iff \exists P. P \models_{\exists a \phi} P' \land O([\psi']_A, P') = [T] \quad \text{(Induction hypothesis)} \\
\iff \exists P. P \models_{\exists a \phi} P' \land O([\psi]_A, P) = [T] &\iff \exists O([\exists a \phi]_A, P) = [T] \quad \text{(Rule 5 and Definition of O)} \\
\iff \exists O([\forall a \phi]_A, P) = [T], &\iff O([\exists a \phi]_A, P) = [T] \quad \text{(Lemma 3 : O(t, p) = O(\exists t, p))} \\
\iff O([\phi]_A, P) = [T].
\end{align*}
\]

\[\Box\]
Theorem. Let $P$ and $p$ be states in an MTS and an LTS, respectively. Then,
\[\forall_{\text{init}} O(t, P) \subseteq O(t, p) \Rightarrow P \preceq p.\]

Proof. Suppose $\forall_{\text{init}} O(t, P) \subseteq O(t, p)$. In lieu of the modal characterization 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, $P \preceq p$. Consider the test $t = \delta\delta\text{SUCC}$. Clearly, $O(t, P) = \{\bot, \top\}$ and $O(t, p) = \{\bot\}$. However, $\{\bot, \top\} \not\subseteq \{\bot\}$. Thus, in order to obtain a full characterization 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:
\[t ::= \text{SUCC} \mid \text{FAIL} \mid \exists t_1 \mid t_1 \land t_2 \mid t_1 \lor t_2.\]

Then, $\forall_{\text{init}} O(t, P) \subseteq O(t, p) \Leftrightarrow P \preceq p.$

It follows also from Corollary 1 that if an LTS is not a valid product of an MTS, i.e., $P \not\equiv p$, 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 apart from the product line.

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 $t = 31d$ sugar coffee pour sugar pour coffee SUCC 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, $P \not\equiv p$.

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 $(P, A, F, \rightarrow, A)$. It should not be surprising that only the semantics of the tests of the form $\delta\delta\text{SUCC}$
and $\hat{t}$ 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.

$$
\begin{align*}
& \frac{P \xrightarrow{\phi} P'}{\hat{t} \parallel P \rightarrow \top \parallel P'} \quad (3') \\
& \frac{\exists Q. P \xrightarrow{\hat{t}} Q \quad \lambda \models \phi}{\hat{t} \parallel P \rightarrow \bot} \quad (4') \\
& \frac{P \xrightarrow{\phi} P'}{\hat{t} \parallel P \rightarrow \top \parallel P'} \quad (5') \\
& \frac{\exists Q. P \xrightarrow{\hat{t}} Q \quad \lambda \models \phi}{\hat{t} \parallel P \rightarrow \top} \quad (6').
\end{align*}
$$

Second, we define the family of observation functions (parameterized by the product configurations) which essentially evaluates an expression experiment interacting with a specification modeled as an FTS.

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

In a similar vein, we also give an interpretation of a test $t \in T$ over a PL-ITS $(\mathcal{P} \times \{ L, R, ? \})^\top A \rightarrow \rightarrow$, where only rules 3–6 are modified and the remaining operational rules of the operators (except at, $\hat{t}$) are unchanged.

$$
\begin{align*}
& \frac{(P, v) \xrightarrow{\phi} (P', v') \quad v' \equiv \theta}{\hat{t} \parallel (P, v) \rightarrow \top \parallel (P, v')} \quad (3'') \\
& \frac{\exists Q. (P, v) \xrightarrow{\hat{t}} (Q, v') \quad v' \equiv \theta}{\hat{t} \parallel (P, v) \rightarrow \bot} \quad (4'') \\
& \frac{(P, v) \xrightarrow{\phi} (P', v') \quad v' \equiv \theta}{\hat{t} \parallel (P, v) \rightarrow \top \parallel (P, v')} \quad (5'') \\
& \frac{\exists Q. (P, v) \xrightarrow{\hat{t}} (Q, v') \quad v' \equiv \theta}{\hat{t} \parallel P \rightarrow \top} \quad (6'').
\end{align*}
$$

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

$$
O_{\theta}(t, P) = \{ T \mid t \parallel (P, v) \rightarrow \top \} \cup \{ \bot \mid t \parallel (P, v) \rightarrow \bot \}.
$$

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

Lemma 5. Let $P$ be a state in an FTS 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 FTS specification, $p$ be state in an LTS, and $\lambda$ be a product. Then,

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

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 $P \xrightarrow{\phi} Q, \lambda \models \phi$, and for all $q$, if $p \xrightarrow{\phi} q$ then $Q \not\vdash_{\lambda} q$. Let $p(a) = \{ q \mid p \xrightarrow{\phi} q \}$. 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,

   $$
   \begin{align*}
   T \in O_\lambda(a \text{SUCC}, P) & \quad (\therefore P \xrightarrow{\phi} Q, \lambda \models \phi) \\
   \Rightarrow \exists O_\lambda(a \text{SUCC}, P) = \{ T \} & \quad (\text{Truth table of } \exists) \\
   \Rightarrow O_\lambda(\exists \text{SUCC}, P) = \{ T \} & \quad (\text{Lemma 5: } O_\lambda(\exists t, s) = \exists O_\lambda(t, s)).
   \end{align*}
   $$

   But, $O(a \text{SUCC}, p) = \{ \bot \}$; thus,

   $$
   O(\text{SUCC}, p) = \exists O(a \text{SUCC}, p) = O(\exists \text{SUCC}, p) = \{ \bot \}.
   $$

   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 $O_\lambda(t_i, Q) \neq O(t_i, q_i)$.

   (⇐) Suppose otherwise, $p (\lambda) = \{ q_1, \ldots, q_n \}$. Then, by induction hypothesis there exists sub-tests $t_1, \ldots, t_n$ such that $O_\lambda(t_i, Q) = O(t_i, q_i)$.
i. Either, $O_s(t_1, Q) = \{ \top \} \land O(t_1, q_i) = \{ \bot \}$. Let $t' = t_1 \land \cdots \land t_n$. Consequently,

$$
\top \in O_s(\bar{a}t', P) \quad \therefore O_s(t_1, Q) = \{ \top \}
$$

$$
\Rightarrow \exists O_s(\bar{a}t', P) = \{ \top \} \quad \text{(Truth table of $\exists$)}
$$

$$
\Rightarrow O_s(\bar{a}t', P) = \{ \top \} \quad \text{(Lemma 5: $O_s(\exists t, P) = \exists O_s(t, P)$)}
$$

But,

$$
O(\bar{a}t', p) = \{ \bot \} \quad \therefore O(t_1, q_i) = \{ \bot \}
$$

$$
\Rightarrow \exists O(\bar{a}t', p) = \{ \bot \} \quad \text{(Truth table: $\exists \{ \bot \} = \{ \bot \}$)}
$$

$$
\Rightarrow O(\exists a t', p) = \{ \bot \} \quad (O(\exists t, p) = \exists O(t, p) [23]).
$$

Hence, a contradiction.

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

$$
\bot \in O_s(\bar{a}t', Q) \quad \therefore O_s(t_1, Q) = \{ \bot \}
$$

$$
\Rightarrow \forall O_s(\bar{a}t', Q) = \{ \bot \} \quad \text{(Truth table of $\forall$)}
$$

$$
\Rightarrow O_s(\forall a t', Q) = \{ \bot \} \quad \text{(Lemma 5: $O_s(\forall t, Q) = \forall O_s(t, Q)$)}
$$

Furthermore,

$$
O(\bar{a}t', p) = \{ \top \} \quad \text{(since $O(t_1', q_i) = \{ \top \}$)}
$$

$$
\Rightarrow \forall O(\bar{a}t', p) = \{ \top \} \quad \forall \{ \top \} = \{ \top \}
$$

$$
\Rightarrow O(\forall a t', p) = \{ \top \}
$$

Hence, a contradiction.

2. Or, there exists a $q$ such that $p \models q$ and for all $Q, \phi$, if $P \models Q \land \lambda \models \phi$, then $Q \models q$. Due to image finiteness assumption, we know that the set $P(a) = \{ Q \mid \exists \emptyset P \models Q \land \lambda \models \phi \}$ is finite.

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

$$
O_s(\bar{a}t', P) = \{ \top \} \quad \therefore P(a) = \emptyset
$$

$$
\Rightarrow \exists O_s(\bar{a}t', P) = \{ \top \} \quad \text{(Truth table: $\exists \{ \top \} = \{ \top \}$)}
$$

$$
\Rightarrow O_s(\forall a t', P) = \{ \top \} \quad \text{(Lemma 5: $O_s(\forall t, P) = \forall O_s(t, P)$)}
$$

But, $O(\exists a t, p) = \{ \bot \}$ (since $p \models q$, $O(\exists a t, q) = \{ \bot \}$). Thus, $O(\forall a t, p) = \forall O(\exists a t, p) = \{ \bot \}$, 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 $O_s(t_1, Q_i) \neq O(t_1, q_i)$.

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

$$
\bot \in O(\bar{a}t', p) \quad \therefore O(t_1, q_i) = \{ \bot \}
$$

$$
\Rightarrow \forall O(\bar{a}t', p) = \{ \top \} \quad \text{(Truth table of $\forall$)}
$$

$$
\Rightarrow O(\forall a t', p) = \{ \top \} \quad \forall O(t, p) = O(\forall t, p) [23].
$$

Hence, a contradiction.

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

$$
O_s(\bar{a}t', P) = \{ \bot \} \quad \text{(since $O_s(t', Q_i) = \{ \bot \}$)}
$$

$$
\Rightarrow \exists O_s(\bar{a}t', P) = \{ \bot \} \quad \exists \{ \bot \} = \{ \bot \}
$$

$$
\Rightarrow O_s(\forall a t', P) = \{ \bot \}.
$$
Furthermore, \( O(t', q) = O(t_1, q) \land \cdots \land O(t_n, q) = \{\top\} \). Thus,

\[
\top \in O(\bar{a}'t', p) \quad \text{(since } O(t', q) = \{\top\})
\]

\[
\Rightarrow \exists \bar{a} O(\bar{a}'t', p) = \{\top\} \quad (\text{Truth table of } \exists)
\]

\[
\Rightarrow O(\bar{a}'t', p) = \{\top\} \quad (\exists O(t, p) = O(\bar{a}t, p) [23])
\]

Hence, a contradiction follows.

\(\Rightarrow\) Suppose \( P \vdash_a p \). We show by induction on \( t \) that \( O_3(t, P) = O(t, p) \). The cases when \( t = \text{Succ}, \text{Fail}, t_1 \lor t_2, t_1 \land t_2, \text{W}' \), \( \bar{a}'t' \) are straightforward. The interesting cases are the following:

1. Let \( t = \bar{a}'t' \).
   (a) Let \( \top \in O_3(at', P) \). Then,
   \[
P \xrightarrow[\phi, Q]{a_0} q \quad \text{for some } \phi, Q
   \]
   \[
\Rightarrow p \xrightarrow{} q \quad (\vdash \top \vdash_a p)
   \]
   \[
\Rightarrow \top \in O_3(t', Q) \Rightarrow \top \in O(t', q) \quad \text{(Induction hypothesis)}
   \]
   \[
\Rightarrow \top \in O(at', p).
   \]

   (b) Let \( \bot \in O_3(at', P) \). Then we have the following cases:
   i. Either \( P \xrightarrow[]{} Q \land \lambda \models \phi \land \bot \in O_3(t', Q) \), for some \( Q, \phi \). Similar to Case (a).
   ii. Or, \( \exists Q, \phi \in P \xrightarrow[]{} Q \land \lambda \models \phi \). Then, \( O_3(at', P) = \{\bot\} \). Suppose otherwise, \( \top \in O(at', P) \). Then, \( \exists Q, p \xrightarrow[\phi, Q]{a} q \). But, \( P \vdash_a p \). Thus, \( \exists Q, \phi \in P \xrightarrow[\phi, Q]{a} Q \land \lambda \models \phi \), which is a contradiction.

(c) Let \( \top \in O(at', p) \). Similar to Case (a).

(d) Let \( \bot \in O(at', P) \). Similar to Case (b).

2. Let \( t = \bar{a}'t' \). Similar to the Case (1).

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,

\[
(P, v) \vdash_\theta p \Leftrightarrow \forall \tau \in \mathcal{T} O_0(t, P, v) = O(t, p).
\]

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

### 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.
References

Appendix C

Paper III
Complete IOCO Test Cases: A Case Study

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

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

Sofia Costa Paiva
Instituíto 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

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

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 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 least 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
as the Body Comfort System [11] and the Turn Indicator Lights [14] from the automotive domain. For these case studies, we produce a set of mutants, within a fault domain, as their incorrect implementations. We then apply the two techniques by applying the respective test case generation algorithms, gathering execution time and fault classification data, and analyzing them. The results point out that both methods reveal all faults seeded in the mutants. The results also indicate that Complete IOCO is more efficient in detecting deeper faults in larger state spaces, since these faults are difficult to reach with the random exploration of JTorX.

This paper is structured as follows. Section 2 presents an overview of (Complete) IOCO test case generation. Section 3 presents the specifications used in the case study. Section 4 reports the methodology used for the case study and Section 5 presents and analyzes the results. Finally, Section 6 concludes and points out future directions.

2. FROM MEALY IOTSS TO TEST CASES

In our context, systems are modelled by Input/Output Transition Systems (IOTS), defined in terms of states and transitions labelled by input and output actions. In this section, we give a brief overview of the relevant concepts for IOTS-based testing. This includes a brief overview of IOTSs, as well as IOCO and Complete IOCO test case generation algorithms.

2.1 IOTSS and Mealy IOTSSs

An IOTS $M$ is a quintuple $(S, I, O, h, s_0)$, where $S$ is a set of states, $I$ and $O$ are disjoint sets of input and output actions, respectively, $h$ is a set of transitions labelled by input and output actions. In this paper, we assume that $S \subseteq S \times (I \cup O \cup \{\delta\}) \times S$ is the transition relation, with the symbol $\delta \notin (I \cup O)$ denoting quiescence (lack of output), and $s_0 \in S$ is the initial state. Figure 1a presents an example of IOTS, where $I=\{\text{button}\}, \{\text{iol}\} \Rightarrow \{\text{iof}\} \Rightarrow \{\text{iofbr}\}$, and $I$ is the input set. The symbol $?b$ precedes inputs and the $!$ precedes outputs. The set of inputs and outputs enabled at an state $s$ are, respectively, denoted by $\text{inp}(s)$ and $\text{out}(s)$. A quiescent state $s$, a state without output actions, is denoted as $s^0(s)$. In Figure 1, quiescent states are designated by adding the $\delta$-transitions.

A sequence of actions $u \in (I \cup O \cup \{\delta\})^*$ of IOTS $M$ from state $s_1 \in S$ is a defined trace, if there exists a path $(s_1, a_1, s_2)(s_2, a_2, s_3)...(s_n, a_n, s_{n+1})$ such that $u = (a_1, ..., a_n)$. The set of all traces defined for state $s$ is denoted by $tr(s)$. We use $tr(T)$ to denote the set of traces from states in $T \subseteq S$. We denote the empty trace by $\varepsilon$. $T_{\text{after-U}}$ denotes the set of states reached from states in $T \subseteq S$ when traces in $U$ are executed. State $s \in S$ is quiescent if no output is enabled in $s$. We use $S_{\text{quiescent}}$ to denote the set of all quiescent states in IOTS $M$. In Figure 1b, states $s^0$ and $s^1$ are quiescent. An IOTS is input-complete if all inputs are enabled in quiescent states; in IOCO, an IOTS is called input-enabled if in each and every state inputs are enabled, possibly after some internal transitions.

Mealy IOTSSs [16, 13] behave similarly to a deterministic Mealy machine in that they only receive inputs in quiescent states, i.e., states where no outputs or internal transitions are enabled. This is an important class of IOTS because several results from IOTS and FSM testing theories, such as the use of fault domains, converge on this class of IOTSs. An IOTS is Mealy if $\text{inp}(s) \neq \emptyset \Rightarrow \exists \text{out}(s) = \emptyset$, i.e., an input is enabled only in quiescent states [16]. Figure 1b presents an example of Mealy IOTS, that shows input-completeness in quiescent states $s^0$ and $s^1$. An important concept in Mealy IOTSSs is bridge trace: given an input, a bridge trace is the sequence of outputs until reaching a quiescent state. A bridge trace for the IOTS in Figure 1b is $\{s^0, ?b\} \Rightarrow s^1$.

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 $I \equiv$ ioce $S$ if, for each trace $\alpha \in tr(I)$, we have $\text{out}(I_{\text{after-}} \alpha) \subseteq \text{out}(S_{\text{after-}} \alpha)$.

Trommons [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 IOTSSs 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 IOTSSs, such as Alternating IOTSSs. Simao [16] proposes a generalization of Alternating IOTSSs, called Mealy IOTSSs, which pave the way for defining a general fault model for IOTSSs.

Paiva and Simao [13] proposed a reformulation of the W-
method for FSMs [5] to IOTs 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 IOTs as test models implies that one can avoid the distortion caused by asynchronous channels in testing, since in Mealy IOTs 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 IOTs with at most as many stable states as the specification, covering output and transfer faults.

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

- **non-oscillating**: the Mealy IOT 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 is recorded, which is compared with that of the implementation during test execution.

Complete IOCO for Mealy IOTs 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],
- **Turn Indicator Lights (TIL)** [14], and
- **Standard Exterior Mirror Component (EM) and Standard Alarm System Component (AS) of The Body Comfort System [11]**.

The remainder of this section briefly describes each specification model.

#### 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 IOTs - 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.
### 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 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\_activated\_off) activates the alarm system and disables the monitoring. The alarm system can be deactivated (as\_deactivated\_##\_deactivated\_and\_re-activated\_) and re-activated again (as\_activated\_on). The alarm monitoring of the alarm system is enabled (as\_active\_on) if the car is locked by using the car key (key\_pos\_lock). If the car is unlocked (key\_pos\_unlock) then the active system is disabled (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 either 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\_bot-##\_bottom, EM\_hor\_left, EM\_hor\_right, and EM\_hor\_pending. The pending position (EM\_hor\_pending) is the initial window position. From the initial state, the exterior mirror moves down (em\_mv\_down), up (em\_mv\_up), right (em\_mv\_right), or left (em\_mv\_left), based on the corresponding movement command (em\_mv\_down, em\_mv\_up, em\_mv\_right, and em\_mv\_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 Mealy IOTSs were expressive enough to capture all specification models.

---

Table 1: Guard conditions of CSM specification model

<table>
<thead>
<tr>
<th>#</th>
<th>Conditions for EBI</th>
<th>Conditions for SBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>V_{i0} \leq V_{MRSP}</td>
<td>V_{i0} \leq V_{MRSP}</td>
</tr>
<tr>
<td>c1</td>
<td>V_{i0} &gt; V_{MRSP}</td>
<td>V_{i0} &gt; V_{MRSP}</td>
</tr>
<tr>
<td>c2</td>
<td>V_{o0} \leq V_{MRSP}</td>
<td>V_{o0} \leq V_{MRSP}</td>
</tr>
<tr>
<td>c3</td>
<td>V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
<td>V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
</tr>
<tr>
<td>c4</td>
<td>V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
<td>V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{o0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
</tr>
<tr>
<td>c5</td>
<td>V_{i0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{i0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
<td>V_{i0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP}) \land V_{i0} &gt; V_{MRSP} + dV_{MRSP}(V_{MRSP})</td>
</tr>
<tr>
<td>c6</td>
<td>V_{i0} \leq V_{MRSP} \land V_{i0} = V_{i0} = 0</td>
<td>V_{o0} \leq V_{MRSP} \land V_{o0} = V_{o0} = 0</td>
</tr>
<tr>
<td>c7</td>
<td>V_{o0} \leq V_{MRSP} \land V_{o0} = V_{o0} = 0</td>
<td>V_{i0} \leq V_{MRSP} \land V_{i0} = V_{i0} = 0</td>
</tr>
</tbody>
</table>

---

Figure 3: EBI model

![Figure 3: EBI model](image)

Figure 4: Turn Indicator Lights model

![Figure 4: Turn Indicator Lights model](image)
models of the previous section and hence, both methods are applicable to them.

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 IOTs 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 is 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, JTorX is more efficient 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 10cc0 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 2012/02232-3), 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 etcs ceiling speed monitor. Work Package 4 OETCS/WP4/CSM – Delta-oriented software product line test models - the body comfort system case study. Technical Report 2012-07, TU Braunschweig, 2013.


7. REFERENCES

Appendix D

Paper IV
Basic Behavioral Models for Software Product Lines: Revisited

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi

Science of Computer Programming Journal (SCP), 2018
Basic behavioral models for software product lines: Revisited

Mahsa Varshosaz a,⁎, Harsh Beohar b, Mohammad Reza Mousavi c

a Center for Research on Embedded Systems (CERES), Holmstad University, Sweden
b University of Duisburg-Essen, Germany
c University of Leicester, UK

A R T I C L E   I N F O

Article history:
Received 28 May 2017
Received in revised form 2 September 2018
Accepted 3 September 2018
Available online 7 September 2018

Keywords:
Software product lines
Behavioral model
Featured transition systems
Calculus of communicating systems
Product line labeled transition systems

A B S T R A C T

In Beohar et al. (2016) [9], we established an expressiveness hierarchy and studied the notions of refinement and testing for three fundamental behavioral models for software product lines. These models were featured transition systems, product line labeled transition systems, and modal 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 changes the expressiveness results, as we demonstrate in this paper. Namely, we show that the original notion of product line labeled transition systems and featured transition systems are equally expressive. As an additional result, we show that there are featured transition systems for which the size of the corresponding product line labeled transition system, resulting 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 a featured transition system, such that the size of featured transition system is linear in terms of the size of the corresponding model. To summarize, featured transition systems are equally expressive as, but exponentially more succinct than, product line labeled transition systems.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Software Product Line (SPL) engineering is a software development technique enabling mass production and customization. Using this technique, a family of software systems is efficiently developed based on a common core and by benefitting 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 that can be used for compactly and efficiently representing the behavior of the products in an SPL; 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 [4–7]. 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).

FTSs [1] are introduced as an extension of LTSs 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).

⁎ Corresponding author.
E-mail addresses: mahsa.varshosaz@lth.se (M. Varshosaz), harsh.breohar@uni-due.de (H. Beohar), mrm7890@le.ac.uk (M.R. Mousavi).

https://doi.org/10.1016/j.scico.2018.09.001
0167-6423/© 2018 Elsevier B.V. All rights reserved.
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.

PL-CCSs [2] are an extension of Milner’s Calculus of Communicating Systems (CCSs) [8]. Using PL-CCSs, it is possible to model alternative behavior. The syntax of PL-CCS is an extension of the syntax of CCS with a 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], Product Line Labeled Transition Systems (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 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 favor of the left-hand side or right-hand side variant.

In [9], we studied the comparative expressiveness of three of the above formalisms, namely FTSs, PL-CCSs and MTSSs, where as a part of the results, we concluded that the class of PL-LTSs is less expressive than the class of FTSs (see Theorem 4 in [9]). In this work, it was assumed that in PL-LTSs 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 [2]. Considering this assumption, it was shown that PL-LTSs cannot capture some types of behavior such as three-way choices which can be captured by FTSs.

In this paper, we relax the above-mentioned restriction and adapt the result to the original and more liberal definition of PL-LTSs [2]. We revisit the comparative expressiveness of FTSs and PL-LTSs 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 the FTS are also valid implementations for the PL-LTS resulting from the encoding. The results show that the class of PL-LTSs is at least as expressive as the class of FTSs. We also show that the results provided in [9], specifying that the class of FTSs is at least as expressive as the class of PL-LTSs still holds. Hence, we conclude that the class of PL-LTSs and the class of FTSs are equally expressive. We also provide a comparative succinctness analysis of the size of the PL-LTSs resulting from any sound encoding in terms of the number of states of the corresponding FTS. The results of the succinctness analysis show that FTSs are more succinct 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 FTSs and PL-CCSs. In Section 3, we provide encodings between FTSs and PL-LTSs. In Section 4, we show that the class of PL-LTSs, i.e., underlying semantic model of PL-CCSs, and the class of FTSs are equally expressive. In Section 5, we provide a comparative succinctness analysis for the models resulting from encoding of FTSs. In Section 6, 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 models

In SPL 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” [10]. Each product in an SPL 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 [11] are a common means to compactly represent the set of products of an SPL in terms of its 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 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 [12].

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 [9]).

In this feature model features such as coin (o), beverage (b), and coffee (c) 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 set of features coffee (c), cappuccino (p) and tea (t) have the or relation. Also, features 1e (e) and 1d (d) have the alternative relation, which means the machine can take only one type of coin (euro or dollar). 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 behavior 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 (FTS).** A feature transition system is a 6-tuple \( (P, A, F, \rightarrow, \Lambda, p_{\text{init}}) \), where

1. \( P \) is a set of states,
2. \( A \) is a set of actions,
3. \( F \) is a set of features,
4. \( \rightarrow \subseteq P \times \mathbb{B}(F) \times A \times P \) is the transition relation satisfying the following condition:
   \[
   \forall F, A, F', \phi, \phi' \ ( (P, \phi, a, P') \in \rightarrow \land (P, \phi', a, P) \in \rightarrow ) \implies \phi = \phi',
   \]
5. \( \Lambda \subseteq \{ \lambda : F \rightarrow \mathbb{B} \} \) is a set of product configurations,
6. \( p_{\text{init}} \in P \) is the initial state.\(^1\)

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

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

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

Considering a feature expression \( \phi \in \mathbb{B}(F) \) and a product configuration \( \lambda \in \Lambda \), we say \( \lambda \) satisfies \( \phi \), denoted by \( \lambda \models \phi \), if the result of every substitution of the value of the variables in the feature expression \( \phi \) according to \( \lambda \) is satisfiable.

As mentioned above, each FTS represents the behavior of a set of products. We use LTSs as another formal structure in this paper to describe the behavior of single products. An LTS is defined as follows.

**Definition 2 (LTS).** A labeled transition system is a quadruple \( (S, A, \rightarrow, s_{\text{init}}) \), 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 the 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 \( \rho = s_0 a_1 s_1 a_2 \cdots a_k s_k \), where \( \forall_{(a_i \in A)} \cdot s_i \xrightarrow{a_{i+1}} s_{i+1} \). We denote the set of all initial finite paths in LTS \( T \) by \( \text{Paths}(T) \). By \( \rho(k) \), we denote the \( k \)-th state in path \( \rho \). For \( \rho = s_0 a_1 \cdots a_k s_k \), we define \( \lambda_{\text{last}}(\rho) = s_k \). Furthermore, for a path \( \rho \), \( \text{Trace}(\rho) \) denotes the sequence of actions on the path. For example, \( \text{Trace}(s_0 a_1 a_2 \cdots a_k s_k) = a_1 a_2 \cdots a_k \). We assume that LTSs denotes the class of all LTSs.

\(^1\) In the original definition of FTSs in \([1]\), an FTS can have multiple initial states. Here, for the sake of a more succinct presentation we have considered a single initial state for FTSs; however, it is straightforward to extend the results to multiple initial states.
2.2.1. Deriving valid products

Each FTS represents the behavior of a set of products. The behavior of each product can be represented using an LTS. Hence, each FTS has a set of valid LTS implementations. Intuitively, an LTS can be considered an FTS in which all the feature expressions on the transitions are true. To capture the behavior of a subset of products (or a single one), a refinement relation is defined. The refinement relation formalizes the notion of product derivation as follows [9]:

**Definition 3 (Product-derivation relation for FTSs).** Given an FTS \( fts = (P, A, F, \rightarrow, \Lambda, \text{products}) \), an LTS \( l = (S, A, \rightarrow, \text{state}) \), and a product configuration \( \lambda \in \Lambda \), a binary relation \( R_\lambda \subseteq P \times S \) is called product-derivation relation if and only if the following transfer properties are satisfied.

1. \( \forall P, Q, s, \lambda (P \stackrel{\Delta}{\rightarrow} s \land P \stackrel{\lambda}{\rightarrow} Q \land \lambda \models \phi) \Rightarrow \exists Q_\lambda, s_\lambda (Q_\lambda \rightarrow s_\lambda) \land Q_\lambda \models \phi \land R_\lambda s_\lambda t \).
2. \( \forall P, s, t (P \stackrel{\Delta}{\rightarrow} s \land s \stackrel{\lambda}{\rightarrow} t \Rightarrow \exists Q_\lambda, s_\lambda (Q_\lambda \rightarrow s_\lambda) \land P \stackrel{\lambda}{\rightarrow} Q \land \lambda \models \phi \land R_\lambda s_\lambda t \).

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

We say that \( l \) is a valid implementation of \( fts \), denoted by \( fts \models l \) if and only if there exists a product configuration \( \lambda \in \Lambda \) such that \( \text{products} \models R_\lambda s \).

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

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 [2].

**Definition 4 (PL-CCS).** Assuming the alphabet \( A = \Sigma \cup \hat{\Sigma} \cup \{\tau\} \), where \( \Sigma \) is a set of symbols and \( \hat{\Sigma} = \{\hat{a} | a \in \Sigma\} \) and \( \tau \notin \Sigma \) is a symbol for internal actions. The syntax of PL-CCS terms \( e \) is defined by the following grammar:

\[
\text{Nil} \mid \alpha. e \mid e + e' \mid e \oplus e' \mid e \| e' \mid e(f) \mid e[L],
\]

where \( \text{Nil} \) denotes the terminating process, \( \alpha. \_ \) denotes the action prefixing for action \( \alpha \in A \), \( + \) and \( \| \) 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 actions in \( L \)), and finally \( \oplus \) denotes a family of binary operators.
The difference between the introduced binary variant operator $@$ 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 $Q = b.Q @ c.Q$: 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 is respected. This means that process $Q$ 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, Gruler et al. assume that the $@$ operators in each term are uniquely indexed with natural numbers. This means in every PL-CCS term, there is at most one appearance of the operator $@$ 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 PL-LTSs [2], using a structural operational semantics. We refer to [2] for the formal semantics of PL-CCS. Each state in a PL-LTS comprises a pair of an ordinary state, e.g., a process term, and a configuration vector. The transitions of a PL-LTS are also labeled with configuration vectors. The configuration vectors are used to keep track of the choices made about alternative behavior and are of type $\langle L, R, ? \rangle^I$ with $I$ being an index set. The formal definition of a PL-LTS is as follows:

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

1. $\forall P, v, Q, v', v'' \ (P, v) \xrightarrow{a, v'} (Q, v'') \Rightarrow v'' = v''$.  
2. $\forall P, v, Q, v', v'' \ (P, v) \xrightarrow{a, v'} (Q, v') \land v(i) \neq ? \Rightarrow v''(i) = v(i)$.  
3. $\forall P_0, v_0, Q_0, v'_0, i, P_1, v_1, Q_1, v'_1 \ (P_0, v_0) \xrightarrow{a_0, v'_0} (Q_0, v'_0) \land (P_1, v_1) \xrightarrow{a_1, v'_1} (Q_1, v'_1) \land v_0(i) = v_1(i) = ? \land v'_0(i) \neq ? \land v'_1(i) \Rightarrow (P_0, v_0) = (P_1, v_1)$.

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 cannot 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 $(P_0, v_0) [a_1, v_1] (P_1, v_1) \cdots [a_n, v_n] (P_n, v_n)$ where $\forall i \in \langle 1, n \rangle: (P_i, v_i) \xrightarrow{a_i, v_{i+1}} (P_{i+1}, v_{i+1})$. We denote the set of all such paths for a PL-LTS plt by Paths(plt). 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 $\subseteq$ on the set $\langle L, R, ? \rangle$ is defined as:

$$\subseteq = \{ (?, ?), (L, L), (R, R), (?, L), (?, R) \}.$$
We lift this ordering relation to the level of configuration vectors by defining \( v \sqsubseteq v' \iff \forall_{v'} v(i) \sqsubseteq v'(i) \), 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 \( \triangleright= \) on the set \( \{L, R, ?\} \) is defined as:

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

We lift this relation to the level of configuration vectors by defining \( v \triangleright= v' \iff \exists_{v'} v(i) \triangleright= v'(i) \), 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 product-derivation relation for PL-LTSs is given as follows.

