Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Learning by Sampling: Learning Behavioral Family Models from Software Product Lines

Carlos Diego Nascimento Damasceno ·
Mohammad Reza Mousavi ·
Adenilso da Silva Simao

Abstract Family-based behavioral analysis operates on a single specification artifact, referred to as family model, annotated with feature constraints to express behavioral variability in terms of conditional states and transitions. Family-based behavioral modeling paves the way for efficient model-based analysis of software product lines. Family-based behavioral model learning incorporates feature model analysis and model learning principles to efficiently unify product models into a family model and integrate the behavior of various products into a behavioral family model. Albeit reasonably effective, the exhaustive analysis of product lines is often infeasible due to the potentially exponential number of valid configurations. In this paper, we first present a family-based behavioral model learning techniques, called $\mathit{FFSM}_{\mathit{Diff}}$. Subsequently, we report on our experience on learning family models by employing product sampling. Using 105 products of six product lines expressed in terms of Mealy machines, we evaluate the precision of family models learned from products selected from different settings of the T-wise product sampling criterion. We show that product sampling can lead to models as precise as those learned by exhaustive analysis and hence, reduce the costs for family model learning.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPESP (grant 2019/06937-0), and University of Leicester (UK), College of Science & Engineering. Research carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0). We have also been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/1.

Carlos Diego Nascimento Damasceno and Adenilso da Silva Simao
Instituto de Ciências Matemáticas e de Computação, Universidade de Sao Paulo, Brazil
E-mail: damascenodiego@alumni.usp.br, adenilso@icmc.usp.br

Mohammad Reza Mousavi
School of Informatics, University of Leicester, UK
E-mail: mm789@leicester.ac.uk

Keywords Software product lines · Model learning · Family model · T-wise sampling

1 Introduction

Several technology companies, such as ABB [117], Boeing [111], Philips [81,80], and Siemens [82], have been facing an increasing demand for mass production and customization of software products [97]. To cope with this need, they have been investing in establishing common platforms to build software families using production line principles [114]. Software product lines (SPL) provide a means to support the mass production and customization of software systems [35]. Unlike traditional systems, which are tailored for a specific use, SPLs are developed for reuse and with reuse. Thus, products are not created anew but derived from assets managed as commonalities and variabilities [83].

Analysing (e.g., validating and verifying) the system-level functionalities of an SPL on a product-based basis is very demanding due to the potentially exponential number of valid products, e.g., the Linux kernel and its 6,320 features [22]. Explicit models have been used to support the analysis and development of high-quality systems. They help software engineers in program comprehension [15], software refactoring [108], model checking [14], and model-based testing [124]. Family-based modeling approaches have been developed to facilitate SPL analysis without going through each and every product individually [120]. Such approaches typically involve two types of family-based models: structural and behavioral [97]. Family-based structural models, such as feature models [70], capture the presence and absence of features in various products. Family-based behavioral models, such as featured finite state machines [58], capture the functionality of features and their interactions. Family-based behavioral models are often referred to as a *family model* [120,93] or *150% model* [107, 24] and are the corner-stone of efficient model-based SPL behavioral analysis.

More specifically, family-based behavioral analysis techniques have been developed for efficient test case generation [13,20,58] and model checking [104,119] of SPLs. Family models have been used for conformance analysis [56], probabilistic model checking [128,30], and real-time software testing [85]. Nevertheless, the creation and maintenance of family models are difficult and time consuming due to crosscutting features [93] and the traceability between the family and feature models can become hard to maintain [107]. Thus, as new requirements emerge and products evolve, the lack of maintenance may lead to outdated and incomplete models [133]. Additionally, in practice, many software development environments do not have a structured SPL development process in place and rely on individual product models without knowledge about their commonalities and variabilities [63]. To remedy these issues, we propose an approach for learning behavioural family models of SPLs.

In recent years, we have seen a resurgence of interest in model learning [125,2], particularly supervised techniques for learning state-based models [7, 110]. This has led to successful applications in industrial practice [2] and empirical studies to evaluate the performance of new algorithms and tools for model learning [90]. Our approach builds upon these recent attempts and aims to abstract a family-based state machine model from individually learned or hand-crafted product models.

We introduce the Featured Finite State Machine Difference ($\mathit{FFSM}_{\mathit{Diff}}$) algorithm, a technique that employs a similarity measure for state-based models [134] to identify similar behavior in various products specified as Mealy machines [57], annotate conditional states and transitions with feature constraints, and integrate them into a succinct family model. Our technique is discussed in terms of a Featured Finite State Machines (FFSM), a family-based formalism that unifies Mealy Machines of SPLs into a single representation to enable an efficient model-based analysis of SPLs [58,56]. However, the ideas surrounding our algorithm can be extended to other family-based notations [19], such as Modal Transition Systems (MTS) [78], various extensions of MTS [52,77,17,16], and Featured Transition System (FTS) [33,20].

Additionally, we evaluate the use of product sampling [95,68] to efficiently choose individual products that are to be analyzed and learn precise family model. Product sampling techniques, such as T-wise [127], should collectively cover the behavior of an SPL using a subset of all valid combinations of T selected features [95,68]. Hence, they should address family model learning with reasonable precision and execution costs lower than in an exhaustive analysis. To evaluate the precision of learned models and the efficiency of learned models by sampling, we compare the sampling and exhaustive approaches.

To evaluate our approach, we perform an empirical study of its efficiency on a benchmark set of SPLs [58,32,105,46,45]. Through this empirical evaluation, we aim to answer the following research questions:

(RQ1) Is our approach effective in learning succinct family models with respect to the total size of the products under learning?
(RQ2) Is our approach effective in learning succinct family models with respect to the total size of the hand-crafted models?
(RQ3) Is the size of learned family models influenced by the configuration similarity degree of the products under learning?
(RQ4) Is our approach effective in learning precise family models compared to those obtained by exhaustive analysis?

Regarding (RQ1) and (RQ2), we evaluate the succinctness of the learned family model with respect to the individual product models and the hand-crafted family-based specifications. We describe *succinctness* in terms of the number of transitions and states as these are factors that influence the complexity of model-based techniques [26,14] and that are used to interpret the language and structure of state-based models [134]. Regarding (RQ3), we show that our approach is effective when it can identify reuse and leads to more succinct family models by integrating the reused features. Hence, we set out to test the correlation between the degree of reuse and the succinctness of learned models. Finally, regarding (RQ4), we test the effectiveness of various sampling techniques in learning precise family models.

This paper builds upon and extends on preliminary results of a conference paper that has been published in the proceedings of the 23rd International Systems and Software Product Line Conference (SPLC 2019) [42]. Besides providing a more detailed explanation throughout the paper, we have introduced new parameters for model comparison and merging; we have incorporated three extra models in our benchmark, including a state machine model from a real system [105]; and we have evaluated the precision of family models learned by sampling. We briefly summarize our contributions as follows:

1. We introduce a technique to learn family models from individual product specifications by means of state-based model comparison and feature model analysis;
2. We present an experiment evaluating our technique and showing its effectiveness for learning succinct family models in terms of numbers of states and transitions;
3. We show that the amount of feature reuse is a factor that affects the size of learned family models;
4. We evaluate the effectiveness of family models learned by sampling against those learned by exhaustive analysis.

The remainder of this paper is structured as follows: In Section 2, we introduce the fundamental background concepts used in this study, such as SPLs, sample-based analysis for SPLs, finite state machines and structural comparison of state-based models. In Section 3, we introduce our family model learning approach and how it incorporates feature model analysis into the process of structural comparison of state machines. In Section 4, we present a process that employs product sampling to reduce the costs for family model learning. In Section 5, we discuss an empirical evaluation to evaluate the effectiveness of our approaches for family model learning and the precision of models learned by sampling. In Section 6, we discuss some works related to our paper. In Section 7, we conclude this paper by presenting our conclusions and future work. To support the reader, a glossary of symbols is available in Appendix A.

2 Background

This section presents the background concepts and formalisms used in this study. We introduce SPLs, finite state machines, featured finite state machines, and the *Labeled Transition Systems difference* (LTS_{Diff}) algorithm for structural comparison of state-based models represented as Labeled Transition Systems (LTS), a well-known variant of FSM [71]. We follow this particular ordering in order not to break the logical flow of the presentation on behavioral modeling for SPLs (i.e., the concepts of FSMs and FFSMs are strongly related by definition) and because no study has ever associated the LTS_{Diff} algorithm to family models.

2.1 Software Product Lines

A software product line is a family of products sharing a common and managed set of *features* developed in a prescribed way to satisfy the needs of a particular market segment. Pohl et al. [97] introduces an SPL engineering framework with two key processes: Domain engineering and Application engineering. This separation of concerns enables to build robust platforms to develop customer-specific applications in a shorter time, at lower cost, and with improved quality.

During the domain engineering, the common and variable artifacts, and the scope of an SPL are defined, managed and constructed. During the application engineering, commonalities and variabilities of an SPL are exploited to achieve the highest possible reuse of domain artifacts. Artifacts generated during the domain engineering are used to support the creation of software products. Thus, most of the application artifacts are not developed anew but reused from domain engineering artifacts with the support of software generators and valid product configurations. In Figure 1, we illustrate the software product line engineering framework and the processes of domain and application engineering.

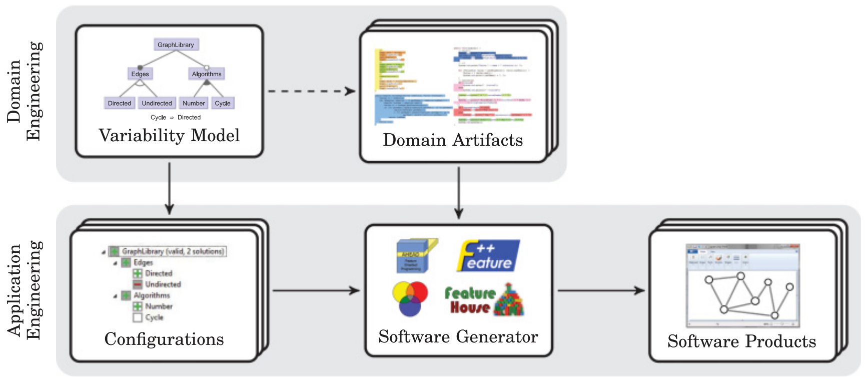

Fig. 1: The Software Product Line Framework [120]

Let F be the set of features of an SPL. A product p is defined by a subset of features $p \subseteq F$ selected from a variability model, such as a feature model [70]. A feature model captures the structural information and dependencies between common and variant features of an SPL. Features are concrete, if they are mapped to any implementation artifact; or abstract, if they are only used to group other features [122]. These dependencies are denoted as a hierarchically arranged set of interconnected features where parent-children relationships indicate dependency relations among features, and cross-hierarchy constraints are typically denoted by propositional logic formulas [48].

There are four basic types of parental relationships among features: *Mandatory*, if a child feature is included in all products in which its parent appears;

Optional, if a child is optionally included; *Alternative*, when only one child feature can be selected; and *Or*, when one or more of features can be included. For cross-hierarchy relationships, we have two typical forms: *Requires*, if the implementation of a feature `A` demands another feature `B`; and *Excludes*, if two features cannot be part of the same product. Propositional logic formulas can be used to describe more complex and advanced cross-hierarchy constraints among features [48]. In fact, propositional logic formulas have been extensively used in the automated analysis of feature models.

Boolean satisfiability solvers have been used as key elements under the hood of many feature model analysis tools [18]. The SAT4J project [79] is an example of a satisfiability solver widely used in feature model analysis. It composes the FeatureIDE [121] library, an Eclipse-based IDE that supports all phases of feature-oriented software development for SPLs [50].

Feature models have been also extended with cardinality constraints and attributes to cope with the need for richer specifications [37,38,22]. In this paper, we investigate the problem of family model learning using an extensive academic benchmarks [58,32] of SPLs that included non-trivial aspects, such as the possibility of infinite behavior and the existence of states with similar or identical behavior in different products.

Let the set of features of a feature model be F, the powerset $\mathcal{P}(F)$ of all feature combinations is constrained to a subset of valid products $P \subseteq \mathcal{P}(F)$ that satisfy its feature constraints. Feature constraints are propositional logic formulae that interpret the elements from F in terms of propositional variables. SAT solvers [79] are often used to detect valid feature models, feature combinations, core features (i.e., features that are part of all products) and redundancies [18]. We denote by $B(F)$ the set of all feature constraints. The subset $\Lambda \subseteq B(F)$ defines all valid product configurations of an SPL. We interchangeably refer to products as sets of features and propositions.

The configuration $\xi \in B(F)$ of a product $p \in P$ is a feature constraint that expresses the conjunction of all features included in p and the conjunction of negated features absent from it, i.e., $\xi = (\bigwedge_{f \in p} f) \wedge (\bigwedge_{f \notin p} \neg f)$. Given a feature constraint $\chi \in B(F)$, a configuration $\xi \in \Lambda$ satisfies χ, denoted by $\xi \vDash \chi$, iff the feature constraint $\xi \wedge \chi$ is *true*. Given two feature constraints ω_a and ω_b from a feature model FM, and $\Lambda_a, \Lambda_b \subseteq \Lambda$ satisfying ω_a and ω_b, respectively, we say that ω_a and ω_b are equivalent under FM if $\Lambda_a = \Lambda_b$. To illustrate the concepts of SPLs, we begin using the Arcade Game Maker feature model as our running example.

Example 1 (The Arcade Game Maker SPL) The Arcade Game Maker (AGM) SPL includes three alternative features (i.e., `Brickle`, `Pong` and `Bowling`) and one optional feature (i.e., `Save`). In Figure 2, we depict the AGM feature model. In the feature expressions to come, we typically use the abbreviated names of features as shown in Figure 2. The AGM feature model has six valid product configurations, among which three satisfy the feature constraint $\neg$`S`, indicating that the `Save` feature is absent.

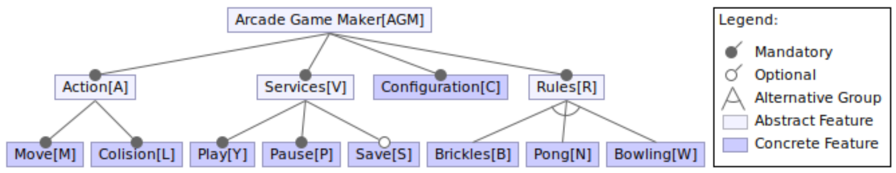

Fig. 2: The AGM feature model

2.1.1 Product-based analysis of SPL

In product-based strategies, valid products of an SPL are individually specified and analyzed. While theoretically possible, these strategies are impractical due to the potentially exponential number of feature combinations [120]. Hence, one should avoid exhaustive and redundant analysis, and cater for valid feature interactions [9]. To tackle these issues, combinatorial interaction testing and similarity analysis have been relevant approaches to optimize the analysis of product lines. In the next sections, we present two criteria that have been employed to optimize the analysis SPLs.

Configuration sampling

Combinatorial interaction testing (CIT) aims at using interaction coverage to sample product configurations [72]. It is based on the observation that most faults emerge by the interaction between a small number of features [73]. For interactions between any t features of SPLs, CIT is often referred to as T-wise testing [95]. The T-wise sampling criterion, defined below, aims at sampling valid configurations from all possible combinations of selected and unselected features. These interactions are called a t-set.