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

1. \( \forall_{P,Q,A,v,v',s} ((P, v) \mathcal{R}_S s \land (P, v) \stackrel{a,v'}{\rightarrow} (Q, v') \land v' \sqsubseteq \theta) \Rightarrow \exists t \mathcal{A} s \stackrel{a}{\rightarrow} t \land (Q, v') \mathcal{R}_S t .
\)

2. \( \forall_{P,a,v,s,t} ((P, v) \mathcal{R}_S s \land s \stackrel{a}{\rightarrow} t) \Rightarrow \exists_{Q,v'} ((P, v) \stackrel{a}{\rightarrow} (Q, v') \land v' \sqsubseteq \theta \land (Q, v') \mathcal{R}_S t .
\)

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

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

2.4. Encoding

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

**Definition 9** (Product line structure). A product line structure is a tuple \( \mathbf{M} = (\mathbf{M}, \mathcal{I}) \), where \( \mathbf{M} \) is the class of the intended product line models (in this paper FTSS and PL-LTSs) and \( \mathcal{I} : \mathbf{M} \rightarrow \text{LTS} \) 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 \( (\text{FTSs}, \mathcal{I}) \), which is a product line structure defined for the class of FTSSs. For an arbitrary FTSS \( \text{fts} \) and arbitrary LTS \( l \), it holds \( l \in [\text{fts}] \Rightarrow \text{fts} \ll l \) (see definition of \( \ll \) in Section 2.2.1). Similarly, consider the tuple \( (\text{PL-LTSs}, \mathcal{I}) \), 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 \( l \in [\text{plt}] \Rightarrow \text{plt} \ll l \) (see definition of \( \ll \) in Section 2.3).

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

We say that a product line structure \( \mathbf{M}' \) is at least as expressive as \( \mathbf{M} \) if and only if there exists an encoding \( E : \mathbf{M} \rightarrow \mathbf{M} ' \). Also, we say that two product line structures \( \mathbf{M} \) and \( \mathbf{M} ' \) are equally expressive if and only if there exists an encoding from \( \mathbf{M} \) to \( \mathbf{M} ' \) and vice versa.

3. Encodings between FTSSs and PL-LTSs

In this section, we provide an encoding from FTSSs to PL-LTSs and thereby show that PL-LTSs are at least as expressive as FTSSs. Furthermore, we provide a slight variation of the encoding from PL-LTSs into FTSSs given in [9], based on the more liberal definition of PL-LTSs. We show that based on the latter encoding, the class of FTSSs is at least as expressive as the class of PL-LTSs; thus, reinstating the results of [9] for the more liberal definition of PL-LTS. The combination of these two encodings shows that PL-LTSs and FTSSs are equally expressive.

**Definition 11** (FTS to PL-LTS encoding). Consider an FTS such as \( \text{fts} = (\mathcal{P}, A, F, \rightarrow, A, p_{\text{init}}) \). The PL-LTS resulting from the encoding, denoted by \( E(\text{fts}) \), is a quintuple \( (\mathcal{P}, A, I, \rightarrow, p_{\text{init}}) \), where:
• \( \mathcal{P} \subseteq \mathcal{P} \times \{1, R, \mathcal{P}\} \),

• \( A \) is the set of actions,

• \( I = \{0, 1, \ldots, |A| - 1\} \) is the index set,

• \( p_{\text{init}} = (p_{\text{init}}, \{\mathcal{P}\}) \) is the initial state,

• \( \rightarrow \subseteq \mathcal{P} \times A \times \{1, R, \mathcal{P}\} \times \mathcal{P} \), is the transition relation which is defined as follows.

Consider an arbitrary bijective function \( \lambda : \Lambda \to \{0 \ldots |\Lambda| - 1\} \), where \( \{0 \ldots |\Lambda| - 1\} \) is the set of all natural numbers not greater than \(|\Lambda| - 1\). For each product configuration \( \lambda \in \Lambda \), we define \( \nu_{\lambda} \) to be the configuration vector such that \( \nu_{\lambda}(\mathcal{P}) = 0 \) \( \forall \mathcal{P} \neq \mathcal{P} \), \( \nu_{\lambda}(\mathcal{P}) = 1 \) \( \forall \mathcal{P} = \mathcal{P} \), \( \nu_{\lambda}(\mathcal{P}) = 0 \) \( \forall \mathcal{P} \neq \mathcal{P} \). Then, the transition relation is the smallest set satisfying the following two conditions:

\[
\begin{align*}
\forall \lambda \in \Lambda : & \quad P \xrightarrow{a, \lambda} Q \land P = p_{\text{init}} \land \lambda \models \phi \Rightarrow (p_{\text{init}}, \{\mathcal{P}\}) \xrightarrow{a, \lambda} (Q, \nu_{\lambda}) \\
\forall \lambda \in \Lambda : & \quad P \xrightarrow{a, \lambda} Q \land P \neq p_{\text{init}} \land \lambda \models \phi \Rightarrow (P, \nu_{\lambda}) \xrightarrow{a, \lambda} (Q, \nu_{\lambda})
\end{align*}
\]

In the above definition, all the choices are resolved in the first step and through the transitions emanating from the initial state. After that, the configuration vectors remain the same.

**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 resulting from removing feature tree from the FTS in Fig. 2 and part (b) represents the PL-LTS resulting form the encoding.

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

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

**Proof.** Consider an arbitrary FTS \( \text{fts} = (P, A, I, \rightarrow, A, p_{\text{init}}) \) and the PL-LTS resulting from the encoding, \( E(\text{fts}) = (\mathcal{P}, A, I, \rightarrow, \mathcal{P}, p_{\text{init}}) \). The first condition of Definition 5, is as follows: \( \forall (P, v), (Q, v') : (P, v) \xrightarrow{a, \lambda} (Q, v') \Rightarrow v' = v' \). 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: \( \forall (P, v), (Q, v') : (P, v) \xrightarrow{a, \lambda} (Q, v') \land v(i) \neq ? \Rightarrow v'(i) = v(i) \).

According to Definition 11, the transitions in \( E(\text{fts}) \) are defined using the following two rules:

1. \( \forall \lambda \in \Lambda : P \xrightarrow{a, \lambda} Q \land P = p_{\text{init}} \land \lambda \models \phi \Rightarrow (p_{\text{init}}, \{\mathcal{P}\}) \xrightarrow{a, \lambda} (Q, \nu_{\lambda}) \)
2. \( \forall \lambda \in \Lambda : P \xrightarrow{a, \lambda} Q \land P \neq p_{\text{init}} \land \lambda \models \phi \Rightarrow (P, \nu_{\lambda}) \xrightarrow{a, \lambda} (Q, \nu_{\lambda}) \)

If the transition is due to rule 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: \( \forall (P, v), (Q, v') : (P, v) \xrightarrow{a, \lambda} (Q, v') \land v(i) = v(i) \Rightarrow v'(i) = v'(i) \neq v'(i) \Rightarrow (P, v) \xrightarrow{a, \lambda} (Q, v') \).

According to Definition 11, only the transitions emanating from the initial state of \( E(\text{fts}) \) and target states with different configuration vectors. Since, \( E(\text{fts}) \) has a single initial state, the third condition in Definition 11 is preserved by \( E(\text{fts}) \).

Next, we give a slight variation of the encoding from PL-LTSs into FTSs given in [9].

**Definition 12 (PL-LTS to FTS Encoding).** Consider a PL-LTS \( \mathcal{P}LTS = (P, A, I, \rightarrow, p_{\text{init}}) \) with the set of product configurations \( \Theta \); the FTS resulting from the encoding, denoted by \( E(\text{plt}) \), is a 6-tuple \( (\mathcal{P}, A, F, \rightarrow', \Lambda, p_{\text{init}}) \), where:

1. \( \mathcal{P} = \bigcup_{\mathcal{P} \in \mathcal{I}} \{I, R, \mathcal{P}\} \),
2. \( \Lambda = \bigcup_{\mathcal{P} \in \mathcal{I}} \{\mathcal{P} \theta(i)\} \)
3. The transition relation \( \rightarrow' \) is defined in the following way:

\[
\begin{align*}
(P, v) \xrightarrow{a, v'} (Q, v) \\
(P, v) \xrightarrow{\lambda} (Q, v) \quad \phi = \bigwedge_{\mathcal{P} \in \mathcal{I}} \theta(i) \quad \Xi(I, v, v')
\end{align*}
\]
where $\Xi(I, \nu, \nu') \iff \forall i \in I \cdot \nu'(i) \neq \nu(i) \land \forall j \in \nu(i) \cdot \nu'(j) = \nu(j)$. (In the above definition we assume the notations $\nu(i)$ and $\theta(i)$, which are used in construction of feature expressions and the product configurations, can be also used for representing variables that stand for features, i.e., $L_i$ or $R_i$ for $i \in I$.)

The difference between the encoding given in Definition 12 and the one given in Theorem 3 in [9], is in the definition of the transition relation. In the definition of transition relation given in Theorem 3 in [9], in the description of function $\Xi$ it is assumed that only one of the elements of $\nu$ changes in each step. In Definition 12, we relax this assumption according to the original and more liberal definition of PL-LTSS given in Definition 5.
4. Comparative expressiveness

In this section, we first prove that the class of PL-LTSs is at least as expressive as the class of FTSs. Then, we show that the result of the proof provided in [9], which shows that the class of FTSs is at least as expressive as the class of PL-LTSs, remains the same considering the more liberal definition of PL-LTSs. Thus, we conclude that the class of PL-LTSs and the class of FTSs are equally expressive.

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

**Proof.** It suffices to show that each LTS $I$ that implements an arbitrary FTS $fts$ is also a valid implementation of $E(fts)$, the PL-LTS resulting from applying the encoding given in Definition 11 to $fts$, and vice versa, i.e., $\forall I \in \text{LTS} : fts \models I \implies E(fts) \models I$. This means the proof of the theorem can be reduced to proving $\lbrack fts \rbrack = \lbrack E(fts) \rbrack$ (see Definition 9).

Consider $fts = (\mathcal{G}, A, I, \rightarrow, \lambda, p_{\text{init}})$ and $E(fts) = (\mathcal{G}, A, I, \rightarrow, \lambda, p_{\text{init}})$; we separate the bi-implication in the proof obligation into the following two implications:

1. $[fts] \subseteq \lbrack E(fts) \rbrack^*$

   In order to prove $[fts] \subseteq \lbrack E(fts) \rbrack^*$, we show that $\forall I \in \text{LTS} : I \models fts \implies I \models E(fts)$. Consider an arbitrary LTS $I = (\mathcal{G}, A, \rightarrow, s_{\text{init}})$ s.t. $I \in \text{LTS}$, which means that $fts \models I$ (see Section 2.4). We prove $E(fts) \models I$ and hence, $I \in \lbrack E(fts) \rbrack^*$.

   Let $\theta$ denote the set of product configuration vectors derived from the set $\Lambda$, i.e., $\theta = \{ v_\lambda | \lambda \in \Lambda \}$, where $v_\lambda$ has the same definition as given in Definition 11. In order to prove $E(fts) \models I$, it suffices to show that for some product configuration vector $\theta \in \theta$, it holds that $p_{\text{init}} \rightarrow^* s_{\text{init}}$ (see Section 2.3).

   Next, we show that the above statement is satisfied by $I$ and $E(fts)$. Considering Definition 8, for any two arbitrary states, $(P,\nu) \in \text{P}$ and $s \in \text{S}$, $(P,\nu) \rightarrow^* s$ holds if a product-derivation relation such as $R_\theta$ satisfies such that $(P,\nu) \rightarrow R_\theta s$ and $R_\theta$ satisfies the following properties:

   (1) $\forall (P,\nu) \in \text{P}$ and $s \in \text{S}$.

   (2) $\forall (P,\nu) \in \text{P}$ and $s \in \text{S}$.

   We define a binary relation $R_\theta \subseteq \text{P} \times \text{S}$ [where $\theta$ is a configuration vector in $\theta$] such that:

   $$\forall P \in \text{P}, s \in \text{S} : \exists \lambda \in \Lambda : ((P = \text{p_{init}} \land P \rightarrow s) \models (p_{\text{init}}, \nu^I) \models \theta) \land ((P \neq \text{p_{init}} \land P \rightarrow s) \models (P,\nu) \rightarrow R_\theta s \land \theta = \nu)$$

   Consider the configuration vector $\lambda$ that derives LTS $I$ from FTS $fts$ (based on Definition 8); let $\lambda = \nu_1$ and $R_\theta$ be a member of the above defined relation. Next, we prove that $R_\theta$ satisfies the properties of a product-derivation relation (products (1) and (2)).

   Consider an arbitrary pair of states in $R_\theta$, such as $(P,\nu) \rightarrow^* s$; based on the definition given above for $R_\theta$, it holds $P \rightarrow R_\theta s$, where we distinguish the following two cases: $(P,\nu) = \text{p_{init}}$ and $(P,\nu) \neq \text{p_{init}}$.

   First, we prove that statement (1) is satisfied by $R_\theta$.

   Thus $\nu = \nu_1$ and $P = \text{p_{init}}$ (see Definition 11). Consider an arbitrary transition of the form $(\text{p_{init}}, \nu^I) \rightarrow^* (Q,\nu_2)$; based on Definition 11, such a transition is resulting from encoding one of the outgoing transitions from $P$ in $fts$, i.e.:

   $$\begin{align*}
   \nu_0 & : \rightarrow^* (Q,\nu) \\
   \nu_0 & \rightarrow^* (Q,\nu_2) \models \phi \land P \rightarrow \nu_0
   \end{align*}$$

   Considering property 1 satisfied by relation $R_\theta$, $P \rightarrow^* s$ implies the following statement:

   $$\begin{align*}
   \nu_0 & : \rightarrow^* (Q,\nu) \land \lambda \models \phi \\
   \nu_0 & \rightarrow^* (Q,\nu_2) \land \theta = \nu_2
   \end{align*}$$

   Based on the definition of $R_\theta$, $Q \rightarrow R_\theta t \models (Q,\nu_2) \models \theta = \nu_2$. Hence, $(Q,\nu_2) \rightarrow R_\theta t$ holds only in case of $\lambda = \lambda'$.

   Given that $\lambda = \nu_2$, based on the definition of the relation $\llbracket$ (see Definition 8) it holds $\nu_2 \equiv \theta$ (notice that for any $\lambda' \in \Lambda$ such that $\lambda \neq \lambda'$, based on the definition of $\nu_2$ it holds $\nu_2 \models \nu_2'$ and hence, $\nu_2 \equiv \theta$). Thus, from (1.1) and (1.11), the following statement is derived:

   $$\begin{align*}
   \nu_0 & : \rightarrow^* (Q,\nu) \land \nu_2 \models \theta \\
   \nu_0 & \rightarrow^* (Q,\nu_2) \land \theta = \nu_2
   \end{align*}$$
* $(P, v) \neq \bar{p}_{init}$.

Thus, (based on the definition of $\mathcal{R}_v$) $v = v_s$. Consider an arbitrary transition emanating from $(P, v_s)$ of the form

$\vdash (P, v_s) \xrightarrow{(Q, v_s)}$;

based on Definition 11, the configuration vector in the target state of the outgoing transitions from $(P, v_s)$ is $v_s$, and such transition is resulting from encoding one of the outgoing transitions from $P$ in $\mathcal{S}$, i.e.:

$\forall Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P \neq \bar{p}_{init}$

(1.iv)

Considering property 1 satisfied by relation $\mathcal{R}_v$, $P \mathcal{R}_v s$ implies the following statement:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(1.v)

Based on the definition of $\mathcal{R}_v$, $Q \mathcal{R}_v t \equiv (Q, v_s) \mathcal{R}_v t \land \theta = v_s$. Using the same reasoning as in the previous case, from (1.iv) and (1.v), the following statement is derived:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} (Q, v_s) \land \theta \equiv \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(vi)

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

$\forall Q, P, v, v', \bar{P}, Q, \vdash (P, v) \mathcal{R}_v s \land (P, v) \xrightarrow{\bar{P}} (Q, v') \land v' \subseteq \theta \Rightarrow \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

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

Next, we prove that statement (2) is satisfied by $\mathcal{R}_v$. Again we distinguish the two cases: $(P, v) = \bar{p}_{init}$ and $(P, v) \neq \bar{p}_{init}$.

* $(P, v) = \bar{p}_{init}$.

Thus, $v = [\emptyset]_I$ and $P = \bar{p}_{init}$ (see Definition 11).

Consider an arbitrary transition emanating from $(\bar{p}_{init}, [\emptyset]_I)$ of the form $(\bar{p}_{init}, [\emptyset]_I) \xrightarrow{\bar{P}} (Q, v_s)$; based on Definition 11, such a transition is resulting from encoding an outgoing transition from $P$, i.e.:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (\bar{p}_{init}, [\emptyset]_I) \xrightarrow{\bar{P}} (Q, v_s) \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(2.i)

Considering property 2 satisfied by relation $\mathcal{R}_v$, $P \mathcal{R}_v s$ implies the following statement:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(2.ii)

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

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(2.iii)

Thus, $(P, v) \neq \bar{p}_{init}$.

Consider an arbitrary transition emanating from $(P, v_s)$ of the form $(P, v_s) \xrightarrow{\bar{P}} (Q, v_s)$, based on the Definition 11, the configuration vector in the target state of the outgoing transitions from $(P, v_s)$ is $v_s$; such a transition is resulting from encoding an outgoing transition from $P$, i.e.:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(2.iv)

Furthermore, consider property 2 satisfied by relation $\mathcal{R}_v$; $P \mathcal{R}_v s$ implies the following statement:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} \exists \bar{P} : P \xrightarrow{\phi} Q \land \lambda \models \phi \land P = \bar{p}_{init}$

(2.v)

Given the definition of $\mathcal{R}_v$, $Q \mathcal{R}_v t \equiv (Q, v_s) \mathcal{R}_v t \land \theta = v_s$. Since, $\theta = v_s$ and based on Definition 6 it holds $v_s \subseteq \theta$. Considering (2.iv) and (2.v), the following statement holds:

$\forall Q, \bar{P}, Q, \bar{P}, Q, \vdash (P, v_s) \xrightarrow{(Q, v_s)} (Q, v_s) \land \theta \equiv (Q, v_s)$

(2.vi)

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

$\forall P, Q, v, v', \bar{P}, Q, \vdash (P, v) \mathcal{R}_v s \land (P, v) \xrightarrow{\bar{P}} (Q, v') \land v' \subseteq \theta \Rightarrow (Q, v') \mathcal{R}_v t$

which means that $\mathcal{R}_v$ satisfies the second property of a product derivation relation.

Based on the assumption $\mathcal{F}_{\mathcal{S}} I$, it holds $\mathcal{F}_{\mathcal{S}} I \rightarrow \mathcal{S}_{\mathcal{S}_I}$. As shown above, $\mathcal{R}_v$ satisfies the properties of a product derivation relation given in Definition 8. Hence, based on the definition of $\mathcal{R}_v$ it holds that $\mathcal{F}_{\mathcal{S}} I \rightarrow \mathcal{F}_{\mathcal{S}} I \rightarrow \mathcal{S}_{\mathcal{S}_I}$. Thus, $\mathcal{F}_{\mathcal{S}} I \rightarrow \mathcal{S}_{\mathcal{S}_I}$. This means $\mathcal{E} \mathcal{F}_{\mathcal{S}} I \mathcal{F}_{\mathcal{S}} I$ and subsequently $I \in \mathcal{E} \mathcal{F}_{\mathcal{S}} I \mathcal{F}_{\mathcal{S}} I$.
• \([E(fts)] \subseteq \{fts\}\).

**Proof.** In order to prove \([E(fts)] \subseteq \{fts\}\), we show that for all LTS \(l \in \{E(fts)\}\), there exists \(l \in \{fts\}\). Consider an arbitrary LTS \(l = (S, A, \rightarrow, s_{init})\), and let \(l \in \{E(fts)\}\) and subsequently \(E(fts) < l\) (see Definition 10). We prove \(fts \rightarrow l\) and hence, \(l \in \{fts\}\). To prove \(fts \rightarrow l\), it suffices to show that for some product configuration \(\lambda \in \Lambda\) the following statement holds: \(p_{\text{init}} \vdash s_{\text{init}}\) (see Definition 3).

Next, we show that the above statement is satisfied by \(l\) and \(fts\). According to Definition 3, \(P_{\rightarrow}\) holds if a product-derivation relation such as \(R_{\lambda}\) exists such that \(P \in R_{\lambda}\) and \(R_{\lambda}\) satisfies the following properties:

\[
\forall P, Q, \lambda, \theta, \theta' \cdot \left(P \in R_{\lambda} \land \lambda \vdash \phi \Rightarrow \exists \lambda' \cdot s \overset{\lambda}{\rightarrow} t \land \Theta_{\lambda} t\right)
\]

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 \(R_{\Theta} \subseteq \mathcal{F} \times S\) exists, which satisfies the following properties:

\[
\forall P, Q, \lambda, \theta, \theta' \cdot \left((P, v) \in R_{\lambda} \land (P, v) \overset{\lambda}{\rightarrow} (Q, v') \land \Theta \vdash \phi \Rightarrow \exists \lambda' \cdot (P, v) \overset{\lambda'}{\rightarrow} (Q, v') \land \Theta \vdash \phi \right)
\]

Next, we define a binary relations \(\mathcal{R}_{\lambda}\) (where \(\lambda\) is a product configuration in \(\Lambda\)) such that:

\[
\forall P, Q, \lambda, \theta, \theta' \cdot \left((P, v) \in R_{\lambda} \land (P, v) \overset{\lambda}{\rightarrow} (Q, v') \land \Theta \vdash \phi \Rightarrow \exists \lambda' \cdot (P, v) \overset{\lambda'}{\rightarrow} (Q, v') \land \Theta \vdash \phi \right)
\]

Assume that LTS \(l\) is derived from PL-LTS \(E(fts)\) with regards to the product configuration vector \(\theta = v_l\). Let \(\mathcal{R}_{\lambda}\) be a relation defined as above. Next, we prove that \(\mathcal{R}_{\lambda}\) satisfies the properties of a product-derivation relation (statements (1) and (2)).

First, we consider the case where \(P = \text{pinit}\):

Based on Definition 11, each transition emanating from \(P\) such as \(\overset{\lambda}{\rightarrow} S\), is encoded as a transition in PL-LTS \(E(fts)\), i.e.:

\[
\forall a, Q, \lambda' \cdot \left(P \overset{\lambda}{\rightarrow} Q \land \lambda' \vdash \phi \Rightarrow (P, a) \overset{\lambda, a}{\rightarrow} (Q, a)ight)
\]

Considering property 1 satisfied by relation \(\mathcal{R}_{\lambda}\), \((\text{pinit}, [\lambda])\) \(\mathcal{R}_{\lambda}\) implies the following statement:

\[
\forall a, Q, \lambda' \cdot \left((\text{pinit}, [\lambda]) \overset{\lambda, a}{\rightarrow} (Q, a) \land \lambda' \vdash \phi \Rightarrow \exists \lambda' \cdot s \overset{\lambda}{\rightarrow} t \land \Theta_{\lambda} t\right)
\]

Based on the definition of \(\mathcal{R}_{\lambda}\), \((Q, v_l) \in \mathcal{R}_{\lambda} t \Rightarrow Q \in \mathcal{R}_{\lambda} t \land v_l = \theta\). Hence, \(Q \in \mathcal{R}_{\lambda} t\) holds only in case \(\lambda' = \lambda\). Thus, from (1.i) and (1.ii), the following statement is derived:

\[
\forall a, Q \cdot \left(P \overset{\lambda}{\rightarrow} Q \land \lambda \vdash \phi \Rightarrow \exists \lambda' \cdot s \overset{\lambda}{\rightarrow} t \land \Theta_{\lambda} t\right)
\]

Next, we assume \(P \neq \text{pinit}\):

Based on Definition 11, each transition emanating from \(P\) such as \(\overset{\lambda}{\rightarrow} S\), is encoded as a transition in PL-LTS \(E(fts)\), i.e.:

\[
\forall a, Q, \lambda' \cdot \left(P \overset{\lambda}{\rightarrow} Q \land \lambda \neq \phi \Rightarrow (P, a, v_l) \overset{\lambda, a, v_l}{\rightarrow} (Q, v_l)\right)
\]

Considering property 1 satisfied by relation \(\mathcal{R}_{\lambda}\), \((P, v_l) \overset{\lambda}{\rightarrow} \Theta_{\lambda} t \Rightarrow \forall a, Q \cdot \left(P \overset{\lambda}{\rightarrow} Q \land \lambda \vdash \phi \Rightarrow \exists \lambda' \cdot s \overset{\lambda}{\rightarrow} t \land \Theta_{\lambda} t\right)
\]

Based on the definition of \(\mathcal{R}_{\lambda}\), \((Q, v_l) \in \mathcal{R}_{\lambda} t \Rightarrow Q \in \mathcal{R}_{\lambda} t \land v_l = \theta\). Thus, from (1.iii) and (1.i), the following statement is derived:

\[
\forall a, Q \cdot \left(P \overset{\lambda}{\rightarrow} Q \land \lambda \vdash \phi \Rightarrow \exists \lambda' \cdot s \overset{\lambda}{\rightarrow} t \land \Theta_{\lambda} t\right)
\]
Considering (1.iii) and (1.vi), the following statement holds:
\[ \forall P, Q, s, \phi : (P \not\sim_Q s \land P \stackrel{\phi}{\rightarrow} Q \land \lambda \models \phi) \Rightarrow \exists t : s \xrightarrow{\lambda} t \land Q \not\sim t, \]
which means that the relation \( \not\sim \) satisfies statement (1).

Next, we prove that \( \not\sim \) satisfies statement (2).

Considering an arbitrary pair of states in \( \not\sim \), such as \( P \not\sim_Q s \). Based on the definition given for \( \not\sim \), it holds \( (P, v) \not\sim_Q s \), where \( \theta = v \), and \( v = \{?\} \) if \( P = \text{init} \) and \( (P, v) \not\sim_Q s \) where \( \theta = v \) and \( v = v \) if \( P \neq \text{init} \). For the sake of clarity we distinguish the following two cases: \( P = \text{init} \) and \( P \neq \text{init} \).

First, we consider the case that \( P = \text{init} \):

Based on Definition 11, each transition emanating from \( P \), such as \( P \not\sim_Q Q \) is encoded as an outgoing transition from \( \text{init} \), i.e.,
\[ \forall a, Q, \lambda : P \not\sim_Q Q \land P = \text{init} \land \lambda \models \phi \Rightarrow (\text{init}, \{?\}) \xrightarrow{\lambda} (Q, \lambda) \]
(2.i)

Considering property 2 of relation \( \not\sim \), \( (\text{init}, \{?\}) \not\sim t \) s implies the following statement:
\[ \forall a, s, t : (s \not\sim t) \Rightarrow \exists Q, \lambda : (\text{init}, \{?\}) \xrightarrow{\lambda} (Q, \lambda) \land \lambda \models \phi \]
(2.ii)

Based on the definition of \( \not\sim \), \( Q \not\sim t \) and \( \lambda \) only in case \( \lambda = \lambda \). Thus, from (2.i) and (2.ii), the following statement is derived:
\[ \forall a, s, t : (s \not\sim t) \Rightarrow \exists Q, \lambda : (P \not\sim_Q Q \land \lambda \models \phi \land Q \not\sim t) \]
(2.iii)

Next, we assume \( P \neq \text{init} \):

Based on Definition 11, each transition emanating from \( P \), such as \( P \not\sim_Q Q \), is encoded as an outgoing transition from \( P \), i.e.,
\[ \forall a, Q, \lambda : P \not\sim_Q Q \land P \neq \text{init} \land \lambda \models \phi \Rightarrow (P, v) \xrightarrow{\lambda} (Q, v) \]
(2.iv)

Considering property 2 satisfied by relation \( \not\sim \), \( (P, v) \not\sim t \) s implies the following statement:
\[ \forall a, s, t : (s \not\sim t) \Rightarrow \exists Q, \lambda : (P, v) \xrightarrow{\lambda} (Q, v) \land \lambda \models \phi \]
(2.v)

Based on the definition of \( \not\sim \), \( Q \not\sim t \) and \( \lambda \) only in case \( \lambda = \lambda \). Thus, from (2.iv) and (2.v), the following statement is derived:
\[ \forall a, s, t : (s \not\sim t) \Rightarrow \exists Q, \lambda : (P \not\sim_Q Q \land \lambda \models \phi \land Q \not\sim t) \]
(2.vi)

Considering two derived statements (2.iii) and (2.vi), the following statement holds:
\[ \forall P, Q, s, t : (s \not\sim t) \Rightarrow \exists Q, \lambda : (P \not\sim_Q Q \land \lambda \models \phi \land Q \not\sim t) \]

Hence, we conclude that \( \not\sim \) satisfies statement (2).

The assumption \( E(\text{fts}) \triangleq \lambda \) implies that \( \text{init} \not\sim \lambda \). Based on the above proof, \( \not\sim \) satisfies the properties of a product derivation relation given in Definition 3. Hence, based on the definition of \( \not\sim \), it holds that \( (\text{init}, \{?\}) \not\sim \lambda \Rightarrow \text{init} \not\sim \lambda \). This means \( \text{fts} \) is reversible and subsequently \( \lambda \in \text{fts} \).

Hence, we conclude that \( [E(\text{fts})] \subseteq \text{fts} \). □

As is shown above \([\text{fts}] \subseteq [E(\text{fts})] \) and \([E(\text{fts})] \subseteq [\text{fts}] \). Thus, \([\text{fts}] = [E(\text{fts})] \), which means the class of PL-LTSs is at least as expressive as the family of FTIs. □

In the next theorem, we show that the class of FTIs is at least as expressive as class of PL-LTSs, which is the same result as provided in [9], but based on a modified version of encoding given in the proof of Theorem 3, in [9] (see Definition 12). The subsequent proof is almost identical to the proof of Theorem 3 in [9], as well.

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

**Proof.** Consider the encoding from PL-LTSs into FTIs given in Definition 12. The proof for the above theorem remains the same as the proof of Theorem 3, in [9] (by considering the modified encoding), which is as follows. Assume \( \text{plt} = ([P \times \mathbb{L} \times \{?\}], F, \lambda, \rightarrow, \text{init}) \) is a PL-LTS and the FTs \( E(\text{plt}) = ([P \times \mathbb{L} \times \{?\}], F, \lambda, \rightarrow, \text{init}) \) is the result of applying encoding \( E \) on \( \text{plt} \), based on Definition 12. For any \( (P, v) \in [P \times \mathbb{L} \times \{?\}], \) we fix \( E(P, v) = (P, v) \). We need to show that \([\text{plt}] = [E(\text{plt})] \) . We divide the proof obligation into the following two cases:
Proof. Let $l \in \mathcal{PH}$, where $l = (S, A, \rightarrow, s_0)$. Then $p_{\text{init}} \vdash_s s_{\text{init}}$ for some $\theta \in \{L, R\}$\footnote{Note that $\mathcal{PH}$ is not a formal language.}. Define a configuration $\lambda_0 \in A$ as follows: $\lambda_0(A) = \top \iff \theta(i) = L$ and $\lambda_0(A) = \top \iff \theta(i) = R$. Furthermore, consider the following relation $\mathcal{R}_{\lambda_0}$ such that $v(Q, v') \mathcal{R}_{\lambda_0} s \iff (Q, v') \gamma \rho$. It is straightforward to show that $\mathcal{R}_{\lambda_0}$ is a product derivation relation with regards to Definition 3.

Based on Theorem 2 and Theorem 3, we give the following corollary.

**Corollary 1.** The class of PL-LTSs and the class of FTSs are equally expressive.

**Proof.** Considering Theorem 2 the class of PL-LTSs is at least as expressive as the class of FTSs. Based on Theorem 3, the class of FTSs is at least as expressive as the class of PL-LTSs. Hence, considering Definition 10, we conclude that the class of PL-LTSs and the class of FTSs are equally expressive.

5. Succinctness analysis

In this section, we provide an analysis of the succinctness (the number of states and the configuration vector size included in the states) of PL-LTSs resulting from encoding FTSs. We prove that for some FTSs the size of the PL-LTS which is resulting from any sound encoding, is exponential in terms of the number of states of the FTS. Furthermore, we show that for each PL-LTS a sound encoding into FTSs exists such that the size of the resulting FTS is linear in terms of the size of PL-LTS. Hence, as a result we conclude that FTSs are in general exponentially more succinct than PL-LTSs. In the rest of this section, we assume that $E$ denotes all sound encodings from the class of FTSs into the class of PL-LTSs. We consider the FTS $fts$ depicted in Fig. 5. In this FTS, in each state $s_i$ there is a variant choice between features $f_x$ and $f_y$, i.e., in each valid product either the transition labeled $f_x$ or the one with $f_y$ (but not both) must be present. We assume $E(fts) = (\vec{P}, A, I, \rightarrow, p_{\text{init}})$ is the PL-LTS resulting from encoding $fts$ using an arbitrary encoding $E \in E$. The FTS $fts$ has 2nd non-trace equivalent LTS implementations each of which has exactly one path. We assume $\text{Imp}$ 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., $\Omega(|I|)$. 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. We formalize this in the following lemma.

**Lemma 1.** Assuming two non-trace-equivalent LTSs, $l_1, l_2 \in \text{Imp}$, such as $l_1 = (S_1, A, \rightarrow, s_1)$ and $l_2 = (S_2, A, \rightarrow, s_2)$, where $p_{\text{init}} \vdash_{s_1} s_{\text{init}}$ and $p_{\text{init}} \vdash_{s_2} s'_{\text{init}}$. It holds that:

\[
\exists \pi, Q, \rho, \alpha, \beta : (Q, v') \triangleright (Q, v') \land (v' \subseteq \theta_1 \land v'' \subseteq \theta_2) \land (Q, v) \triangleright (Q, v') \land (v' \subseteq \theta_1 \land v'' \subseteq \theta_2)
\]

**Proof.** Assuming that $\text{Paths}(l_1) = \rho_1$, $\text{Paths}(l_2) = \rho_2$; $\Sigma = \text{pref}(\text{Trace}(\rho_1)) \cap \text{pref}(\text{Trace}(\rho_2))$, where $\text{pref}(\cdot)$ denotes the set of the finite prefixes of a sequence. We consider $\sigma \in \Sigma$ such that $|\sigma| < |\sigma'|$ i.e., a maximal trace $\sigma'$ in $\Sigma$. Assume $|\sigma| = k$; let $\rho_1(k) = \alpha$ and $\rho_2(k) = \beta$, given that $l_1, l_2 \in \text{Imp}$ are distinct it holds that: $s_1 \xrightarrow{\alpha} s_{k+1}$ and $s_2 \xrightarrow{\beta} s_{k+2}$ where $\alpha \neq \beta$. Based on the condition (2) in Definition 8, it holds:

\[
s_1 \xrightarrow{\alpha} s_{k+1} \rightarrow \exists \pi, Q, \rho, \alpha, \beta : (Q, v') \triangleright (Q, v') \land (v' \subseteq \theta_1)
\]

\[
s_2 \xrightarrow{\beta} s_{k+2} \rightarrow \exists \pi, Q, \rho, \alpha, \beta : (Q, v') \triangleright (Q, v') \land (v' \subseteq \theta_2).
\]
Given that $I_1, I_2 \in \text{Imp}$ it holds $|\text{Out}(s^1_1)| = 1$ and $|\text{Out}(s^2_1)| = 1$; hence $v'_1 \notin \theta_2$ and $v'_2 \notin \theta_1$ (otherwise, $s^1_2$ and $s^2_2$ should have more than one outgoing transitions). Thus, it can be concluded that:

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \subseteq \theta_1 \land v' \notin \theta_2) \land$$

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \notin \theta_1 \land v' \subseteq \theta_2) \square$$

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

**Lemma 2.** Let $E \in \mathbb{E}$ be an arbitrary encoding. The size of the configuration vector included in the states of $E(\text{fts})$ (i.e., $\Omega(1^n)$) is at least $n$.

**Proof.** Consider two non-trace-equivalent LTSs $I_1, I_2 \in \text{Imp}$, such as $I_1 = (S_1, A, \rightarrow_1, s^1_0)$ and $I_2 = (S_2, A, \rightarrow_2, s^2_0)$, where $P_{\text{init}} \vdash \gamma_n s^1_0$ and $P_{\text{init}} \vdash \gamma_2 s^2_0$. According to Lemma 1, it holds that:

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \subseteq \theta_1 \land v' \notin \theta_2) \land$$

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \notin \theta_1 \land v' \subseteq \theta_2),$$

which means for any two arbitrary LTSs $I_1, I_2 \in \text{Imp}$ it holds that: $\exists_{P \in \text{Paths}(E(\text{fts}))} \cdot \text{last}(P) = (Q, v') \land (v' \subseteq \theta_1 \land v' \notin \theta_2)$ and $\exists_{P \in \text{Paths}(E(\text{fts}))} \cdot \text{last}(P) = (Q, v') \land (v' \notin \theta_1 \land v' \subseteq \theta_2)$. Thus, $\exists_{\gamma_n \gamma_1} \cdot v_1 \Rightarrow v_2$ (see Definition 7). This means $\forall_{1, 2, \text{Imp}} \exists_{\gamma_{1, 1}, \gamma_{2, 2}, \gamma_{1, 2}} \cdot v_1 \Rightarrow v_2$, hence for each two products selected from Imp, there are two states in the PL-LTS that the configuration vectors in these states are conflicting. As $|\text{Imp}| = 2^n$, the minimum size of the configuration vector included in the state of the PL-LTS is $\log(2^n) = n$. $\square$

Next, we prove the following theorem regarding the succinctness of the PL-LTSs resulting from encoding FTSs.

**Lemma 3.** Consider the class of all possible encodings from FTSs into PL-LTSs, denoted by $\mathbb{E}$. 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 $E \in \mathbb{E}$ be an arbitrary encoding and $E(\text{fts}) = (\tilde{P}, A, I, \rightarrow, \tilde{P}_{\text{init}})$ be the PL-LTS resulting from the encoding. Consider two distinct LTSs, $I_1, I_2 \in \text{Imp}$, such as $I_1 = (S_1, A, \rightarrow_1, s^1_0)$ and $I_2 = (S_2, A, \rightarrow_2, s^2_0)$, where $P_{\text{init}} \vdash \gamma_n s^1_0$ and $P_{\text{init}} \vdash \gamma_2 s^2_0$; according to Lemma 1, it holds that:

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \subseteq \theta_1 \land v' \notin \theta_2) \land$$

$$\exists P, Q, a, v, v' \cdot (P, v) \xrightarrow{a} (Q, v') \land (v' \notin \theta_1 \land v' \subseteq \theta_2),$$

Hence, for each two arbitrary LTSs $I_1, I_2$ it holds that $\exists_{\gamma_{1,1}, \gamma_{2,2}, \gamma_{1,2}} \cdot v_1 \Rightarrow v_2$. As $|\text{Imp}| = 2^n$ it holds that the size of the set of states in $E(\text{fts})$ is at least $2^n$.

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

Next, we prove that each PL-LTS can be encoded into an FTS using a sound encoding such that the size of the FTS is linear in the size of the PL-LTS.

**Theorem 4.** An encoding $E \in \mathbb{E}$ from PL-LTSs into FTSs exists such that for any PL-LTS $P$, the size of the model resulting from the encoding of $P$ is linear in terms of the size of $P$, i.e., $|E(P)| = O(|P|)$ where $|.|$ represents the number of states.

**Proof.** Let $(\tilde{P} \times (L, R, ?))_I, A, \rightarrow, \tilde{P}_{\text{init}}$ be an arbitrary PL-LTS. We consider the encoding given in Definition 12. The corresponding FTS is denoted by $(\tilde{P} \times (L, R, ?))_I, A, F, \rightarrow, \tilde{P}_{\text{init}}$. The result of encoding a state in the PL-LTS such as $(P, v) \in \tilde{P} \times (L, R, ?)_I$ is state $(P, v)$ in the FTS. Considering the transition relation given in Definition 12, 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 straightforward to see that the size of the FTS resulting from the encoding is linear in terms of the size of the original PL-LTS. $\square$
6. Conclusion

In this paper, we compared the expressiveness of the PL-CCSs and FTSs. To this end, we used a more liberal definition for PL-FTSs (which are considered as the semantic domain for PL-CCS terms) in comparison with our previous work [9]. We described an encoding from the class of FTSs into the class of PL-FTSs. Then, we proved that the set of LTSs that implement an FTS, are also valid implementations of the PL-FTS resulting from encoding the FTS and vice versa. Furthermore, we showed that the class of FTSs is at least as expressive as the class of PL-FTSs, which is the same result as provided in [9]. Thus, we conclude that the class of PL-FTSs and the class of FTSs are equally expressive. We also provided a succinctness analysis of the models resulting from the encoding. The results show that for some FTSs the size of the PL-FTS resulting from encoding the FTS (using any sound encoding) is exponential in terms of the number of the states of the FTS. Furthermore, the results show that there exists an encoding from PL-FTSs to FTSs for which the size of the FTS resulting from the encoding is linear in terms of the size of the original PL-FTS. Hence, as a result we conclude that FTSs are in general exponentially more succinct than PL-FTSs.

Both in the present paper and in [9], 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 and succinctness given in [9] and in the present paper, by comparing the expressiveness and succinctness of other formalisms, such as MTSs (for succinctness) and their extensions such as 1MTSs [4], DMTSs [5], PMTs [6] and MTSs with variability constraints [7], and also variations of process algebras such as Variant Process Algebra [13] and DeltaCCS [14] with formalisms included in the lattice, is another avenue for our future work.

Acknowledgements

This work has been supported by grants from the Swedish Knowledge Foundation/Stiftelsen för Kunskaps- och Kompetensutveckling in the context of the AUTO-CAAS HoG project (number: 20140312), Swedish Research Council (Vetenskapsrådet) award number: 621-2014-5057 (Effective Model-Based Testing of Concurrent Systems), and the ELLIT Strategic Research Environment.

References

Appendix E

Paper V
A Classification of Product Sampling for Software Product Lines

Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer

International Conference on Software Product Line (SPLC)
Gothenburg, Sweden, September 10–14, 2018
A Classification of Product Sampling for Software Product Lines

Mahsa Varshosaz, 1 Mustafa Al-Hajjaji, 2 Thomas Thüm, 3 Tobias Runge, 3 Mohammad Reza Mousavi, 4,1 and Ina Schaefer 3

1 Halmstad University, Sweden 2 Pure-Systems GmbH, Germany 3 TU Braunschweig, Germany 4 University of Leicester, UK

ABSTRACT
The analysis of software product lines is challenging due to the potentially large number of products, which grow exponentially in terms of the number of features. Product sampling is a technique used to avoid exhaustive testing, which is often infeasible. In this paper, we propose a classification for product sampling techniques and classify the existing literature accordingly. We distinguish the important characteristics of such approaches based on the information used for sampling, the kind of algorithm, and the achieved coverage criteria. Furthermore, we give an overview on existing tools and evaluations of product sampling techniques. We share our insights on the state-of-the-art of product sampling and discuss potential future work.

CCS CONCEPTS
• Software and its engineering → Software product lines

KEYWORDS
Sampling Algorithms, Software Product Lines, Testing, Feature Interaction, Domain Models

ACM Reference Format:

1 INTRODUCTION
Software product lines (SPLs) have become common practice for mass production and customization of software systems. In an SPL, products are developed based on a common core. The main goal of using SPLs is to enable systematic reuse in different phases of development by considering the commonalities and variabilities among the products in an SPL [56].

Testing and analysis of software product lines is known to be challenging due to the sheer number of possible products, which makes exhaustive testing and analysis practically impossible. To alleviate this problem, one may resort to product sampling techniques [82] that provide a subset of all valid products. These products are supposed to collectively cover the behavior of the product line and hence for example testing them should reveal most faults in all other products.

Several approaches have been proposed for product sampling in the context of software product lines, in order to search the vast space of valid products [28, 56, 66]. For such approaches, a myriad of search algorithms for finding a sample to cover a product line have been proposed, where the notion of coverage may also vary from one approach to another. Different algorithms use different types of information sources to find a covering sample. Moreover, the proposed algorithms have typically been evaluated with respect to different criteria and with different degrees of tool support and reproducibility.

We aim for bringing more structure onto the extensive literature on product sampling. In contrast to existing surveys on product sampling [49, 71] or product-line testing [56], we do not follow a systematic process in which all interesting research questions is defined up-front. In contrast, our goal is to provide more insights for readers by means of a detailed classification of existing sampling techniques. We envision that our insights can be used to have a better understanding of such techniques for education and research and also for recognizing their requirements and shortcomings to apply such techniques in practice. To this end, we considered a literature catalog with 48 publications [1–48]. We limited our search to find studies that are focusing on new sampling algorithms [1–38] or evaluations of existing ones [39–48].

Our contributions are threefold. First, we propose a classification for product sampling, involving input data used for sampling, the actual algorithm and achieved coverage, as well as its evaluation (cf. Section 3). Second, we surveyed and classified the literature with respect to the classification (cf. Section 4–6). The list of studies and their classification can be found online.7 We plan to update this list in the future and welcome any pointers by the community. Third, we identify underrepresented research areas to be addressed by future work.

Our synthesis results indicate that most techniques used problem-space input information, in terms of feature models in generating product samples. Solution space information, such as test artifacts or code coverage, has rarely been used and we think bridging this gap may lead to novel research results. Regarding the developed techniques and algorithms, greedy and evolutionary algorithms have been developed most in this domain. Also, there are no techniques that consider the history of feature models and evolution in software product lines. Regarding evaluation, there are very few

References are sorted by author names but grouped into proposed algorithms, evaluations of sampling algorithms, and other references.

7 http://thomas.xn--thm-iaa.de/sampling/
evaluations on industrial-scale systems and there is a clear need for a benchmark (with different types of information, including feature models, test suites and test results) and agreed-upon metrics for efficiency and effectiveness. We elaborate on these observations in the remainder of the paper.

2 MOTIVATING EXAMPLE

In software product line engineering, the variabilities and commonalities among the products are described in terms of features. A feature is defined as a user-visible aspect or characteristic of a software system [62]. Features in a product line have different relations that can be described compactly by means of feature models [54], which can be graphically represented as a feature diagram. As an example, Fig. 1, represents the feature diagram of an Elevator product line. (This is a simplified version of the example provided by Kröter et al. in [64].) Each node in this diagram represents a feature. There are some features in this diagram such as Behavior and Mode that are mandatory. This means that any elevator in this product line should include these features. There are also optional features such as the feature Service, which is used to facilitate the maintainability of the elevator, that are optional, which means that there can be valid products with or without this feature. Additionally, there are different types of relationships between sub-features. For example, Sabbath and FIFO that are two possible modes for an elevator have alternative relationship. This means an elevator in this product line can only include one of these modes. Furthermore, a feature can require or exclude other features. For example, in an elevator with FIFO mode, there should be a button for the user inside the elevator and in each floor. Hence, there is a requirement relation between the CallButton and FIFO features. This relation is represented by the dashed arrow in Fig. 1. The products in a product line can be described as subsets of features. As an example the elevator product line with feature diagram represented in Fig. 1, consists of 10 valid products.

In Listing 1, we represent a part of the code related to the control unit of the elevator product line. (The code is in Java and preprocessing directives are used to add variability.) In Lines 2–11, a set of variables are defined that are used by methods and based on the set of selected features. In this code there is a main method, called run() (lines 12–20), which triggers the execution of the functions that embody the main parts of the functionality of an elevator. Based on the code, in each step the next state of the elevator and its direction is calculated based on the current state and the mode of the elevator. This is done using methods calculateNextState() and setDirection(). As shown in lines 21–31, the calculation of the next state can be done in different ways based on the features included in a product. The preprocessing directives are used to separate the parts of the code related to each feature. The execution of these parts is depending on the features included in a product. Assume that the goal is to test the behavior of the products in the elevator product line. A subset of these products is selected as a representative to be tested. The products can be selected in a random manner or with regards to a specific criteria. The quality of the selected set can be specified considering different measures, e.g., code coverage or the number of faults that are revealed by testing the sample set. Consider two sample sets $S_1 = \{ [e, b, m, c, s, h, a], [e, b, m, c, s, h, a], [e, b, m, c, s, f, d] \}$, and $S_2 = \{ [e, b, m, h, i], [e, b, m, f, c, d], [e, b, m, f, c, u] \}$; the size of the two sets is the same but, testing the products in $S_2$ results in revealing a compile error which is resulted from interaction of two features FIFO and UndirectedCall. This is due to the inclusion of a call in line 33 to a method which is not included in the code in case that feature UndirectedCall is selected (the implementation of the method in line 39 for this feature). As another example consider another possible sample set $S_3 = \{ [e, b, m, c, s, h, a] \}$. Using $S_3$ for testing will not reveal the existing fault (null pointer reference) in line 8 in the code. In order to reveal such a fault the feature DirectedCall should be included at least in one of the products in the sample set.

3 CLASSIFICATION OF PRODUCT SAMPLING

In this section, we give an overview of our classification of product sampling. The set of publications considered in this classification has been reached after several iterations. We first performed a targeted search and produced a sample of the key results in the field and we started with an initial classification given our experience in this area [2, 3, 39, 40, 81, 82] and adapted it in the process of studying the literature. Consequently, we had to repeatedly reclassify surveyed publications. Similarly, we expanded our studied literature using existing surveys [49, 56, 71]. We reached our literature catalog with 48 publications [1–48] by filtering out irrelevant publications and adding others by means of snowballing. In particular, we have limited our search to studies that are focusing on sampling algorithms that incorporate a feature model or evaluations of existing algorithms. In order to give a structure to our classification, we identified three research questions as follows.

**(RQ1)** What type of data is used as input for product sampling?

**(RQ2)** How are sample sets selected?

**(RQ3)** How are the sampling techniques evaluated?

Considering the above questions, we describe three main characteristics of product sampling approaches, namely, input data, sampling technique, and the evaluation. We refined these main characteristics iteratively by investigating the more fine-grained characteristics of the studied papers. The details about these three main characteristics and inferred sub-characteristics are explained in the following.

---

Figure 1: Elevator SPL feature diagram

---
A Classification of Product Sampling for Software Product Lines

3.1 Input Data for Product Sampling

The input data is one of the considered characteristics that is specified to address the types of input that sampling approaches exploit.

We consider two main sub-characteristics for input data, namely, problem space and solution space. In general, the problem space concerns the system requirements and specification handled during the domain analysis and the solution space refers to the concrete artifacts that are created through design and implementation processes. The problem space includes two sub-characteristics that are feature model and expert knowledge. We also refine the solution space characteristic into two sub-characteristics, namely, test artifacts and implementation artifacts. These characteristics are represented in Fig. 2 and are explained with more details in the following.

3.1.1 Feature Model. As mentioned in Section 2, feature models represent different relations among features in a product line. Hence, feature models can be used to recognize valid products from invalid ones throughout the sampling process. According to feature diagram in Fig. 1, a sample set that contains a configuration including both features Sabbath and FIFO is not valid as these two features are in an alternative relation. The feature model can be also considered as a kind of expert knowledge, we separate this sub-characteristic, as many studies use feature models as a part of information that they use.

3.1.2 Expert Knowledge. In general, expert knowledge is the knowledge about the characteristics of the environment that the system operates in. Recognizing important feature interactions and generation of products based on that can be considered as a case for which expert knowledge is used. As an example, consider the elevator product line given in Section 2. An expert might know that the products that include a combination of features Sabbath, Service and DirectedCall are more prevalent and in demand in the market. Then the products containing combinations of these features can be included in the sample set to make sure that they are analyzed.

3.2 Techniques for Product Sampling

While there are different inputs are necessary for sampling, there are also different principal techniques for computing sample sets. We distinguish four main sub-characteristics for the technique characteristic, namely, automatic selection, semi-automatic selection, manual selection and coverage, which are represented in Fig. 3 and explained in the following.

3.2.1 Automatic Selection. We consider the two following general types of automatic selection approaches:

Greedy: In greedy algorithms, a locally optimal choice is made in each stage of the problem solving [59]. In these algorithms, in each step, a new member is added to the sample set that is the best choice considering the current sample set and the defined criteria for sampling and the process continues with the resulting set. Consider
the Elevator product line. Assume that in the current step the sample set is \{ (c, b, m, h) \, (e, b, m, h, s) \} and that the criterion considered throughout the sampling process is feature interaction (Feature interaction is a software engineering concept which addresses the occurrence of changes in the behavior of the system related to a feature in presence and absence of other features). Then, the solution in the next step after applying a greedy algorithm can be \{ (c, b, m, h) \, (e, b, m, h, s) \, (e, b, m, f, d, c, s) \} since a new set of interactions among features \( f \) and \( d \) is covered.

**Meta-Heuristic Search:** The problem of finding a representative subset of products in a product line can be formulated as an optimization problem. Meta-heuristic algorithms aim at selecting a subset of products as an optimal solution for this problem using computational search in a configuration space. The search space can be potentially large due to high variability and complex feature combinations in an SPL. Many meta-heuristic approaches have been inspired by biological evolution [8]. Based on whether the meta-heuristic searches operate on individual states or population of states [11], the meta-heuristic approaches are divided into the following:

(a) **Local search**

Such approaches start with a preliminary set of products as a solution for the optimization problem and the search algorithm will gradually evolve the current set of products to reach a near optimal solution. Examples of such approaches are simulated annealing and tabu search.

(b) **Population-Based Search**

Such approaches start with a preliminary set of sample sets (sets of products) and then the algorithm will evolve the current sets of products to reach a final solution. Examples of such approaches are genetic algorithms and swarm techniques. In such approaches the primary set of solutions are mutated and recombined into new sets in order to find a near optimal solution. A fitness function is usually used as a measure for evolving the set of solutions during the process. As a general example consider the elevator product line and the code given in Listing 1. Assume that in the current step of the sampling process the generated solution consist of two sets, namely \( S_1 \) and \( S_2 \), each with one member \{ (c, b, m, h) \, (e, b, m, f) \}. In the next step the intermediate solution can be evolved to \{ (c, b, m, h) \, (e, b, m, h) \, (e, b, m, f) \, (e, b, m, f, c, d) \}. Considering different criteria one of these sets can be used to continue the sampling based upon.

### 3.2.2 Semi-Automatic Selection

In semi-automatic selection, different type of data such as the required number of products to be generated, the time for sampling, and a degree for coverage e.g., coverage on feature interactions can be considered.

**Figure 3:** Technique of product sampling

Furthermore, in such techniques, the full sample set or a primary set that is generated by other sampling techniques can be used as a starting point for sampling. An expert can provide the set based on information, such as feature model and domain knowledge.

#### 3.2.3 Manual Selection

The set of products can be selected manually. In this approach a set of products are selected by a domain expert and based on the knowledge that they have about the possible and common feature combinations.

#### 3.2.4 Coverage

Coverage criteria are often used to assure the quality of product sampling. One widely used criterion is feature interaction coverage. Considering this criterion, the main goal during the sampling process is to provide coverage for different kinds of feature interactions such as feature-wise (aka. 1-wise), pair-wise (aka. 2-wise), and t-wise. As an example, when considering the pair-wise coverage criterion, all the valid/possible pairs of features should be covered by configurations in the sample set. A common mean for extracting the sample set based on the feature interactions is a covering array [19]. A t-wise covering array is a subset of products that covers all the valid t-wise feature combinations in the product line. A covering array is commonly represented using a table where each row represents a feature and each column represents a product. As an example consider the elevator product line given in Section 2. A sample set that fulfills the pair-wise criterion for this example is \{ (e, b, m, h, s) \, (e, b, m, h, c, d) \, (e, b, m, h, f, d) \}. Code coverage criteria. Considering this criteria, the code should be covered with some percentage using the sample set. As an example, consider the code for the elevator product line given in Listing 1. One notion of code coverage could be that each ifdef block should be included at least once as a part of the code of the products in the sample set. As an example a sample set such as \{ (e, b, m, f, c, d) \} will cover all the ifdef blocks in the implementation given in Listing 1 at least once.

Additionally, there are some techniques which do not consider any notion of coverage during the sampling process. Hence, we address this in the classification of the selected studies as well (not as an explicit characteristic).

### 3.3 Evaluation of Product Sampling

Evaluation is another high-level characteristic in our classification. This characteristic mainly addresses the artifacts and the process taken to evaluate the sampling techniques. Furthermore, we refine this characteristic into three sub-characteristics, namely, tool support, evaluation criteria, and subject system, which are represented in Fig. 4, in addition to a set of sub-characteristics which are explained in detail in the following.
3.3.1 Evaluation Criteria. The evaluation of sampling techniques is performed by considering several criteria. Two major criteria that are recognized in our classification are efficiency and effectiveness. By considering different experiments we observe that several interpretations of efficiency and effectiveness are provided. As for the efficiency, the criterion can be addressing the efficiency of the sampling technique, which is related to the computation of the sample set or the efficiency of the testing technique that is used combined with the product sampling. An example of sampling efficiency is the time to generate the sample set. Also, the efficiency of the testing technique can be measured in terms of the sample size (the size of the sample set can affect the required time for testing) and the testing execution time. On the other hand, effectiveness addresses the quality of the sample set mainly with regards to a notion of coverage, e.g., fault coverage, feature interaction coverage.

3.3.2 Subject System. In this classification, we consider the subject systems and the case studies that are considered during the evaluation of the sampling technique. The type of subject systems can be an indication of the practicality and the scalability of the technique. In our classification, we classify three types of subject systems, namely artificially generated, which are subject systems generated by random combination of a set of features or using a program with regards to specific rules, academic, which are small subject systems mostly used as examples in academic work and provided by researchers, and real-world, which are systems that are used in practice. As an example the elevator product line example given in Section 2, can be considered as an academic subject system. Furthermore, another factor that we consider about the subject systems is the size of the corresponding feature model.

3.3.3 Tool Support. Another sub-characteristic in our classification is the tool support, which we use to indicate whether the sampling technique is supported by an implementation. We also specify if the tool set is open source and/or available for public use. We distinguish these characteristic since this type of information can be useful for users who are interested in application of sampling techniques in practice.

4 CLASSIFICATION BASED ON INPUT DATA

Before discussing how sampling techniques work internally, we answer RQ1 by giving an overview which input these techniques require to work (cf. Section 3.1). This input typically needs to be available for them to work (e.g., provided by the user).

Feature Model. All sampling techniques that we surveyed take the feature model as input, due to the scope of the survey. Sampling techniques rely on the feature model to distinguish valid from invalid configurations, as only valid configurations are desirable for many applications of sampling. To check validity, feature models are often translated to boolean satisfiability problems (e.g., [2, 21, 46, 78]) or to constraint satisfaction problems (e.g., [29, 46]) to make use of dedicated and efficient solvers. In some cases, the feature model is translated into the set of all valid configurations [5, 6, 26], which does not scale to large product lines. There are also sampling techniques that take advantage of the hierarchy in the feature model (i.e., the feature diagram) [4, 14, 31, 35] or support numerical attributes in feature models [12, 43]. In particular, Reuling et al. and Arcaini et al. apply typical change operations on the hierarchy to imitate faults introduced by domain engineers [4, 31]. If no feature model is available, it is often sufficient to have the list of features and take a tautology as input to the sampling algorithm, which basically means that there are no constraints on the features.

Expert Knowledge. While feature models are a common input to sampling, there are further sources of domain knowledge being used as input. A largely manual, but quite common technique in practice is to let domain experts identify a set of typical products manually. For example, changes to the Linux kernel are typically sent to the mailing list and a continuous integration server applies each patch automatically to compile and test it with a set of ten pre-defined and ten randomly selected products. Besides this continuous integration, Linux developers often only test the kernel with the all-yes-config, a configuration in which almost all features are selected [37, 60]. Many other product lines come with a default configuration being sufficient for many users [24]. Default configuration and all-yes-config are instances of single-configuration heuristics [24]. Multiple pre-defined configurations are supported when compiling or testing products in FeatureIDE [3, 72].

Oster et al. were the first to compute a sample based on a set of pre-defined products [29]. That is, users provide some products as input, which will be in the sample, and the sampling technique extends this set towards a representative set of products. With InCLing, Al-Hajjaji et al. presented a further technique for that purpose [2]. However, those pre-defined products are optional for both sampling techniques. Whereas all those discussed techniques include pre-defined configurations, Johansen et al. allow to define partial configurations (i.e., a subset of the product line) that are used for two purposes [20]. First, to rule out configurations that are valid according to the feature model but not expected for the sample. Second, to assign weights to those partial configurations to distinguish more relevant from less relevant configurations. Ensan et al. enable experts to also rule out certain configurations, but in a much simpler manner. They let experts define a subset of all features defined in the feature model, which is then used for sampling [7]. Similarly, Kowal et al. propose to filter feature models based on priorities assigned to features and known feature interactions, such as shared resources and data [23]. Henard et al. allow experts to specify test costs for each feature, which is then used in the objective function to better estimate the testing effort for specific configurations [17].

Implementation Artifacts. Whereas most sampling techniques are only based on problem space information, there are a few instances that also consider solution space information. Tartler et al. propose a sampling technique for product lines implemented with preprocessors [38]. They analyze the source code to ensure that every lexical code fragment (i.e., #ifdef block) is included in at least one sample product. This technique is also an instance of a code-coverage heuristic [24]. In contrast, Shi et al. use control flow graphs to identify which features can interact with each other [33]. An underlying and quite restrictive assumption of their work is that each feature is implemented by a single method and this method is called only once from a common code base. Whereas both sampling techniques are rather different, both use implementation artifacts as input to reduce the search space defined by the feature model.
Test Artifacts. Similar to implementation artifacts, also test artifacts have been used as input for sampling [21, 22]. Kim et al. proposed to exhaustively consider all configurations, for which a unit test [21] or a runtime check [22] can lead to different results. In particular, they use test and implementation artifacts to detect which features can influence the result of the test. Then they produce a sample for each unit test or runtime check, which covers all combinations of those features, whereas they exclude combinations that are invalid according to the feature model. In contrast to all other sampling techniques, which derive one sample for a product line, Kim et al. derive a separate sample for each test.

Summary and Insights. Without even going into detail how the sampling algorithms work, we have already noticed a large diversity in terms of the input used for sampling. In Table 1, we give an overview all surveyed algorithms and their classification. All surveyed sampling techniques use the feature model to distinguish valid from invalid configurations. Whereas feature models represented as propositional formulas are sufficient to feed them into a SAT solver, there are recent approaches that also incorporate the hierarchy in addition [4, 14, 31, 35]. Other sampling techniques can incorporate further domain knowledge from experts in terms of predefined or partial configurations [2, 3, 7, 17, 20, 23, 24, 29, 37, 38]. Besides those inputs, domain knowledge has also been extracted from implementation artifacts [21, 22, 33, 37, 38] and test artifacts [21, 22]. Input besides the feature model is typically used to further restrict the set of valid configurations or derive more realistic samples.

While our survey identified numerous interesting inputs for sampling, the vast majority of techniques only consumes the feature model and cannot incorporate any further input. The advantage of those sampling techniques is the better applicability. There is no need to have further domain knowledge or access to implementation and tests. Also, those techniques are completely independent of the variability mechanism used for domain artifacts and do not require experts. However, from an algorithm point-of-view it is possible that with more input, we can produce better samples with fewer resources.