Definition 1 (Valid t-set) A valid t-set is a set of features $\{\pm f_1, \pm f_2, ..., \pm f_t\}$ satisfying the constraints defined by the feature model FM over the set of features F, where $t < |F|$, $+f_i$ indicates a selected feature i and $-f_i$ an unselected one. A T-set is *invalid* if it does not satisfy the constraints of FM.

Definition 2 (T-wise coverage) The t-wise coverage of a set of configurations $TCS = \{PC_1, PC_2, \ldots, PC_m\}$ is the ratio $T_t = \frac{\# \cup_{i=1}^{m} T_{t,PC_i}}{\# \ T_{t,FM}}$, where T_{t,PC_i} is the set of t-sets included within the configuration PC_i, $T_{t,FM}$ is the set of all possible valid t-sets in FM, and $\#A$ denotes the cardinality of a set A.

Sample-based techniques are known to improve the efficiency of SPL analysis by discarding products that may already be covered by other products [120]. However, such analysis may be incomplete and miss product specific behaviors. Higher-order feature interaction coverage is known for its improved fault detection capabilities [115,96]. Thus, for larger T values, the T-wise coverage should lead to more complete analysis. The Chvatal algorithm [31] is an example of technique for T-wise product sampling [68] that is available in the FeatureIDE workbench [121].

Configuration similarity

Studies in software testing have shown that similar test cases tend to have equivalent fault detection capabilities, and no additional gain should be expected when these are simultaneously executed [27]. To mitigate these issues, similarity metrics have been used as test prioritization criteria [137,27] for access control systems [23,40] and SPLs [61,4].

In configuration similarity, a similarity metric describes a similarity relation between two configurations as a numeric value. Similarity metrics often range from zero, if product configurations are totally distinct; to one, if they implement the same set of features.

The Hamming distance is a well-known measure [47] that has been used in the context of SPLs to calculate the similarity between product configurations. It is represented as the normalized number of common selected and unselected features for the two configurations as follows:

Definition 3 (Configuration similarity) The configuration similarity between two product configurations p_i, p_j from a feature model FM with the set of features F is defined as shown in Equation 3.1.

$$confSim(p_i, p_j, F) = \frac{|p_i \cap p_j| + |(F \backslash p_i) \cap (F \backslash p_j)|}{|F|} \tag{3.1}$$

In the $confSim()$ metric, $|p_i \cap p_j|$ denotes the number of common features selected between p_i and p_j and $|(F \backslash p_i) \cap (F \backslash p_j)|$ represents the number of common unselected features between them. These two values are normalized by the total number of features $|F|$.

2.1.2 Family-based analysis of SPLs

Family-based analysis relies on domain artifacts that incorporate knowledge about valid feature combinations to perform efficient model-based analysis of SPLs, e.g., model-based testing [21] and model checking [104,119]. Thus, not every individual product has to be analyzed, and redundant computations are minimized or avoided [120]. The performance family-based strategies is mainly influenced by the number of features, the size of feature implementations, and the amount of reuse during feature combinations [25].

In this section, we introduce the Featured Finite State Machine notation [58,54] to express the individual features and feature combinations as finite state machines extended with feature constraints. We start defining finite state machines [57] as a behavioral model to specify product families and hence, its featured extension for family-based modeling.

Definition 4 (Finite state machine) A finite state machine (FSM) is a septuple $\mathcal{M} = \langle S, s_0, I, O, D, \delta, \lambda \rangle$ where S is the finite set of states, $s_0 \in S$ is the initial state, I is the set of inputs, O is the set of outputs, $D \subseteq S \times I$ is the specification domain, and $\delta : D \to S$ and $\lambda : D \to O$ are the transition and output functions, respectively.

Initially, an FSM is in the initial state s_0. Given a current state $s_i \in S$, when a defined input $x \in I$, such that $(s_i, x) \in D$, is applied, the FSM responds by moving to state $s_j = \delta(s_i, x)$ and producing output $y = \lambda(s_i, x)$. The concatenation of two input sequences α and ω is denoted by $\alpha \cdot \omega$. An input sequence α is a prefix of β, denoted by $\alpha \leqslant \beta$, when $\beta = \alpha \cdot \omega$, for some sequence ω. An input sequence α is a proper prefix of β, denoted by $\alpha < \beta$, when $\beta = \alpha{\cdot}\omega$, for $\omega \neq \epsilon$. The prefixes of a set T of input sequences are denoted by $pref(T) = \{\alpha | \exists \beta \in T, \alpha < \beta\}$. When $T = pref(T)$, it is *prefix-closed.*

An input sequence $\alpha = x_1 \cdot x_2 \cdot ... \cdot x_n \in I^*$ is defined in state $s \in S$ if there are states $s_1, s_2, ..., s_{n+1}$ such that $s = s_1$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq n$. Transition are often represented as a quadruple (s_i, x, y, s_j) with the source state, input, output, and destination states, respectively, as their components; or by directed edges labeled with input and output symbols, i.e., $s_i \xrightarrow{\text{i/o}} s_j$. Transition and output functions are lifted to sequences of input in the standard way. Namely, for the empty input sequence ϵ, $\delta(s, \epsilon) = s$ and $\lambda(s, \epsilon) = \epsilon$. For an input $\alpha{\cdot}x$ defined in state s, we have $\delta(s, \alpha{\cdot}x) = \delta(\delta(s, \alpha), x)$ and $\lambda(s, \alpha \cdot x) = \lambda(s, \alpha)\lambda(\delta(s, \alpha), x)$.

An input sequence $\alpha \in I^*$ is a transfer sequence from s to s' when $\delta(s, \alpha) = s'$. An input sequence γ is a separating sequence for $s_i, s_j \in S$ when $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$. Two states $s_i, s_j \in S$ are equivalent when for all $\alpha \in I^*$, defined in both in s_i and s_j, we have that $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$, otherwise they are distinguishable. An FSM is *complete* when $D = S \times I$, otherwise it is *partial.*

An FSM is *deterministic* when, for each state s_i and input x, there is at most one possible state $s_j = \delta(s_i, x)$ and output $y = \lambda(s_i, x)$. When all states of an FSM are pairwise distinguishable, it is *minimal.* When all states of an FSM are reachable from s_0, it is *initially connected.* When every state is reachable from all states, it is *strongly connected.*

Example 2 (FSM for an AGM product) In Figure 3, we show an FSM for the AGM product derived from the feature constraint $\xi = \mathtt{B} \land \neg\mathtt{S}$. In this product FSM, we have the states $S = \{\textit{Start Game}, \textit{Bowling Game}, \textit{Pause Game}\}$, inputs $I = \{\textit{Start}, \textit{Pause}, \textit{Exit}\}$ and outputs $O = \{\textit{0}, \textit{1}\}$. Transition and output functions are represented by directed edges labeled with input and output symbols. The initial state is indicated by an edge from a black dot.

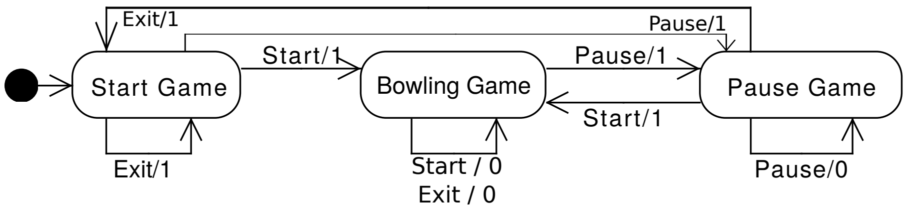

Fig. 3: Example of FSM [58]

In this study, we focus on complete, deterministic, minimal, and initially connected FSMs, which are hereafter referred to as finite state machines. This is a reasonable assumption as FSMs are suitable abstraction models for testing reactive systems [57,26,29] and specifying the semantics of richer notations [59,112]. Furthermore, the ideas surrounding our proposal can be extended to non-connected, non-minimal and non-deterministic product models [134].

Definition 5 (Featured Finite State Machine) An FFSM is a septuple $\langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where: F is a finite set of features, Λ is the set of product configurations, $C \subseteq S \times B(F)$ is a finite set of conditional states, where S is a finite set of state labels, $B(F)$ is the set of all feature constraints, and C satisfies the condition:

$$\forall (s, \phi) \in C,\ \exists \xi \in \Lambda | \xi \vDash \phi \tag{5.1}$$

$c_0 = (s_0, true) \in C$ is the initial conditional state of the FFSM, $Y \subseteq I \times B(F)$ is a finite set of conditional inputs, where I is the finite set of input symbols, O is the finite set of output symbols, and $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying the condition:

$$\forall ((s, \phi), (x, \phi''), o, (s', \phi')) \in \Gamma, \exists \xi \in \Lambda | \xi \vDash (\phi \wedge \phi' \wedge \phi'') \tag{5.2}$$

Conditions (5.1) and (5.2) ensure that all conditional states and transitions are present in at least one valid product of the SPL. A conditional state $c = (s, \phi) \in C$ is alternatively denoted by $s[\phi]$.

A conditional transition $(c, (x, \phi), o, c')$ from conditional state c to c' with conditional input x and output o is alternatively denoted $c \xrightarrow{x[\phi]/o} c'$. The logical operators `and`, `or` and `not` are denoted by the symbols &, |, and ¬, respectively. An omitted condition means that the condition is *true*.

Given an FFSM $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ and a configuration $\xi \in \Lambda$, the product derivation operator Δ_ξ [58] parameterized by the configuration ξ derives a product FSM $\Delta_\xi = (S, s_0, I, O, D, \delta_\xi, \lambda_\xi)$, where: $S = \{s | (s, \phi) \in C \wedge (\phi \vDash \xi)\}$ is the set of states; $s_0 = s$ is the initial state where $(s_0, \phi) = c_0$; and $D = \{(s, x, o, s') | ((s, \phi), (x, \phi'), o, (s', \phi'')) \in \Gamma \wedge \xi \vDash (\phi \wedge \phi' \wedge \phi'')\}$ is the set of completely defined transitions derived from Γ to ξ. The transition and output functions δ_ξ and λ_ξ are defined in terms of the transitions in D.

Example 3 (The Arcade Game Maker FFSM) Figure 4 depicts an FFSM for the AGM SPL. In this example, the conditional state `Save Game`[S] and all conditional transitions reaching or leaving it are implemented by all products implementing feature S. In Figure 3, the FSM is an example of product derived using the configuration $\xi = (AGM \wedge A \wedge M \wedge L \wedge V \wedge Y \wedge P \wedge W \wedge \neg S \wedge \neg B \wedge \neg N)$.

To make FFSMs suitable for model-based testing [124], Fragal, Simao and Mousavi [58] proposed a validation techniques to check if it satisfies the basic properties of FSMs, i.e., determinism, completeness, initially connectedness, and minimality, at the product line level. Added to this, they also show that

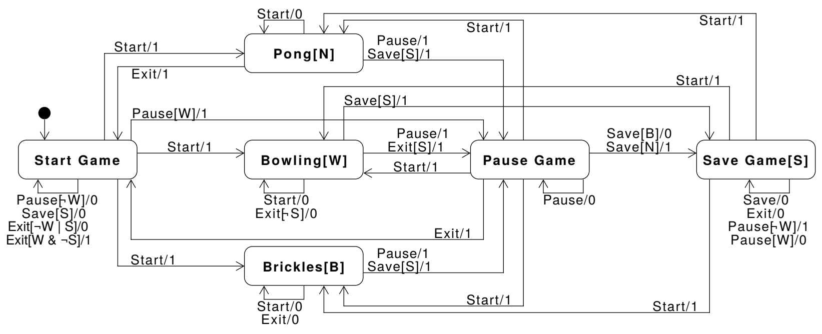

Fig. 4: FFSM of the AGM [54]

the SPL-level validation is sound, i.e., if an FFSM satisfies these properties, so do all the FSM products that can be derived from it.

Recently, FFSMs have been employed to generate configurable test suites that can be pruned using feature constraints and product configurations [56]. The readability of FFSMs also has been improved by grouping up conditional states and transitions into hierarchical entities [55]. Thus, FFSMs have the prospect of serving as a suitable models basis for family-based analysis.

2.2 Structural comparison of state-based models

According to Walkinshaw and Bogdanov [134], structurally comparing two state machines is a difficult task which involves establishing equivalence relationships between states and transitions. To achieve this goal, they proposed LTS_{Diff}, an algorithm to compute the precise difference between two LTSs, a well-known variant of FSM [71]. In this section, we discuss the LTS_{Diff} algorithm in terms of FSMs.

2.2.1 Similarity score

In the LTS_{Diff} algorithm, the differences between two FSM models $M_r = \langle S_r, s_{0_r}, I_r, O_r, D_r, \delta_r, \lambda_r \rangle$ and $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$ are described in terms of states and their surrounding transitions matching input and output symbols. To achieve this, it first calculates the set of matching transitions for all states $a \in S_r, b \in S_u$ using the individual number of pairs of states that can be reached by matching transitions, as follows:

$$\begin{aligned} Succ_{a,b} = \{&(c, d, i, o) \in S_r \times S_u \times (I_r \cup I_u) \times (O_r \cup O_u), \text{ such that} \\ &\delta_r(a, i) = c,\ \delta_u(b, i) = d,\ \textit{and} \\ &\lambda_r(a, i) = \lambda_u(b, i) = o\} \end{aligned}$$

Second, a global similarity score is calculated by aggregating the scores of states connected to the original pair as follows:

$$S^G_{Succ}(a,b) = \frac{1}{2} \frac{\sum_{(c,d,i,o)\,\in\, Succ_{a,b}} (1 + k \times S^G_{Succ}(c,d))}{|\sum^{out}_r(a) - \sum^{out}_u(b)| + |\sum^{out}_r(b) - \sum^{out}_u(a)| + |Succ_{a,b}|}$$

An attenuation ratio k is used to give precedence to state pairs that are closer to the original pair of states and the notation $\sum^{out}_r(a)$ refers to the set of labels of outgoing transitions for state a of M_r. Thus, the expression $|\sum^{out}_r(a) - \sum^{out}_u(b)| + |\sum^{out}_r(b) - \sum^{out}_u(a)|$ denotes the number of outgoing transitions from both states a and b that do not match each other.

Given two FSMs M_r and M_u, the global similarity score $S^G_{Succ}(a,b)$ is used to build a system of linear equations, such that each equation corresponds to the $S^G_{Succ}(a,b)$ for one specific pair of states $(a,b) \in S_r \times S_u$.

The global similarity is calculated both in terms of future behavior (i.e., outgoing transitions) and past behaviors (i.e., incoming transitions). The global similarity score for incoming transitions $S^G_{Prev}(a,b)$ is calculated in a similar manner. Consider the systems of equations for $S^G_{Succ}(a,b)$ and $S^G_{Prev}(a,b)$, the similarity scores for all pairs (a,b) are averaged as follows:

$$S(a,b) = \frac{S^G_{Succ}(a,b) + S^G_{Prev}(a,b)}{2}$$

Example 4 (Illustration of a system of linear equations) In Table 1, we depict the coefficients of our system of equations for the comparison of the two FSMs shown in Figures 3 and 5. State pairs are represented by the first two letters of their respective names.