Here are some future research directions that we identified with respect to further input. First, there is not a single technique incorporating the history of the product line. The history of the feature model may reveal new features or combinations, whereas changes to the pre-defined configurations or domain artifacts indicate what should be covered. While there are numerous techniques for SPL regression testing [36, 66], they typically take a fixed sample for all versions or do not use sampling. There are even applications that require the computation of samples being stable over the history. For example, if we aim to analyze the performance of a product line over time, we should probably not consider a completely different sample for each revision. Second, already known feature interactions derived with static analyses or even defects or vulnerabilities occurred in the past (e.g., documented in issue trackers) may enhance samples even further. Third, we could try to include especially those configurations that were outliers before (e.g., configurations with most defects as well as fastest and slowest configurations). Finally, requirements as well as informal or formal specifications can be used as input for sampling.

Table 1: Overview on the literature on product sampling, grouped into algorithms [1–38] and evaluations [39–48].

5 CLASSIFICATION BASED ON TECHNIQUE

To answer RQ2, we present in this section a classification of the selected studies based on the technique used for product sampling.
Greedy Techniques. Several greedy sampling algorithms have been used to sample products in product lines [1–5, 7, 14, 18–24, 28, 29, 31, 33, 37]. In the following, we explain the main greedy sampling techniques that are addressed in the above studies.

One of the first greedy algorithms that have been proposed to generate a minimal set of configurations [18]. With Chvatal, the combinations of features are generated to be considered during the sampling process. The configurations are added to the sample set in a greedy manner and each newly added configuration should cover at least an uncovered combination. Similarly, the ICPL algorithm is introduced for generating covering arrays for large scale feature models [19]. In fact, the ICPL algorithm is built upon the Chvatal algorithm [18] with additional performance improvements, such as parallelizing the sampling process. Similar to Chvatal, the ICPL algorithm receives a feature model and a coverage strength \( t \) as input. Then it generates a covering array that produces the \( t \)-wise coverage as output. MoSo-PoLiTe (Model-based Software Product Line Testing) is another greedy algorithm that has been proposed to generate a set of products using feature models [29]. In MoSo-PoLiTe, the pair-wise combinations are extracted. The algorithm starts with a pair of features and incrementally adds the rest of pairs by applying forward checking to check whether the selected pair can be combined with the remaining pairs of features to generate valid products. Moreover, MoSo-PoLiTe considers the pre-defined products in the sampling process. Al-Hajaji et al. [2] propose an algorithm, called IncLing, where the products are generated incrementally one at a time. Similar to MoSo-PoLiTe [29], IncLing considers the set of products that are already selected and tested. This algorithm aims to increase the diversity among selected products by choosing dissimilar pair-wise feature combinations to be covered in the next product in order to increase the possibility of fault detection [2, 3].

A divide and conquer strategy is used in [30] to generate \( t \)-wise combinations from a feature model. Using the divide and conquer technique, a set of sub-problems is given to a constraint solver which are then solved automatically instead. In this approach the features in the feature models are translated to Alloy specifications. A set of products are generated using the resulting model which provide \( t \)-wise coverage.

To select optimal products with respect to the non-functional properties of product line, several approaches have been proposed [32, 34, 35, 33, 36]. For instance, Siegmund et al. [35] propose an approach that predicts footprint and main-memory consumption of products. In their work, they generate a small set of products to approximate the influence of features. In particular, they consider feature-wise sampling, where a product for each feature is generated, to measure the influence of each feature separately, interaction-wise sampling, where a product for each interacting features that are given based on domain knowledge, is generated, to measure the influence of feature interaction, and pair-wise sampling, where a product for each pair of features is generated, to measure the influence of the pair-wise feature interaction. In the same direction, Sarkar et al. [32] predict the performance of configurable systems by adapting sampling strategies that generate samples with the aim of estimating the learning behavior of the prediction model as well as the cost of building, training, and testing it. In particular, they adapt progressive sampling strategy, where in each iteration a fixed set of sample is added to the training set, and projective sampling strategy, where the learning behavior is approximated using a minimal set of initial samples. To do so, they exploit the feature frequencies (i.e., how often the features appear in the current set of samples) to generate the initial set of samples for both sampling strategies.

To reduce the number of products to be tested, several greedy search-based reduction techniques have been proposed [7, 14, 20–23, 33]. For instance, Kim et al. [21] propose a technique, where the features that do not have any impact when the test is performed are removed. Similarly, a compositional symbolic execution technique has been proposed to reduce the number of generated products by identifying the possible interaction between features [33]. Another algorithm is proposed where a product line is divided into sub-product lines with weights given by domain experts [20]. Then, the products that have more weights are selected to be tested first. Similarly, Ensan et al. [7] propose to reduce the size of the sample set by covering the most desirable features, which are usually given by the domain experts. Hashinger et al. [14] propose reduction rules based on the constraints in feature models, which applies during the sampling and results in reducing the number of feature combinations that are used to generate the set of configurations. In the same direction, Kowal et al. [23] propose to cover only features that interact with each other in the generated sample. They propose to model the additional information about the interaction between features into the feature models. Thus, the number of generated products is reduced as a result of reducing the number of combinations of features that are required to be covered. The aforementioned greedy techniques sample products based on different criteria, such as coverage [2, 19, 29]. However, additional sampling algorithms have been proposed [1, 3, 6, 15, 24, 37]. For instance, random sampling, which can be considered also a greedy technique, generate a set of products randomly [3, 7, 15, 24]. In this technique the configurations that are not valid according to the feature model are discarded. However, with random sampling, no specific coverage criteria through the generation of products can be guaranteed. Most-enabled-disabled (i.e., configurations where most of features are enabled or disabled), all-most-enabled-disabled (i.e., all possible configurations where most of features are enabled or disabled), one-enabled (i.e., configurations where one feature is enabled and the others are disabled) are sampling techniques in which special combinations of features are obtained by enabling/disabling features [1]. The sample sets can be computed in a greedy manner using these techniques. Several mutation-based approaches have been exploited to sample products [4, 15, 31]. These approaches aim to generate products that have a high probability of containing faults. Several meta-heuristic approaches have been proposed to sample product lines [5, 6, 8–10, 15, 17, 25, 27]. In this subset of studies, an initial set of products is selected randomly and then gradually evolves by considering the constraints in the optimization problem. Usually the size of the initial set is given by testers. Based on the search process of finding solutions, we classify these approaches into local search and population-based search (cf. Section 3).
Local Search Techniques. Several local search techniques have been proposed to sample products [5, 8, 10, 15, 16, 26]. For instance, the 1+1 evolutionary algorithm has been used to sample products [15, 16]. While 1+1 evolutionary algorithm, the main evolution operator is the mutation. In the fitness function, they consider the similarity between products. Their goal is to generate products that are dissimilar to each other with respect to the selected features in each product [15]. CASA is a local search technique that uses simulated annealing to generate a set of products, which achieves a certain degree of t-wise coverage. In this approach, in the first step, the number of configurations are minimized and in the next step it is ensured that a certain degree of coverage is achieved [11]. Ensan et al. [6] propose a meta-heuristic approach to generate optimal products with respect to some criteria, such as feature coverage. While it can be seen as a population-based technique as it considers genetic algorithm, we consider it in this paper as a local search technique, as it operates on individual states. Marijani et al. [26] propose an optimization technique to generate a set of products that achieves pairwise coverage. In particular, the algorithm first, transforms the feature models into an internal constraint model. This model is expressed as a matrix, where columns represent features and rows represent configurations. Given this matrix, an optimization algorithm is proposed to generate a minimal set of products for a given time.

Population-Based Techniques. Several population-based algorithms have been used to sample products [6, 8, 10, 17, 25, 27, 41]. Multi-objective evolutionary optimization is recognized among evolutionary approaches for sampling. In such approaches, the product sampling is defined as a multi-objective optimization problem and multi-objective algorithms are used for solving the problem [9]. In such techniques, the objective function is usually defined based on a coverage criteria [25, 27]. Genetic algorithms, which are extensively used, fall into evolutionary techniques category. These algorithms form a correspondence between the genetic structure of chromosomes and the representation of solutions in the optimization problem [8, 17]. Considering this correspondence, the solutions of the optimization problem (i.e., the set of configurations) are represented as chromosomes, and hence by using natural biological evolution operators, such as mutation and crossover, a new near optimal solution can be generated. The Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) [6] is another evolutionary algorithm which decomposes a multi-objective optimization problem into a set of scalable optimization sub-problems and solves them simultaneously [6]. MOEA/D-DRA is an extension of this algorithm that enables dynamic resource allocation [6].

Semi-Automatic Selection Techniques. Oster et al. [29] and Al-Hajaji et al. [2] consider pre-defined configurations, which can be given by domain experts, in the sampling process. One semi-automatic sampling technique is the single configuration technique where a single configuration is selected as a representative for all the products to be analyzed [24]. This configuration is typically selected by a domain expert, where usually part of the sampling is performed automatically, such as checking the satisfiability. One disadvantage of this technique is that mutually exclusive behavior/code can not be covered by one selected configuration. Note that the other sampling techniques, where the user can be involved in the sampling process, can also be classified as semi-automatic techniques.

Manual Selection Techniques. Several approaches also support manual sampling, where domain experts are usually generate products based on domain knowledge [5, 24, 37]. All-yes-config is a sampling approach, where all the possible features are selected to be included in a product [37, 38]. The main limitation of using this approach is that the selected configuration can include alternative behavior which is not valid according to the constraints between features. Note that the generated all-yes-config may not be the optimal configuration due to the dependencies between features which may affect the sampling process. Another manual selection approach, where a set of configurations in Linux is selected iteratively, is used [38]. In particular, they aim to add more blocks of the code by additionally considering the constraints between the blocks.

Feature Interaction and Coverage. Several greedy algorithms generate a set of products that guarantee a certain degree of coverage. For instance, the following algorithms achieve 1-wise coverage [3, 5, 32, 36]. While some of the algorithms achieve the pair-wise coverage [2, 26, 29, 36], other techniques scale to t-wise interaction coverage [11, 14, 18, 19, 23, 30, 33]. Except for the work of Garvin et al. [11], where they cover up to 6-wise coverage, most of the sampling techniques that consider meta-heuristic algorithms do not guarantee 100% of a coverage degree [6, 8, 10, 15–17, 25, 27, 41]. Note that the random-based techniques [3, 8, 15, 24], semi-automatic selection techniques [2, 24, 72], mutation-based sampling [4, 15, 31], and some of the greedy techniques, such as [7, 26], do not achieve degrees of coverage. Compared to the feature interaction coverage, only a few studies consider the code coverage [21, 22, 33, 38]. With these techniques, a minimal sample set is selected such that every lexical code fragment of the whole system code is covered by at least one product.

Summary and Insights. Based on our classification, we observe that the greedy techniques [1–3, 7, 14, 18–24, 29, 31, 33, 37] and meta-heuristic algorithms [5, 6, 8–10, 15, 17, 25–27, 41] are used more often compared to other techniques (cf. Table 1). However, other semi-automatic selection techniques have been also proposed [2, 24, 29, 72], where the users have the ability to influence the sampling process. Moreover, some of the greedy techniques aim to guarantee a small set of configurations that achieves a degree of feature interaction coverage (e.g., pair-wise coverage [2, 26, 29, 36] and t-wise [11, 14, 18, 19, 23, 30]). However, the main limitation of most of algorithms is that they do not scale to large product lines [28]. Thus, several approaches have been proposed to improve the scalability by reducing the configuration space [14, 21, 33]. Most of these reduction techniques require more domain knowledge to guide the sampling process.

As an alternative to the t-wise sampling techniques, meta-heuristic algorithms have been proposed to sample products of product lines. These meta-heuristic algorithms aim to handle many objectives during the sampling, which is not the case with most of the greedy algorithms, where only the coverage is often considered in the sampling process. However, the limitation of these meta-heuristic algorithms is that they are not deterministic, which may influence...
the testing badly, as they often cannot reproduce the same products to check whether detected faults are fixed, especially when feature models of the corresponding product lines are modified. Other greedy techniques (e.g., random sampling) are proposed, which do not guarantee a certain degree of feature interaction coverage. Moreover, we observed that only two greedy algorithms consider the prioritization of the generated products [2, 29], while the order in the other greedy algorithms are often influenced implicitly by the coverage, as they try to cover as many of the uncovered combinations as soon as possible.

In future, we argue that combining different techniques can improve the testing effectiveness and efficiency by avoiding or diminishing the limitations of existing approaches and benefiting from their advantages. For instance, combining the meta-heuristic algorithms with the greedy ones may be helpful to avoid being trapped in the local optima. Furthermore, we noticed that only a few of the existing sampling algorithms consider the code coverage. Thus, sampling techniques that guarantee a degree of code coverage are required, because most of the faults exist in the source code [1]. Moreover, there are no sampling techniques that consider the evolution of product lines, such as the history of existing samples. Taking the history of previous samples into consideration can be used to increase the diversity to cover more interactions or to reduce diversity over time for other applications.

6 CLASSIFICATION BASED ON EVALUATION

The techniques proposed in the literature have been evaluated for different measures of efficiency and effectiveness, against various subject systems, and using various types of tools. With respect to RQ3, this section provides a synthesis of these aspects of evaluation for the surveyed techniques.

Efficiency. Most of the analyzed sampling approaches have been evaluated to assess some measure of efficiency. In two cases [5, 7], no evaluation was done. Regarding efficiency, we distinguish between sampling and testing efficiency.

(a) Sampling efficiency. This notion measures the time (and possibly memory and computational resources) to generate the sample. We have identified several studies [2, 4, 5, 9, 11, 14–19, 21–26, 29, 31, 33, 38, 40, 42, 46] that evaluated the sampling efficiency by measuring the execution time of the algorithms to generate the sample. Kowal et al. [23] reduce the feature model to operate on smaller input data. They compare the efficiency of their sampling approach against existing ones by measuring the computation time. Other measures of resource consumption such as volatile or persistent memory consumption are also relevant in this respect, but none of these studies evaluated any other measures of efficiency.

(b) Testing efficiency. Testing efficiency focuses on the resulting sample, such as the number- and the size of configurations in the sample set. There are studies evaluating the testing efficiency by counting the number of generated configurations [1, 2, 4, 5, 7, 8, 10–17, 19–32, 34–36, 40–48]. Henard et al. [17] also check the size of the configurations to evaluate the testing efficiency. By the size of a configuration, we mean the number of selected features of the feature model. In all these papers, the problem space was evaluated but not the solution space. Liebig et al. [24] used the solution space, code artifacts besides the feature model, to evaluate the algorithm.

Testing efficiency was evaluated by measuring the analysis time, i.e., the time required for type checking and data-flow analysis.

Effectiveness. By effectiveness, we mean the quality of the generated sample set and its capability in detecting faults. Effectiveness typically measure some notion of coverage. We subdivided the measurement of effectiveness into three types, such as (a) feature (interaction) coverage, (b) fault coverage (also incarnated in measures such as mutation score), or (c) code coverage.

(a) Feature (interaction) coverage. The feature (interaction) coverage was measured in [2, 6, 8–10, 17, 20, 25, 30, 42, 45] to rate the effectiveness of algorithms. Johansen et al. [20] weighted the feature interactions in the covering array. In a special case [16], the feature coverage of the sample set is measured for a specific amount of time. All of these approaches checked if a pairwise/t-wise feature coverage is achieved. This means, every pair (respectively, every t-wise tuple) of features has to occur in at least one configuration. Here, only the problem space is used in the evaluation. In some cases, no 100% coverage is achieved because evolutionary algorithms are used [6, 8, 42].

(b) Fault coverage. Other studies measured fault coverage [1, 4, 8, 28, 31, 37, 39, 42, 44], i.e., the capability of detecting certain faults. Ensar et al. [8] and Tartler et al. [37] use a program which marks those features or combinations that contain a fault. The evaluation then analyzes if all errors are discovered by the configurations in the sample. To discover features which contain an error, only the specific feature has to be in a configuration, but to discover errors stemming from feature interactions, all features of the corresponding combination have to be in a configuration. Ferreira et al. [9], Filho et al. [27], and Henard et al. [15] evaluate their approach using a mutation score (i.e., the discovered mutants in a feature model). Al-Hajjaji et al. [39, 40] introduce two prioritization approaches but they also measure how effective the default order of sampling algorithms is w.r.t. the fault detection rate. Al-Hajjaji et al. in [39] use delta-modeling for the evaluation. They analyze differences between products and select a sample based on these information. Halin et al. [44] use a real subject system and associate test cases to measure the fault coverage instead of only feature models. Abal et al. [1] also detect real faults. They use the commit history of the Linux kernel and map fixed bugs to features. Then, they evaluate if their sampling approach covers these features to find the bug.

(c) Code coverage. We analyzed the studies based on which input data they used for the evaluation. Most of them only use the feature model, but two studies also use code as input [38, 40]. For example, Tartler et al. [38] use code block coverage as an evaluation metric. Siegmund et al. [34, 36, 47, 48], Grebhalh et al. [12, 43] and Sarkar et al. [32] evaluate the effectiveness of sampling algorithms in a different way. They use the sampling result to predict the performance of products and compare against real performance measurements. The prediction error for different sampling strategies is evaluated. Siegmund et al. [35, 36] also used the approach to predicted footprints of products.

Subject Systems. The subject systems used for evaluation vary significantly. We distinguish between real [1, 2, 13, 16, 18–22, 24, 28, 30, 32, 34–38, 40, 41, 47, 48, 78], academic [2, 6, 8–10, 16, 17, 19, 23, 26, 27, 32, 34–36, 39, 41, 44, 45, 47, 48, 78] and artificial [2, 4, 16, 29, 31, 40, 46] feature models used as subjects. As
mentioned in Section 3, real feature models are models of existing projects used in practice (e.g., in open source or industrial projects), such as the Linux kernel. The difference between academic and artificially-generated feature models is that academic feature models are created by researchers, while artificially-generated feature models are constructed by a program according to some rules; in other words, to generate artificial feature models no specific domain knowledge and manual intervention is necessary. SPLOT [74] is a popular tool for the generation of artificial feature models.

For example, Oster et al. in [29] use artificial feature models to evaluate the algorithms. The feature model of the Linux kernel is also used in many cases [1, 2, 16, 18, 19, 24, 37, 38, 40]. Furthermore, the E-Shop is an example of an academic feature model which is used in some studies for evaluations [6, 9, 10, 23].

The size of the feature models is distinguished in a range from very small ones with fewer than 20 features to huge feature models containing more than 10,000 features. We collected the size of all evaluated feature models if it was specified by the authors. Therefore, we can only present the lower bounds of the tested feature models. More than 1,000 feature models with less than 100 features were evaluated [2, 4–6, 8–23, 26–36, 39–45]. We collected more than 100 feature models with a size between 100 and 500 features [2, 4, 11, 14–16, 18, 19, 23, 28, 29, 31, 35, 36, 40, 46]. Bigger feature models are less evaluated. In the considered studies approximately 15 models with a size bigger than 5,000 [1, 2, 16, 18, 19, 24, 28, 38, 40, 78] are evaluated. The biggest evaluated feature model of the Linux kernel has 26,427 features [1, 28].

**Tool support.** Considering tool support, we observe that several studies have an implementation [2–6, 8–22, 24–38, 40, 43, 45–48], and some of these implementations are either open source or publicly available [2, 3, 8, 11–13, 15, 17–20, 24, 25, 28, 32, 34–38, 40, 43]. The implementations are based on various programming languages. The most popular programming language for implementations is Java [8, 17, 19, 30]. There are implementations which cover a set of algorithms, e.g., ICPL [19], and IncLing [2] are implemented in the FeatureIDE framework [72].

**Summary and Insights.** Most papers evaluate one or more algorithms using some measure of efficiency or effectiveness. Efficiency mostly focuses on the time of sample generation or the size of the sample [1, 2, 4, 5, 7–36, 38, 40–48], while effectiveness mostly addresses feature (interaction) coverage [2, 6, 8–10, 17, 20, 25, 30, 42, 45] and less prominently, fault coverage [1, 4, 8, 28, 31, 37, 39, 42, 44] or code coverage [38, 40]. The used subject systems are equally distributed between real and academic feature models. Artificially generated ones are less used. The size of the feature models also ranges from small feature models with only a few features to very large ones with almost 7,000 features to over 26,000 features. Another interesting point is that many authors do not name their tool. Unique names would be helpful to distinguish implementations.

Regarding efficiency, measuring other resources such as memory consumption or amount of required storage are missing in the literature. We also identified lack of evaluations using the solution space. Most papers only use the feature model as input data in the evaluation. For example, the evaluation of real faults in the code are under-studied. More research seems to be necessary to establish a common measure of effectiveness in terms of fault coverage.

Comparing the different measures of effectiveness and studying the compromise between efficiency (sampling and testing time and resources) versus effectiveness in different domains seems to be under-studied. Providing a common benchmark of subject systems based on real examples from various domains will facilitate the evaluation of different algorithms on a common ground. Furthermore, evaluation of sampling algorithms, which consider evolution of product lines, could be an interesting topic for future work.

7 RELATED WORK

There are numerous strategies to cope with many software products during analysis and testing of software product lines [56, 81, 82]. Considering all products separately in an unoptimized product-based analysis is typically not feasible [81]. In optimized product-based analyses, the situation is improved by reducing the number of products or by reducing the effort for each product. Product sampling as discussed in this survey aims to reduce the number of products and is also known as sample-based analysis [81] or simply as sampling [82]. The effort for each product can be reduced by applying regression techniques to software product lines [33, 55, 58, 68–70, 79]. As such regression-based analyses already assume that a sufficiently small number of products is given, they are often combined with product sampling.

In contrast to product-based analyses, a software product line can also be analyzed in a feature-based or family-based fashion [81]. Feature-based means that the implementation of each feature is analyzed separately without taking other features into account. However, this way feature interactions are not covered by the analyses [81]. Family-based analyses also consider domain artifacts instead of generated products, but incorporate valid combinations of features as defined in the feature model [56, 81]. Family-based analyses have the inherent problem that they require special tools, whereas product-based analyses can always use tools from single-system engineering. Special tools are needed for the analysis itself or at least to transform the product line into a metaproduct simulating the behavior of all products [63, 73, 76]. While the family-based strategy has been extensively studied for static analyses [81] and is known to outperform sample-based analyses [24, 51], recent applications to testing indicate that product sampling is still necessary to complement family-based testing [63, 73, 76].

In principle, product sampling can have numerous applications, but it is typically proposed in the context of product-line testing. Consequently, existing surveys on product-line testing discuss product sampling [56, 65], but not as detailed as we do. While our focus is on sampling for product lines, the roots of this research area are in combinatorial interaction testing [77]. In contrast to combinatorial interaction testing, product sampling is specific to product lines and is typically based on feature models. Ahmed et al. [49] conducted a systematic literature study on interaction testing supporting constraints. Their scope is different, as they also incorporate constraints on input parameters for testing of single systems and miss applications of product sampling beyond testing. Furthermore, our classification is more detailed and gives more insights. Lopez-Herrejon et al. [71] perform a systematic literature review about the combinatorial interaction testing approaches in product lines. They briefly classify the proposed techniques into different categories.
In our classification, we provide more details about the discussed techniques, such as the required input as well as the criteria that are used to evaluate these techniques. Johansen et al. [61] present a survey of empirics of product line testing strategies. They report that only a few research conduct an empirical study to evaluate the corresponding strategies. Lamachana et al. [65] and Carmo Machado et al. [56] conducted a systematic literature review, where sampling techniques have been discussed as part of the approaches that have been proposed to select products for testing. In this paper, we distinguish with more details between the proposed approaches that sample products of product lines with respect to many characteristics, such as the required information for sampling, the considered criteria, and the tool support. Medeiros et al. [28] present a comparative study of 10 sampling algorithms for product lines with respect to the size of the generated samples and the fault detection rate. In our work, we classify the state of the art product sampling approaches with respect to many characteristics, including the sample size and the rate of fault detection.

Numerous of the distinguished sampling approaches in this paper have been used in the evaluation of the existing product prioritization approaches [39, 67]. For instance, Al-Hajjaji et al. [39] aim to find faults faster by prioritizing products based on their dissimilarity with respect to solution artifacts. They evaluate their work against the default order of the sampling algorithm MoSoPolite [29]. On the contrary, Lity et al. [67] prioritize products based on their similarity for testing. While this approach is not mainly about sampling products, it involves selecting test cases, test requirements, products, and their interrelations to reduce the effort during the incremental analysis by minimizing the differences between the consecutive analyzed products.

Ballar et al. [52] propose a multi-objective approach that considers test cases, test requirements, products, and their interrelations with respect to the corresponding cost and profit of selecting them. In particular, they use a mapping from requirements (i.e., test goals) to test cases and a mapping from test cases to products as input. They evaluate the effectiveness of their approach by measuring the accuracy of selecting a set of products and test cases to test a product line, with respect to certain restrictions, such as reducing the cost as well as maximizing the profit of testing. While this approach is not mainly about sampling products, it involves selecting products to be tested from a large set of products. In single-system engineering, Yoo et al. [84] surveyed the efforts have been made to select test cases. NIE et al. [77] discuss the combinatorial interaction testing techniques with respect to their important issues, methods, and applications. In this paper, most of the classified sample approaches are considered as instances of combinatorial interaction testing approach.

8 CONCLUSION

In this paper, we presented an overview of the literature regarding product sampling for efficient analysis of software product lines. To this end, we classified the literature in terms of the type of information used in the technique, the algorithms employed, and the methods and measures used to evaluate them. In each characteristic, we identified the areas of strength and the understudied areas, which can help researchers and practitioners to choose an appropriate technique based on their constraints.

We gained numerous insights by means of this survey. A vast majority of techniques only use feature models as input. However, there are other types of input data that can be incorporated in sampling which need more investigation in the future, such as the product-line evolution, different forms of specification, known feature interactions extracted using static analyses, as well as test artifacts and implementation artifacts. Greedy and meta-heuristic techniques are more commonly used among different techniques. However, as scalability is the main issue in many sampling approaches, using semi-automatic selection techniques could help with reducing the configuration space by incorporating the expert knowledge. Moreover, techniques that consider code coverage or notions of diversity seem to be understudied. Considering evaluation, the efficiency, mostly measured by time of sample generation, sample size, and the effectiveness, mostly measured by feature interaction, are most commonly used. Further efficiency measures, e.g., memory or storage consumption, as well as incorporating solution space in evaluation need more investigation. Furthermore, providing a benchmark of subject systems based on real examples facilitates future evaluations.

ACKNOWLEDGMENTS

We gratefully acknowledge constructive discussions on the classification at the FOSD meeting in Gothenburg. In particular, we thank Myra Cohen for her suggestions with respect to the classification of meta-heuristic search techniques. This work has been partially supported by the European project ITEA 3 15010 ReVAMP and by the German Research Foundation within the project ImoTEP (grant agreement LO 2198/2-1).

REFERENCES


A Classification of Product Sampling for Software Product Lines

SPLC ’18, September 10–14, 2018, Gothenburg, Sweden


Norbert Siegmund, Alexander von Rhein, Sven Apel, and Gunter Saake. 2016. On Essential Con-

Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Con-

Norbert Siegmund, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyrer. 2013. Strategies for Product-Line Verification: Case Studies and Experi-


Appendix F

Paper VI
Comparative Expressiveness of Product Line
Calculus of Communicating Systems and
1-Selecting Modal Transition Systems

Mahsa Varshosaz and Mohammad Reza Mousavi

Conference on Current Trends in Theory and Practice
of Computer Science (SOFSEM)
Novy Smokovec, Slovakia, January 27–30, 2019
Comparative Expressiveness of Product Line Calculus of Communicating Systems and 1-Selecting Modal Transition Systems

Mahsa Varshosaz1 and Mohammad Reza Mousavi2

1 Centre for Research on Embedded Systems
Halmstad University, Sweden,
mahsa.varshosaz@hh.se

2 Department of Informatics,
University of Leicester, UK
mm7@le.ac.uk

Abstract. Product line calculus of communicating systems (PL-CCS) is a process calculus proposed to model the behavior of software product lines. Modal transition systems (MTSs) are also used to model variability in behavioral models. MTSs are known to be strictly less expressive than PL-CCS. In this paper, we show that the extension of MTSs with hyper transitions by Fecher and Schmidt, called 1-selecting modal transition systems (1MTSs), closes this expressiveness gap. To this end, we propose a novel notion of refinement for 1MTSs that makes them more suitable for specifying variability for software product lines and prove its various essential properties.

Keywords: Product line calculus of communicating systems (PL-CCS), Modal transition system (MTSs), 1-selecting modal transition system (1MTS), Comparative expressiveness

1 Introduction

Variability modeling is a cornerstone of software product line (SPL) engineering, through which an inventory of commonalities and differences among different products are specified in a structured manner. Efficient analysis of variability-intensive systems is a major challenge due to the potentially large number of valid products. To this end, many techniques have been adapted, which exploit variability in different types of analysis. A basic building block of many of these techniques is a model for capturing variability at the structural or behavioral level. In this paper, we focus on formal behavioral models that can be used to capture variability; examples of such models include modal transition systems (MTSs) [18], product line calculus of communicating systems (PL-CCS) [14], and featured transition systems (FTSs) [10].

In a previous paper [9], we studied the comparative expressiveness of these formalisms with respect to the set of products (labeled transition systems (LTSs)) they can specify. There, we proved that MTSs are strictly less expressive than
PL-CCS (and its underlying semantic model, product line labeled transition systems (PL-LTSs)). A formalism that was not studied in our previous paper [9] is 1-Selecting Modal Transition System (1MTS) [12], which extends modal transition systems with (must/may) hyper transitions. Such hyper transitions bundle a number of possible behavior, of which exactly one will be included in each valid product. Using 1MTSs it is possible to model alternative behaviour (choices with XOR relation) in products, which cannot be modeled using MTSs. Intuitively, this seems the very missing modeling feature in order to fill the expressiveness gap between MTSs and PL-LTSs.

In this paper, we show that this extension is indeed sufficient to close the expressiveness gap between MTS and PL-LTS (see Section 5). Furthermore, we observed that by considering the current refinement relation provided for 1MTSs, some aspects of behavioral variability, such as persistent choices in recursive specifications, cannot be modeled satisfactorily (see Section 3). Hence, we propose a new refinement relation for 1MTSs which addresses these concerns (see Section 4) and also leads to more succinct models, and we show that the new refinement relation enjoys the same intuitive properties as the original one [12]. The other direction of comparison (from 1MTSs to PL-LTSs) is left as a future work. However, we conjecture that encoding 1MTSs into PL-LTSs is also possible.

2 Preliminaries

In this section, we explain some basic concepts regarding software product lines, 1-selecting modal transition systems, and product-line labeled transition systems that are used throughout the rest of the paper.

2.1 Software Product Lines

The products in a software product line are developed from a common core. The commonalities and variabilities among products are usually described in terms of features. A feature is a distinctive user-visible aspect or characteristic of the system [15]. The products in a product line can be described as sets of features. There are different types of relations between features in a product line. We explain some of these relations using an example of a vending machine product line. The product line includes three mandatory features, namely, Coin, Drink, and Coffee, which means all the products in this product line should include these three features. There are two types of coin, namely, Dollar and Euro which have alternative relation. This means that a product in this product line can either accept dollar or euro coins but not both. The Drink feature has two sub-features as well, namely, Tea and Coffee. The Tea is an optional feature. This means that a product in this product line can offer both tea and coffee or only coffee as drinks (since, Coffee is a mandatory feature).
Comparative Expressiveness of PL-CCS and 1-Selecting MTS

2.2 1-Selecting Modal Transition Systems

Fecher and Schmidt [12] introduced the following definition of 1MTSs.