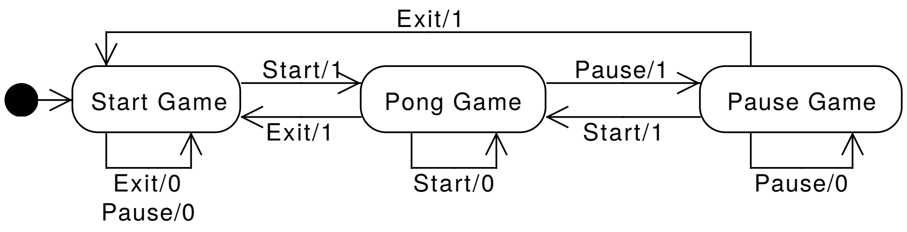

Fig. 5: FSM of an alternative product from the AGM SPL

In the leftmost column, we indicate the respective state pair of each row. In the mid columns, we denote the coefficients for state pairs reachable via matching transitions outgoing from its respective row's state pair. In the rightmost column, we indicate the number of matching transitions for an specific row's state pair. The rightmost value is calculated as the summation of the unitary value in the global similarity score's numerator. A solution for this system of equations indicates the similarity degrees for each state pair.

Pair	(St,St)	(St,Po)	(St,Pa)	(Bo,St)	(Bo,Po)	(Bo,Pa)	(Pa,St)	(Pa,Po)	(Pa,Pa)	#Match
(St,St)	10.0	0.0	0.0	0.0	-0.5	0.0	0.0	0.0	0.0	1
(St,Po)	-0.5	8.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.5	2
(St,Pa)	-0.5	0.0	8.0	0.0	-0.5	0.0	0.0	0.0	0.0	2
(Bo,St)	0.0	0.0	0.0	9.5	0.0	0.0	0.0	0.0	0.0	1
(Bo,Po)	0.0	0.0	0.0	0.0	7.5	0.0	0.0	0.0	-0.5	2
(Bo,Pa)	0.0	0.0	0.0	0.0	0.0	12.0	0.0	0.0	0.0	0
(Pa,St)	0.0	0.0	0.0	0.0	-0.5	0.0	7.5	0.0	0.0	2
(Pa,Po)	-0.5	0.0	0.0	0.0	0.0	0.0	0.0	10.0	0.0	1
(Pa,Pa)	-0.5	0.0	0.0	0.0	-0.5	0.0	0.0	0.0	5.5	3

Table 1: Illustration of a system of linear equations

2.2.2 The LTS_{Diff} algorithm

The comparison of two FSMs is performed in a similar fashion to how we manually navigate in an unfamiliar landscape using a map. In Algorithm 1, we describe this process as proposed by Walkinshaw and Bogdanov [134].

Algorithm 1: The LTS_{Diff} algorithm [134]

```
1  Input: FSM $M_r$, FSM $M_u$, $k$, $t$, $r$;
2  $PairsToScore = computeScores(M_r, M_u, k)$;
3  $KPairs = identifyLandmarks(PairsToScore, t, r)$;
4  if $KPairs == \emptyset$ and $S(s_{0_r}, s_{0_u}) > 0$ then
5      $KPairs = (s_{0_r}, s_{0_u})$;
6  end
7  $NPairs = \cup_{(a,b)\in KPairs} Surr(a,b) - KPairs$;
8  while $NPairs \neq \emptyset$ do
9      while $NPairs \neq \emptyset$ do
10         $(a,b) = pickHighest(NPairs, PairsToScore)$;
11         $KPairs = KPairs \cup (a,b)$;
12         $NPairs = removeConflicts(NPairs, (a,b))$;
13     end
14     $NPairs = \cup_{(a,b)\in KPairs} Surr(a,b) - KPairs$;
15 end
16 $Add = \{b_1 \xrightarrow{a/b} b_2 \in D_u \mid \nexists (a_1 \xrightarrow{a/b} a_2 \in D_r \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs)\}$;
17 $Rem = \{a_1 \xrightarrow{a/b} a_2 \in D_r \mid \nexists (b_1 \xrightarrow{a/b} b_2 \in D_u \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs)\}$;
18 $Kpt = KPairs$;
19 return $(Add, Rem, Kpt)$;
```

First, we compute the similarity scores for all state pairs of the models, as indicated the previous section. Hence, in Line 3, we use a filtering function denoted by *identifyLandmarks()* to select the top t% most equivalent pairs, i.e., those pairs with score above t. If one state is matched to several others, a ratio r includes only those pairs that are at least r times as good as any other match. If no state pair is identified as landmark, then in Line 4-6 the initial states are mapped and selected as initial landmark. Parameters t, k and r can be adapted depending on how many similar transitions the models have. If there are many similar transitions, the threshold should be higher, to make sure that we start the matching process from state pairs that clearly stand out compared to the others pairs.

Second, in Line 7, the algorithm starts from the initial landmarks to find surrounding states reachable via incoming and outgoing transitions matching input/output labels. Once the initial members of the $KPairs$ set are found, we use the $Surr()$ function to search for all surrounding state pairs reachable via incoming and outgoing transitions matching labels. These surrounding states are added to the $NPairs$ set of matched state pairs to be analyzed.

Third, we begin an iterative process where we pick one state pair of highest similarity degrees $(a, b) \in NPairs$, incorporate (a, b) into the $KPairs$ set of common transitions, and remove every state pair conflicting with (a, b) from the $NPairs$ set, i.e., (x, y) is conflicting with (a, b) if $(x, y) = (a, \cdot)$ or $(x, y) = (\cdot, b)$. Each of these steps are shown in Lines 10, 11 and 12, respectively, and repeated until there are no elements left in the $NPairs$ set. In Line 14, we discard all surrounding state pairs reachable via matching transitions. This iterative process is repeated until there are no pairs left in the $NPairs$ set, as indicated between Lines 8-15. At the end of this process, the $KPairs$ set is used to derive the sets of transitions added, removed and kept by checking matching transitions for all states in the $KPairs$ set.

The worst-case complexity for solving this system of linear equations is $O((|S_r| \times |S_u|)^3)$, where $|S_r|$ and $|S_u|$ are the number of states in each of the compared FSM models. However, in practice, the average complexity is lower due to the sparse nature of the produced matrices [134].

2.2.3 Model precision

Originally, the LTS_{Diff} algorithm [134] has been proposed to identify structural differences between two state-based models reverse engineered by model learning algorithms [76,36]. This structural difference is categorized in terms of a confusion matrix [113]. In Table 2, we show the confusion matrix used to compute the structural difference between two FSMs, namely the reference model M_r and the target M_u. These respectively indicate the system's internal behavior and reverse engineered model.

		Target M_u	
		in D_u	not in D_u
Reference M_r	in D_r	$TP = D_r - Rem$	$FN = Rem$
	not in D_r	$FP = Add$	$TN = \emptyset$

Table 2: The confusion matrix to compute the performance metrics for model learning algorithms

In this confusion matrix, the set of true positives TP is derived from the set of correctly learned transitions, i.e., $D_r - Rem$. The set of false positives FP

is denoted by the set of extra transitions *Add*, i.e., those that were incorrectly hypothesized. The set of false negative FN is defined as the set of removed transitions *Rem*, i.e., those that should be in the learned model M_u but are missing. The set of true negatives TN is empty because it refers to impossible transitions, i.e., those that should be in none of the models M_r or M_u.

Based on these sets, performance metrics, such as Precision, Recall and F-measure, can be computed for model learning algorithms [134]. *Precision* tells the proportion of transitions in D_u that are also in D_r, and *Recall* tells the proportion of transitions in D_r that are also in D_u. In Table 3, we show the formula used to calculate the aforementioned performance metrics.

Measure	Formula	Description
Precision	$\frac{\lvert TP\rvert}{\lvert TP \cup FP\rvert}$	Proportion of transitions from M_u that are in M_r
Recall	$\frac{\lvert TP\rvert}{\lvert TP \cup FN\rvert}$	Proportion of transitions from M_r that are in M_u
F-Measure	$\frac{2 \times Precision \times Recall}{Precision + Recall}$	Harmonic mean between Precision and Recall

Table 3: Performance metrics for comparing FSMs

3 Learning family models from product specifications

Family models have been exploited as theoretical foundation for efficient SPL analysis techniques, e.g., model-based testing [21] and model checking [104, 119]. Albeit reasonably efficient, family-based analysis is a challenging task because the creation and maintenance of family models is time consuming and error-prone, especially if there are crosscutting features and large models [107]. Additionally, as requirements change and product instances evolve, the lack of maintenance may render outdated family models [133].

In this section, we introduce the $FFSM_{Diff}$ algorithm, a family model learning technique that builds succinct FFSM models [58, 56] for an SPL by integrating feature model analysis into the process of structural comparison of state-based models [134]. Although our technique is discussed in terms of FFSMs, it can be extended to non-connected, non-minimal and non-deterministic models [134] and other family-based modeling approaches [19], such as FTSs [33, 20], as the FSM notation is a variant of LTS where labels indicate input/output pairs.

The $FFSM_{Diff}$ algorithm allows to (i) learn a new FFSM model from two product FSMs, or (ii) include a product FSM into an existing FFSM. The former approach is applicable when there is no FFSM existing a priori, and the latter if there is a new configuration $\xi_u \notin \Lambda_r$ not included in an FFSM FF_r specifying a set of configurations Λ_r, respectively. In both cases, we assume that the feature model, the FSMs, and the configurations of each product under learning are known a priori. This means that the product FSMs shall be previously hand-crafted or learned using some variant of model learning [7, 110, 41]. Furthermore, the product FSMs shall satisfy the basic properties of testing. These properties are also assumed to be valid for the existing FFSM that will incorporate new product behavior [58].

3.1 The $FFSM_{Diff}$ algorithm

In the $FFSM_{Diff}$ algorithm, we employ the structural comparison of state-based models proposed by Walkinshaw and Bogdanov [134] to match product models. Hence, we learn FFSMs by merging them into an unified family model where differences are indicated by feature constraints. To date, this is the first study to employ automata learning principles for learning family models.

In Algorithm 2, we depict the pseudocode for the $FFSM_{Diff}$ algorithm and discuss the main changes that we incorporated to employ the idea of state-based model comparison as a family model learning algorithm.

Algorithm 2: The $FFSM_{Diff}$ algorithm

```
1  Input: Model $M_r$, Model $M_u$, $k$, $t$, $r$;
2  $PairsToScore = computeScores(M_r, M_u, k)$;
3  $KPairs = identifyLandmarks(PairsToScore, t, r)$;
4  $NPairs = \cup_{(a,b)\in KPairs} Surr(a,b) - KPairs$;
5  while $NPairs \neq \emptyset$ do
6      while $NPairs \neq \emptyset$ do
7          $(a,b) = pickHighest(NPairs, PairsToScore)$;
8          $KPairs = KPairs \cup (a,b)$;
9          $NPairs = removeConflicts(NPairs, (a,b))$;
10     end
11     $NPairs = \cup_{(a,b)\in KPairs} Surr(a,b) - KPairs$;
12 end
13 $Add = \{b_1 \xrightarrow{a/b} b_2 \in D_u \mid \nexists (a_1 \xrightarrow{a/b} a_2 \in D_r \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs)\}$;
14 $Rem = \{a_1 \xrightarrow{a/b} a_2 \in D_r \mid \nexists (b_1 \xrightarrow{a/b} b_2 \in D_u \wedge (a_1,b_1) \in KPairs \wedge (a_2,b_2) \in KPairs)\}$;
15 $Kpt = KPairs$;
16 $FF_{r,u} = \texttt{mergeAndAnnotate}(M_u, M_r, Add, Rem, Kpt)$;
17 return $(FF_{r,u})$;
```

First, to identify the landmarks between product models, we have adapted the *identifyLandmarks*() function from Algorithm 1 to assume the state pair (s_{0_r}, s_{0_u}) as a default landmark. Hence, we search for all pairs likely to be equivalent, given the threshold t and ratio r parameters. The state pairs identified as satisfying the threshold t and ratio r are returned to the set $KPairs$ of common transitions (i.e., commonalities), as shown in Line 3. Since the state pair (s_{0_r}, s_{0_u}) is taken by default as an initial landmark, any pair with s_{0_r} or s_{0_u} is also discarded. Additionally, we eliminated the risk for having an empty $KPairs$ set, as shown in Lines 4-6 of Algorithm 1.

Second, we employ the resulting sets *Add*, *Rem* to identify product-specific states that will receive a presence condition indicating the particular product that it is associated with, and the set *Kpt* is used to indicate the matched states that will be annotated with the conjunction of both simplified configurations. Conditional transitions departing from matching states that also match I/O labels must be unified, otherwise they shall be represented by distinct transitions with their respective simplified configurations. The process of matching and annotating states and transitions is indicated by the `mergeAndAnnotate()` function shown in Line 16. As the LTS_{Diff}, the $FFSM_{Diff}$ also has a (worst-case) complexity of $O((|S_r| \times |S_u|)^3)$ that, in practice, the is often lower due to the sparse nature of the produced matrices.

In the next section, we formally describe how this `mergeAndAnnotate()` process is performed for learning a new FFSM from two products. Afterwards, we extend the formalities for this idea to the task of incorporating new product-specific behavior into an existing FFSM.

3.2 Learning a new FFSM

Let $M_r = \langle S_r, s_{0_r}, I_r, O_r, D_r, \delta_r, \lambda_r \rangle$ and $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$ be the FSMs of two products p_r and p_u that implement configurations $\xi_r = (\bigwedge_{f \in p_r} f) \wedge (\bigwedge_{f \notin p_r} \neg f)$ and $\xi_u = (\bigwedge_{f \in p_u} f) \wedge (\bigwedge_{f \notin p_u} \neg f)$. To learn a new FFSM from M_r and M_u, there are two assumptions: (i) M_r and M_u are FSMs built a priori (e.g., using automata learning [7,125]), (ii) their respective feature model and configurations ξ_r and ξ_u are known a priori. To learn new FFSMs from two product FSMs, we proceed as follows:

Definition 6 (FFSM learned from two configurations) An FFSM learned from $\langle M_r, M_u \rangle$ is a septuple $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where

- $F = (p_r \cup p_u)$ is the set of features implemented by the two products
- $\Lambda = \{\xi_r, \xi_u\}$ is a smallest set composed by the two configurations,
- $C \subseteq (S_r \cup S_u \cup (S_r \times S_u)) \times B(F)$ is the set of conditional states where

$$\begin{aligned}
\forall s_i \in S_r, s_j \in S_u \;&|\; (s_i, s_j) \in KPairs \cdot ((a,b), \xi_r | \xi_u) \in C, \\
\forall s_i \in S_r, \nexists s_j \in S_u \;&|\; (s_i, s_j) \in KPairs \cdot (s_i, \xi_r) \in C, \\
\forall s_j \in S_u, \nexists s_i \in S_r \;&|\; (s_i, s_j) \in KPairs \cdot (s_j, \xi_u) \in C
\end{aligned} \tag{6.1}$$

- $c_0 = ((s_{0_r}, s_{0_u}), true) \in C$ is the initial conditional state,
- $Y \subseteq (I_r \cup I_u) \times B(F)$ is a finite set of conditional input symbols,
- $O = (O_r \cup O_u)$ is the finite set of output symbols
- $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions where

– two transitions $(s_i, x) \in D_r$ and $(s_j, x) \in D_u$ are unified in the same conditional transition if

$$\begin{aligned} \forall (s_i, x) \in D_r, (s_j, x) \in D_u \ \mid \ & \lambda_r(s_i, x) = \lambda_u(s_j, x) = o, \\ & \delta_r(s_i, x) = s_k, \delta_u(s_j, x) = s_l, \\ & (s_i, s_j), (s_k, s_l) \in KPairs \ \cdot \\ ((s_i, s_j), \phi), (x, (\xi_u | \xi_r)), & o, ((s_k, s_l), \phi'')) \in \Gamma \end{aligned} \tag{6.2}$$

– otherwise, for two transitions $(s_i, x) \in D_r$ and $(s_j, y) \in D_u$, there are two independent conditional transitions defined as follows:

$$\begin{aligned} \forall (s_i, x) \in D_r, (s_j, x) \in D_u \ \mid \ & \lambda_r(s_i, x) = o_r, \delta_r(s_i, x) = s_k, \\ & \lambda_u(s_j, y) = o_u, \delta_u(s_j, y) = s_l, \ \cdot \\ ((s_i, \phi_r), (x, \xi_r), & o_r, (s_k, \phi''_r)) \in \Gamma, \\ ((s_j, \phi_u), (y, \xi_u), & o_u, (s_l, \phi''_u)) \in \Gamma \end{aligned} \tag{6.3}$$

Condition 6.1 ensures that product states are either unified into one or two distinct conditional states. These are annotated either with the disjunction or individual configurations, respectively. Condition 6.2 denotes when two transitions shall be unified due to their matching labels and conditional states. Finally, Condition 6.3 describes the case where two transitions cannot be merged and hence, there are two distinct conditional transitions one for each configuration.