**Definition 1 (1MTS).** A 1-selecting modal transition system, is a tuple $(S, A, \rightarrow, \cdot\cdot\cdot, s_{init})$, where:

- $S$ is a set of states or processes,
- $A$ is a set of actions,
- $\rightarrow \subseteq S \times (2^{A \times S} \setminus \emptyset)$, is the must hyper transition relation,
- $\cdot\cdot\cdot \subseteq S \times (2^{A \times S} \setminus \emptyset)$ is the may hyper transition relation,
- $s_{init} \subseteq S$, is a non-empty set of initial states.

In each 1MTS, the relation $\rightarrow \subseteq \cdot\cdot\cdot$ holds between the sets of may- and must hyper transitions. This means that must hyper transitions also implicitly represent may hyper transitions.

We use $\text{IMTS}$ to denote the class of all 1MTSs.

Based on the above definition, there are two types of hyper transitions in a 1MTS, called may- and must hyper transitions. A may hyper transition represents a set of alternative choices which are optional (at most one of the choices can be selected). On the other hand, a must hyper transition represents a set of alternative choices where selecting one of the choices is obligatory. Furthermore, we assume that for each state $s$, $(s, \cdot\cdot\cdot) = \{\gamma \mid (s, \gamma) \in \cdot\cdot\cdot\}$. A simple example of a 1MTS is provided in Fig. 1. This 1MTS represents the behavior of products in the vending machine product line.

In order to define how a transition among those in a hyper transition is chosen, the following notion of choice function is used.

**Definition 2 (Choice Function).** Let $A$ be a set, and $B \subseteq 2^A$ and $\gamma : B \rightarrow A$. Then $\gamma$ is a choice function if $\forall b \in B : \gamma(b) \in b$. The set of all choice functions on $B$ is defined by $\text{choice}(B)$.

As 1MTSs are abstract models, one can associate with each 1MTS a set of 1MTSs that refine it by allowing for fewer optional choices. The refinement relation on 1MTSs is defined as follows [12].
4 M. Varshosaz and M. R. Mousavi

Fig. 2: (1) 1MTS and (2) LTS refining the model in Fig. 1 and Fig. 2(1).

Definition 3 (Refinement for 1MTSs). A refinement relation between two 1MTSs such as \( M = (S, L, \rightarrow, s_{init}) \) and \( \bar{M} = (\bar{S}, \bar{L}, \bar{\rightarrow}, \bar{s}_{init}) \), is defined as a relation \( R_{1MTS} \subseteq \bar{S} \times \bar{S} \) such that \( \forall \bar{s} \in \bar{S}, \exists s \in S \) \( R_{1MTS} \bar{s} \) and \( \forall (s, \bar{s}) \in R_{1MTS} \), such that:

1. \( \forall \bar{s} \in \bar{S}, \exists s \in S \) \( R_{1MTS} \bar{s} \), where \( R_{1MTS} \) is a relation relating each of the initial states of \( M \) to one of the initial states of \( \bar{M} \).

As a simple example in Fig. 2(1), a 1MTS is shown which refines the 1MTS in Fig. 1. In this 1MTS, the may hyper transitions are not present.

We define the concrete implementations of a 1MTS as labeled transition systems, defined below.

Definition 4 (LTS). An LTS is a tuple \( (S, A, \rightarrow, s_{init}) \), where \( S \) is a set of states, \( A \) is a set of actions, \( \rightarrow : S \times A \times \bar{S} \) is the transition relation, and \( s_{init} \) is the initial state. We denote the class of LTSs by \( LTS \). (We follow the definition given for LTSs as implementations of 1MTSs with single initial states in [12]).

As a simple example in Fig. 2(2), an LTS is shown which refines the 1MTS in Fig. 1 and 2(1). In this LTS, the may hyper transitions are not present and the alternative choice among the \( \text{insert\_euro} \) and \( \text{insert\_dollar} \) is resolved by choosing the former.

2.3 Product Line Process Algebras

Milner’s Calculus of Communicating Systems (CCS) [20] is extended by Gruler et al. [14] into PL-CCS by introducing a new operator, called binary variant, to represent the alternative behavior. The introduced binary variant operator \( \oplus_1 \) 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 \( s = a.(b.s + c.s) \) and \( t = a.(b.t \oplus_1 c.t) \); recursive process \( s \)
keeps making choices between \(b\) and \(c\) in each recursion, while process \(t\) makes a choice between \(b\) and \(c\) in the first recursion after performing \(a\), and the choice is recorded and respected in all the following iterations. This means that process \(t\) behaves deterministically after the first iteration with respect to the choice between \(b\) and \(c\). To simplify the formal development of the theory, Gruler et. al. assume that in every PL-CCS term, there is at most one appearance of the operator \(\Downarrow\), 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 PL-LTSs [14], using a structural operational semantics, which is explained informally next. The states of a product line labeled transition system are pairs of ordinary states, i.e., process terms, and \(configuration\) \(vectors\). The transitions of a PL-LTS are also labeled with configuration vectors. These vectors are of type \([L, R, ?]I\) with \(I\) being an index set, \(L\) and \(R\), respectively, denoting that the choice has been made in favor of the left- or right-hand-side term and \(?\) denoting that the choice has not been made yet.

**Definition 5 (PL-LTS).** 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 is a 5-tuple \((P \times [L, R, ?]I, A, I, \rightarrow, \text{post})\) consisting of a set of states \(P \times [L, R, ?]I\), a set of actions \(A\), and a transition relation \(\rightarrow \subseteq (P \times [L, R, ?]I) \times (A \times [L, R, ?]I) \times (P \times [L, R, ?]I)\), and an initial state \(\text{post} \in P \times [L, R, ?]I\), satisfying the following restrictions:

1. \(\forall_{P, a, Q, v, v'} (P, a) \xrightarrow{a, v} (Q, v') \quad \rightarrow \quad v' = v'\).
2. \(\forall_{P, a, Q, v, v'} (P, a) \xrightarrow{a, v} (Q, v') \wedge v(i) \neq ? \quad \rightarrow \quad v'(i) = v(i)\).
3. \(\forall_{P, a, Q, v, v', i, P_0, v_0} (P_0, v_0) \xrightarrow{a, v, i} (Q_0, v_0') \wedge (P_1, v_1) \xrightarrow{a, v', i} (Q_1, v_1') \wedge v_0(i) = v_1(i) \wedge v_0'(i) = v_1'(i) \quad \rightarrow \quad (P_h, v_h) = (P_l, v_l)\).

In Definition 5, the conditions follow from the operational rules given by Gruler et al. [14]. The first condition indicates that the change in the configuration is identically reflected in the label and the target. The second condition indicates that a decision made on a choice is recorded as \(L\) or \(R\) in the configuration vector and would not change in the future. The third condition reflects that the configuration at index \(i\) can be resolved in at most one state; this follows immediately from the uniqueness of indices in PL-CCS terms.

In order to define the valid implementations of a PL-LTS, we start with the following relation between the configuration vectors [9].

**Definition 6 (Configuration Ordering).** The ordering relation \(\sqsubseteq\) on the set \([L, R,?]I\) is defined as \(\sqsubseteq = \{(?, ?), (L, L), (R, R), (?, L), (?, R)\}\). We lift this ordering relation to the level of configurations by defining \(v \sqsubseteq v' \iff \forall_{i \in I} v(i) \sqsubseteq v'(i)\), for any \(v, v' \in [L, R,?]I\).

Considering the above definition, for each \(v, v' \in [R, L,?]I\), we say \(v(i) \triangleright v'(i) \iff v(j) \triangleright v'(j) \lor v(j) \sqsubseteq v(i)\), for each \(i, j \in I\). We lift this ordering relation
resented in Fig. 3(1). Then, consider the 1MTS shown in Fig. 3(2). Intuitively, next section, that is more suitable for the setting of software product lines. These issues lead us to design decisions for a new notion of refinement, introduced in the lead to design decisions for a new notion of refinement, introduced in the next section, that is more suitable for the setting of software product lines.

In this section, we study the refinement relation provided for 1MTSs by Fecher and Schmidt [12] (see Definition 3) and use some examples to point out a few issues in using this notion of refinement for product derivation. These issues lead us to design decisions for a new notion of refinement, introduced in the next section, that is more suitable for the setting of software product lines.

The first example concerns alternative behavior. Consider the PL-CCS terms $s_0 = a.s_1$ and $s_1 = b.s_2 \oplus c.s_3$. The corresponding underlying PL-LTS is represented in Fig. 3(1). Then, consider the 1MTS shown in Fig. 3(2). Intuitively,
Comparative Expressiveness of PL-CCS and 1-Selecting MTS

This model may be considered as a solution to represent the same set of products using 1MTSs: it bundles the choice between the b- and c-labeled transitions into a must hyper transition. (Recall from Definition 1 that must hyper transitions intuitively represent mandatory choices.) However, in Fig. 3(3), a valid implementation of this 1MTS based on the refinement relation in Definition 3 is depicted. (The dashed arrows show how the states of the LTS and 1MTS are related using the refinement relation.) In the LTS implementation, both the b- and c-labeled transitions are included. A 1MTS that has the same implementations as the PL-LTS in Fig. 3(1), is given in Fig. 3(4): namely, the choice has been lifted to the initial states. This way, the exclusive behavior can be separated among the two parts of the model initiated in these two states.

The process of lifting choices to the initial states can lead to an exponential blow up in 1MTS representation of product lines. This is already hinted at by the 1MTS given in Fig. 3(4) and can be generalized as follows. Consider the 1MTS shown in Fig. 3(5). This model is similar to the 1MTS given in Fig. 3(2) with \( k = n/2 \) independent exclusive choices (modeled by \( k \) must hyper transitions). There are \( 2^k \) possible combinations of all choices. This model suffers from the same problem as described above, namely, the alternative transitions can be included simultaneously in some LTS implementations. As mentioned above, in order to model alternative behavior the solution is to use a model with several initial states where each part of the model includes one of the possible combinations. Hence, the model should include \( 2^k \) separate parts each with a different initial state. This issue severely compromises succinctness in 1MTS representation of product lines.

Another issue in using 1MTSs for modeling product lines concerns persistent choices. Assume that we add the term \( s_3 = d.s_1 \) to the aforementioned PL-CCS process term. This will lead to having a new state in the PL-LTS \( (s_1, R) \) and a transition from \( (s_3, R) \) to this state. As mentioned in Section 2.3, the decisions made about the exclusive choices are stored in configuration vectors. Hence, when going back again to \( s_1 \), the choice that was made before, which is \( R \), will not change. Using the current notion of refinement for 1MTSs, it is not possible to keep track of the choices that are made in the past. Assume that we want to model the same behavior (as in Fig. 3(1)) using 1MTSs. Assume a transition from state \( s_3 \) to state \( s_1 \) with label \( d \) is added to the 1MTS represented in Fig. 3(2). One of the valid implementations of such 1MTS is an LTS where \( b \) is chosen the first time reaching state \( s_1 \) and then \( c \) is chosen the next time that this state is reached. The solution to solve this problem is the same as above (using several initial states) in addition to unrolling loops.

To address these 3 issues, namely, alternative behavior, succinct representation of choice, and persistence choice, we introduce a new notion of refinement for 1MTSs in the next section.


4 Revisiting the Refinement Relation

In this section, we propose a new refinement relation for 1MTSs to address the issues pointed out in the previous section regarding the original refinement relation [12]. Then, we show that our new refinement relation preserves the intuitive properties posed for the original one [12].

4.1 New Refinement Relation

We revisit the refinement relation in Definition 3, and provide a new refinement relation for 1MTSs as follows. First, we define an auxiliary function, namely, the choice resolution function.

Definition 8 (Choice Resolution Function). Consider a 1MTS $M=(S, L, \rightarrow, \cdot \rightarrow, s_{\text{init}})$. A choice resolution function is a total function $f : S \rightarrow \bigcup_{i \in \mathbb{N}} \text{choice}(s_i \rightarrow t_i)$.

We denote the set of all choice resolution functions of the 1MTS $M$ by $\mathcal{M}_M$.

The purpose of defining the choice resolution function is to assign a choice function to each state of the 1MTS once and for all. Next, we give the refinement relation for 1MTSs as follows.

Definition 9 (New Refinement for 1MTS). Consider two arbitrary 1MTSs $M=(S, L, \rightarrow, \cdot \rightarrow, s_{\text{init}})$ and $M'=(S', L, \rightarrow', \cdot \rightarrow', s'_{\text{init}})$, we say $M$ refines $M'$, denoted by $M \triangleright M'$, if there exists a refinement relation $\mathcal{R}_{1\text{MTS}} \subseteq S \times S' \times L \times L'$ such that if $f \in \mathcal{M}_M \exists f' \in \mathcal{M}_{M'} \forall s_0 \in s_{\text{init}} \exists s'_0 \in s'_{\text{init}} \forall (s_0, s'_0, f, f') \in \mathcal{R}_{1\text{MTS}}$ and $\forall (s, s', f, f') \in \mathcal{R}_{1\text{MTS}}$, the following conditions hold:

(i) $\forall \omega \in (s \rightarrow s') \exists s'' \in (s' \rightarrow s'') \exists a \in L, s'' \in S, s'' \in S'$ \hspace{1cm} $f(s)(\omega) = (a, s'')$

(ii) $\forall s'' \in (s' \rightarrow s'') \exists \omega \in (s \rightarrow s'') \exists a \in L, s'' \in S, s'' \in S'$ \hspace{1cm} $f(s)(\omega) = (a, s'')$

We use $\mathcal{R}_{1\text{MTS}}$ to denote a 1MTS refinement relation that follows the above definition (that uses choice resolution functions $f$ and $f'$). In Fig. 4(1), an example of a 1MTS is given. Based on the Definition 3, the 1MTS in Fig. 4(2) is refining this 1MTS. However, based on the Definition 9, this is not a valid refinement for the 1MTS in Fig. 4(1). Hence, the problem with modeling alternative behavior that was mentioned in Section 3 is solved in the new definition. Similarly the problems with modeling the conciseness and the persistent behavior are solved.

4.2 Refinement Relation Properties

We prove a set of properties for the new refinement relation as follows. This is the same set of properties proven for the original 1MTS refinement relation by Fecher and Schmidt in [12]. (Due to space limitation, the proofs are omitted and we will include them in an extended version of the paper.) First, we show that the new refinement relation is a preorder.
Comparative Expressiveness of PL-CCS and 1-Selecting MTS

Proposition 1. The refinement relation given in Definition 9, is a preorder.

Next, we show that all the LTS implementations of a 1MTS also implement the 1MTSs that are refined by this 1MTS.

Proposition 2. Consider two 1MTSs $M$ and $M'$ such that $M > M'$, Then $\forall lts \in LTS \cdot lts \triangleright M \Rightarrow lts \triangleright M'$.

Next, we prove that the bisimulation relation satisfies the properties of the refinement relation in Definition 9.

Proposition 3. Consider two arbitrary LTSs $lts_1$ and $lts_2$ such that $lts_1 \simeq lts_2$, where $\simeq$ denotes strong bisimilarity; it follows that $lts_1 \triangleright lts_2$.

5 Encoding PL-LTSs into 1MTSs

In order to compare the expressiveness of PL-LTSs with 1MTSs, following the approach provided by Beohar et al. in [9], we define an encoding from PL-LTSs into 1MTSs. The main idea of giving an encoding is to define a transformation from one class of models into the other class of models that is semantic preserving. First, we give the following auxiliary definitions taken from [9].

Definition 10 (Product Line Structure). A product line structure is a tuple $M = (M, []),$ where $M$ is the class of the intended product line models (in this paper 1MTSs and PL-LTSs) and $[] : M \rightarrow LTS$ is the semantic function mapping a product formalism to a set of product LTSs that can be derived from each product line model.

Next, we give the formal definition of an encoding.

Definition 11 (Encoding). An encoding from a product line structure $M = (M, [])$ into $M' = (M', []')$, is defined as a function $E : M \rightarrow M'$ satisfying the following correctness criterion: $[] = []' \circ E$. We say a product line structure $M'$ is at least as expressive as $M$ if and only if there exists an encoding $E : M \rightarrow M'$.

Before elaborating on the proposed encoding, we give two auxiliary definitions which are used for encoding the transitions of a PL-LTS into must/may hyper transitions of a 1MTS. As (hyper) transitions in a 1MTS are transitions with multiple targets (see Definition 1), we need to group some of the transitions in a
PL-LTS, which correspond to resolving the same alternative choice, and encode them as a (may/must) hyper transition. To this end, we consider the type of transitions that is made by a transition to the configuration vector of a PL-LTS. A transition for which the configuration vectors in the source and target states are not identical, is corresponding to resolving a choice (making a decision about one of the variant operators). We formally define the hyper must closed set and hyper may closed set as follows.

**Definition 12 (Hyper Must Closed Set).** Consider a state \((P, \nu)\) of a PL-LTS such as \((\mathbb{P}, \{L, R, ?\}, A, I, \rightarrow, p_{out})\); we assume that Out\((P, \nu)\) denotes the set of all outgoing transitions from state \((P, \nu)\) and Out\(_{(P, \nu)}^i\) denotes the set of outgoing transitions form \((P, \nu)\) that make a change in at least one of the elements of the configuration vector of the source state, i.e., for each \((P, \nu) \xrightarrow{a} (P', \nu') \in \text{Out}_{(P, \nu)}^i\), there exists an \(i \in I\) s.t. \(\nu(i) =? \land \nu'(i) \\

**Definition 13 (Hyper May Closed Set).** The hyper may closed set for a state \((P, \nu)\), denoted by \(T^i_{(P, \nu)}\), is defined the same as the hyper must closed set as given in Definition 12, with the only difference that the first condition is replaced with the following condition:

- For each \((P, \nu) \xrightarrow{a_1, a_2} (Q_0, \nu_0) \in T\), for some \(i \in I\) s.t. \(\nu(i) \neq \nu_0(i)\) there exists a \((P, \nu) \xrightarrow{a_1, a_2} (Q_1, \nu_1) \in T\) s.t. \(\neg(\nu_0(i) \lor \nu_1(i))\) and for all \(j \neq i\), \(\nu_0(j) \lor \nu_1(j)\).
- For each two different transitions \((P, \nu) \xrightarrow{a_1, a_2} (Q_0, \nu_0) \in T\) and \((P, \nu) \xrightarrow{a_1, a_2} (Q_1, \nu_1) \in T\), there exists \(i \in I\) s.t. \(\neg(\nu_0(i) \lor \nu_1(i))\).

We denote the set of all such maximal subsets for a state \((P, \nu)\), by \(S^i_{(P, \nu)}\).

**Definition 14 (PL-LTS to 1MTS Encoding).** Let \((\mathbb{P}, A, I, \rightarrow, p_{out})\) be a PL-LTS. We construct a 1MTS \(M = (S, A, \rightarrow, s_{\text{init}}, s_{\text{out}})\) as an encoding of such a PL-LTS as follows.

- The set \(S\) of states is defined as \(\mathbb{P}\), i.e., the set of states in the PL-LTS, \(p_{\text{out}} = s_{\text{out}}\). \(A\) is the same set of actions,
- We construct the \(\rightarrow\) and \(\rightarrow^*\), which, respectively, denote the must and may hyper transition relations for each state of the the encoding 1MTSs as follows. Given Definition 12 and Definition 13, we define the following transition rules:

\[
((P, \nu) \rightarrow) = \bigcup \limits_{(P', \nu') \in \mathbb{P}} \bigcup \limits_{a \in A} \{(a, (P, \nu))\} \cup \{(P, \nu) \xrightarrow{a} (P', \nu') \in A\} \cup
\]
Comparative Expressiveness of PL-CCS and 1-Selecting MTS

Given the above encoding, we prove that the class of 1MTSs is at least as expressive as the class of PL-LTSs. (Due to space limitation, the proofs are omitted and we will include them in an extended version of the paper.)

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

### 6 Related Work

In this section, we discuss related work regarding formalisms used for modeling product lines and the comparison of their expressiveness. We limit our consideration to the models which have LTSs as the semantic domain.

Considering the comparison of the expressiveness of the formalisms used for modeling variability, Beohar et al. in [9] provide a comparison between the expressiveness of three fundamental models, namely, MTSs, PL-CCSs, and Feature Transition Systems (FTSs). (FTSs [11] are extensions of LTSs with propositional formulas called feature expressions.) A novel notion of encoding, based on the set of implementing LTSs, from one class of models to the other is provided. The existence of mutual encodings between two classes of models is described as having the same expressiveness. As a result a hierarchy of formalisms based on their expressiveness is provided. Furthermore, Benduhn et al. in [7], provide a survey on formalisms focusing on the suitability of these models in applying different analysis techniques.

Considering the formalisms proposed for modeling product lines; In [13], Fischbein et al. for the first time argued that MTSs are adequate for modeling variability. In several works, MTSs have been used for modeling variability in the behavior of product lines [2, 1, 3, 19, 16]. As shown in [9], MTSs are the least expressive in the provided hierarchy. In order to tackle the limited expressiveness of MTSs, several extensions of such models have been proposed. In a set of works, MTSs are used with variability constraints [6], which are constraints expressed in Modal-Hennessy-Milner-Logic (MHML) [2, 1, 3]. In [17], an extension of MTSs, namely, Disjunctive Modal Transition Systems (DTMSs) are introduced which provides the possibility to model an or relation between choices in the behavior using hyper transitions. Fecher and Schmidt in [12], introduce 1MTSs, which (as mentioned in Section 2) can be used for modeling alternative choices. Furthermore, in this work, a comparison between the expressiveness of these two models is provided, which shows that the two classes of models have
the same expressiveness concerning the sets of implementing LTSs. Benes et al. in \cite{8}, introduce an extension of MTSs, namely, parametric modal transition systems in which the concept of obligation functions is used. The obligation functions are defined upon atomic propositions of states, the transitions, and a set of parameters, which can be used for representing features. By setting the valuation of parameters the presence or absence of states and transitions in a specific product model can be specified. Moreover, an extension of contract automata with modality \cite{5} is introduced by Basile et al. in \cite{4}. In this extension of the model, permitted and necessary requests are distinguished using feature constraints. There have been other approaches introduced that use some interface theories principles to indicate the set of derivable variants from an MTS as the ones that are compatible under parallel composition with regards to a given environmental specification \cite{16, 19}.

As mentioned in Section 2, PL-CCS \cite{14}, introduced by Gruler et al. \cite{14}, is an extension of Milner’s CCS \cite{20} by means of an alternative choice operator called “binary variant”. This operator provides the possibility of modeling persistent choices in the behavior. The validity of variants can be further restricted using the multi-valued modal \(\mu\)-calculus \cite{21}.

To the best of our knowledge, the provided encoding from PL-LTSs into 1MTSs, the results regarding the expressiveness, and the provided refinement relation for 1MTSs that addresses the limitations of such models in modeling variability in the behavior in this paper are novel.

7 Conclusion

In this paper, we compared the expressiveness of PL-LTSs and 1MTSs. To this end, we defined the set of products for specifications in both formalisms, of which the behaviors are commonly specified in the domain of LTSs. We then showed that 1MTSs can capture all products that can be specified by the product line calculus of communicating systems. Furthermore, we provided a set of observations regarding the limitations in modeling variability in the behavior which are enforced by the refinement relation given for 1MTSs. We proposed a new refinement relation for 1MTSs to tackle these limitations and proved a set of properties for the new refinement relation.

An immediate question to ask is whether the two formalism have the same expressive power or not. We conjecture that the answer is positive and leave this for immediate future work. We also would like to combine the results of this paper with our earlier results in \cite{9} and present a comprehensive lattice of expressive power among all fundamental behavioral models for software product lines. As another part of our future work, we plan to provide a stronger relation between PL-LTSs and PL-CCS terms by introducing a set of conditions (on the configuration vectors of states) in a PL-LTS which guarantee that the PL-LTS is induced from a PL-CCS term.
Comparative Expressiveness of PL-CCS and 1-Selecting MTS

References


Appendix G

Paper VII
Modal Transition System Encoding of Featured Transition Systems

Mahsa Varshosaz, Lars Luthmann, Paul Mohr, Malte Lochau, and Mohammad Reza Mousavi

Submitted to Journal of Logical and Algebraic Methods in Programming
Modal Transition System Encoding of Featured Transition Systems

Mahsa Varshosaz\textsuperscript{a}, Lars Luthmann\textsuperscript{b}, Paul Mohr\textsuperscript{b}, Malte Lochau\textsuperscript{b}, Mohammad Reza Mousavi\textsuperscript{c}

\textsuperscript{a}Halmstad University, Sweden
\textsuperscript{b}TU Darmstadt, Germany
\textsuperscript{c}University of Leicester, UK

Abstract

Featured transition systems (FTSs) and modal transition systems (MTSs) are two of the most prominent and well-studied formalisms for modeling and analyzing behavioral variability as apparent in software product line engineering. On one hand, it is known that for finite behavior FTSs are strictly more expressive than MTSs, essentially due to the inability of MTSs to express logically constrained behavioral variability such as exclusive behaviors. On the other hand, MTSs enjoy many desirable formal properties such as compositionality of semantic refinement and parallel composition. In order to finally consolidate the two formalisms for variability modeling, we establish a rigorous connection between FTSs and MTSs by means of an encoding of one FTS into an equivalent set of multiple MTSs. To this end, we split the structure of an FTS into several MTSs whenever it is necessary to denote exclusive choices that are not expressible in a single MTS. Moreover, extra care is taken when dealing with infinite behaviour: loops may have to be unrolled to accumulate FTS path constraints when encoding them into MTSs. We prove our encoding to be semantic-preserving (i.e., the resulting set of MTSs induces, up to bisimulation, the same set of derivable variants as their FTS counterpart) and to commute with modal refinement. We further give an algorithm to calculate a concise representation of a given FTS as a set of MTSs. Finally, we present experimental results gained from applying a tool implementation of our approach to a collection of case studies.

Keywords: Featured Transition Systems, Modal Transition Systems, Expressiveness Power, Product Lines, Modeling

Email addresses: mahsa.varshosaz@hh.se (Mahsa Varshosaz), lars.luthmann@es.tu-darmstadt.de (Lars Luthmann), malte.lochau@es.tu-darmstadt.de (Malte Lochau), mm789@le.ac.uk (Mohammad Reza Mousavi)
1. Introduction

Different formal models have been proposed to capture the behavior of software product lines (SPLs), e.g., for model-based testing or model checking. Examples of such formal models include featured transition systems (FTSs) [1], modal transition systems (MTSs) [2] and various extensions thereof [3, 4, 5, 6, 7, 8, 9, 10]. The expressive power of some of the aforementioned formalisms has been assessed in [11]. The comparison of the expressiveness is established based on proving the (non-)existence of an encoding, which is a transformation from one class of models to the other by preserving the set of derivable model variants in terms of implementing Labeled Transition Systems (LTSs). (The provided results cover models with infinite state or finite behavior.) As a part of the results, it is shown that FTSs with finite behavior are more expressive than plain MTSs with finite behavior (i.e., MTS without any additional constructs to express variability constraints), essentially because those plain MTSs cannot specify persistently exclusive behavior. However, the theory of MTSs has been extensively studied [10] and based on that, various tools have been developed to support their analysis [5, 12, 13, 14, 15]. In addition, MTSs enjoy many desirable formal properties such as inherent notions of semantic refinement being compatible with parallel composition, thus enabling compositional reasoning. Hence, it makes sense to further explore the connection between FTSs and MTSs and to come up with semantic-preserving encoding of FTSs into MTSs.

We address this problem by providing a translation of one FTS into a set of multiple MTSs. The MTSs considered in this work are congruent with the ones defined by Larsen et al. in [2], which extend LTSs by a may-/must-modality of single transitions. An alternative approach [16, 17, 18] is to encode FTSs into MTSs by annotating the target MTSs with variability constraints when needed. Our encoding only splits the structure of a given FTS into multiple mutually excluding MTSs when it is necessary, i.e., when there is an exclusive choice among transitions in the FTS that cannot be captured by one single MTS. We prove that our translation allows for step-wise refinement, i.e., it is consistent with the existing notions of refinement on MTSs and FTSs and it is semantic preserving, i.e., it induces, up to bisimulation, similar sets of products for the resulting set of MTSs as the original FTS. We also give an algorithm to calculate the translated MTSs and prove it correct with respect to our definition. A number of essential concerns are addressed in the definition of this encoding: firstly, the path constraints accumulated through different paths may turn out to be inconsistent with each other and hence, such paths have to be split into different MTSs. Moreover, paths are accumulated and potentially strengthened through loops and hence, loops may have to be unrolled to cater for this. We further consider the issue of minimality of FTS encodings into sets of MTSs and show that our proposed algorithm satisfies this notion for structurally deterministic FTSs.

In addition, we present experimental evaluation results gained from applying a tool implementation of our approach to a collection of case studies from recent literature on FTS [19] as well as from a collection of synthetically generated, yet realistic FTS models. The goal of this empirical study is to show scalability of our tool also to larger models and to investigate efficiency and effectiveness of generating a minimal MTS encoding from FTS input models. In particular, we investigate the computational effort...
for generating MTS models from a given FTS as well as the average number of MTS as compared to the overall number of variants.

The rest of this paper is organized as follows. In Section 2, we introduce a running example, namely the Arcade Game Maker product line [20, 21], that is used to illustrate the concepts and notions. In Section 3, we formally introduce the basic concepts regarding labeled transition systems (LTSs), MTSs, and FTSs. In Section 4, a valid encoding of FTSs into sets of MTSs is characterized in a declarative way and is shown to be semantic preserving. In Section 5, an algorithmic view of the encoding is provided and its correctness is proven. Moreover, a notion of minimality is proposed in the same section and the outcome of the algorithm is shown to satisfy this notion for structurally deterministic FTSs. In Section 6, we evaluate our algorithm on a number of (real-world as well as synthetic) case studies. In Section 7, an overview of the literature in this area is given and different pieces of the literature are related to the present work. Finally, in Section 8, the paper is concluded and some avenues for future research are discussed.

2. Running Example

In this section, we first introduce an illustrative example, a simple software product line of an Arcade Game Maker (AGM), which will be used throughout this paper. The AGM product line comprises two different games (rules), namely pong and bowling.

In addition, the AGM application enables the player to use different services such as pausing and exiting a running game as well as saving the recent game.

A common notation for a compact representation of the set of features and the relations between them are feature models, usually visualized in terms of feature diagrams [22]. A feature diagram for our AGM example is depicted in Figure 1. A feature diagram is a tree-like structure in which each node represents a feature. Each single feature is either mandatory, if it is included in all the products of the product line in which its parent feature is included, or it is optional, otherwise. In addition, a feature can have groups of sub-features of two different kinds. First, a set of sibling features can be in an or-relation, which means that at least one of the features has to be selected whenever the parent feature of that group is selected (not contained in our example). Second, a set of sibling features can be in an xor-relation which means that exactly one of the features has to be selected whenever the parent feature is selected (alternative group in our example). Finally, a feature may be in a require- or (mutual) exclusion-relation with another feature, represented by a (respectively, solid
and dashed) cross-tree edge. Concerning the feature model in Figure 1, the diagram contains compound features for the configuration of rules and services. Features pong and bowling have an xor-relationship, whereas the service features are all optional. In addition, feature pong requires feature pause, whereas feature bowling excludes feature pause. A valid configuration of a product line corresponds to a subset of features satisfying all the constraints of the feature model. For example, the AGM feature model in Figure 1 has 8 valid configurations.

3. Foundations

In this section, we explain the constructs and concepts used throughout this paper. In particular, we consider two existing formalisms for product-line modeling namely, MTSs and FTSs. Each abstract model belonging to one of these two classes of models comprises several concrete implementation variants in terms of labeled transition systems (LTSs). The notion of LTS is defined as follows (cf. [23]).