To guarantee the mapping between the initial states of the products, we set the state pair (s_{0_r}, s_{0_u}) as the initial conditional state for the learned FFSM model. This state pair also helps to steer the identification of commonalities between product FSMs. To reduce the complexity of feature constraints, the product configurations are simplified by discarding the core features [18] expressed in their associated formulas.

Example 5 (FFSM learned from two product configurations) In Figure 6, we depict a fragment of the FFSM learned by comparing and merging the two product FSMs shown in Figures 3 and 5.

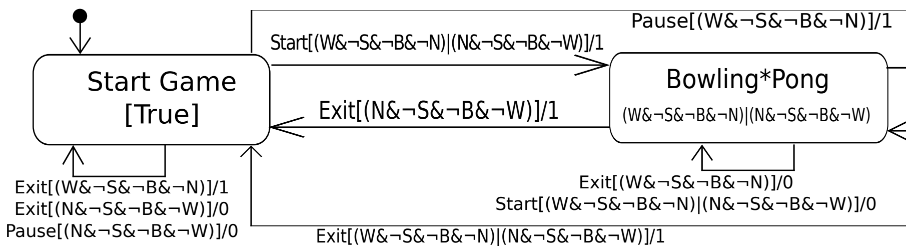

Fig. 6: Fragment of the FFSM learned for the AGM SPL

In this example, the states `Pong Game` and `Bowling Game` were merged into one state `Bowling*Pong` where there is one conditional transition with input symbol `Exit` for each configuration. The feature constraint $(W \wedge \neg S \wedge \neg B \wedge \neg N)$ is an example of simplified configuration for the product in Figure 3.

3.3 Including new product behavior into an existing FFSM

Let the model $FF_r = \langle F_r, \Lambda_r, C_r, c_{0_r}, Y_r, O_r, \Gamma_r \rangle$ be an FFSM learned from a set of product configurations Λ_r. If the FFSM FF_r does not include the behavior of a product FSM $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$ specifying a configuration $\xi_u \notin \Lambda_r$, a new FFSM FF that includes the product behavior from ξ_u can be learned by matching and merging the models $\langle FF_r, M_u \rangle$.

To include a new product into an existing FFSM, we adapted the Definition 6 to compare product models against FFSMs, we introduce another definition of how an existing family model incorporates novel product behavior described in terms of a product FSM. Thus, there are three required assumptions: (i) FF_r and M_u are state machine models built a priori, (ii) configuration ξ_u is known in advance, and (iii) the FSM and FFSM under learning share a feature model that is known a priori. To include a new product-specific behavior into an existing FFSM, we proceed as follows:

Definition 7 (FFSM learned from FF_r and configuration ξ_u)
An FFSM learned from $\langle FF_r, M_u \rangle$ is a septuple $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ where FF_r is a reference FFSM and M_u is the FSM specifying an updated product p_u where

- $F = F_r \cup \{p_u\}$ is the set of features in FF_r and implemented by p_u
- $\Lambda = \Lambda_r \cup \{\xi_u\}$ are the configurations in FF_r and implemented by p_u,
- $C \subseteq (S_r \cup S_u \cup (S_r \times S_u)) \times B(F)$ is the set of conditional states where

$$\begin{aligned} \forall (s_i, \phi_a) \in C_r, s_j \in S_u \;&|\; (s_i, s_j) \in KPairs \cdot ((s_i, s_j), \phi_a | \xi_u) \in C, \\ \forall (s_i, \phi_a) \in C_r, \nexists s_j \in S_u \;&|\; (s_i, s_j) \in KPairs \cdot (s_i, \phi_a) \in C, \\ \forall s_j \in S_u, \nexists s_i \in S_r \;&|\; (s_i, s_j) \in KPairs \cdot (s_j, \xi_u) \in C \end{aligned} \tag{7.1}$$

- $c_0 = ((c_{0_r}, s_{0_u}), true) \in C$ is the initial conditional state,
- $Y \subseteq (Y_r \cup I_u) \times B(F)$ is a finite set of conditional input symbols,
- $O = (O_r \cup O_u)$ is the finite set of output symbols
- $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions where
  - two transitions $((s_i, \phi_i), (x, \phi_r), o, (s_k, \phi_k)) \in \Gamma_r$ and $(s_j, x) \in D_u$ are combined into the same conditional states if

$$\begin{aligned} \forall ((s_i, \phi_i), (x, \phi_r), o, (s_j, \phi_j)) \in \Gamma_r, (s_k, x) \in D_u \;|\; \lambda_u(s_j, x) = o, \\ \delta_u(s_j, x) = s_l, \\ (s_i, s_j), (s_k, s_l) \in \; KPairs \cdot \\ ((s_i, s_j), (\phi | \xi_u)), (x, (\phi_r | \xi_u)), o, ((s_k, s_l), (\phi'' | \xi_u)) \in \Gamma \end{aligned} \tag{7.2}$$

- otherwise, for a conditional transition $((s_i, \phi_i), (x, \phi_r), o_r, (s_k, \phi_k)) \in \Gamma_r$ and a transition $(s_j, y) \in D_u$, there are two conditional transitions defined as follows:

$$\begin{aligned} \forall (s_i, \phi_i) \in C_r, (s_j, x) \in D_u \ \mid \ & \lambda_u(s_j, y) = o_u, \\ & \delta_u(s_j, y) = b_2 \ \cdot \\ ((s_i, \phi_r), (x, \phi_r), o_r, (s_k, & \phi_r'')) \in \Gamma \\ ((s_j, \phi_u), (y, \xi_u), o_u, (s_l, & \phi_u'')) \in \Gamma \end{aligned} \tag{7.3}$$

In addition to the procedure of including new product behavior, we also have extended our $FFSM_{Diff}$ algorithm [42] to identify the sets of transitions added, removed and kept. Thus, we can quantify the behavioral overlap between an updated product model M_u and a reference FFSM FF_r. To identify the amount of behavioral overlap, we use the concept of precision between two models M_u and FF_r in terms of their sets of common transitions [134].

Family model learning has been proposed as an approach to build feature finite state machine models and learn presence conditions indicating feature-specific and product-specific behavior in terms of conditional states and transitions [42]. Since family models are expected to represent all product line variants within the same artifact [107], the most straightforward method should be to analyze all valid configurations in a brute-force fashion. However, this is only feasible for product lines with not too many members [120]. To address this issue, we present an approach that employs product sampling on family model learning.

4 Incorporating Product Sampling into Family Model Learning

Analysing an SPL on a product-based basis is very demanding and cumbersome as there is a huge number of possible product configurations. Thus, we propose to incorporate product sampling into the family model learning so it can be ran without the need for exhaustive learning.

For large SPLs, a typical software analysis approach is to sample product configurations such that reasonable statements on the behavior of the entire product line are possible [120,127]. Product sampling techniques, such as T-wise [95,68], shall collectively cover the behavior of a product line. Hence, they should pave the way for learning family models with reasonable precision and cost lower than exhaustive analysis.

In our approach, we assume there is an arbitrary product sampling technique *sample*() that generates a minimal subset of configurations C_{smpl} to be considered during the sampling process, such as the Chvatal algorithm [68]. In Algorithm 3, we depict our approach for learning family models by sampling.

Let $C_{smpl} = \{\xi_1, \xi_2, \ldots, \xi_n\}$ be the list of sampled product configurations under learning by an arbitrary product sampling technique, referred to as *sample*(), In the family model learning by sampling process, we start by first

Algorithm 3: Learning Family Models by Sampling Configurations

```
1  Input: Feature model $FM$ and Learning parameters $k, t, r$;
2  Output: Family model learned by sampling $FF_n$;
3  $C_{smpl} = \{\xi_1, \xi_2, \ldots, \xi_n\} = sample(FM)$; // List of sampled configurations
4  $M_1 = buildFSM(\xi_1, FM)$; // Build a product FSM $M_i$ for $\xi_i$
5  $M_2 = buildFSM(\xi_2, FM)$;
6  $FF_1 = FFSM_{Diff}(M_1, M_2, k, t, r)$; // Learn new FFSM
7  foreach $j \in \{2, \ldots, n-1\}$ do
8      $M_{j+1} = buildFSM(\xi_{j+1}, FM)$;
9      $FF_j = FFSM_{Diff}(FF_{j-1}, M_{j+1}, k, t, r)$; // Learn partial family model
10 end
11 $return(FF_n)$;
```

building a new family model FF_1 for the two product models M_1 and M_2. Second, given an initial FFSM FF_1, the learning by sampling enters into an iterative stage where novel product-specific behavior expressed in terms of a state machine M_{j+1} is included in a partial family model FF_j. This family model is said to be *partial* as it describes only a subset of valid product instances. Again, product FSMs M_{j+1} may be nonexistent and hence, $buildFSM()$ may be required to build FSMs. In the $buildFSM()$ step, product-specific FSM models can be either hand-crafted or built by means of automata learning [7, 125]. At the end of this iterative stage, a family model FF_n learned from all product configurations $\xi_i \in C_{smpl}$ is constructed. To evaluate the benefits of the product sampling criteria, in the next section we present few experiments to quantify the precision of such models learned by sampling.

5 Empirical evaluation

Several studies have underscored the importance of feature interaction coverage in product sampling [127]. Thus, we designed a set of experiments to analyze our family-based learning technique with the purpose of evaluate its effectiveness in learning succinct models using exhaustive analysis. Hence, we extended our investigation to evaluate whether feature interaction coverage metrics can alleviate the cost of family model learning and collectively cover the behavior of SPLs. Particularly, we applied the T-wise coverage criteria and analyzed the precision of models learned by sampling [95, 68].

In this section, we present the context of our experiment, selected variables, formulated hypotheses, experiment design, and subject systems. Next, we present the analysis and interpretation of results and threats to validity for our empirical evaluation. We close this section discussing the implications and limitations of our study. This section is organized based on recommendations by Wohlin et al. [136]. For the sake of reproducibility, we have made a web page describing the artifacts (e.g., source-code, scripts, FFSMs, FTSs, FSMs, feature models) used and generated in this study available at `https://damascenodiego.github.io/learningFFSM/`. The repository has been structured based on recommendations by Mendez et al. [89].

5.1 Methodology

According to Thüm et al. [120], the effectiveness of family-based analysis should be mainly influenced by the number of features, the size of feature implementations (e.g., modeling, coding artifacts) and the amount of reuse among configurations, rather than the number of valid configurations [120]. Therefore, for our technique to qualify as an effective family-based learning technique, we expect to learn *succinct* FFSMs where states and transitions are annotated with simplified configurations.

By succinct, we mean that the FFSMs learned are smaller than the products under learning and hand-crafted models, especially if there is high feature sharing. By simplified, we mean that product configurations are modified by discarding core features from feature constraints using SAT solvers [79].

Additionally, we expect that family models learned by sampling product configurations shall collectively cover the behavior of a product line and be at least as precise as those models recovered by exhaustive analysis. Thus, we designed a set of experiments to measure the succinctness and precision the learned family models and answer our research questions. In Table 4, we present our hypotheses about each proposed research question.

RQ	Hypotheses	Description
RQ1	H_0^{RQ1}	The size of learned FFSMs is equal to the total size of the pairs of products under learning
	H_1^{RQ1}	The size of learned FFSMs is smaler than the total size of the pairs of products under learning
RQ2	H_0^{RQ2}	The learned FFSMs are larger than hand-crafted models
	H_1^{RQ2}	The learned FFSMs have at most the same size as hand-crafted models
RQ3	H_0^{RQ3}	The size of learned FFSMs is not influenced by configuration similarity
	H_1^{RQ3}	The size of learned FFSMs is influenced by configuration similarity
RQ4	H_0^{RQ4}	The FFSMs learned by sampling configurations are less precise than those learned by exhaustive analysis
	H_1^{RQ4}	The FFSMs learned by sampling configurations can be as precise as those learned by exhaustive analysis

Table 4: Hypotheses

As a measure of *succinctness*, we used the size of the FFSMs learned from product pairs. We describe *size* in terms of *number of transitions* as it is one of the factors that influence the complexity of model-based techniques [26, 14] and that is used to interpret the language and structure of FSMs [134]. To complement our analysis, we also measured the number of states.

To measure the *statistical significance*, we used the Mann-Whitney test to check if there was significant difference ($p < 0.05$) between the sizes of the learned FFSM and the reference model, i.e., the product pair or the hand-crafted family model. To measure the *scientific significance* [69,10], we used the Vargha-Delaney's $\hat{A}$ effect size [126,135] to assess the probability of the learned FFSMs being more succinct than the reference model. If $\hat{A} < 0.5$, the learned FFSM is smaller than the pair of products. If $\hat{A} = 0.5$, they have equivalent sizes. To categorize the magnitude of the $\hat{A}$ effect size, we used the intervals between $\hat{A}$ and 0.5 implemented in the `effsize` package [62,123]: $\textit{negligible} < 0.147 \leq \textit{small} < 0.33 \leq \textit{medium} < 0.474 \leq \textit{large}$.

As a measure of *configuration similarity*, we applied the Hamming distance between product configurations with respect to normalized number of common selected and unselected features [4]. Thus, we analyzed the impact of configuration similarity on family model succinctness by calculating the Pearson's correlation coefficient between the ratio of the size of the learned FFSM with respect to the total size of the product pairs, on one hand, and the similarity between configurations, on the other hand.