**Definition 1 (Labeled Transition System),** A labeled transition system is a tuple \((S, A, \rightarrow, s_{\text{init}})\), where:

- \(S\) is a finite set of states,
- \(A\) is a finite set of actions,
- \(\rightarrow \subseteq S \times A \times S\) is a (labeled) transition relation,
- \(s_{\text{init}} \in S\) is an initial state.

As an example, consider the LTS in Figure 2a depicting a configuration of the AGM. A similar, yet slightly different LTS model can be given for each of the 7 other
valid product configurations of the AMG product line, each sharing certain common behaviors and differing in variable behaviors.

An LTS-based formalism considered for expressing behavioral commonality and variability in a product line are modal transition systems (MTSs), as shown in Figure 2b for our AGM example. In an MTS model, the set of transitions is subdivided into subsets of must-transitions denoting mandatory (core) behavior to be included in every configuration, and may-transitions denoting optional behaviors. Note that every must-transition also requires an “underlying” may-transition as a must-transition always needs to be allowed as well. Hence, we call must-transitions with an underlying may-transition mandatory transitions, whereas may-transitions with no corresponding must-transition are called optional transitions. In figures, we use solid lines to denote mandatory transitions and dashed lines to denote optional transitions.

(The MTSs that we consider in this work are complying with the original definition given by Larsen et al. in [2]. Different extensions of MTSs have been provided e.g. MTSs with variability constraints [17] and parametric MTSs [24] that provide means to explicitly relate features to the behavior similar to FTS. However, these extensions of MTSs are out of the scope of this preliminary work of MTS encodings of FTSs.)

An MTS may be formally defined as follows (cf. [2]).

Definition 2 (Modal Transition System). A modal transition system is a 5-tuple \( (S, A, \rightarrow_{\rightarrow}, \rightarrow_{\exists}, s_{init}) \) where:

- \( S \) is a finite set of states,
- \( A \) is a finite set of actions,
- \( \rightarrow_{\rightarrow} \subseteq S \times A \times S \) is a may-transition relation,
- \( \rightarrow_{\exists} \subseteq S \times A \times S \) is a must-transition relation,
- \( s_{init} \in S \) is an initial state.

Hence, those transitions being contained in the set of may-transitions but not in the set of must-transitions express optional (or variable) behaviors of an SPL. Hence, an MTS integrates a set of LTSs which can be obtained via modal refinement (i.e., every optional transition either becomes a mandatory transition, or it is removed from the model). In this regard, an LTS may be considered as an MTS in which \( \rightarrow_{\rightarrow}=\rightarrow_{\exists} \) holds.

In order to formally define the set of valid implementations of an MTS, we employ the (modal) refinement relation for MTS, based on Larsen et al. [2], as follows.

Definition 3. Consider two MTSs, \( mts_0 = (S, A, \rightarrow_{\rightarrow}, \rightarrow_{\exists}, s_{init}) \) and \( mts = (T, A, \rightarrow_{\rightarrow}, \rightarrow_{\exists}, t_{init}) \). A binary relation \( R \subseteq S \times T \) is a modal refinement relation if and only if the following properties are satisfied.

1. \( \forall t, t_0 \in T, s, a \in A \) \( sRt \land t \rightarrow_{\rightarrow} t_0 \) \( \implies \exists s' \in S \) \( s \rightarrow_{\exists} s' \land s'Rt_0 \), and
2. \( \forall s, s_0 \in S, t, a \in A \) \( sRt \land s \rightarrow_{\rightarrow} s' \) \( \implies \exists t' \in T \) \( t \rightarrow_{\rightarrow} t' \land s'Rt' \).
The modal specification $\text{mts}'$ refines the modal specification $\text{mts}$, denoted $\text{mts}' \preceq \text{mts}$, if there exists a modal refinement relation $\mathcal{R}$ such that $s_{\text{init}} \mathcal{R} s_{\text{init}}$. We denote all the MTSs that refine the MTS $\text{M}$ by $[\text{M}]$.

For each of the 8 valid product configurations of the AGM product line, a corresponding LTS model can be derived from the MTS example in Figure 2b via modal refinement. For instance, the LTS in Figure 2a depicts a product where all optional transitions related to bowling become mandatory and all other optional transitions are removed. However, the converse statement does not hold as, in addition to those 8 valid LTS variants, further LTS variants may be derived from the MTS in Figure 2b that do not correspond to any valid configuration of the AGM product line. For instance, both the behaviors for feature $p$ and feature $b$ may be either preserved or removed under modal refinement which clearly contradicts the exclusive-or dependency among $p$ and $b$ as stated in the feature model in Figure 1. This example illustrates the inherent inability of MTS to express persistently exclusive choices among variable behaviors.

Another LTS-based formalism for expressing behavioral commonality and variability in a product line are featured transition systems (FTSs), as shown in Figure 3 for our AGM example. Similar to an LTS or MTS, an FTS consists of a set of states and a set of transitions, labeled with actions. In addition to actions, transitions of an FTS are further labeled with presence conditions over (Boolean) feature variables. The presence conditions determine those product configurations in which the transition in hand is included. In this way, an FTS incorporates an explicit notion of behavioral variability by virtually integrating a set of similar, yet well-distinguished LTS models into one product-line model.

The transition labels in the FTS in Figure 3 for the AGM product line are of the form "presence condition / action". In particular, the atomic proposition in the presence conditions refer to the (abbreviated) feature names in the feature model in Figure 1. If residing in initial state $\text{init}$, the AGM either enters a new game $\text{bowling}$, or a new game $\text{pong}$, respectively, whenever action $\text{start}$ occurs. If the user triggers action $\text{pause}$, both types of games may be suspended by entering a $\text{pause}$ state. In this particular example,
a pong game may be re-entered again via action `start`, which is, however, not supported in case of a bowling game. Instead, a bowling game may be `saved` during the game, whereas a pong game has to be `paused` before it can be `saved`. In addition, both kinds of games may be stopped by the `exit` action which leads the FTS back to the `init` state. For instance, choosing features `¬u, e, s, ¬p, b` results in the LTS being depicted in Figure 2a.

As described above, feature diagrams are frequently used for representing the set of features of a product line and the relations between them. Alternatively, the configuration constraints as (graphically) imposed in a feature diagram may be also represented as propositional formula over features, represented as Boolean variables. By \( B(F) \), we denote the set of propositional formulae over a set \( F \) of (Boolean) feature variables. We now give the formal definition of FTS based on [1], as follows.

**Definition 4 (Featured Transition System).** A featured transition system is a 6-tuple \((S, A, F, !, \leftrightarrow, p_{\text{init}})\), where

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

In order to define the set of valid implementations of an FTS, we first give the following auxiliary definition.

**Definition 5.** Considering a set of feature variables \( F \) and a set of product configurations \( \Lambda \); for a propositional formula \( e \in B(F) \), we say \( \text{Sat}(e) \), iff \( \exists \lambda : F \rightarrow B \) such that \( \lambda \land e \) is satisfiable.

Next, we define a product derivation relation \([11]\), that is used for extracting the set of valid implementations (or, LTS variants) of an FTS, as follows.

**Definition 6.** Given an FTS \( \text{fts} = (P, A, F, !, \Lambda, p_{\text{init}}) \), and LTS \( l = (S, A, \rightarrow, p_{\text{init}}) \), and a product \( \lambda \in \Lambda \). A binary relations \( R_{\lambda} \subseteq P \times S \) (parameterized by product configurations) are called product-derivation relations if and only if the following transfer properties are satisfied.

1. \( \forall P, Q, a, s, t \ (P \ R_{\lambda} s \land P \overset{\phi/\lambda}{\rightarrow} Q \land \lambda \models \phi \Rightarrow \exists t : s \overset{a}{\rightarrow} t \land Q \ R_{\lambda} t) \)
2. \( \forall P, Q, a, s, t \ (P \ R_{\lambda} s \land s \overset{a}{\rightarrow} t) \Rightarrow \exists Q, \phi : P \overset{\phi/\lambda}{\rightarrow} Q \land \lambda \models \phi \land Q \ R_{\lambda} t \)

A state \( s \in S \) derives the product \( \lambda \) from an FTS-specification \( P \in P \), denoted by \( P \vdash_{\lambda} s \), if there exists a product derivation relation \( R_{\lambda} \) such that \( P \ R_{\lambda} s \).
We say that \( l \) is a valid implementation of \( \text{fts} \), denoted by \( \text{fts} \models l \) if and only if there exists a product configuration \( \lambda \in \Lambda \) such that \( p_{\text{init}} \models_{\lambda} s_{\text{init}} \). We denote all LTSs being derivable from the FTS \( \text{fts} \) by \( [\text{fts}] \).

Please note that Classen et al. in [1] provide a different “projection” operator for deriving the individual product models from an FTS. Based on their definition, an FTS is projected onto a product configuration, and as the result of projection, those transitions of the FTS for which the corresponding feature expression satisfies the product configuration are included in the product model whereas the other transitions are eliminated. This definition provides a syntactical description for deriving product models while the product-derivation relation given in Definition 6, constitutes a semantical notion of product-model derivation similar to modal refinement of MTS. The sets of LTSs derived from an FTS using either of these definitions are equal modulo bisimilarity (see Theorem 4 and its proof in Appendix B.). In this work, we use Definition 6 due to its declarative nature; for example, it allows for implementations that reduce the number of states while constructing the LTSs. It is also more suitable for providing the foundation for our encoding of FTSs into MTSs and more specifically for constructing the proofs to show that the encoding is semantic preserving.

For each of the 8 valid product configurations of the AMG product line, a corresponding LTS model can be derived from the FTS model by deleting those transitions whose presence conditions are not satisfied by the corresponding product configuration (and by omitting the presence conditions of the remaining transitions).

It has been proven in recent literature [11], that FTSs with finite behaviour are strictly more expressive than MTSs with finite behaviour. More precisely, the comparison of the expressive power is based on the (non-)existence of a one-to-one encoding from one class of models into the other. In particular, such an encoding should define a translation of one model into another model having equal (modulo bisimilarity) sets of implementing LTSs. As illustrated by our example, there exist FTSs for which no single MTS can induce the same set of LTS models as valid product-line configurations [11]. However, if we consider multiple MTS models to characterize sets of valid LTSs corresponding to an FTS, then every FTS is expressible in terms of (a set of) MTSs. A general result about this relationship will be constructively proven in the remainder of this paper.

As an alternative line of work, we refer to the extension of MTSs with feature constraints [16, 18]; in this line of work, the authors provide a translation from FTSs to MTSs by annotating the target MTSs with feature constraints when necessary.

4. From FTS to MTSs

The goal of this section is to define a semantic preserving translation, called an encoding. We first set the scene by motivating the basic concepts used in our encoding from an FTS to a set of MTSs. Subsequently, we define our encoding and prove its correctness.

4.1. Encoding Concepts

As stated before, MTSs are inherently incapable of capturing mutually exclusive behavior that is naturally expressible in FTSs. To illustrate this in terms of a minimal
example, consider the FTS in Figure 4a; there is an excludes dependency between the two transitions emanating from initial state $s_0$. Assume towards a contradiction that an MTS could model the behavior of the same product line. Then, the initial state of the purported MTS must include both an $a$-labeled and a $b$-labeled outgoing may-transition (otherwise, it fails to produce one of the LTSs, either having an outgoing $a$- or an outgoing $b$-labeled transition). However, in such a case, there is an LTS product of the purported MTS that has both outgoing $a$- and $b$-labeled transitions from the initial state, which is not a valid product of the FTS.

Hence, whenever the presence conditions of the emanating may- or must-transitions are not consistent in the FTS (i.e., there are dependencies such as excludes or requires relations between presence conditions of transitions in the FTS), we have to split the MTS structure into maximal subsets of transitions without such conflicts, thus leading to a set of MTSs for a given FTS.

For instance, for the FTS in Figure 4a, this leads to the 2 MTSs as depicted in Figure 5a.

However, consistency of transitions in one MTS is not only dependent on their presence conditions as stated in the FTS, but also on the path conditions accumulated from the presence conditions of other transitions traversed on the different possible paths reaching the source state of the transition under consideration.

For example, consider the FTS in Figure 4b: state $s_1$ has 3 outgoing transitions of which the presence conditions appear to be inconsistent at first sight. However, state $s_1$ is only reachable from the initial state through the $a$-labeled transition having presence condition $f_2$. Hence, when arriving at state $s_1$, the path condition $f_2$ must hold and hence, the outgoing $d$-labeled transition is not present. The remaining two outgoing transitions are mutually dependent such that all derivable LTSs containing the $b$-labeled
transition with presence condition $f_1 \land f_2$ must also include the $c$-labeled transition. Conversely, if the $c$-labeled transition is present in an LTS, then both features $f_1$ and $f_2$ are present in the product configuration and hence the $b$-labeled transition must also be included in the LTS. Hence, we require two MTSs to interpret the respective FTS, which are shown in Figure 5b: one representing the behavior of products that include feature $f_1$ and the other representing the behavior of the remaining products.

To generalize, not only consistency of the path conditions of transitions leaving the same state must hold as illustrated in the previous example, but rather consistency of path conditions of all transitions in one MTS must hold. Figure 4c represents an FTS in which there is a configuration dependency between the $c$-labeled and $d$-labeled transitions. This dependency is similar to the one between the transitions in Figure 4b, but the concerned transitions are now located on different paths. Hence, we again require two MTSs to interpret the FTS, as shown in Figure 5c.

Finally, special care is required for handling loops in the state-transition graph of FTS in the respective MTS encodings. Here, loops may have to be unrolled to a certain depth in order to correctly encode the accumulated path constraints for the transitions involved. For instance, consider the FTS depicted in Figure 4d: for reaching state $s_2$, the path condition $f_1 \land f_2$ must hold. When going from here back to state $s_1$, the path constraint is stricter now than when we visited $s_1$ for reaching $s_2$ for the first
time. This is reflected in the MTSs depicted in Figure 5d, where the loop has to be unrolled once to distinguish the different path conditions. The intuition behind this is that if the optional transition labeled with $d$ is included in a variant derived from the MTS on the left via refinement, then this action must always be enabled, again, afterwards whenever reaching some state related to FTS state $s_2$ in the same variant (as enforced by the presence condition in corresponding FTS). This can be encoded into MTS only by unrolling the respective transition loop such that the first occurrence of an action is attached to an optional transition, whereas all subsequent occurrence(s) are attached to mandatory transitions. Figures 4e, 5e, and 6 provide another example, where we assume feature $f$ to be optional. Hence, the FTS in Figure 4e has exactly two variants: one without any actions and a second one in which action $a$ may be performed arbitrarily often. These are also exactly the variants derivable from MTS $mts_c$ (cf. Figure 5e). Additionally, at a first glance, $mts_a$ as depicted in Figure 6a seems to be a smaller MTS (in the number of states and transition), yet having the same variants. However, in contrast to $fts$ and $mts_c$, due to modal refinement, MTS $mts_i$ further comprises an infinite number of different variants, each permitting action $a$ to be performed at most $k$ times, with $k \in \mathbb{N}$, which is clearly not permitted by $fts$.

Given these basic cases, we now formally characterize MTS encoding of FTSs in a declarative way. We therefore introduce the notion of context of an FTS to contain those valid sets of MTSs having the same (union of) sets of LTS implementations as the given FTS. Please note that the context of an FTS is not necessarily unique, i.e., there may be multiple valid sets of MTSs which represent the same behavior as an FTS.

To define the notion of context for FTS, we first need to specify the set of valid products that a set of MTSs can specify. The states of MTS in the context of an FTS consist of pairs of states of the respective FTS together with a propositional formula (up to logical equivalence) denoting the path condition for reaching this state in the FTS. Based on this additional information, we are able to define the set of product configurations corresponding to the set of products implemented by an MTS by means of the set of FTS implementations implying the resulting propositional formula. The overall propositional formula for the whole MTS with respect to the given FTS is constructed using a recursive function defined using a fixed point construction, named context, as follows.

Starting from the initial state of the MTSs at hand, we assume that the set of outgoing transitions from the corresponding state; in the FTS, is partitioned into three different sets of transitions, namely must-, may- and excluded-transitions. Consider
ering must-transitions (i.e., transitions being present in all valid implementations of the FTS), we build the conjunction of the path condition of the current state and the resulting propositional formula for the target state of the transition (being computed by recursively applying the context-function to that state). Since must transitions are present in all considered valid products, the product configurations corresponding to implemented products imply this conjunction. Instead, may-transitions (i.e., transitions present in some but not all valid implementations of the FTS), are represented by disjunction of the negated path condition of the current state and the resulting propositional formula for the target state as described for the must-case. Finally, considering excluded-transitions (i.e., transitions inconsistent with the included transitions of the FTS), we build the conjunction of the negation of the presence conditions.

The MTS constraint given as the conjunction of the formulas constructed for all three sets therefore characterizes the set of LTS subsumed by the current MTS such that all product configurations implying this constraint correspond to products of the FTS implemented by this MTS.

**Definition 7 (MTS Constraint).** Consider an FTS \( fts = (P, A, F, !, \Lambda, p_{init}) \), and an MTS \( mts = (Q, A, !, \Lambda, q_{init}) \), where \( Q = P \times B(F) \). We define the corresponding MTS constraint as follows. We first define for each state \((p, e) \in Q\) the following notations:

- \( \text{exc}((p, e)) = \{(p, a, p', e') \in ! \mid \exists (p, e) \in Q \text{ Sat}(e \land f) \land ((p, e), a, (p', e') \land f) \notin q_{init}\} \),
- \( \text{must}((p, e)) = \{(p, a, f, p', e') \in ! \mid \exists (p, e), a, (p, e \land f) \in q_{init}\} \),
- \( \text{may}((p, e)) = \{(p, a, f, p', e') \in ! \mid \exists ((p, e), a, (p, e \land f)) \in q_{init}\} \).

By \( \text{const}(q_{init}) \) we denote the MTS constraint for \( mts \), where for each \((p, e) \in Q\), \( \text{const}((p, e)) \), is the maximal fixed point (w.r.t. logical implication ordering) for the following function:

\[
\text{const}_i((p, e)) = e \land \bigwedge_{(p, a, f, p') \in \text{must}((p, e))} \left( \text{const}_{i-1}((p', e \land f)) \right) \land \bigwedge_{(p, a, f, p') \in \text{may}((p, e))} (-f \lor (\text{const}_{i-1}((p', e \land f)))) \land \bigwedge_{(p, a, f, p') \in \text{exc}((p, e))} -f
\]

where \( Y_{(p, e) \in Q} \text{ const}((p, e)) = e \). Furthermore, we say \( \Lambda_{mts} \) denotes the set of product configurations corresponding to the products implementing \( mts \), which is \( \Lambda_{mts} = \{\lambda \in \Lambda \mid \lambda \Rightarrow \text{const}(q_{init})\} \).

As a property of the function \( \text{const}() \), based on the following lemma we prove that this function is monotone and hence always has a fixed point.
Lemma 1. Considering the definition of the function const(), given in Definition 7, this function always has a maximal fixed point.

Proof. The proof is included in the appendix.

Next, we give the definition of a consistent MTS with respect to a given FTS.

Definition 8 (Consistent MTS). Given an FTS \( \text{fts} = (\mathcal{P}, A, F, \rightarrow, \lambda, q_{\text{init}}) \), an MTS \( \text{mts} = (\mathcal{Q}, A, \rightarrow, \lambda, q_{\text{init}}) \) is a consistent MTS with respect to \( \text{fts} \), iff the following properties hold:

1. \( \mathcal{Q} \subseteq \mathcal{P} \times \mathcal{B}(\mathcal{F}) \) is a set of states s.t.:
   \[
   \forall p' \in \mathcal{P}, e' \in \mathcal{B}(\mathcal{F}) \ (p', e') \in \mathcal{Q} \iff \exists (p, f, p') \in \mathcal{Q} \exists (p, f, p') \in \mathcal{E} \ (e' = e \land f).
   \]
   Here, we only consider the set of states that are reachable from the initial state.

2. \( A \) is a set of actions.

3. \( q_{\text{init}} = (p_{\text{init}}, W) \in \mathcal{Q} \) is the initial state.

4. \( \rightarrow, \lambda \) are maximal sets satisfying the following properties.

   (a) \( \forall ((p, e), l, (p', e')) \in \rightarrow, \exists (p, f, p') \in \mathcal{E} \ (e' = e \land f) \)

   (b) \( \forall (p, e) \in \mathcal{P}, \forall (p', e') \in \mathcal{B}(\mathcal{F}) \forall \lambda \in \lambda_{\text{max}} \ (\lambda \models e \implies \lambda \models f) \iff (p, e) \xrightarrow{\lambda} (p', e') \)

   (c) Considering any subset of may-transitions \( T \) such that \( \rightarrow, \lambda \subseteq \rightarrow, \lambda \), it holds that:

   \[
   \exists \lambda \in \Lambda_{\text{max}} : \lambda \models \bigwedge_{((p, e), l, (p', e')) \in \mathcal{T}} f \land \neg \left( \bigwedge_{((p, e), l, (p', e')) \in \not\mathcal{T}} g \right)
   \]

   Furthermore, for each \( \lambda \in \Lambda_{\text{max}} \), a set of transitions \( T \neq \emptyset \) with the property stated above exists.

Given the definition of a consistent MTS with regard to an FTS, we define the set of conditions that a set of MTSs must satisfy in order to be a valid part of the FTS context of a given FTS.

Definition 9 (FTS Context). Given an FTS \( \text{fts} = (\mathcal{P}, A, F, \rightarrow, \lambda, q_{\text{init}}) \), a set of MTSs \( \mathcal{M} = \bigcup_{\text{mts} \in \mathcal{M}} \text{mts} \), where \( \text{mts} = (\mathcal{Q}, A, \rightarrow, \lambda, q_{\text{init}}) \) is in the context of \( \text{fts} \), denoted by \( \mathcal{M} \in \text{context}(\text{fts}) \) iff all the MTSs in \( \mathcal{M} \) are consistent according to Definition 8, and the following conditions hold.

1. \( \Lambda = \bigcup_{\text{mts} \in \mathcal{M}} \Lambda_{\text{max}} \)
2. \( \forall q, f, q', e, e' \in Q \)\( (p, (q, f), (q', e')) \in E_1 \cup E_2 \Rightarrow e' = e \land f \)

In the above definition, the first condition indicates that the union of all products implementing at least one MTS in the considered set of MTSs must be equal to the set of products of the FTS. The second condition indicates that each transition in the FTS must be included in at least one MTS in the set of MTSs.

As an example, consider the MTSs \( mts \) and \( mts' \), respectively, on the left- and right-side in Figure 5d. First, \( \lambda_{mts} \) is computed as described above. Assume the set of states in these MTSs are \( Q = \{ q_0 = (s_0, 1); q_1 = (s_1, f_1), q_2 = (s_2, f_1 \land f_2), q_3 = (s_1, f_1 \land f_2 \land f_3) \} \). Then, in general we have:

\[
\text{const}_1(q_0) = \top \land (\neg (f_1) \lor (\text{const}_{i-1}((s_1, f_1)))
\]

Then, we calculate \( \text{const}_{i-1}((s_1, f_1)) \):

\[
\text{const}_{i-1}((s_1, f_1)) = f_1 \land (\text{const}_{i-2}((s_2, f_1 \land f_2)) \land (\neg f_1 \land \neg f_2))
\]

Next, we compute \( \text{const}_{i-2}((s_2, f_1 \land f_2)) \):

\[
\text{const}_{i-2}((s_2, f_1 \land f_2)) = f_1 \land f_2 \land (\neg f_1 \lor \text{const}_{i-3}((s_1, f_1 \land f_2 \land f_3)))
\]

Then, \( \text{const}_{i-3}((s_1, f_1 \land f_2 \land f_3)) \) is computed:

\[
\text{const}_{i-3}((s_1, f_1 \land f_2 \land f_3)) = (f_1 \land f_2 \land f_3) \land (\text{const}_{i-4}((s_2, f_1 \land f_2 \land f_3))) \land \neg (f_1 \land \neg f_2)
\]

In the next step, \( \text{const}_{i-4}((s_2, f_1 \land f_2 \land f_3)) \) is computed as:

\[
\text{const}_{i-4}((s_2, f_1 \land f_2 \land f_3)) = (f_1 \land f_2 \land f_3) \land (\text{const}_{i-5}((s_1, f_1 \land f_2 \land f_3)))
\]

Hence, considering the calculations, in the first step we have:

\[
\begin{align*}
\text{const}_0(q_0) &= \top, \quad \text{const}_0(q_1) = f_1, \quad \text{const}_0(q_2) = f_1 \land f_2 \\
\text{const}_0(q_3) &= f_1 \land f_2 \land f_3, \quad \text{const}_0(q_4) = f_1 \land f_2 \land f_3
\end{align*}
\]

We include a part of the next iterations that are relevant to obtaining the final results:

\[
\begin{align*}
\text{const}_1(q_1) &= f_1 \land f_2 \land f_3 \land (\text{const}_0(q_1)) = f_1 \land f_2 \land f_3 = \text{const}_2(q_4) \\
\text{const}_1(q_2) &= f_1 \land f_2 \land f_3 \land (\text{const}_0(q_2)) = f_1 \land f_2 \land f_3 = \text{const}_2(q_3) \\
\text{const}_2(q_2) &= f_1 \land f_2 \land (\neg (f_3) \lor \text{const}_1(q_1)) = (f_1 \lor \neg f_3) \land (f_2 \lor \neg f_3) \land (f_1 \land f_2) = \text{const}_3(q_2) \\
\text{const}_3(q_4) &= f_1 \land f_2 \land (\neg f_1 \land \neg f_2) = f_1 \land (f_1 \lor \neg f_3) \land (f_2 \lor \neg f_3) \land (f_1 \land f_2) \land (\neg f_1 \lor \neg f_2) = \text{const}_4(q_4) \\
\text{const}_4(q_0) &= (f_1 \lor \neg f_1) = f_1 \lor f_2 = \text{const}_5(q_0)
\end{align*}
\]
Considering the fixed points in the above computations, it holds
\[
\Lambda_{\text{mts}} = \{ \neg f_1 \land f_2 \land f_3, \neg f_1 \land \neg f_2 \land f_3, \neg f_1 \land f_2 \land \neg f_3 \},
\]
By performing similar computations for \( \text{mts}' \), we can conclude from Definition 8 that \( \{ \text{mts}, \text{mts}' \} \) is in the context of the FTS in Figure 4d. As another example, consider Figure 7 to represent the MTSs in the context of the FTS in Figure 3. Note that we, again, have to perform loop unrolling here in order to obtain the correct MTS encoding as described before.

We next prove that, given a set of MTSs in the context of a given FTS according to Definition 9, the union of the sets of products implemented by the MTSs in this set is equal (up to bisimulation) to the set of products implemented by the FTS. In both cases, the product implementations are represented as LTSs. To this end, we first define the (set of) configuration vector(s) corresponding to an LTS implementing an MTS that belongs to a set of MTSs in the context of the given FTS. This definition is then used in the proof of Theorem 1. Intuitively, the construction of this set follows the same idea as the one given in Definition 7.

**Definition 10.** Given an MTS \( \text{mts} = (Q, A \rightarrow, \neg \sigma, q_{\text{init}}) \) from a set of MTSs in the context of a given FTS \( \text{fts} = (P, A, F, !, \nabla, p_{\text{init}}) \). For each LTS \( \text{lts} = (S, A \rightarrow, \sigma, s_{\text{init}}) \) being a valid implementation of \( \text{mts} \), the (set of) corresponding configuration vector(s)
is defined as follows. Considering \( \text{lts} \), there exists a class of relations, denoted by \( R \subseteq S \times Q \), where each \( R^i \in R \) is a refinement relation (cf. Definition 3) that relates states of \( \text{lts} \) to states of \( \text{mts} \). We first define the following auxiliary sets:

\[
\text{impMust}^i = \{(p, e) \stackrel{\text{!}}{\Rightarrow} (p', e') \mid \exists s, s' \in S, a \in A \cdot (s, a, s') \in \rightarrow \land s R^i(p, e) \land s' R^i(p', e')\}
\]

\[
\text{impMay}^i = \{(p, e) \stackrel{\text{!}}{\Rightarrow} (p', e') \mid \exists s, s' \in S, a \in A \cdot (s, a, s') \in \rightarrow \land s R^i(p, e) \land s' R^i(p', e')\}
\]

\[
\text{impExc}^i = \{(p, f, a, p') \mid \exists s, s' \in S, a \in A \cdot (s, f, a, s') \in \rightarrow \land \neg f R^i(p', e')\} \cup \{(p, e) \in Q \cdot \text{Sat}(e \land f) \land ((p, e), (p', e \land f)) \notin \rightarrow_q \}
\]

Given a refinement relation \( R^i \in R \), the (set of) configuration vector(s) corresponding to \( \text{lts} \) is defined as:

\[
\text{conf}^i(\text{lts}) = \left \{ \begin{array}{l}
\text{impMust}^i
\end{array} \right \}
\]

where \( \text{conf}^i(\text{lts}) \) is defined as:

\[
\text{conf}^i(\text{lts}) = \bigwedge_{(p, e) \stackrel{\text{!}}{\Rightarrow} (p', e') \in \text{impMust}^i} e' \land \bigwedge_{(p, e) \stackrel{\text{!}}{\Rightarrow} (p', e') \in \text{impMay}^i} e' \land \bigwedge_{(p, f, a, p') \in \text{impExc}^i} \neg f
\]

Based on this definition, we are now able to prove the correctness of our encoding from FTSs into MTSs.

In particular, we can reduce this problem to a mutual comparison of the sets of LTSs corresponding to product implementations derivable from both representations.

**Theorem 1.** Given an FTS \( \text{fts} \), the set of LTSs implementing \( \text{fts} \) is equal to the union of sets of LTSs implementing each sets of MTSs being the context of \( \text{fts} \), i.e.

\[
\forall_{\text{MF} \subseteq \text{context}(\text{fts})} [\text{fts}] = \bigcup_{\text{mts} \in \text{MF}} [\text{mts}].
\]

**Proof.** We divide the proof into two following obligations, one for each direction. First, we prove that \([\text{fts}] \subseteq \bigcup_{\text{mts} \in \text{MF}} [\text{mts}]\) holds. Given \( \text{fts} = (P, A, F, \rightarrow, \Lambda, p_{init}) \), the set of MTSs \( \text{MF} \in \text{context}(\text{fts}) \) and an LTS \( \text{mts} = (S, A, \rightarrow, s_{init}, s_{init}) \), s.t. \( \text{mts} \in [\text{fts}] \). We prove \([\text{fts}] \subseteq \bigcup_{\text{mts} \in \text{MF}} [\text{mts}]\) by showing that

\[
\exists_{\text{mts} \in \text{MF}} lts \in [\text{mts}] .
\]

To prove \( lts \in [\text{mts}] \), and assuming that \( \text{mts} = (Q, A, \rightarrow_q, q_{init}, q_{init}) \), based on Definition 3, it suffices to show that a refinement relation such as \( R^i_{\text{mts}} \subseteq Q \times S \) exists such that
Thus, property (3) holds. Second, we prove it holds that based on the condition given in property (ii) and the definition of transition correspond to a transition (on the right hand side of the above statement with Definition 9, there exists (i), (ii), and (iii) exists for some \( \lambda \in \Lambda_{mts} \), where \( \lambda \) is a product configuration for which there exists \( R_\lambda \subseteq P \times S \) that satisfies:

1. \((p_{init}, s_{init}) \in R_{mts}\)

2. \(\forall p, q \in Q, s' \in S, s \in S : (q R_{mts} s \land s \xrightarrow{a} s') \implies \exists p' \in Q : q \xrightarrow{p' a} p' \land q R_{mts} s'\)

3. \(\forall p, q \in Q, s' \in S, s \in S : (q R_{mts} s \land q \xrightarrow{f, a} q') \implies \exists s'' : s \xrightarrow{f} s'' \land q' R_{mts} s''\)

We choose \( mts = (Q, A, \rightarrow_\lambda, \rightarrow_\lambda, q_{init}) \) from \( M \) such that \( \lambda \in \Lambda_{mts} \), where \( \lambda \) is a product configuration for which there exists \( R_\lambda \subseteq P \times S \) that satisfies:

1. \((p_{init}, s_{init}) \in R_\lambda\)

2. \(\forall p, q \in P, A, s' \in S, f \in E : (p R_\lambda s \land p \xrightarrow{f, a} p' \land \lambda = f) \implies \exists q' \in Q : s \xrightarrow{\lambda} s' \land q' = \lambda\)

3. \(\forall p, q \in P, A, s' \in S, f \in E : (p R_\lambda s \land s \xrightarrow{f, a} s') \implies \exists p' \in P, A, f \in E : p \xrightarrow{f} p' \land \lambda = f\)

Given \( lts \in \llbracket mts \rrbracket \) based on Definition 6 as defined above, a \( R_\lambda \) with properties (i), (ii), and (iii) exists for some \( \lambda \in \Lambda \). Furthermore, based on the first condition in Definition 9, there exists \( mts \in M \) such that \( \lambda \in \Lambda_{mts} \).