As a measure of *precision*, we used the concept of model precision proposed by Walkinshaw and Bogdanov [134] for evaluating the performance of reverse engineering techniques. Inspired by their observations, we gradually changed our parameter values to constraint the possibilities of matches and applied the same value for all product lines. If product states were too homogeneous, with many similar transitions, we increased the thresholds to make sure that we started from state pairs that clearly standed out, as Walkinshaw and Bogdanov [134] indicate. Hence, we set the learning parameters for the attenuation ratio as $k = 0.5$, the threshold of most equivalent pairs as $t = 0.4$ and the ratio for best matches as $r = 1.4$.

5.2 Experiment Design

To answer **RQ1** and **RQ2**, we implemented the $\textit{FFSM}_{\textit{Diff}}$ algorithm on top of the LearnLib framework [67] for dealing with the state machine models, the SAT4J solver [79] for feature model analysis, the FeatureIDE [121] library for product sampling and configuration similarity, and the Apache Commons Mathematics library [8] for solving the systems of linear equations. We used the $\textit{FFSM}_{\textit{Diff}}$ implementation to combine the FSM models into FFSMs for all pairs of product configurations. Then, we checked whether there were significant and relevant differences between the sizes of the learned FFSM, the pair of products under learning and the hand-crafted models. In Figure 7, we illustrate our experiment to answer the RQ1 and RQ2.

To answer **RQ3**, we normalized the size of the learned FFSMs by the total size of the product pairs to the interval between 0.5, if both product FSMs are equivalent; and 1.0, otherwise. Based on this normalized size, we calculated the Pearson's correlation coefficient between the normalized size of learned FFSMs and configuration similarity to measure the impact of similarity between product configurations on the size of learned family models. For the statistical analysis, we used the R statistical package [101].

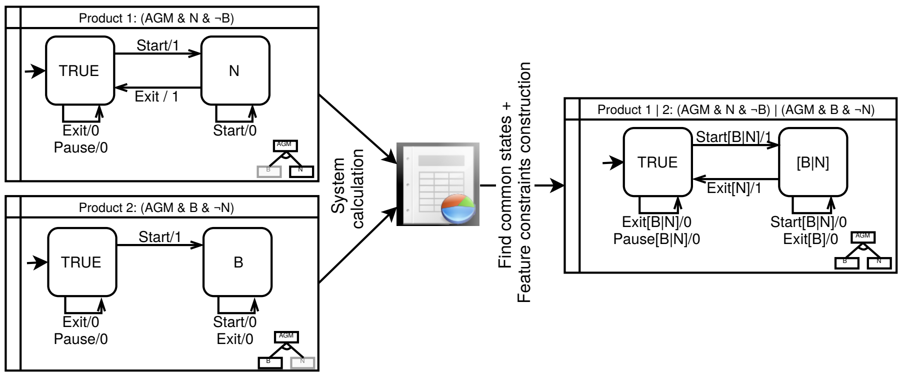

Fig. 7: Experiment design - Learning FFSMs from product pairs

To answer **RQ4**, we used the FeatureIDE workbench [121] to generate subsets of valid products satisfying the feature-wise (aka. 1-wise), pair-wise (aka. 2-wise), 3-wise, 4-wise and all-valid configurations criteria. Particularly, we used the Chvatal algorithm [31] to perform T-wise product sampling [68] that is available in the FeatureIDE workbench [121]. In Figure 8, we illustrate our experiment to answer the RQ4.

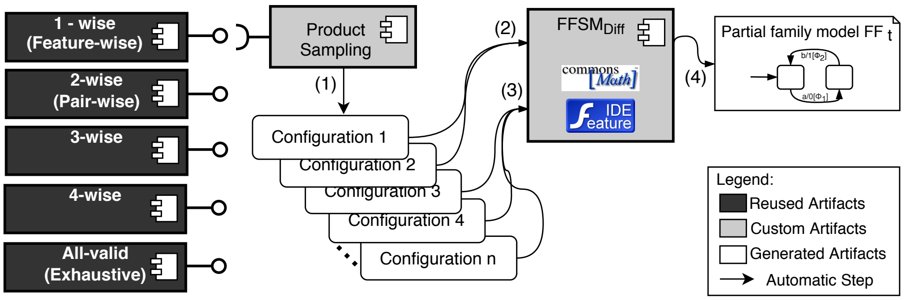

Fig. 8: Experiment design - Learning FFSMs by product sampling

Let $\{\xi_0, \xi_1, \ldots, \xi_m\} \subseteq B(F)$ be a subset of valid configurations generated by some arbitrary sampling criteria, such that they are sorted by configuration similarity [4]. For each sampled subset, we iteratively learned partial FFSMs by merging the FSMs of the configurations $\bigcup_{i=0}^{j-1}(\xi_i)$ with its next configuration ξ_j. To evaluate the precision of the partial family model learned by sampling, we used the $FFSM_{Diff}$ to measure how many transitions from all valid products were included into the FFSMs learned by sampling. The same t, k, r parameters used for learning family models were taken to calculate the precision of models learned by sampling.

5.2.1 Subject systems

In order to evaluate our hypotheses, we searched in the literature of model-based SPL analysis, existing open source projects, and benchmarks for subject systems accompanied by 1) a feature model, 2) models of individual products, and preferably, 3) a behavioral family model. Items 1 and 2 form the basis for the application of our technique and item 3 was used to evaluate our learning technique against the provided models.

We selected 105 Mealy machines derived from six abstract representations of SPLs [58,32,105,46,45]. While one of these abstract representations of SPLs has been already made available as a set of FSMs [58], the other five sets of FSMs had to be hand-crafted from LTS models instantiated from academic benchmarks of SPLs [32,105,46,45]. In Table 5, we present the SPLs in terms of numbers of features, valid configurations, and total of states and transitions in its family model.

SPL		Feature model		Family model	
ID	Name	Features	Valid conf.	States	Transitions
AGM	Arcade Game Maker	13	6	6	35
VM	Vending Machine	9	20	14	197
WS	Wiper System	8	8	13	112
AEROUC5	Aero UC5	7	9	25	450
CPTERMINAL	Card Payment	13	30	11	176
MINEPUMP	Minepump	9	32	25	575

Table 5: Description of the SPLs under learning - Feature and family models

To instantiate these product FSMs, we used the VIBeS tool [44] to derive LTSs for every valid product of each SPL. For each LTS state, we created one FSM state. For every valid input of an LTS state, we added an FSM transition returning `1`. For every missing transition, we added a self-loop transitions returning `0`. This process was hand-crafted by the first author of this paper, who has former experience in modeling software systems for model-based testing [39,40] and automata learning [41]. Moreover, this process was partially automated using Bash and Python scripts that are included in our lab package.

Although these are not fully realistic systems, we believe these academic benchmarks are more representative than random models, which may constitute very rare and special cases in which our techniques do not perform well. They comprise many non-trivial aspects, such as the existence of infinite behaviour, states with similar or identical behaviour in different products [42] and distinct input alphabets [32] that can make family model learning more difficult. Additionally, these models constitute widely used benchmarks for family-based analysis techniques [34,12,33,21,20,45,118]. Since the AGM has been discussed before, in the next sections, we briefly present the other five subject systems used in this study.

The Vending Machine SPL

The Vending Machine (VM) is an SPL that we hand-crafted [42] based on LTSs derived from a collection of illustrative examples of FTS models [32]. In Figure 9, we depict the VM feature model.

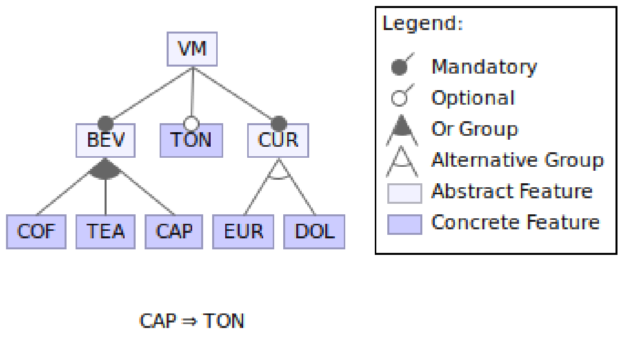

Fig. 9: The VM feature model

In the VM SPL, product instances shall feature at least one and at most three beverages (i.e., *Coffee* - `COF`, *Tea* - `TEA`, and *Cappuccino* - `CAP`), they support one currency (i.e., *Dollar* - `DOL` or *Euro* - `EUR`) and can play one optional *Ringtone* - `TON`. The VM SPL constitutes an interesting case as it can derive FSMs with distinct input alphabets and languages. Among the main characteristics of the derived product FSMs, we highlight two main differences: the possibility to add extra states for each beverage; and changes in the valid input symbols of outgoing transitions departing from the initial state depending of the supported currency. Finally, the VM SPL also shows a "requires" relationship explicit in the feature model as its corresponding propositional formula.

The Wiper System SPL

The Wiper System (WS) is another SPL that we hand-crafted [42] based on models from the same collection of SPLs aforementioned [32]. In Figure 10, we depict the feature model of the WS SPL.

Our WS SPL has two subsystems: the `Sensor` to detect rain and the `Wiper` itself; available in two qualities, namely, high and low; and one optional feature for permanent movement `PermanentWiper`. A high quality sensor `sHigh` can discriminate between heavy and light rain, whereas a low quality sensor `sLow` can only distinguish between rain and no rain. Similarly, the `wHigh` and `wLow`

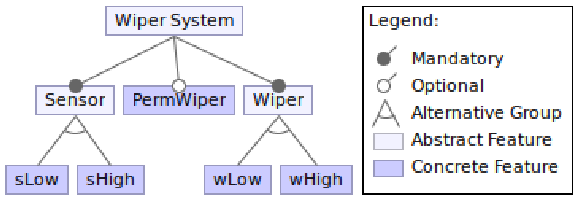

Fig. 10: The WS feature model

quality wipers can operate at two and one speeds, respectively. Each of these features lead to significant changes in the structure and language of its derived product FSM models.

The Aero UC5 SPL

The Aero UC5 (AEROUC5) model has been originally presented by Samih et al. [105] as a set of extended Markov models designed by engineers. It is an industrial situational awareness system for helicopters flying in degraded visual environments that has been employed as a benchmark in SPL research studies [46,45]. The AEROUC5 feature model has been originally composed by 25 features and more than 5 million valid configurations [130].

We adapted the AEROUC5 SPL because there were only four features that were used in the behavioural model of the products [130]. Hence, the original model had a huge amount of identical product models. We have thus restricted the feature model to only those four concrete features used in the behavioural models as shown in Figure 11.

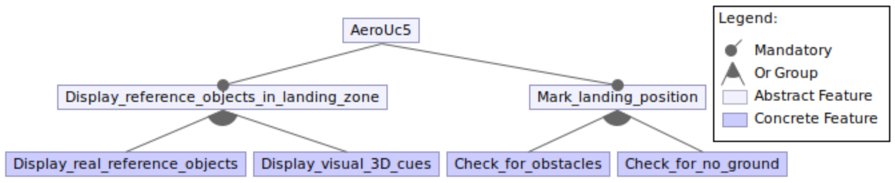

Fig. 11: The Aero UC5 feature model

Our adapted version of the feature model is composed by four features related to displaying (i) real object or (ii) 3D conformal visual cues on a head-tracked Helmet, and marking (iii) intended landing positions or (iv) obstacles on ground using an Obstacle Warning System. This SPL is intended to be a more realistic subject as it has been designed by engineers and is one of our largest behavioral model in terms of number of states and transitions.

The Card Payment Terminal SPL

The Card Payment Terminal (CPTERMINAL) is another SPL originally designed as an FTS [46,45]. In Figure 12, we depict the feature model for the Card Payment Terminal product line.

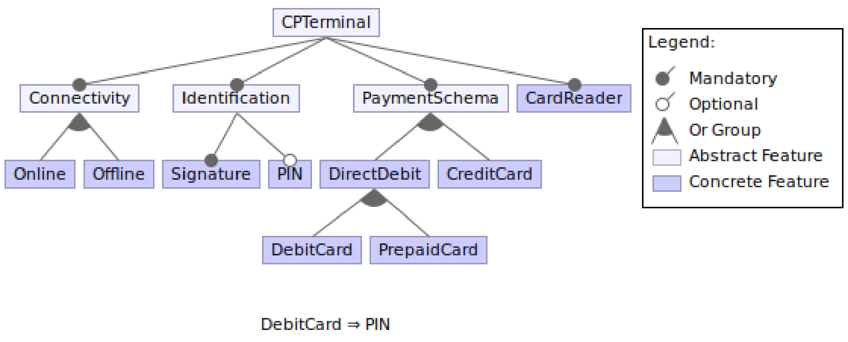

Fig. 12: The Card Payment Terminal feature model

This product line has been defined by a software engineer based on EMV and PCI norms [46,45]. The Card Payment Terminal FTS describes the behavior of one terminal that accepts card payment with `DirectDebit` and/or `CreditCard`. It accepts a card owner authentication method (i.e., `Signature` and optionally `PIN` code), and with a synchronous (`Online`) or asynchronous (`Offline`) connection to the payment service [131]. The CPTERMINAL SPL also includes a "requires" relationship explicit as its corresponding propositional formula. To derive FSMs, we have used the same approach applied to the AEROUC5 SPL.

The Minepump SPL

The Minepump (MINEPUMP) product line has been presented by Classen et al. [33]. The purpose of this system is to keep a mine shaft clear of water while avoiding the danger of methane related explosions. In Figure 13, we show the feature model for the Minepump SPL.

It monitors the mine shaft using the `WaterRegulator` and `MethaneDetect` features. The system is activated once the water level reaches a preset threshold, but only if the methane is below a critical limit. Similarly to the AEROUC5 and CPTERMINAL SPLs, the FSMs for the Minepump SPL were derived from an FTS model [132].

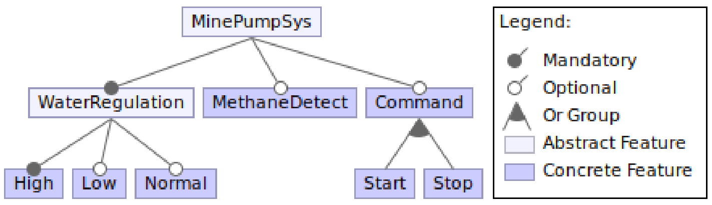

Fig. 13: The Minepump feature model

5.3 Analysis of Results

In this section, we discuss the main results of our experiments in terms of the four defined RQs and the Hypotheses shown in Table 4. For the sake of space, we will only plot and highlight the main findings of our experiments. The full set of plots and tabulate results are available in our online repository under the **EMSE** tag[1]. In the boxplots, the red dashed lines indicate the number of transitions or states in the products under learning and the original hand-crafted model.

*5.3.1 **RQ1** – Is our approach effective in learning succinct family models with respect to the total size of the products under learning?*

Regarding the succinctness of the learned FFSMs, we observed that on average all learned FFSMs presented fewer transitions than their respective pairs of products under learning. In Figure 14, we show boxplots for the sizes of the learned FFSMs and the total size of the pairs of products under learning in terms of number of transitions. The number of transitions of the original hand-crafted family model is indicated by a red dashed line.

In terms of number of states, we also found that the learned FFSMs had fewer states than their pairs of products under learning. Figure 15 shows the boxplots for the numbers of states in the learned FFSMs and the total number of states in the pair of products under learning.

To assess the statistical difference and significance of our results, we ran the Mann-Whitney test and Vargha-Delaney's $\hat{A}$ effect size to check the significance ($p < 0.05$) and magnitude of the difference between the sizes of the learned FFSMs and the pairs of products under learning. In Table 6, we present the p-values and effect sizes comparing the sizes of our learned family models against the pairs of product under learning in terms of states and transitions.

As indicated by Figures 14 and 15, as well as by Table 6, there were statistically significant differences between the sizes of the learned FFSMs and the pair of products under learning. For the effect sizes, we also found that the differences had *large* magnitude. Thus, our results support the hypothesis

[1] `https://github.com/damascenodiego/learningFFSM/releases/tag/EMSE`

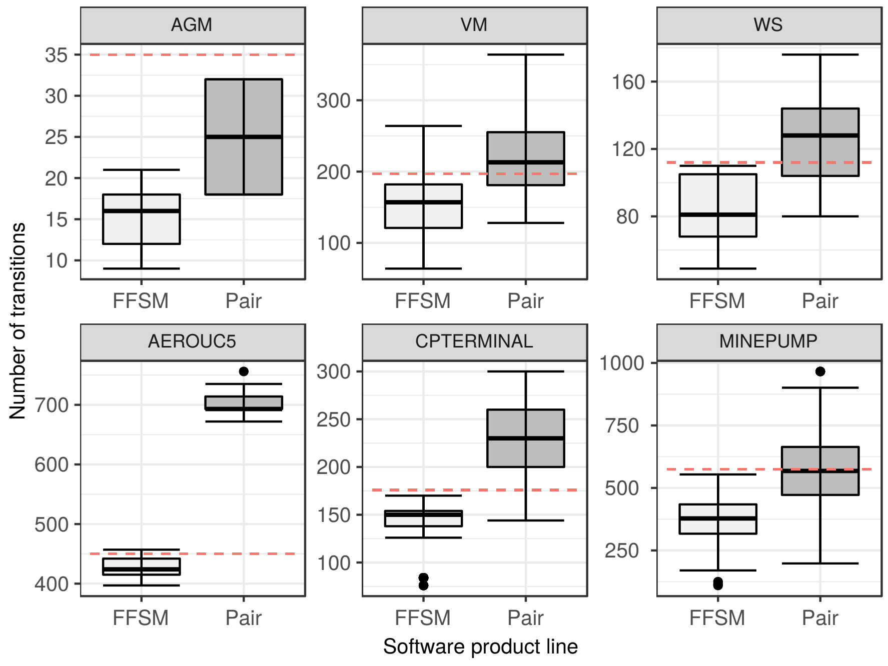

Fig. 14: Number of transitions in the learned FFSMs and pairs of products

# of	Test	AGM	VM	WS	CPTERMINAL	MINEPUMP	AEROUC5
States	MW	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
	VD	0	0.037	0.050	0	0.005	0
Transitions	MW	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
	VD	0.075	0.174	0.146	0.032	0.120	0

VD: Vargha-Delaney's effect size **MW**: Mann-Whitney test

Table 6: Results for the Mann-Whitney test and Vargha-Delaney's effect size: Learned FFSM vs. Product pair

H_1^{RQ1} that the sizes of learned FFSMs is at most equal to the total size of products under learning.

5.3.2 ***RQ2*** *– Is our approach effective in learning succinct family models with respect to the total size of the hand-crafted models?*

To evaluate the succinctness of the learned FFSMs, we also compared the size of hand-crafted models against the FFSMs learned from pairs of products. In Figures 14 and 15, the size of the original hand-crafted FFSM in terms of number of transitions and states is indicated by red dashed lines.

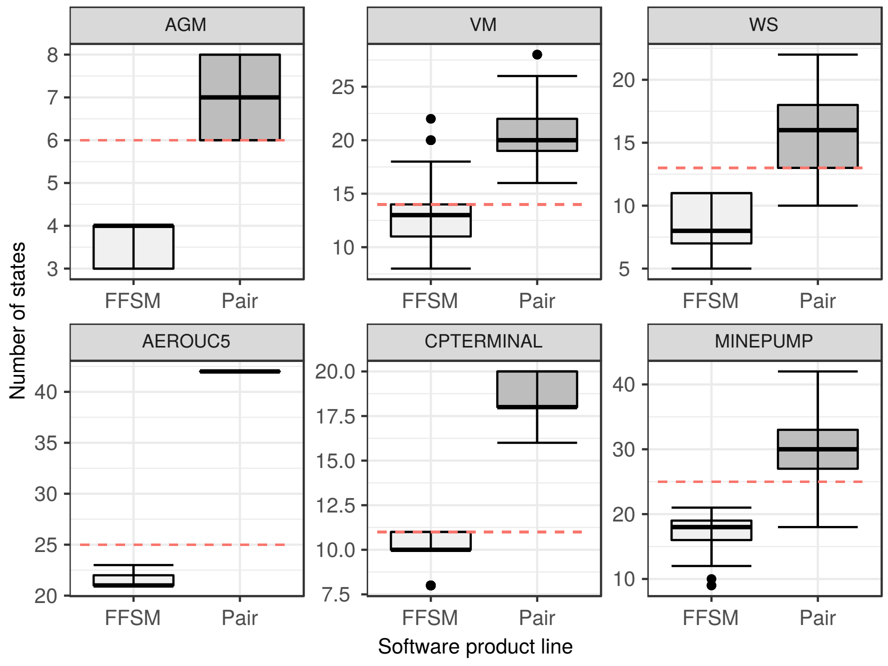

Fig. 15: Number of states in the learned FFSMs and pairs of products

To compare the sizes of the hand-crafted models and the FFSMs learned from product pairs, we used the Mann-Whitney test and $\hat{A}$ effect size. Table 7 shows the results for the Mann-Whitney test and effect size comparing the size of the learned FFSMs against the size of hand-crafted models.

# of	Test	AGM	VM	WS	CPTERMINAL	MINEPUMP	AEROUC5
States	MW	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
	VD	0	0.35	0	0.22	0	0
Transitions	MW	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$	$< .001$
	VD	0	0.09	0	0	0	0.13

VD: Vargha-Delaney's effect size **MW**: Mann-Whitney test

Table 7: Results for the Mann-Whitney test and Vargha-Delaney's effect size: Learned FFSM vs. Hand-crafted model

By analyzing the results of the Mann-Whitney test, we found statistically significant differences ($p < 0.01$) between the sizes of FFSMs learned from all SPLs. The Vargha-Delaney's effect sizes indicated differences of *large* magnitude where FFSMs learned from product pairs included fewer transitions than their hand-crafted versions. These findings persisted for the number of states, except for the VM SPL where we found a small magnitude on the difference between the number of states of the FFSM models learned from product pairs. Thus, our results support the hypothesis H_1^{RQ2} that learned FFSMs have at most the same size as hand-crafted FFSMs.

5.3.3 ***RQ3*** *– Is the size of learned family models influenced by the configuration similarity degree of the products under learning?*

In addition to comparing the size of learned FFSMs against the size of products under learning, we analyzed the relationship between learned family model size and configuration similarity using the Pearson's correlation coefficient. In Figure 16, we show scatter plots for the configuration similarity degree against the size of learned FFSMs for all pairs of products of each SPL.

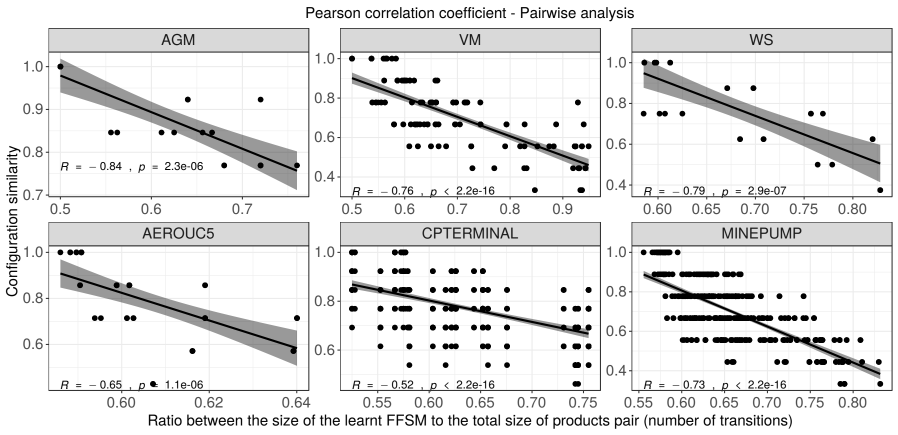

Fig. 16: Scatter plots for the relationship between the normalized size of the learned FFSM and configuration similarity

A configuration similarity equal to 1.0 means that both products have the same feature configuration. A ratio between the size of learned FFSM and total size of products equal to 0.5 means that the products analyzed implement equivalent behavior, otherwise they have some variability expressed by mismatching transitions.

By analyzing the Pearson correlation coefficient, we found *strong* negative correlations between FFSM size and configuration similarity for the VM, WS, AEROUC5 and MINEPUMP product lines; *very strong* negative correlation for the AGM product line; and *moderate* negative correlation for the CPTERMINAL. These results indicate that FFSMs learned from product models with high configuration similarity tend to be smaller than those built from products implementing distinct sets of features. Therefore, our results support the hypothesis H_1^{RQ3} that the size of FFSMs is influenced by configuration similarity and our approach can exploit common features and produce more succinct FFSM models when these are prone to behavioral similarity.

5.3.4 ***RQ4*** *– Is our approach effective in learning precise family models compared to those obtained by exhaustive analysis?*

For each $T \in \{1, 2, 3, 4\}$, we have used the T-wise sampling criteria to generate subsets of valid product configurations and learn FFSM models by sampling. To evaluate the precision of learning by sampling, we used the all-valid configurations criteria to derive all product FSMs and build reference FFSMs for each SPL. In Table 8, we depict the sizes of the subsets of products generated by each configuration sampling criteria.

SPL	Size of the sampled subset generated by T-wise				
	Feature-wise	Pair-wise	3-wise	4-wise	All-valid
AGM	3	6	6	6	6
VM	2	6	13	19	20
WS	2	5	8	8	8
AEROUC5	3	6	9	9	9
CPTERMINAL	3	8	16	24	30
MINEPUMP	3	7	13	24	32

Table 8: Number of configurations in the subsets generated by each criteria

To evaluate the precision of the models *learned by sampling*, we have used the $FFSM_{Diff}$ to measure the proportion of transitions from the analyzed models (i.e., learned by sampling) that are also in the reference models (i.e., individual FSMs of all valid products). Thus, a precision equals to 1 indicates that all transitions from all valid products are included into the FFSM learned by sampling. Figure 17 shows the precision of the FFSM learned by each sampling criteria compared against the full set of models from all valid products.

As our results indicate, model precision turned out to be higher for larger values of T. For most of the product lines, excluding the AGM, we found a significant difference between the models learned by feature-wise sampling and all-valid configurations, i.e., exhaustive analysis. By comparing exhaustive analysis against feature-wise sampling, we found effect sizes categorized as medium to large with the exhaustive criteria reaching higher precision.

Higher interaction strengths are known by their improved fault detection capabilities [115, 96]. Similarly, our results corroborate to these findings as they indicate that family models learned by 3- and 4-wise sampling tend be more precise than those built by feature-wise and pairwise sampling.

As shown in Table 8, the 3- and 4-wise sampling criteria generated the same number of configurations as the exhaustive criteria for the AGM, WS and AEROUC5 product lines. Thus, similar precision levels should be expected.

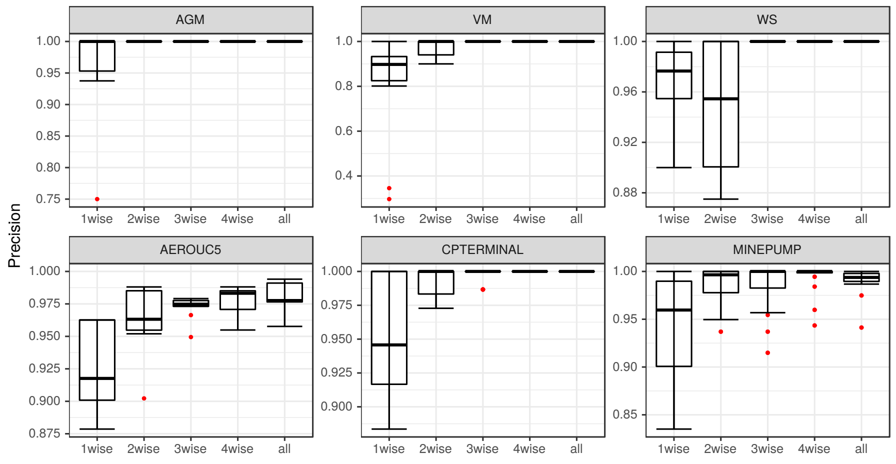

Fig. 17: Model precision by sampling criteria

The results for the Mann-Whitney and Vargha-Delaney's tests corroborate these findings as they indicated no significant difference between the precision of models learned by 3-wise, 4-wise and all-valid sampling criteria.

For the VM, CPTERMINAL and MINEPUMP product lines, we found that models learned by 3-wise and 4-wise sampling reached precision levels similar to those learned by using the exhaustive criteria. For these product lines, we found either no significant differences or effect sizes categorized as negligible to small between the models learned by the 3-wise, 4-wise and all-valid sampling criteria. These findings indicate that product sampling can be helpful at reducing the costs for recovering family models from product families without analysing all-valid products.

For the AEROUC5 and MINEPUMP product lines, we found that FFSMs learned by exhaustive analysis did not reach precision levels equal to 1. We associate this to a possibly high number of state pairs with equal scores returned by the *identifyLandmarks*() function. Thereafter, multiple possible maps between states pairs were found byour algorithm where the selected pairs deemed some transitions as removed and affected precision. These results support our hypothesis H_1^{RQ4} that FFSMs learned by sampling can be at least as precise as those learned by running exhaustive analysis.

5.4 Threats to validity

In this section, we discuss the threats to validity of the methods used in this paper. To do this, we follow the recommendations by Wohlin et al. [136].

Conclusion validity: These threats concern the relationship between treatment and outcome of our investigation. To avoid the risk of violating assumptions of statistical tests, we have opted for the Mann-Whitney nonparametric statistical test. Despite these actions, there are still threats to conclusion validity due to the risk of random heterogeneity in our subject systems as these are academic models in their majority.

External validity: These concern the generalization of our results to industrial SPLs. Our results are based on six subjects, of which one of them has been inspired by a real system [105]; the small number of real product lines and the fact that most feature models did not have complex constraints pose a threat to external validity. Another variable that will form a threat to external validity is the variability inherent to the valid products of our subject systems. For some of our SPLs, the behavioral difference between products made the exhaustive analysis the only criteria able to recover family models fully precise. In these cases, sampling techniques may not be applicable and hence, configuration prioritization [61] may be required. The impact of prioritization techniques in family model learning is out of the scope of this study.