Given \( R_\lambda \) with the above properties, we define a relation \( R_{mts} \), as follows:

\[
\forall (p, e) \in Q, e \in E : (p R_{mts} s \Leftrightarrow p R_\lambda s \land \lambda = e) \quad (2)
\]

Next, we prove that \( R_{mts} \) satisfies property (1). As \( \lambda = \bigvee_{\lambda : \lambda \in \Lambda} \lambda \) and \( p_{init} R_{mts} s_{init} \), based on the definition of \( R_{mts} \), it holds that \( q_{init} R_{mts} mts \). Hence, property (1) holds.

We further prove that \( R_{mts} \) satisfies property (2). Consider an arbitrary pair of states \((p, e), s) \in R_{mts} \). Based on property (iii) (ii) that holds that

\[
(p R_\lambda s \land s \xrightarrow{\lambda} s') \implies \exists p' \in P, f \in E : p \xrightarrow{f} p' \land \lambda = f \land p' R_\lambda s'.
\]

From \( \lambda \in \Lambda_{mts} \) and from Definition 7 we conclude that each transition \( p \xrightarrow{f} p' \) on the right hand side of the above statement with \( \lambda = f \) is translated into a may transition \( mts \) emanating state \((p, e)\) (cf. Lemma 2). Hence, property (2) holds.

Next, we prove that \( R_{mts} \) satisfies property (3). Consider an arbitrary pair of states \((p, e), s) \in R_{mts} \). A must-transition \((p, e) \xrightarrow{\lambda} (p', e')\). Based on Definition 8, this transition correspond to a transition \( p \xrightarrow{f, a} p' \) in the \( lts \) such that \( e' = e \land f \). Hence, from the respective definition of \( const \), it follows that \( \lambda = e \) and \( \lambda = f \). In addition, based on the condition given in property (ii) and the definition of \( R_{mts} \), in Equation (2), it holds that

\[
\exists p' \in P, f \in E : s \xrightarrow{\lambda} s' \land (p', e') \in R_{mts} s'.
\]

Thus, property (3) holds. Second, we prove \( \bigcup_{mts \in M} \llbracket mts \rrbracket \subseteq \llbracket lts \rrbracket \). Given \( lts = (P, A, F, \rightarrow_\lambda, \mu, p_{init}) \) and the set of MTSs \( M \in context(lts) \), consider \( mts = (Q, A, \rightarrow_\lambda, \mu, q_{init}) \) such that \( mts \in M \), and an LTS \( lts = (S, A, \rightarrow, s_{init}) \) such that \( lts \in \llbracket mts \rrbracket \). In order to prove \( \bigcup_{mts \in M} \llbracket mts \rrbracket \subseteq \llbracket lts \rrbracket \) it suffices to show that

\[
lts \in \llbracket lts \rrbracket \quad (3)
\]
holds. Given the above definitions for \( lts \) and \( lts \) and the definition of product derivation for FTSs (cf. Definition 6), it is sufficient to show that for a product configuration \( \lambda \in \Lambda \) a relation \( R_{\lambda} \subseteq P \times S \) exists such that the following holds:

1. \((p_{\text{init}}, s_{\text{init}}) \in R_{\lambda} \).

2. \( \forall p, p' \in P, A, s, e : S, e : E(f) \ (p R_{\lambda} s \land p \xrightarrow{e/s} p' \land \lambda \models \varphi) \Rightarrow \exists p', s' \in S : s' = p' R_{\lambda} s \).

3. \( \forall p, p' \in P, A, s, e : S, e : E(f) \ (p R_{\lambda} s \land s \xrightarrow{e} s') \Rightarrow \exists p', s'' : s' = p' R_{\lambda} s'' \).

Given \( lts \in [mts] \) and, based on Definition 10, there exists a set of refinement relations such as \( R = \bigcup_{\lambda \in \Lambda} R_{\lambda}^{mts} \), where for each relation \( R_{\lambda}^{mts} \), the following properties hold (see Definition 3).

For \( \lambda \in \Lambda_{mts} \) and relation \( R_{\lambda}^{mts} \in R \), we define \( R_{\lambda}^{lts} \) as follows.

\[
R_{\lambda}^{lts} = \{ (p, e) \in Q \times R_{\lambda}^{mts} | (p, e) \land (\lambda \models \text{conf}^f(lts)) \land \lambda \models e \} \tag{4}
\]

First, we prove that \( R_{\lambda}^{lts} \) satisfies property (1). Based on Definition 8, we have \( q_{\text{init}} = (p_{\text{init}}, \bigvee_{\lambda \in \Lambda} \lambda) \) and thus \( \lambda \models \bigvee_{\lambda \in \Lambda} \lambda \). Given property (i) and the definition of \( R_{\lambda}^{lts} \) given in Equation (4), it holds that \( R_{\lambda}^{lts} \) satisfies property (1).

Next, we prove that \( R_{\lambda}^{lts} \) satisfies property (2). Consider a pair of states \((p, e) \in R_{\lambda}^{lts} \) and a transition \( p \xrightarrow{lts} p' \), where \( \lambda \models f \). We can assume the following three cases.

- The transition is included as must-transition in \( mts \). Based on property (ii), for all such transitions there exists \( s \xrightarrow{e} s' \) such that \( s' R_{\lambda}^{mts} s \). According to the definition of \( R_{\lambda}^{lts} \) given in Equation (4), we have that \( \lambda \models e \) and as \( \lambda \models f \) holds, it further holds that \( \lambda \models e \land f \). Hence, considering \( \lambda \in \Lambda_{mts} \), due to Equation (4) it holds that \( s' R_{\lambda}^{lts} p' \).

- The transition is included as a may-transition in \( mts \). Hence, we have \( (p, e) \xrightarrow{\lambda} (p', e \land f) \). Given \( lts \in [mts] \), due to property (iii) it holds that \( s \xrightarrow{e} s' \) such that \( s' R_{\lambda}^{mts} (p', e \land f) \). Otherwise, according to Definition 10, \( \neg f \) would be part of the conjunction included in the construction of \( \text{conf}^f(lts) \), which results in \( \lambda \not\models f \). According to the definition of \( R_{\lambda}^{lts} \) given in Equation (4), it holds that \( \lambda \not\models e \) and, based on the assumption \( \lambda \models f \), it follows that \( \lambda \not\models e \land f \). Hence, considering \( \lambda \in \Lambda_{mts} \), due to Equation (4) it holds that \( s' R_{\lambda}^{lts} p' \).

- The transition excluded from \( mts \). This case is not valid according to Lemma 2.
Based on the three above cases, we conclude that $R_i$ satisfies property (2). Finally, we prove that $R_i$ satisfies property (3). We consider a pair of states $(s, p) \in R_i$ and a transition $s \xrightarrow{(p, e)} s'$. Based on property (iii), there exists a transition $(p, e) \xrightarrow{\text{mts}} (p', e')$ in the mts such that $s' \xrightarrow{\text{mts}} (p', e')$ holds. Based on the Definition 8, each transition $(p, e) \xrightarrow{\text{mts}} (p', e')$ results from encoding a transition $p \xrightarrow{\text{f/a}} p'$. Based on Equation (4), it holds that $\lambda \models f$. Considering that $\lambda \models e$ holds, it can be concluded from Equation (4) that $s' \xrightarrow{\text{mts}} (p', e')$ holds. Hence, $R_i$ also satisfies property (3).

5. Generating Minimal MTS Encodings of FTS

The MTS encoding of FTSs as defined in the previous section always permits, as a trivial solution, to simply interpret a given FTS as a set of LTSs (i.e., MTS with $\xrightarrow{\text{mts}} = \xrightarrow{\text{f/a}}$) corresponding to the set of valid implementations of the FTS. This solution may be considered as the maximal encoding.

In this section, we provide an constructive algorithm for computing the declarative definition of context (Definition 9) and prove its correctness. Furthermore, we show that using this algorithm, we not only generate valid, but also minimal (i.e., most succinct [11]) MTS encoding of a given FTS.

5.1. Generation of MTS Encodings

An operational characterization of generating an MTS encoding from a given FTS as defined in a declarative manner in the previous section is given in Algorithm 1. The algorithm receives as input an FTS $\text{fts}$ and returns as output a set $M$ of MTS being an MTS encoding of $\text{fts}$. We describe the two procedures $\text{MAIN}$ (lines 1–7) and $\text{NEWMTS}$ (lines 8–45) in more detail in the following.

Procedure $\text{MAIN}$. The procedure $\text{MAIN}$ repeatedly calls procedure $\text{NEWMTS}$ for generating further MTSs to be added to the result set $M$ until every valid implementation of $\text{fts}$ is finally covered by some LTS variant of at least one MTS in the set $M$. First, result set $M$ is initialized as empty set (line 2). Next, a presence condition $m \in B(F)$ is introduced (line 3). This so-called blocking clause is used throughout the algorithm to represent the set of configurations which are not yet covered by some MTS within the current result set $M$ (i.e., initially all configurations $\lambda \in \Lambda$ of $\text{fts}$, cf. line 3). The main loop then adds further MTS into result set $M$ until the set of not-yet-covered configurations is empty (i.e., the blocking clause becomes unsatisfiable, cf. line 4). To this end, procedure $\text{NEWMTS}$ is invoked with the current blocking clause $m$ and returns a further MTS $\text{mts}$ to be added to $M$ (line 6) together with a feature expression $\text{blockingClause}$ defining the set of additional configurations covered by the new MTS (line 5). Hence, the blocking clause $m$ is updated by conjunction of the negated $\text{blockingClause}$ expression before starting the next iteration.
Algorithm 1 MTS Generation

Input: \( M := (\mathcal{P}, \mathcal{A}, F, \rightarrow, A, p_{\text{init}}) \)

Output: \( M' := (\mathcal{P}', \mathcal{A}', F', \rightarrow', A', p_{\text{init}}) \)

1. procedure MAIN
2. \( M := \emptyset \)
3. \( m := \emptyset \cup \lambda \)
4. while \( m \neq \emptyset \) do
5. \( \text{(mts, blockingClause)} := \text{NEW MTS}(m) \)
6. \( M := M \cup \{ \text{mts} \} \)
7. \( m := m \land \neg \text{blockingClause} \)

8. procedure \( \text{NEW MTS}(\text{featureExpression } m) \)
9. \( \beta_{\text{mns}} := (\mathcal{P}, \mathcal{A}, F, \rightarrow, \lambda, p_{\text{init}}), \text{ where } \rightarrow_{\text{mns}} := \emptyset, \Lambda_{\text{mns}} := \{ \lambda \in \Lambda \mid \lambda \rightarrow_{\text{mns}} m \} \) and \( p_{\text{init}} := p_{\text{init}} \)
10. \( \text{mts} := (\mathcal{P} \times \mathcal{B}(\mathcal{F}), \Lambda, b, \{ p_{\text{init}} \}) \)
11. \( m := m \cup \{ \text{mts} \} \)
12. while \( \text{WS} \neq \emptyset \) do
13. \( q := (p, e) \in \text{WS} \)
14. \( \text{WS} := \text{WS} \setminus \{ q \} \)
15. \( \text{DS} := \text{DS} \cup \{ (p, e) \} \)
16. for each \( (p, a, f, p') \in \rightarrow \) do
17. \( \rightarrow_{\text{mns}} := \rightarrow_{\text{mns}} \cup \{ (p, f, a, p') \} \)
18. \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
19. if \( \{ (p', e), (p, e', f) \} \in \rightarrow_{\text{mns}} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
20. if \( \{ (p, e), (p, e', f) \} \in \rightarrow_{\text{mns}} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
21. if \( \{ (p, e), (p, e', f) \} \in \rightarrow_{\text{mns}} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
22. else
23. \( T := \text{UPDATE T OPOREL}(T, \text{mts}) \)
24. \( \text{UT} := \text{UPDATE ENROLLED OPTIONAL LOOPS}(\text{UT}, \text{mts}) \)
25. if \( \{ (p, e), (p, e', f) \} \in \rightarrow_{\text{mns}} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
26. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
27. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
28. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
29. for each \( a \in \rightarrow \) do
30. if \( \{ (p, e), (p, e', f) \} \in \rightarrow_{\text{mns}} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
31. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
32. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
33. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
34. if \( \{ (p, e), (p, e', f) \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
35. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
36. break
37. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \}
38. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
39. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \)
40. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \)
41. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \)
42. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
43. if \( \{ (p, e), (p', e') \} \in \mathcal{U} \) then \( m := m \land \neg \{ (p, e, a, (p', e') \} \)
44. \( \text{WS} := \text{WS} \cup \{ (p', e') \} \)
45. return \( \text{NEW MTS}(m \land \neg \{ (p, e, a, (p', e') \} \}) \)
Procedure NewMTS. The procedure NewMTS constructs the next MTS $\text{mts}_m$ from FTS $\text{fts}$ with respect to the current blocking clause $m$ by starting with an empty $\text{fts}_m$ and by incrementally traversing (and potentially adding) all transitions of $\text{fts}$ into $\text{fts}_m$ being reachable from the initial state. Each traversed transition either becomes a may-transition, a must-transition or an excluded transition in $\text{mts}_m$, depending on the presence conditions of the previously added transitions. To this end, a call to the helper-function $\text{const}$ returns the MTS constraint denoting the maximal fixed point (cf. Definition 7) with respect to the current (intermediate) models $\text{mts}_m$ and $\text{fts}_m$.

First, $\text{fts}_m$ is initialized without any transitions and the set of configurations being restricted by the blocking clause $m$ (line 9). Similarly, $\text{mts}_m$ is also initialized with no transitions (line 10). Furthermore, additional temporary data structures are initialized, namely a set $\neg \gamma$ to store those transitions from $\text{fts}$ being excluded from $\text{mts}_m$, and a set $\Delta$ (done-set) to store those states of $\text{mts}_m$ already visited during the traversal (line 11).

Moreover, we utilize the relation $T \subseteq \gamma \times \neg \gamma$ containing those pairs of transitions of an MTS being in a topological order (i.e., $(t, t_0) \in T$ either iff transition $t$ always precedes transition $t_0$ on every path leading from the initial state to the first occurrence of $t_0$, or if $t = t_0$ holds. The worst-case complexity of computing $T$ is quadratic in the number of transitions. In addition, we further initialize a set $U$ (done-set) for memorizing those transitions from the FTS being added as unrolled (optional) transition into the MTS under construction as described in Section 4. This set is used to check whether such previously performed unrollings may become obsolete in subsequent steps of the MTS construction due to dependencies among presence conditions of FTS transitions involved (see below for details).

In addition, the set $W$ (working-set) is used to store those (still-to-be-processed) states of $\text{mts}_m$, that are directly reachable via previously added states in $\text{mts}_m$, either by optional or mandatory transitions. This set initially contains the initial state of $\text{fts}$, being restricted by $m$ (line 12).

The main iteration (line 13) then proceeds as long as the working-set $W$ contains further states, by picking-and-removing an arbitrary next state $q = (p, e)$ from $W$ (line 14) and by adding component $q$ (i.e., the respective state in $\text{fts}$) into the done-set $\Delta$.

Next, we iterate over the set of all outgoing transitions of state $q$ in $\text{fts}$ (line 16) and add them to $\text{fts}_m$ (line 17). We first try to add those transitions as a may-transitions into $\text{mts}_m$ (line 18). Here, $(p', e \land f)$ denotes an MTS state in which the presence condition $e \land f$ holds (i.e., if there already exists a state $(p', e')$ in the MTS with $e'$ being equivalent to $e \land f$, the target state of the newly added transition is that existing state). In the next step, we check whether adding the currently considered transition of the FTS would lead to an inconsistent MTS model (line 19). This is done by checking compatibility of the conjunction of all presence conditions of transitions from $\text{fts}$ already added as may-transitions into $\text{mts}_m$ in previous steps including the current one. In case of non-satisfiability, the transition is instead added to the exclude-set (line 20) and removed from the set of may-transitions of $\text{mts}_m$ (but it remains in the set of transitions of $\text{fts}_m$, in order to mark it as already processed and explicitly excluded). In case the condition in line 19 is not satisfied (i.e., the new transition can be added to $\text{mts}_m$), we incrementally update relation $T$ (line 23) and set $U$ (line 24) of $\text{mts}_m$ by taking the newly added transition into account.

Furthermore, we have to prevent unnecessary loop unrollings which may occur in
two different possible ways throughout the construction steps performed by the algorithm up to this point.

The first case occurs if an FTS transition added as mandatory transition into the MTS in a previous step (due to dependencies of its presence conditions to those other transitions) is additionally added as unrolled transition in a subsequent step. For instance, this unrolling may happen if the previously added mandatory transition is followed by an optional transition being located within the same loop (including that transition itself). As adding the optional transition may result in an updated path condition not being equivalent to the path condition holding at the beginning of the loop, the loop will be unrolled in the MTS. However, to handle those cases, we have to distinguish between transitions being (correctly) mandatory solely due to their presence condition and transitions (incorrectly) becoming mandatory due to dependencies between their presence conditions to those of other mandatory transitions thus potentially resulting in unnecessary unrollings as described above. In order to avoid the latter case, we check for each newly added transition in the MTS if the corresponding FTS transition has already been inserted into the MTS in a previous step (as a mandatory transition) thus encountering a case of loop unrolling (cf. line 25). If this is the case, we restart the current call of procedure newMTS (cf. line 26) with an adapted initial condition such that the (falsely) unrolled transition immediately becomes mandatory. To avoid infinite recursion in case of correctly unrolled mandatory transitions, we further have to check if its presence condition is already implied by the initial condition before restarting (cf. second part of line 25).

The second case occurs if an FTS transition added as unrolled optional transition into the MTS in a previous step later becomes mandatory due to dependencies of its presence conditions to those of other transitions added in subsequent steps. Hence, whenever a transition becomes mandatory (lines 29, 34 and 40), we have to check whether this transition is part of an unrolled loop in the MTS (lines 30, 35 and 41). If this is the case, we also restart newMTS (lines 31, 36 and 42), again, by additionally conjuncting the presence condition of the respective transition to the initial condition (thus making the transition to an a-priori mandatory transition). Hence, the loop now only consists of mandatory transitions and is therefore not unrolled anymore as the path condition holding after the loop is equivalent to the path condition already holding at the beginning of the loop. Figure 8 provides an example for the necessity of this restart (where all features are assumed optional). Here, state s0 of the FTS depicted in Figure 8a has a self-loop transition which will be unrolled as feature f is optional. As a result, all following transitions will be duplicated, too. For instance, mtsi (cf. Figure 8b) illustrates an intermediate result where the transition labeled with a is unrolled such that the transition labeled with b is duplicated and therefore becomes mandatory. Note that the first transition labeled a is optional such that whenever a is included in an MTS variant, it may be performed arbitrarily often afterward (as induced by the FTS). When proceeding the constructions in Algorithm 1, we will reach the transition labeled with d at some point. As this transition has the same presence condition as the transition labeled with a, both transitions will then become mandatory as there are only variants permitted having either both a and d included or none of both. Hence, the previously unrolled optional transition labeled with a becomes mandatory and we restart newMTS with the refined feature condition m = \top \land \neg f. As a result, the loop of the
initial state is not unrolled anymore (as $\top \land f$ is obviously equivalent to $\top \land \neg f$) and therefore no transitions will be duplicated. As a consequence, the transition labeled with $b$ (amongst others) also remains optional. In this way, we only restart \textsc{newMTS} if we encounter cases of (unnecessarily) unrolled transitions (e.g., the transition with label $a$), but not in case of (necessarily) duplicated transitions due to (necessary) unrollings.

Furthermore, we check whether the presence condition of the newly added transition is implied by either the presence condition of some other FTS transition already added to $\text{mts}_n$ or by the current MTS constraint of $\text{mts}_n$ (line 28). If one of both cases holds, then the newly added transition has to become mandatory (line 29). Figure 9 provides an example for this issue (note that Example 1 on page 24 provides a full description of applying Algorithm 1 to this example FTS). While generating the MTS in Figure 9b, we have an intermediate step where the transitions labeled with actions $a$ and $c$ are optional and the transition labeled with action $b$ is mandatory. As a consequence, the transition labeled with action $c$ has to become mandatory, too, as the presence condition of the respective FTS transition labeled with action $c$ (cf. Figure 9a) is implied by the presence condition of the FTS transition labeled with action $b$. Hence, every variant containing the transition labeled with action $b$ must also contain the transition labeled with action $c$.

In addition, we have to check whether an optional transition has to become mandatory if this is implied by a particular combination of other optional transitions (lines 32 to 37). For this, we consider each subset of optional transitions (line 32, where $\mathcal{P}$ denotes power set) in ascending order, starting with the smallest sets. First, we conjugate the negated presence conditions of all transitions being in $s$, and then conjugate the
presence conditions of all optional transitions not being in $s$ (line 33) and check the resulting formula for satisfiability. If this check fails, then the corresponding combination of optional transitions is not permitted to not include those being in $s$ in a valid variant while including those being in $s$. We then pick one element from $s$ to become mandatory (line 34) and are then able to immediately terminate the check due to the ascending traversal (line 37).

Figure 10 (with feature model $m = f_1 \lor f_2 \lor f_3$) provides an example for the checks in lines 32 to 37. Here, a variant only containing a transition labeled with action $c$ is the only variant not permitted by the FTS. Without the check in lines 32 to 37, Algorithm 1 would produce $mts_1$ with all transitions being optional. However, by adding this check, subset $s$ containing the transitions labeled with actions $a$ and $b$ will yield the unsatisfiable formula $\neg f_1 \land \neg f_2 \land \neg f_3 \land (f_1 \lor f_2 \lor f_3)$. As a result, we will pick either the transitions labeled with $a$ or with $b$ and make it mandatory thus leading to a correct solution. The same holds for $mts_2$ which is generated next. Without lines 32 to 37, both transition would be optional, thus again allowing an (invalid) variant only containing a transition labeled with action $c$.

In the next step, we have to check whether the update of $mts_m$ (i.e., either adding a transition from $fts_m$ as optional or mandatory transition or excluding it from $mts_m$) potentially causes existing optional transitions in $mts_m$ to become mandatory. For this, we have to check for each optional transition if the presence condition of the corresponding transition from $fts_m$ is implied by (combinations of) presence conditions of other transitions added to $mts_m$ (which is, again, done by invoking $const_i$ on the current model in line 39). In this case, the set of must-transitions of $mts_m$ is updated, accordingly (line 40). However, for obtaining the minimal solution, this only holds for transitions not topologically preceding the transition under consideration. In those cases, reachability of the current transition already depends on the presence/absence of the topologically preceding transition. Figure 9 gives an example for this exception. Here, the transition labeled $a$ of the MTS in Figure 9b will not become mandatory although the respective presence condition of the FTS (cf. Figure 9a) is implied by the presence condition of $b$.

Finally, we have to check whether the target state of the currently processed transition is already subsumed by some state in the done-set $DS$ (line 43). Otherwise, we add the state into the working-set $WS$ (line 44).

Example 1. Figure 9 provides an example for a complete application of Algorithm 1.
Here, Figure 9a depicts the FTS and Figures 9b and 9c show the resulting set of MTS. After initialization, it holds that \( m = \top \) as both \( f_1 \) and \( f_2 \) are optional features being independent from each other; thus invoking \( \text{NEWMTS}(\top) \). This procedure starts with \( q = (s_0, \top) \), as initial state in the working set \( WS \). State \( s_0 \) has one outgoing transition in \( \text{fts} \) labeled \( a \), which is therefore added to \( mts_1 \) as a \( \text{may-transition} \) as well as to \( \text{fts}_m \) (lines 16 to 18), whereas the set of excluded transitions of \( mts_1 \) remains empty (lines 19 to 21). As this newly added \( \text{may-transition} \) does not (yet) depend on any other transition in \( mts_1 \) or on blocking clause \( \varphi \), it does not become mandatory (lines 39 and 40). Next, the target state \( (s_1, f_2) \) of the transition is added to the working set \( WS \) (line 43 and 44). As state \( q = (s_0, \top) \) does not have further outgoing transitions (line 16) and \( WS \) is not empty (line 13), the next iteration of the while-loop, starts, for instance, by picking the transition labeled with \( b \) (line 16). Here, lines 17 to 21 yield similar results as before. Additionally, the while-loop in line 39 does not add a new \( \text{must-transition} \) although the transition labeled with \( b \) (line 16). Here, lines 17 to 21 yield similar results as before. As state \( q = (s_0, \top) \) does not have further outgoing transitions (line 16) and \( WS \) is not empty (line 13), the next iteration of the while-loop, starts, for instance, by picking the transition labeled with \( b \) (line 16). Here, lines 17 to 21 yield similar results as before.

Additionally, the while-loop in line 39 does not add a new \( \text{must-transition} \) although the presence condition \( f_2 \) of the previously added transition labeled \( a \) is implied by the presence condition \( f_1 \land f_2 \) of the newly added transition labeled \( b \). This is due to the transitions of \( a \) and \( b \) being in the topological relation, i.e., every path leading to \( b \) also visits \( a \). Therefore, \( a \) may remain optional. When the transition labeled \( c \) is added to \( mts_1 \), it becomes mandatory as the presence condition of \( b \) implies the presence condition of \( c \) (lines 28 to 29). Furthermore, \( b \) becomes mandatory as there is no variant with \( b \) but without \( a \) (lines 32 to 37). A variant with \( a \) and \( c \) (as \( c \) is mandatory) has features \( f_1 \) and \( f_2 \), selected, and hence, \( c \) must be included in this variant as well. As a result, the new fixed point is now given as \( \text{const}(s_0, \top) = \neg f_2 \lor (f_1 \land f_2) \). In contrast, the transition labeled \( d \) has to be excluded from \( mts_1 \), as its presence condition is not compatible with those of the transitions labeled \( a \) and \( b \) (line 19). The new fixed point is therefore given as

\[
\text{const}(s_0, \top) = (\neg f_2 \lor (f_1 \land f_2)) \land \neg (f_1 \land \neg f_2)
\]

which is equivalent to \( \neg f_2 \lor (f_1 \land f_2) \). This leads to termination of \( \text{NEWMTS} \) with \( mts_1 \) and the respective fixed point was returned (line 45). The next invocation \( \text{NEWMTS}(\top \land \neg (f_2 \lor (f_1 \land f_2))) \) being equivalent to \( \neg f_1 \land \neg f_2 \) then returns \( mts_2 \) as shown in Figure 9d. This causes the updated blocking clause \( \varphi \) to become unsatisfiable and procedure \( \text{MAIN} \) to terminate with \( \mathcal{M} = \{ mts_1, mts_2 \} \).

We now prove correctness of Algorithm 1 with respect to the definition of MTS encoding of FTS (cf. Definition 8).

**Theorem 2.** Let \( \mathcal{M} \) be the MTS encoding of an FTS \( \text{fts} \) as generated by Algorithm 1. Then it holds that \( \mathcal{M} \in \text{context}(\text{fts}) \).

**Proof.** We prove Theorem 2 by showing that (1) \( \forall mts \in \mathcal{M} : (\forall t\text{ts} : mts \geq \text{fts} \lor \text{lt}s) \) and (2) \( \forall t\text{ts} : \text{fts} \geq \text{lt}s : (\exists mts \in \mathcal{M} : \text{lt}s \leq mts) \).

1. Proof by induction. Initially, \( mts_m \) contains no transitions. In each iteration of procedure \( \text{NEWMTS} \), the sets \( \rightarrow_o, \rightarrow_c \) and \( \rightarrow_e \) correspond to \( \text{may}, \text{must} \) and \( \text{exc} \) of Definition 7 for the current models \( mts_m \) and \( \text{fts}_m \). All those sets are initially empty. In every iteration, procedure \( \text{NEWMTS} \) checks for every transition of \( \text{fts} \) if its presence condition is compatible with the current model.
to avoid LTS variants with mutual excluding combinations of transitions. Additionally, adding transitions of \( mts \) as new may-transitions into \( mts \) results in an increased number of derivable LTS variants. Therefore, we have to ensure that (initially) optional transitions become mandatory if there are new variants derivable \( mts \) which are not included in \( fts \). In particular, we have to consider two cases where a newly added optional transition \( t \) causes \( mts \) to have more variants than \( fts \).

(a) The presence condition of \( t \) in \( fts \) is implied by aggregated model condition resulting from other transitions processed (either added as optional/mandatory or excluded) in previous iterations of constructing \( mts \). Hence, such logical dependencies between presence conditions of transitions of \( fts \) must be reflected by setting \( t \) mandatory in \( mts \). As those cases might arise whenever a transition modality of the current \( mts \) is adapted (namely before and after the second case), we have to perform a corresponding check twice (see lines 26–29 as well as lines 39–40).

(b) After adding \( t \) as optional transition into \( mts \), the set of all optional transitions in \( mts \) including \( t \) added in previous iterations yields variants which are not included in \( fts \) (lines 32–33). Hence, \( t \) has to be set to mandatory to exclude those variants from \( mts \).

(c) The presence of \( t \) may be implied by the feature expression describing the variants of the current MTS, by the presence condition of another transition, or by the combination of presence conditions of other transitions. Hence, \( t \) has to become mandatory in \( mts \) to ensure the restrictions on LTS variants as imposed in \( fts \).

Procedure \text{NewMTS} terminates after having processed every transition of \( fts \) this way. Hence, it holds that \( fts \in \text{context}(fts) \) and thus \( \forall \text{lts} \subseteq mts : \text{lts} \triangleright lts \).

2. Proof by induction. Initially, \( M \) contains no transitions MTS. In each iteration of procedure \text{MAIN}, the blocking clause \( m \) specifies exactly those configurations of \( fts \) not yet being covered by some LTS variant of an MTS in set \( M \). The blocking clause is initialized with the set of all valid configurations of \( fts \) and is refined after every invocation (including recursive restarts) of \text{NewMTS} by excluding those configuration being covered by the newly generated \( mts \) (line 7). Hence, procedure \text{MAIN} terminates only after having covered every LTS variant of \( fts \) by at least one MTS in \( M \) and thus \( \forall \text{lts} : \text{fts} \triangleright \text{lts} : (\exists mts \in M : \text{lts} \subseteq mts) \). \( \square \)

We next explore the notion of \textit{minimality} of MTS encodings of FTS in more detail and investigate whether Algorithm 1 is able to generate a minimal MTS encoding.

5.2. Minimality of MTS Encodings

As already mentioned before, there always exists a trivial encoding \( M \in \text{context}(fts) \) in which every MTS \( mts \in M \) constitutes an LTS such that \( |M| = |\lambda| \). In some cases, however, this \textit{maximal} solution is also the only valid solution (e.g., example in Figure 4a). Conversely, we intuitively expect a \textit{minimal} solution \( M \in \text{context}(fts) \) to
consist of a minimum number of MTSs. For instance, assume the transitions of the FTS in Figure 4a to be annotated with two different features $f$ and $f'$, both being optional and independent. A minimal solution would consist of one MTS having both transitions as optional transitions. Now, assume both transitions to be annotated with the same mandatory feature $f$. Then, a minimal solution would, again, consist of one MTS having both transitions, but now as mandatory transitions (i.e., being an LTS).