Internal validity: These threats concern issues that may indicate a causal relationship, when there is none. As the validity of experiments is highly dependent on the reliability of the measures and treatment implementation, we designed our experiments on top of three widely used tools for state-machine learning [99], SAT solving [79], and SPL analysis [121]. The number of product models and the diverse characteristics in the academic benchmarks used in our study support that the internal validity of our results is good.

Construct validity: These are concerned the ability to draw correct conclusions about the treatment and outcomes. Two factors that will form threats to construct validity are the nature of the hand-crafted FFSMs used as ground-truth models and the subsets of product configurations sampled using T-wise criteria. Highly experienced modellers will be able to produce more concise representations and subsets of product configurations better than professionals with less experience. In addition to that, configuration subsets sampled by T-wise criteria may be still large, compared to the set of all-valid products. In these cases, domain-specific expertise may be useful to optimize family model learning. The fact the modeller in our case was an expert both in SPL and in the formal modelling language, mitigates the risk for our results.

5.5 Discussion

What are the implications for practitioners and researchers? While exhaustive learning may be suitable for small product lines, in large SPL projects, it becomes impractical. Our proposal aims to recover domain-level artifacts (i.e., family models) from application-level artifacts (i.e., finite state machines). Thus, we believe that our technique can enable model-based analysis techniques, such as regression testing, performance analysis, and product sampling, to cases where family models are missing or incomplete. To employ our technique, we expect engineers to have skills on model learning, reverse engineering, and feature model analysis.

In regression testing, family model learning could be employed to support test suite optimization and reduce the potentially large number of test cases generated from product-based techniques [56]. In performance analysis, family model learning could be employed to support non-functional testing by incorporating conditional probabilities for family-based probabilistic model checking [128] and conditional time guards for stochastic real-time analysis of software product lines [86]. In product sampling, iterative techniques, such as IncLing [3], could incorporate partial family models to check for feature interactions, e.g., by testing if richer product variants subsume the behavior/properties of its constituents without unexpected behavioral changes, i.e., interaction problems [127].

What types of systems may it work/not work? In our learning by sampling technique, there is an assumption that sampled products shall collectively cover the behavior of product families and have their models specified a priori. However, if there is no such behavioral overlap, then products learned by sampling may never be precise enough and exhaustive learning should be required. If that is the case, an iterative sampling process could be employed for prioritizing product configurations for learning novel and unseen behavior and testing, if a partial family model already includes the behavior of a given product.

Regarding the size of product models, the worst-case complexity indicates a cubic growth in the cost for learning. However, due to the sparse nature of the produced matrices, the $FFSM_{Diff}$ algorithm tends to scale well. Figure 18 shows the distribution of times required to learn an FFSM for all pairs of product models. The average time to learn models from products with total size of 40 states lies below 3000 milliseconds and these results corroborate the results by Walkinshah and Bogdanov [134] where their algorithm was comparatively cheap.

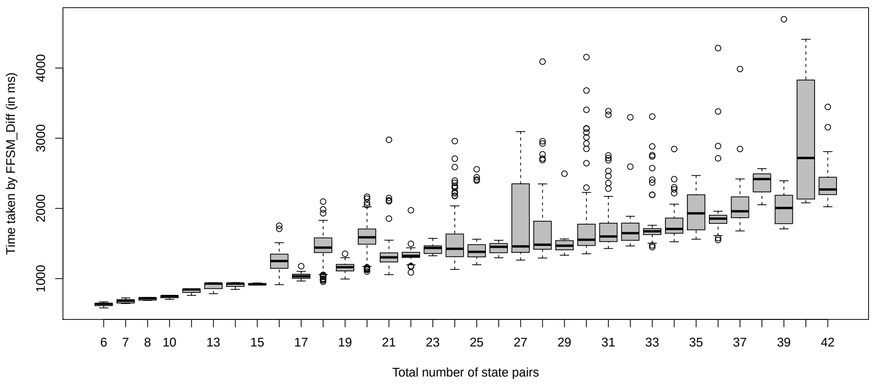

Fig. 18: Average times for learning FFSMs from all product pairs

How are the different notions of variability represented? Currently, our approach annotates state and transitions using the disjunction of simplified configurations. As a result of this design decision, the representation of feature constraints is limited to a unique format (i.e., OR with ANDs). To overcome this limitation, more sophisticated presence-condition simplification techniques [100] could be used to reduce the complexity of feature constraints. Other possible solutions are the usage of feature model refactoring and specialization [18] to come up with the constrains for conditional state and transitions.

To which other models this technique can be applied? We expect that our technique can be applied to learn other behavioral models for software product lines such as MTS [78,52,77,17,16] and FTS [33,20]. Regarding deterministic subsets of these two models, we expect that our technique can be readily applied without much modifications. Learning non-deterministic models, however, requires further investigation.

6 Related Work

In this section, we discuss our approach in terms of related work and how it can be helpful in the respective context. Studies related to ours are in the fields of state-machine learning, product sampling, family-based analysis, comparison of state models, reverse engineering feature models, and SPL evolution.

6.1 State-machine learning

As software requirements change and systems evolve, the lack of maintenance may render outdated and incomplete models [133] and hamper the application of model-based techniques [87]. To tackle these issues, state-machine learning, also known as automata learning [7], has become popular technique to automate the construction of behavioral models.

State-machine learning has been harnessed for black box model checking [94], real-world protocols [1,53], software evolution [65,43], automatic test generation [98], and generalization of failure models [28,74]. For an overview of state-machine learning and applications, we refer the reader to [66,116,2]. The problem of learning models from SPLs becomes more complex as it has to cope with products and features that may have their own models, requirements and code.

Our study improves upon the state-of-the-art by evaluating the quality of models learned by sampling subsets of valid products. Thus, we pave the way for more efficient and precise family model learning approaches, which is a topic that is still understudied [42].

6.2 Product sampling for SPLs

Due to the number of valid configurations that usually grows exponentially with the number of features, the exhaustive analysis of SPLs is impractical [120]. To alleviate this issue, sampling techniques that provide subsets of all valid products are being used to cover the behavior of SPLs and hence reveal most faults in all other products [95].

According to Varshosaz et al. [127], product sampling techniques often rely on feature models [70] and SAT solvers [79] to distinguish valid from invalid configurations [18]. To support the sampling process, techniques can use meta-heuristics (e.g., genetic algorithms [49,84]), coverage criteria (e.g., T-wise [95]), manual selection and semi-automatic selection.

In our work, we have used product sampling to generate subsets of valid configurations satisfying T-wise coverage. We have used the Chvatal algorithm [31] implemented in the FeatureIDE workbench [121]. This algorithm has been adapted by Johansen et al. [68] for product sampling by generating all T-wise feature combinations. Incremental product sampling algorithms, such as the IncLing [3], could be employed for family model learning, but this has been left as future work.

6.3 Family-based analysis of SPLs

Family-based analysis operates on domain artifacts and incorporates knowledge about valid feature combinations, given a feature model. Thus, not every individual product has to be analyzed [120], as opposed to traditional analysis strategies that are influenced by the number of valid feature combinations [25]. To achieve this goal, family-based analysis techniques rely on *family models*. For an overview on techniques for family-model analysis, testing and modeling, we refer the reader to recent surveys [120,19,21].

Family models have been exploited as theoretical foundation to perform efficient model-based testing of SPLs [13,20], family model checking [104,119], to automate the generation of specifications for individual products [11], to efficiently validate families of products [58], and to describe fine-grained differences among product variants [106]. We believe that our approach is complementary to the aforementioned techniques as it can give insights about optimizing family model learning to scenarios where there is a large number of valid product configurations. Our technique is discussed in terms of FFSMs, but it can be extended to other family-based notations, like FTSs [33,20], as FSMs can be represented as a variant of LTS labeled with input/output pairs.

6.4 Comparison of state-based models

The comparison of FSMs is an important task for software engineering [134] such as conformance testing [26], and performance analysis of state-machine learning techniques [7,125]. Studies related to ours are by Damasceno et al. [42], Nejati et al. [91], and Walkinshaw and Bogdanov [134].

Damasceno et al. [42] introduced an approach to compare product FSMs and build family models [42]. In this paper, we evaluate how product sampling can help to reduce the costs for learning family models by sampling product configurations. Product lines may have an exponential number of valid configurations and hence, sampling techniques can be helpful to reduce the effort required to recover family models.

Nejati et al. [91] presented an approach for matching and merging Statecharts [59]. Their approach relies on two operators for matching and merging transitions. The latter uses static and behavioral properties to match state pairs. The former produces a combined model in which variant behaviors are parameterized using guards on their transitions where temporal properties are preserved. The authors showed that relying on both operators produces higher precision than relying on them independently.

Walkinshaw and Bogdanov [134] evaluated two approaches to compute the precise difference between LTSs in terms of their language and structure. To compare the language of state-based models, the authors have proposed an approach based on the proportion of test sequences [29,129] that are classified in the same way by two models M_r and M_u. Thus, performance metrics, (e.g., precision, recall, and F-measure) can be used to compare the languages of LTS models. A major issue on comparing the language of FSMs is the fact that some minor differences can mask structural similarities. To tackle this issue, the authors have proposed an algorithm to compare the structure of FSMs. The aforementioned approaches are *complementary* as two models may have similar state transition structure, but completely different languages, or vice-versa.

The family model learning process may face scalability issues in large SPLs as a result of the worst-case complexity required to solve the system of linear equations. Alternatively, search-based techniques could be used for product sampling [49] and matching and merging states and transitions of product and family models [5]. Furthermore, expert knowledge [127] or prioritization techniques [61] could also be incorporated to identify what products should be analyzed first. These are left as future work.

6.5 Reverse engineering feature models

Feature models play a central role on the variability management for SPLs [18]. By using SAT solvers [79], feature models can be analyzed to detect invalid relationships or product configurations, core or dead features, redundancies, and enumerate or quantify all valid products of an SPL. Unfortunately, companies often develop software variants in an unstructured way and may lack feature models as their construction is time-consuming and error prone [60].

In this context, several approaches have been proposed to automatically build feature models from sets of product configurations [60,103,6]. Approaches based on Formal Concept Analysis (FCA) show promising possibilities on reverse engineering feature models as they can detect interdependencies and hierarchies between features [6].

Our proposal focuses on the problem of "reverse engineering" family models from sampled product configurations. In our study, we assume that the feature model is known a priori. However, we believe that our technique can be extended to cope with non-existent feature models and learn family and feature models at once, but the succinctness of the feature constraints may be compromised. Thus, investigations combining feature model and behavioral model learning are still required.

6.6 SPL evolution

The tasks of SPL reengineering and refactoring are vital to the maintenance and evolution of their software products. For an overview on product line evolution, refactoring and reengineering, we refer the readers to [75,51,88].

A large variety of artifacts have been considered in SPL evolution, but feature models are by far the most researched ones [88]. Moreover, recent studies have shown that there is a need for reengineering approaches specifically tailored for agile processes [88], and migration of SPL paradigms [75].

Several studies have investigated model learning techniques to cope with traditional software evolution and regression testing [109,64]. However, to the best of our knowledge, there are no works investigating model learning in the setting of SPLs. Combined with state-machine learning [7], we believe that our algorithm can support model-based regression testing in SPLs [102] and family model checking [104,119] in agile processes [92].

7 Conclusion

alumni. In this paper, we present a technique for learning behavioral family models in terms of Featured Finite State Machines (FFSMs). Our technique builds upon a known feature model for a product line and its individually learned or hand-crafted finite state machines, corresponding to product-specific models with their respective known sets of features. We presented the $\mathit{FFSM}_{\mathit{Diff}}$ algorithm, that unifies these product models into an FFSM by employing a state-based model comparison technique and feature model analysis. Furthermore, we combined our approach with product sampling to reduce the cost of exhaustive learning and integrate sampled product models into an accurate family model.

We performed an empirical study of the effectiveness of our approach by analyzing the succinctness and the accuracy of the learned models. We show that the learned family models are more succinct than the total size of the individual product models, particularly when there is a high degree of reuse among these products. In addition, we also performed a set of experiments to investigate whether feature interaction criteria (e.g., T-Wise) can alleviate the costs for family model learning by sampling valid products to collectively cover the behavior of product families. Our empirical analysis showed that family models learned by sampling can be as precise as those learned from exhaustive analysis. These results pave the way for reducing the costs for recovering family models from product lines.

This paper extends our previous conference publication [42] by including three extra models into our empirical evaluation. Our results corroborate our previous findings where product models were effectively merged into succinct FFSMs with fewer states, especially if there is high feature sharing among products. Also the integration of product sampling into the learning process and the empirical study of the accuracy of the learned models in this respect are novel in the present paper.

As future work, we envision to investigate three problems: how to incorporate family models in active model learning, and how to improve the readability of our family models.

Adaptive model learning is a variant of automata learning [7] that attempts to reuse input sequences from existing models to speed up state coverage and identification [64, 41]. We believe that the performance of automata learning algorithms could be improved by reusing partial family models describing subsets of valid products, in a similar fashion to the standard adaptive model learning.

For incremental family model learning, we believe that search-based or interactive techniques could be used to recommend product configurations to be analyzed and pave the way for an incremental family model learning framework. Incremental product sampling algorithms, e.g., IncLing [3], could be employed in combination with model-based testing techniques to test-and-learn behavioral variability of black-box product instances and incorporate new product behavior in partial family models.

Finally, to improve the readability of the learned family models, we aim at investigating alternative approaches for presence-condition simplification. Currently, our approach annotates conditional state and transitions using the disjunction of simplified configurations. As a result of this process, the representation of feature constraints is limited to a unique format (i.e., OR with ANDs). To overcome this limitation, more sophisticated presence-condition simplification techniques [100] could be used to reduce the complexity of feature constraints. Alternatively, feature model refactoring and specialization [18] could also be employed to redesign constrained feature models as conditions of conditional state and transitions.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPESP (grant 2019/06937-0), and University of Leicester (UK), College of Science &

Engineering. Research carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0). The authors are grateful to the anonymous reviewers, the *VALidation and VErification* (VALVE) group of the University of Leicester and *Laboratory of Software Engineering* (LabES) at the University of Sao Paulo for their insightful comments.

Appendix

A Glossary of Symbols

This glossary section lists the main symbols and abbreviations that are used in this manuscript with their following meanings.