In general, it is not obvious how to characterize an MTS encoding as minimal. Intuitively, we require for an MTS encoding to be minimal that each MTS $M \in \text{context}(fts)$ contains as many optional transitions as possible as every optional transition doubles the number of FTS implementations subsumed by a single MTS (i.e., an MTS $m \in M$ with $k$ may-transitions subsumes $2^k$ LTS variants). However, simply counting the number of optional transitions may be misleading as the set of LTS variants derivable from two different MTS may be overlapping or even be similar. For instance, when removing the optional transitions from the MTS in Figure 5d both result in the same LTS. Additionally, we should require each pair of MTS of an MTS encoding to not contain any mutually bisimilar variants.

Another, more technical, issue arises from the possible unrolling of loops: even if the number of MTS in an MTS interpretation $M \in \text{context}(fts)$ is minimal, the number of transitions in MTS $mts \in M$ may be arbitrarily increased as compared to the FTS due to (redundant, yet valid) unrollings of loops. The FTS in Figure 4d and the corresponding MTS in Figure 5d provide an example. Here, the FTS contains three states and several loops between the states $s_1$ and $s_2$, whereas the corresponding MTSs both contain five states due to the adaptions of the path conditions throughout the construction steps performed by the algorithm as described above.

To summarize, minimality of MTS encodings may be characterized by lifting the notion of modal refinement to sets of MTS as follows.

**Definition 11 (Minimal MTS Encoding).** Let $M, M'$ be sets of MTS. By $M' \subseteq M$ we denote that

$$\forall mts' \in M' : \exists mts \in M : mts' \preceq mts$$

holds. An MTS encoding $M \in \text{context}(fts)$ is minimal for FTS $fts$ iff it is a greatest element of set $\text{context}(fts)$ with respect to $\subseteq$ and it holds that

$$\forall mts, mts' \in M : (mts \preceq mts' \Rightarrow mts = mts').$$

Note, that $M \in \text{context}(fts)$ is minimal for $fts$ iff it is a greatest element of $\text{context}(fts)$ thus subsuming a maximum number of (more refined) MTS and therefore also LTS variants. Furthermore, the first condition of Definition 11 does not imply the second one as the first condition does not forbid having MTSs with bisimilar variants in $M$ (or $M'$). Additionally, $\subseteq$ is a preorder on the set $\text{context}(fts)$ as a minimal MTS encoding of an FTS $fts$ is not necessarily unique. Furthermore, for Algorithm 1 to produce minimal MTS encodings, we have to impose a restriction on the corresponding input FTS models, referred to as structurally deterministic FTS. In particular, we call an FTS is structurally deterministic there exists no state having more than one outgoing transitions labeled with the same action, regardless of the (in-)compatibility of their presence conditions.
Definition 12 (Structurally Deterministic FTS). FTS $\langle S, A, F, \rightarrow, \lambda, \text{p_alternate} \rangle$ is structurally deterministic if $\forall t = (p, f, a, p') \in \rightarrow, (\forall t' = (p, f', a', p'') \in \rightarrow, t \neq t' \Rightarrow a \neq a')$.

Figure 11 provides an example (with all features being optional) for a structurally non-deterministic FTS to illustrate the problem such an FTS causes to Algorithm 1. In particular, the FTS (cf. Figure 11a) is structurally non-deterministic as state $s_0$ has two outgoing transitions being labeled with $a$ (although having mutually excluding presence conditions). As a result, we obtain two MTSs $mts_1$ and $mts_2$ (cf. Figures 11b and 11c) each with a mandatory transition similarly labeled $a$ which, however, correspond to different FTS transitions with mutually excluding presence conditions. As a consequence, it holds that $mts_2 \preceq mts_1$ and hence the result is not minimal according to Definition 11. To avoid those cases, we require structurally deterministic FTS for Algorithm 1 to derive a minimal MTS encoding. However, as a future work we plan to extend Algorithm 1 with a post-processing step to check for each generated MTS if is already covered by another MTS (between lines 5 and 6). We omitted this (presumably very expensive) check in the current version of our tool as it does no occur in any of our subject systems.

Next, we prove that Algorithm 1 derives a minimal MTS encoding for any given structurally deterministic FTS.

**Theorem 3.** Let $M \in \text{context}(fts)$ be the MTS encoding of a structurally deterministic FTS $fts$ generated by Algorithm 1. Then $M$ is the minimal MTS encoding of $fts$.

**Proof.** We prove Theorem 3 by showing that (1) $\forall m_i \in M : \nrightarrow_i$ is minimal, and (2) $\forall m_i, m_j \in M : \nrightarrow_i \not\subseteq m_j \Rightarrow m_i \nrightarrow_j m_j$, both by contradiction.

1. Let us assume that there exists an $m_i \in M$ such that $\nrightarrow_i$ is not minimal. In this case, there must be a step in Algorithm 1 where an optional transition unnecessarily becomes mandatory. Initially, every transition from $fts$ which has not to be excluded from $m_i$ is added as an optional transition. Optional transitions may become mandatory due to two reasons as already shown for Theorem 2.

(a) The presence condition of a newly added optional transition $t$ is implied by the presence condition of another transition or by the combination of presence conditions of a set of other transitions (lines 28–29). Hence, $t$ must be mandatory as $t$ must be included in every variant in which these other transition(s) are also included.

28
Due to the presence condition of the newly added transition $t$, a combination of optional transitions (including $t$) yields an invalid variant (lines 32–33). Hence, $t$ must be mandatory as otherwise the set of generated MTSs contains more variants than the FTS.

In addition, we have to consider those cases in which optional transitions might (unnecessarily) become mandatory due to loop unrolling. However, in these cases, procedure `NEWMTS` is restarted with a refined blocking clause to avoid unnecessary unrollings (lines 25–26, 30–31, 35–36, and 41–42). As a consequence, Algorithm 1 results in $\mathcal{M}$ being minimal.

2. Assume that there exists $m_i, m_j \in \mathcal{M}$ with $m_i \preceq m_j$. Hence, there exists at least one variant $v$ of the product line with $v \preceq m_i$ and $v \preceq m_j$. This is only possible if the FTS is structurally non-deterministic as this would require two bisimilar variants with different feature configurations. For a structurally deterministic FTS, this is avoided by imposing, and iteratively refining, the blocking clause in each new call of procedure `NEWMTS` (lines 5–7). In this way, any MTS having at least one variant already covered by some previously generated MTS will no more be (re-)generated in any subsequent run. Hence, we have $m_i \npreceq m_j$.

From (1) and (2) it follows that $\mathcal{M}$ is minimal for structurally deterministic FTSs according to Definition 11.

6. Implementation and Evaluation

In this section, we present experimental evaluation results gained from applying our approach to a collection of FTS models. To this end, we have implemented Algorithm 1 in a tool which allows us to generate a minimal MTS encoding from a given input FTS model as described in the previous section.

6.1. Experimental Setup

The first goal of our evaluation is to investigate general applicability of the algorithm to differing input FTS models. In addition, we are interested in the computational effort for generating a minimal set of MTSs from an FTS as well as the average number of MTSs required for a minimal MTS encoding of an FTS as compared to the maximum number of MTSs (i.e., the number of LTS variants derivable from the FTS). In particular, we consider the following research questions.

**Research Questions.**

- **RQ1 (Efficiency).** What is the computational effort for generating a minimal set of MTSs for a given FTS, as compared to the maximal set?
- **RQ2 (Effectiveness).** What is the average number of MTSs in a minimal set for a given FTS, as compared to the maximal set?

To address both questions, we applied our tool to a collection of subject systems comprising both existing case studies from the research community on FTS as well as synthetically generated FTS models.
Table 1: Subject Systems from this paper (1–5) and Real-World Subject Systems [19] (6–14)

<table>
<thead>
<tr>
<th>Subject System</th>
<th># Features</th>
<th># Variants</th>
<th># States</th>
<th># Transitions</th>
<th># Annotated Transitions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Figure 4a</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Figure 4a of this paper</td>
</tr>
<tr>
<td>2: Figure 4b</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>Figure 4b of this paper</td>
</tr>
<tr>
<td>3: Figure 4c</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>Figure 4c of this paper</td>
</tr>
<tr>
<td>4: Figure 4d</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>Figure 4d of this paper</td>
</tr>
<tr>
<td>5: Arcade Game Maker</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>13</td>
<td>13</td>
<td>Running example of this paper</td>
</tr>
<tr>
<td>6: Simple Traffic Light</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>Simple traffic light loop</td>
</tr>
<tr>
<td>7: Complex Traffic Light</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>Variant of the simple traffic light loop</td>
</tr>
<tr>
<td>8: Hot Drink Machine</td>
<td>9</td>
<td>28</td>
<td>14</td>
<td>21</td>
<td>17</td>
<td>Coffee/tea machine with multiple currencies</td>
</tr>
<tr>
<td>9: Sensor Subsystem</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>15</td>
<td>6</td>
<td>Sensor subsystem of a car wiper system</td>
</tr>
<tr>
<td>10: Wiper Subsystem</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>14</td>
<td>6</td>
<td>Wiper subsystem of a car wiper system</td>
</tr>
<tr>
<td>11: Modified Wiper Subsystem</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>14</td>
<td>7</td>
<td>Wiper subsystem of a car wiper system with permanent wiping</td>
</tr>
<tr>
<td>12: Mine Pump Controller</td>
<td>4</td>
<td>4</td>
<td>25</td>
<td>36</td>
<td>35</td>
<td>Controller of a water pumping system</td>
</tr>
<tr>
<td>13: Mine Pump System States</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>18</td>
<td>0</td>
<td>Supplement to the mine pump controller</td>
</tr>
<tr>
<td>14: Refined Mine Pump Controller</td>
<td>9</td>
<td>40</td>
<td>25</td>
<td>36</td>
<td>35</td>
<td>Mine pump controller with additional water level readings</td>
</tr>
</tbody>
</table>

Implementation. We implemented Algorithm 1 in a JAVA-tool called MooSE (Modal Transition System Encoding). As part of our tool, we utilize the SAT solver SAT4J [25] for reasoning about satisfiability of feature constraints. Furthermore, we use the Java Universal Network/Graph (JUNG) framework1 to create a GUI front-end for easily operating our tool and for visualizing FTS and MTS models. Besides Algorithm 1, our tool further incorporates an automated bisimulation check between FTS and MTS models as described in Section 4 which, for instance, allows the user to verify correctness of generated models.

In order to make our results reproducible, we provide our tool implementation (together with a manual) and our case studies on a supplementary web page2.

Subject Systems. We applied our experiments to 51 subject systems, where the first group comprises 14 FTSs which are taken from existing case studies initially published in [19] (cf. Table 1) as well as the examples created for this paper. In addition, the other group consists of 37 synthetically generated FTSs. In particular, the first group consists of FTS case studies modeling traffic light controls, a vending machine for coffee and tea, subsystems of a wiper system, and several different parts of mine-pump control systems. The smallest case study, Simple Traffic Light, consists of one (mandatory) feature, where the corresponding feature model hence defines one valid configuration and a corresponding LTS variant. The FTS of the Simple Traffic Light consists of four

1http://jung.sourceforge.net
2https://www.es.tu-darmstadt.de/fts2mts/
states and four transitions from which none is annotated by a presence condition. We added this example to our corpus to check whether our algorithm produces correct results also for such corner cases. The largest case study, Refined Mine Pump Controller, has nine features with 40 possible configurations and the respective FTS consists of 25 states and 36 transitions from which 35 are annotated by a presence condition thus constituting variable behavior.

In the second group, we further consider randomly generated FTS models in order to investigate in more detail the impact of different FTS properties on the resulting MTS encoding. We do this by first applying an existing tool for generating feature models and Featured Finite State Machines (FFSM) [21] being (non-hierarchical) Finite States Machines where transitions are, similar to FTS models, annotated with presence conditions. We translate FFSM to FTS by simply copying the set of states and transitions together with their presence conditions. In contrast, the transition labels on FFSM, which are much more compound than those of FTS, are treated as one atomic action per transition as the actual labeling is not relevant for Algorithm 1. In particular, we generated three FTSs as well as three feature models (i.e., one feature model for each FTS). We then adapted each FTS, e.g., by removing transitions, and each feature model, e.g., by changing an or-group to an alternative group, to obtain a diverse corpus. Here, the case studies vary between twelve to 16 features, 50 to 10 states, 21 to 79 transitions, and five to 50 variants.

Experiment Design and Measurement Setup. In order to evaluate our approach, we generated a minimal MTS encoding as well as the maximal set of MTSs (i.e., the set of all LTS variants of the input FTSs) for each of our subject systems. To answer research question \textbf{RQ1}, we measured CPU times required for applying Algorithm 1 as compared to generating all LTS variants from the given FTS models. Concerning \textbf{RQ2}, we additionally counted the number of MTS models of the minimal encoding as well as the number of LTS variants of the maximal encoding for the given FTS models. We used SAT4J version 2.3.4, and we applied all experiments on a machine with Windows 10 x64 and 12GB of RAM running on an Intel Xeon E3-1230v3 (4x3.3GHz) machine. We reran the experiments several times and observed that the deviation of the results among the different runs are negligible.

6.2. Results and Discussion

We next present the measurement results of our experiments together with a discussion of the results with respect to our research questions.

Results. The measurement results addressing \textbf{RQ1} and \textbf{RQ2} are shown in Figure 12 (real-world case studies) and Figure 13 (synthetic case studies), respectively.

- **RQ1 (Efficiency).** The average CPU time required for the real-world case studies is 5.2 s with a geometric mean of 65.9 ms, ranging from less than 1 ms (Simple Traffic Light) to 59.7 s (Refined Mine Pump Controller). The average CPU time required for the synthetic case studies is 8.1 min with a geometric mean of 170.5 s, ranging from 5.2 s (case study 1 having six variants) to 36.9 min (case study 35 having 20 variants). In contrast, generating the maximal set
Figure 12: Results for the Real-Word Case Studies (cf. Table 1) with Logarithmic y-Axis

Figure 13: Results for the Synthetic Studies with Logarithmic y-Axis
of MTS (i.e., directly deriving the set of LTS variants) takes at most 2 ms for all case studies (except for the synthetic case study 36 taking 6 ms).

- **RQ2 (Effectiveness).** Concerning the real-world case studies, the minimum number of MTSs equals the maximum number of MTSs in some cases (namely 1, 6, 7, 8, 9, 10, 11, and 13) while for other case studies, the minimum number of MTSs is considerably smaller than the maximum number (e.g., case study 14 can be encoded as 20 MTSs comprising 32 LTS variants). Here, the geometric mean is 2.7 for the minimum number of MTSs and 3.5 for the maximum number. Note that we three case studies (6, 7, and 13) each having only one variant to demonstrate that our algorithm does indeed not produce sets of MTSs with overlapping variants. Without these special cases, the geometric mean is 3.6 for the minimum number of MTSs and 5.0 for the maximum number. For the synthetic case studies, Algorithm 1 produces 14.8 MTSs and a maximum number 18.6 MTSs, again, considering the geometric mean. Again, note that we observe a number of cases where the minimum and maximum number of MTSs coincide (1, 2, 21, 22, 26 to 30, 32, 33, 34, and 37) as no optional transitions can ever be produced in any MTS. For the rest of the synthetic case studies, the ratio between the minimum and the maximum number of MTSs ranges between 10 (minimum) to 12 (maximum) and 48 (minimum) to 80 (maximum).

**Discussion and Summary.** We now discuss the results of our experimental evaluation with respect to the research questions **RQ1** and **RQ2**.

- **RQ1 (Efficiency).** From the results obtained from both the real-world case studies as well as the synthetically generated case studies, we can conclude that there is no obvious correlation between the different structural size measures (e.g., number of states and transitions) of the FTS models and the CPU time consumed by Algorithm 1. Instead, we observe a potential correlation between CPU time and the number of variants for most of the case studies. To summarize, we may conclude that our approach is capable to also scale to FTS models comprising considerably larger sets of variants as the average CPU time is between 5.2 s (real-world) and 8.1 min (synthetic). Even the largest synthetic case study only takes about 37 min. However, it should be noted that generating a minimum set of MTSs is considerably slower than generating the maximum set, taking only 6 ms in the worst case (synthetic case study 36).

- **RQ2 (Effectiveness).** Concerning effectiveness of Algorithm 1, there is no obvious correlation between the minimal and maximal solution. As described above, there are structurally small as well as larger case studies for which the number of variants and the number of MTSs, however, is equal (for both real-world and synthetic case studies). Instead, we observe that the number of generated MTSs not only depends on the structure of the FTS but also on the dependencies between features and presence conditions as also illustrated by the different examples in Section 4). For our real-world case studies, the minimum number of MTSs is 23% smaller than the maximum number. When leaving out the special cases 6, 7, and 13 (only consisting of one variant) the minimum number of
MTSs is 28% smaller than the maximum number. For the synthetic case studies, we observe similar results. Here, the minimum number of MTSs is 20% smaller than the maximum number. Additionally, when leaving out the case studies 26 to 37 (where the minimum and maximum are very similar due a high degree of dependencies between presence conditions of different transitions), the minimum number of MTSs is, again, 28% smaller than the maximum number. To summarize, there are no obvious (i.e., syntactic) structural properties of FTS models indicating a clear correlation between the sizes of the minimum and maximum MTS encoding.

6.3. Threats to Validity

We conclude this section with a brief discussion of threats to validity potentially obstructing our evaluation results.

Internal Threats. Concerning the correctness of our approach, we provide a detailed proof in Section 4 showing that the MTS encoding is, up to bisimulation equivalence, both sound and complete with respect to the set of variants represented by the input FTS model. In addition, we proof in Section 5 that the MTS encoding generated by Algorithm 1 constitutes a minimal solution. In this regard, one possible threat to internal validity might arise from the correct implementation of the approach in our tool. To address this issue, we exhaustively tested our tool implementation using a variety of different examples including default cases as well as corner cases. In addition, the only major external component used in our tool is SAT4J which is a mature SAT-solver widely used in practice. Another potential threat to the internal validity of our evaluation results may arise from the inherent non-determinism of Algorithm 1 concerning, for instance, the ordering in which the state-transition graph is traversed in an iteration. As a result, the resulting minimal MTS encoding is not unique thus leading to different measurement results for different runs with the same input model.

External Threats. One potential threat to external validity might arise from the lack of comparison with our approaches. However, we are not aware of any competitive approach so far in recent literature aiming at generating a minimal MTS encoding as pursued in our approach (see Section 7 for details). However, one interesting path to follow in a future work would be to consider MTS with variability constraints as recently proposed in [17]. As this extension increases expressiveness of MTS, it may permit an even more succinct encoding of FTS as compared to the plain MTS considered in our setting.

Finally, the selection of subject systems might always threaten external validity. For our experiments, we selected well-known community benchmarks as well as synthetically generated models. However, although we are confident that our collection covers a variety of crucial cases and model sizes, the lack of real-world FTS models might obstruct any generalization of our evaluation results.
7. Related Work

In this section, we discuss related work on relating the semantic models (and therefore comparing the expressiveness of) different modeling formalisms for software product lines from the recent literature. To this end, we limit our considerations to (operational) behavioral-variability modeling formalisms based on (variations of, or extensions to) LTSs as the underlying semantic foundation. The goal of all considered approaches is, in general, to avoid that every possible model variant derivable from a software product line has to be explicitly modeled as a dedicated LTS. Instead, the different approaches propose (syntactic and/or semantic) mechanisms for integrating several model variants into one concise model.

To our knowledge, our approach in encoding product-line formalisms with strictly different expressive power is novel and unique. Based on this concept, expressiveness of the different approaches can be characterized by the minimum number of models required to cover all variants. In this regard, FTSs and MTSs can be seen as two extrema of an expressiveness spectrum, as one FTS always suffices to comprise all possible LTS variants (but, with the disadvantage of a complex representation), whereas MTSs are inherently limited by only being capable of distinguishing between mandatory and optional transitions (but, with the advantage of a simple representation).

As an alternative way of comparing expressiveness, Beohar et al. have recently proposed to define encodings between formalisms such that a hierarchy of expressiveness is naturally built upon the (non-)existence of (mutual) encodings [11]. In contrast, Benduhn et al. survey different modeling formalisms in terms of their suitability for applying different product-line analysis strategies [26].

Concerning MTSs [2] in particular, Fischbein et al. [27] were the first to argue that these models are adequate for modeling behavioral variability in software product lines. Thereupon, several researchers used MTSs as well-suited formalism to perform rigorous analysis of software product lines [4, 5, 6, 28, 7]. In order to cope with the limited expressive power of MTSs and to further restrict the set of valid model variants derivable from an MTS, various approaches combine MTSs with additional constraints expressed in a deontic logic called Modal-Hennessy-Milner-Logic (MHML) [4, 5, 6] as well as so-called variability constraints [17]. Furthermore, for this extension of MTSs with feature constraints the authors provide a translation from FTSs to MTSs by annotating the target MTSs with feature constraints when necessary [18]. Furthermore, Benes et al. in [24], introduce an extension of MTSs with a set of parameters and define obligation functions on the set of atomic propositions, which are related to each state and contain transitions emanating the state and parameters. By setting different valuations for the parameters and also using different atomic propositions the presence or absence of transitions can be specified. Using this formalism global/persistent choices can be made throughout a model. Křečinský and Sickert [29] show that this extension of MTS may be translated to Boolean MTS, whereas they do not consider a translation into (sets of) plain MTSs as done in our work. Basile et al. in [30], introduce an extension of contract automata with modality [31], in which necessary requests are distinguished from permitted requests, with feature constraints. These models can be used for modeling the behavior of contract-based dynamic service product lines. Other approaches exploit principles from interface theories to restrict the set of derivable vari-
ants from MTS to only those being compatible under parallel composition to a given environmental specification [7, 28]. In [9], two variants of MTSs, namely disjunctive modal transition systems (DMTSs) [32] and 1MTSs are compared from the expressiveness point of view. DMTSs are similar to 1MTSs in that both rely on the notion hyper transitions, in order to explicitly relate transitions in MTSs to (sets of) features of a product line. The difference is in the interpretation of such transitions. In DMTSs, must-hyper-transitions represent an or-relation between multiple choices, whereas this restriction is not made in 1MTS. In [9], it is shown that both formalisms have the same expressive power, i.e., they induce the same sets of LTSs as their implementations.

Concerning FTSs, as initially proposed by Classen et al. [33], those models have been mostly utilized for efficient temporal model-checking of entire product lines by solely considering one FTS model [1]. Thereupon, Cordy et al. [34] extended this earlier work by combining non-Boolean features and multi-features in a high-level specification language called TVL$^\ast$. An algorithm for constructing an FTS from a behavioral specification written in TVL$^\ast$ was also given.

Finally, PL-CCS [35], introduced by Gruler et al. [35], constitutes an extension of Milner’s CCS [36] by means of an alternative choice operator called “binary variant" to choose (and memorize) behavioral variations in CCS step semantics. Similar to MTSs, the validity of variants can be further restricted using the multi-valued modal $\mu$-calculus [37].

To summarize, none of these existing approaches for comparing product-line modeling formalisms yet followed the idea as proposed in this paper, by relating one more expressive (FTS) model to less expressive (MTS) models requiring sets of models such that both comprise equivalent sets of variants.

8. Conclusions and Future Work

In this paper, we presented an encoding of FTSs into sets of MTSs with an equivalent set of LTS variants. We also gave an algorithmic interpretation of this translation and proved it to be correct. Moreover, we discussed the issue of minimality of the computed translations and proved a particular notion of minimality for the outputs of our algorithm.

The concept developed in this paper allows for a novel assessment concerning the expressiveness of variability-modeling formalism in terms of the number of models required for covering all variants. Based on this new concept, we aim at defining new encoding and expressiveness criteria as future work. As a result, we are targeting the definition of a dense spectrum of variability-modeling formalisms, having FTSs and MTSs (or, plain LTSs, respectively) as its extrema. In the regard, also the possible influence of of potentially infinite sets of states of these formalisms on expressiveness shall be taken into account. Furthermore, we plan to adapt the algorithm such that the requirement of structurally deterministic FTSs may be dropped to obtain minimal MTS encodings.

Acknowledgment. The work of Mahsa Varshosaz and Mohammad Reza Mousavi has been partially supported by grants from the Swedish Knowledge Foundation (Stiftelsen
for Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAAS HoGpro-
tect (number: 20140312), Swedish Research Council (Vetenskapsradet) award number:
621-2014-5057 (EffectiveModel-Based Testing of Concurrent Systems), and the EL-
LIIT Strategic Research Environment. The work of Lars Luthmann, Paul Mohr and
Malte Lochau has been supported by the German Research Foundation (DFG) in the
Priority Programme SPP 1593: Design For Future – Managed Software Evolution (LO
2198/2-1).

References

tured Transition Systems: Foundations for Verifying Variability-Intensive Sys-
tems and Their Application to LTL Model Checking, IEEE Transactions on Soft-

[2] K. Larsen, B. Thomsen, A modal process logic, in: Proc. of the 3rd Annual Sym-

Proceedings of ACM/IEEE Symposium on Logic in Computer Science (LICS 1990),

in product families, in: Proceedings of the 15th International Software Product
Line Conference (SPLC ’11), IEEE, 2011, pp. 130–139.

ilies of services, in: Proc. of the International Conference on Formal techniques
for distributed systems (FMOODS’11/FORTE’11), Vol. 6722 of Lecture Notes in

derive product line behavioural descriptions, in: Proceedings of the 5th Interna-
tional Symposium on Leveraging Applications of Formal Methods, Verification
and Validation. Technologies for Mastering Change (ISoLA ’12), Vol. 7609 of

Symposium on Programming Languages and Systems (ESOP’07), Vol. 4421 of

[8] M. H. ter Beek, A. Lluch-Lafuente, M. Petrocchi, Combining declarative and
procedural views in the specification and analysis of product families, in: Pro-
ceedings of the 17th International Software Product Line Conference co-located
workshops (SPLC ’13 workshops), ACM, 2013, pp. 10–17.


URL http://doi.acm.org/10.1145/3109729.3109741

39


Appendix A.

Lemma 1. Considering the definition of the function const(), given in Definition 7, this function always has a maximal fixed point.

Proof. Consider an FTS $fts = (P, A, F, \rightarrow, \Lambda, p_{init})$; given $mts = (Q, A, \rightarrow, \square, \Lambda, q_{init})$, where $Q = P \times B(F)$; we prove $const((p, e))$, for $(p, e) \in Q$, always has a maximal fixed point as follows. We prove this function is monotone and hence has a fixed point. To this end, we show $const((p, e)) = const((p, e))$ by applying induction on the index of the function. In the base case for any $(p, e) \in Q$ we have:

\[
const_0((p, e)) = e
\]

\[
const_1((p, e)) = e \land \bigwedge_{(p.a,f,p') \in \text{max}(p,e)} \left( const_0((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \lor \left( const_0((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \right) \right)
\]

Thus, it holds $const_1((p, e)) \implies const_0((p, e))$.

In the inductive step we consider: for any $(p, e) \in Q$, $\forall j \leq i - 1$ $const_j((p, e)) \implies const_j-1((p, e))$. Then, we prove $const_i((p, e)) \implies const_{i-1}((p, e))$ as well. Based on the above definition:

\[
const_i((p, e)) = e \land \bigwedge_{(p.a,f,p') \in \text{max}(p,e)} \left( const_{i-1}((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \lor \left( const_{i-1}((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \right) \right)
\]

and also it holds:

\[
const_{i-1}((p, e)) = e \land \bigwedge_{(p.a,f,p') \in \text{max}(p,e)} \left( const_{i-2}((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \lor \left( const_{i-2}((p', e \land f)) \land \bigwedge_{(p,a,f,p') \in \text{max}(p,e)} \neg f \right) \right)
\]
Given the premise in the inductive step it holds: \( V(p, e') \in Q \Rightarrow const_{k-1}((p', e')) \Rightarrow const_{k-2}((p', e')) \). Hence, it holds:

\[
\forall (p, a, f') \in \mathbb{M}(p, e) (const_{i-1}((p', e \land f))) \Rightarrow \forall (p, a, f') \in \mathbb{M}(p, e) (const_{i-2}((p', e \land f))) \Rightarrow \forall (p, a, f') \in \mathbb{M}(p, e) (\neg f' \lor (const_{i-1}((p', e \land f))))
\]

Hence, it holds \( const_{i}((p, e)) \Rightarrow const_{i-1}((p, e)) \).

**Lemma 2.** Consider an arbitrary fs = \((P, A, \rightarrow, A, p_{out})\) and a set of MTSs \( M \in context(fs)\). Consider \( mts = (Q, A, \rightarrow_{Q}, \rightarrow_{\circ}, q_{init}) \) s.t. \( mts \in M \) and \( \lambda \in \Lambda_{mts} \), it holds:

\[
\forall (p, e) \in Q \Rightarrow e \land \lambda \Rightarrow f
\]

\[
\forall (p, e, a, f') \in \rightarrow_{Q} ((p, e), a, (p', e \land f)) \in \rightarrow_{Q}
\]

**Proof.** Based on item 4.1.a in Definition 8, there exists a path in the set of finite paths of mts such as \( p : q_{init} a_{0} (p_{1}, e_{1}) \ldots (p_{n-1}, e_{n-1}) p_{n-1} (p, e) \), in which \( \forall 1 \leq i \leq n-1 e_{i} = e_{i-1} \land f_{i} \), and \( e = e_{n-1} \land f_{n-1} \). Based on Definition 8, in each transition \( (p, e) \overset{a}{\rightarrow} (p', e') \), it holds \( e' \Rightarrow e \) and as \( \lambda \Rightarrow e \), it can be concluded that \( \forall 1 \leq i \leq n \lambda \Rightarrow e_{i} \), and also as \( q_{init} = (p_{init}, \mathbb{V}_{\lambda, A} \lambda) \) it holds that \( \lambda \Rightarrow \mathbb{V}_{\lambda, A} \lambda \). Hence, the premise of the above implication holds for all the states in \( \rho \), that is an initial path that ends in \( (p, e) \). Next, we use induction through the path to prove the above implication holds. Since \( \lambda \in \Lambda_{mts} \), then \( \lambda \Rightarrow const(q_{init}) \). Assume that the iterations for computing the function \( const \) are fixed in \( k \) steps that is \( const_{k}(q_{init}) = const_{k-1}(q_{init}) \) (a fixed point exists according to Lemma 1).

We consider the base step of induction:

\[
const_{k}(q_{init}) = e \land \bigwedge_{(p, a, f') \in \mathbb{M}(p, e)} (const_{k-1}((p', e \land f))) \land \bigwedge_{(p, a, f') \in \mathbb{M}(p, e)} (\neg f' \lor (const_{k-1}((p', e \land f)))) \land \bigwedge_{(p, a, f') \in var(p, e)} \neg f
\]

Based on item 4.1.a in Definition 8, \( e_{1} = \mathbb{V}_{\lambda, A} \lambda \land f_{0} \). Since, \( \lambda \Rightarrow const_{k}(q_{init}) \), and \( \lambda \Rightarrow e_{1} \), from the above formula it can be concluded that \( \forall (q_{init}, a, p', e') \in \rightarrow_{Q} (q_{init}, a, (p', e')) \in \rightarrow_{Q} \). Otherwise \( \neg f_{0} \), is considered as one of the conjunctions in construction of \( const_{k}((p, e)) \), and as \( \lambda \Rightarrow const_{k}(q_{init}) \) then \( \lambda \Rightarrow \neg f_{0} \), which contradicts \( \lambda \Rightarrow f_{0} \).