SPL	Software Product Line
SPLC	Software Product Line Conference
SAT4J	The boolean satisfaction and optimization library in Java
SAT	Boolean satisfiability problem
$FFSM_{Diff}$	The Featured Finite State Machine Difference algorithm
LTS_{Diff}	The Labeled Transition System Difference algorithm
F	The set of features of an SPL
p	A product of an SPL
$\mathcal{P}(F)$	The powerset of all feature combinations
$\mathcal{M}$	Symbol identifying a finite state machine
TP	True positives
TN	True negatives
FP	False positives
FN	False negatives
$FF_{r,u}$	Featured Finite State Machine learned from two products r and u
FF_r	Featured Finite State Machine taken as reference model
$Surr(a,b)$	Operation to find surrounding state pairs via matching transitions
$buildFSM()$	Operation to build an FSM for a given configuration of a product line
$B(F)$	The set of all feature constraints
ξ	A product configuration
χ	A feature constraint
$\hat{A}$	Vargha-Delaney's effect size
AGM	The Arcade Game Maker product line
VM	The Vending Machine product line
WS	The Wiper System product line
AEROUC5	The Aero UC5 product line
CPTERMINAL	The Card Payment product line
MINEPUMP	The Minepump product line
CIT	Combinatorial Interaction Testing
FSM	Finite State Machine
FFSM	Featured Finite State Machine
δ	Transition function for an FSM
λ	Output function for an FSM
ϵ	Empty input sequence
Λ	The set of product configurations specified that an FFSM specifies
Γ	The set of conditional transitions of an FFSM model
FCA	Formal Concept Analysis
LTS	Labeled Transition System
MTS	Modal Transition System
FTS	Featured Transition System

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: D. Giannakopoulou, D. Méry (eds.) FM 2012: Formal Methods, pp. 10–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-32759-9_4. URL https://doi.org/10.1007/978-3-642-32759-9_4
2. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: A. Bennaceur, R. Hähnle, K. Meinke (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers, pp. 74–100. Springer International Publishing, Cham (2018). DOI 10.1007/978-3-319-96562-8_3. URL https://doi.org/10.1007/978-3-319-96562-8_3
3. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: Incling: Efficient product-line testing using incremental pairwise sampling. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, p. 144–155. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2993236.2993253. URL https://doi.org/10.1145/2993236.2993253
4. Al-Hajjaji, M., Lity, S., Lachmann, R., Thüm, T., Schaefer, I., Saake, G.: Delta-oriented product prioritization for similarity-based product-line testing. In: Proceedings of the 2nd International Workshop on Variability and Complexity in Software Design, VACE '17, p. 34–40. IEEE Press (2017). DOI 10.1109/VACE.2017.8. URL https://doi.org/10.1109/VACE.2017.8
5. Al-Khiaty, M.A.R., Ahmed, M.: Matching UML class diagrams using a hybridized greedy-genetic algorithm. In: 2017 12th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT), pp. 161–166. IEEE (2017). DOI 10.1109/STC-CSIT.2017.8098759. URL https://doi.org/10.1109/STC-CSIT.2017.8098759
6. Al-Msie'deen, R., Huchard, M., Seriai, A., Urtado, C., Vauttier, S.: Reverse engineering feature models from software configurations using formal concept analysis. In: K. Bertet, S. Rudolph (eds.) Proceedings of the Eleventh International Conference on Concept Lattices and Their Applications, Košice, Slovakia, October 7-10, 2014, *CEUR Workshop Proceedings*, vol. 1252, pp. 95–106. CEUR-WS.org (2014). URL http://ceur-ws.org/Vol-1252/cla2014_submission_13.pdf
7. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87 – 106 (1987). DOI 10.1016/0890-5401(87)90052-6. URL https://doi.org/10.1016/0890-5401(87)90052-6
8. Apache: Commons Math: The Apache Commons Mathematics Library. http://commons.apache.org/proper/commons-math/ (2016). [Online; accessed 28-Mar-2019]
9. Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., Garvin, B.: Exploring feature interactions in the wild: The new feature-interaction challenge. In: Proceedings of the 5th International Workshop on Feature-Oriented Software Development, pp. 1–8. ACM, New York, NY, USA (2013). DOI 10.1145/2528265.2528267. URL https://doi.org/10.1145/2528265.2528267
10. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, p. 1–10. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/1985793.1985795. URL https://doi.org/10.1145/1985793.1985795
11. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A compositional framework to derive product line behavioural descriptions. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 146–161. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-34026-0_12. URL https://doi.org/10.1007/978-3-642-34026-0_12
12. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal description of variability in product families. In: 2011 15th International Software Product Line Conference, pp.

130–139. IEEE. DOI 10.1109/SPLC.2011.34. URL https://doi.org/10.1109/SPLC.2011.34

13. Atlee, J.M., Beidu, S., Fahrenberg, U., Legay, A.: Merging Features in Featured Transition Systems. In: Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, *CEUR Workshop Proceedings*, vol. 1514, pp. 38–43. Ottawa, Canada (2015). URL https://hal.inria.fr/hal-01237661
14. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
15. Bauer, O., Geske, M., Isberner, M.: Analyzing program behavior through active automata learning. International Journal on Software Tools for Technology Transfer **16**(5), 531–542 (2014). DOI 10.1007/s10009-014-0333-2. URL http://dx.doi.org/10.1007/s10009-014-0333-2
16. ter Beek, M.H., Damiani, F., Gnesi, S., Mazzanti, F., Paolini, L.: On the expressiveness of modal transition systems with variability constraints **169**, 1–17. DOI 10.1016/j.scico.2018.09.006. URL http://www.sciencedirect.com/science/article/pii/S0167642318303769
17. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints **85**(2), 287–315. DOI 10.1016/j.jlamp.2015.11.006. URL http://www.sciencedirect.com/science/article/pii/S2352220815001431
18. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6), 615–636 (2010). DOI https://doi.org/10.1016/j.is.2010.01.001. URL http://www.sciencedirect.com/science/article/pii/S0306437910000025
19. Benduhn, F., Thüm, T., Lochau, M., Leich, T., Saake, G.: A survey on modeling techniques for formal behavioral verification of software product lines. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '15, p. 80–87. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2701319.2701332. URL https://doi.org/10.1145/2701319.2701332
20. Beohar, H., Mousavi, M.R.: Input–output conformance testing for software product lines. Journal of Logical and Algebraic Methods in Programming **85**(6), 1131 – 1153 (2016). DOI 10.1016/j.jlamp.2016.09.007. URL https://doi.org/10.1016/j.jlamp.2016.09.007. NWPT 2013
21. Beohar, H., Varshosaz, M., Mousavi, M.R.: Basic behavioral models for software product lines: Expressiveness and testing pre-orders. Science of Computer Programming **123**, 42 – 60 (2016). DOI 10.1016/j.scico.2015.06.005. URL https://doi.org/10.1016/j.scico.2015.06.005
22. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering **39**(12), 1611–1640 (2013). DOI 10.1109/TSE.2013.34
23. Bertolino, A., Daoudagh, S., El Kateb, D., Henard, C., Le Traon, Y., Lonetti, F., Marchetti, E., Mouelhi, T., Papadakis, M.: Similarity testing for access control. Information and Software Technology **58**, 355 – 372 (2015). DOI 10.1016/j.infsof.2014.07.003. URL http://dx.doi.org/10.1016/j.infsof.2014.07.003
24. Beuche, D., Schulze, M., Duvigneau, M.: When 150% is too much: Supporting product centric viewpoints in an industrial product line. In: Proceedings of the 20th International Systems and Software Product Line Conference, SPLC '16, p. 262–269. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2934466.2934493. URL https://doi.org/10.1145/2934466.2934493
25. Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P.: Intraprocedural dataflow analysis for software product lines. In: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, AOSD '12, pp. 13–24. ACM, New York, NY, USA (2012). DOI 10.1145/2162049.2162052. URL http://doi.acm.org/10.1145/2162049.2162052
26. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Part i. testing of finite state machines. In: M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, A. Pretschner

(eds.) Model-Based Testing of Reactive Systems: Advanced Lectures, pp. 1–3. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI 10.1007/11498490_1. URL `https://doi.org/10.1007/11498490_1`

27. Cartaxo, E.G., Machado, P.D.L., Neto, F.G.O.: On the use of a similarity function for test case selection in the context of model-based testing. Software Testing, Verification and Reliability **21**(2), 75–100 (2011). DOI 10.1002/stvr.413. URL `https://doi.org/abs/10.1002/stvr.413`
28. Chapman, M., Chockler, H., Kesseli, P., Kroening, D., Strichman, O., Tautschnig, M.: Learning the language of error. In: B. Finkbeiner, G. Pu, L. Zhang (eds.) Automated Technology for Verification and Analysis, pp. 114–130. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-24953-7_9. URL `https://doi.org/10.1007/978-3-319-24953-7_9`
29. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978). DOI 10.1109/TSE.1978.231496. URL `https://doi.org/10.1109/TSE.1978.231496`
30. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: ProFeat: feature-oriented engineering for family-based probabilistic model checking. Formal Aspects of Computing **30**(1), 45–75. DOI 10.1007/s00165-017-0432-4. URL `https://doi.org/10.1007/s00165-017-0432-4`
31. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research **4**(3), 233–235 (1979). DOI 10.1287/moor.4.3.233. URL `https://doi.org/10.1287/moor.4.3.233`
32. Classen, A.: Modelling with fts: a collection of illustrative examples (2010). URL `https://researchportal.unamur.be/en/publications/modelling-with-fts-a-collection-of-illustrative-examples`
33. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. IEEE Transactions on Software Engineering **39**(8), 1069–1089 (2013). DOI 10.1109/TSE.2012.86. URL `https://doi.org/10.1109/TSE.2012.86`
34. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, p. 335–344. Association for Computing Machinery, New York, NY, USA (2010). DOI 10.1145/1806799.1806850. URL `https://doi.org/10.1145/1806799.1806850`
35. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
36. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3), 215–249 (1998). DOI 10.1145/287000.287001. URL `https://doi.org/10.1145/287000.287001`
37. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: D. Batory, C. Consel, W. Taha (eds.) Generative Programming and Component Engineering, vol. 2487, pp. 156–172. Springer Berlin Heidelberg. DOI 10.1007/3-540-45821-2_10. URL `http://link.springer.com/10.1007/3-540-45821-2_10`. Series Title: Lecture Notes in Computer Science
38. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization **10**(1), 7–29. DOI 10.1002/spip.213. URL `http://doi.wiley.com/10.1002/spip.213`
39. Damasceno, C.D.N., Masiero, P.C., Simao, A.: Evaluating test characteristics and effectiveness of fsm-based testing methods on rbac systems. In: Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES '16, pp. 83–92. ACM, New York, NY, USA (2016). DOI 10.1145/2973839.2973849. URL `http://doi.acm.org/10.1145/2973839.2973849`
40. Damasceno, C.D.N., Masiero, P.C., Simao, A.: Similarity testing for role-based access control systems. Journal of Software Engineering Research and Development

6(1), 1 (2018). DOI 10.1186/s40411-017-0045-x. URL https://doi.org/10.1186/s40411-017-0045-x

41. Damasceno, C.D.N., Mousavi, M.R., da Silva Simao, A.: Learning to reuse: Adaptive model learning for evolving systems. In: W. Ahrendt, S.L. Tapia Tarifa (eds.) Integrated Formal Methods: 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings, pp. 138–156. Springer International Publishing, Cham (2019). DOI 10.1007/978-3-030-34968-4_8. URL http://doi.acm.org/10.1007/978-3-030-34968-4_8
42. Damasceno, C.D.N., Mousavi, M.R., Simao, A.: Learning from difference: An automated approach for learning family models from software product lines. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, SPLC '19, pp. 52–63. ACM, New York, NY, USA (2019). DOI 10.1145/3336294.3336307. URL http://doi.acm.org/10.1145/3336294.3336307
43. De Ruiter, J., Poll, E.: Protocol state fuzzing of tls implementations. In: Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15, pp. 193–206. USENIX Association, Berkeley, CA, USA (2015). DOI 10.5555/2831143.2831156. URL https://doi.org/10.5555/2831143.2831156
44. Devroey, X., Perrouin, G.: Vibes: Variability intensive system behavioural testing (2014). URL https://projects.info.unamur.be/vibes/
45. Devroey, X., Perrouin, G., Legay, A., Schobbens, P.Y., Heymans, P.: Search-based similarity-driven behavioural SPL testing. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16, pp. 89–96. ACM Press. DOI 10.1145/2866614.2866627. URL http://dl.acm.org/citation.cfm?doid=2866614.2866627
46. Devroey, X., Perrouin, G., Legay, A., Schobbens, P.Y., Heymans, P.: Covering spl behaviour with sampled configurations: An initial assessment. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, pp. 59:59–59:66. ACM, New York, NY, USA (2015). DOI 10.1145/2701319.2701325. URL http://doi.acm.org/10.1145/2701319.2701325
47. Deza, M.M., Deza, E.: Distances and similarities in data analysis. In: Encyclopedia of Distances, pp. 291–305. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-30958-8_17. URL https://doi.org/10.1007/978-3-642-30958-8_17
48. Don Batory: Feature models, grammars, and propositional formulas. In: H. Obbink, K. Pohl (eds.) Software Product Lines, vol. 3714, pp. 7–20. Springer Berlin Heidelberg. DOI 10.1007/11554844_3. URL https://doi.org/10.1007/11554844_3. Series Title: Lecture Notes in Computer Science
49. Ensan, F., Bagheri, E., Gašević, D.: Evolutionary search-based test generation for software product line feature models. In: J. Ralyté, X. Franch, S. Brinkkemper, S. Wrycza (eds.) Advanced Information Systems Engineering, pp. 613–628. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-31095-9_40. URL https://doi.org/10.1007/978-3-642-31095-9_40
50. FeatureIDE: FeatureIDE: An extensible framework for feature-oriented software development. https://featureide.github.io/ (2004). [Online; accessed 21-May-2020]
51. Fenske, W., Thüm, T., Saake, G.: A taxonomy of software product line reengineering. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2556624.2556643. URL https://doi.org/10.1145/2556624.2556643
52. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis, ROSATEA '06, pp. 39–48. Association for Computing Machinery. DOI 10.1145/1147249.1147254. URL https://doi.org/10.1145/1147249.1147254
53. Fiterău-Broştean, P., Howar, F.: Learning-based testing the sliding window behavior of tcp implementations. In: L. Petrucci, C. Seceleanu, A. Cavalcanti (eds.) Critical Systems: Formal Methods and Automated Verification: Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems and 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS

2017, Turin, Italy, September 18-20, 2017, Proceedings, pp. 185–200. Springer International Publishing, Cham (2017). DOI 10.1007/978-3-319-67113-0_12. URL https://doi.org/10.1007/978-3-319-67113-0_12
54. Fragal, V.H.: Automatic generation of configurable test-suites for software product lines. Ph.D. thesis, Universidade de São Paulo (2017). [Online] http://www.teses.usp.br/teses/disponiveis/55/55134/tde-10012019-085746/
55. Fragal, V.H., Simao, A., Mousavi, M.R.: Hierarchical featured state machines. Science of Computer Programming **171**, 67–88 (2019). DOI 10.1016/j.scico.2018.10.001. URL https://doi.org/10.1016/j.scico.2018.10.001
56. Fragal, V.H., Simao, A., Mousavi, M.R., Turker, U.C.: Extending HSI Test Generation Method for Software Product Lines. The Computer Journal **62**(1), 109–129 (2018). DOI 10.1093/comjnl/bxy046. URL https://doi.org/10.1093/comjnl/bxy046
57. Gill, A.: Introduction to the Theory of Finite State Machines. McGraw-Hill, New York (1962)
58. Hafemann Fragal, V., Simao, A., Mousavi, M.R.: Validated test models for software product lines: Featured finite state machines. In: O. Kouchnarenko, R. Khosravi (eds.) Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers, pp. 210–227. Springer International Publishing, Cham (2017). DOI 10.1007/978-3-319-57666-4_13. URL https://doi.org/10.1007/978-3-319-57666-4_13
59. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987). DOI 10.1016/0167-6423(87)90035-9. URL https://doi.org/10.1016/0167-6423(87)90035-9
60. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Reverse engineering feature models from programs' feature sets. In: 2011 18th Working Conference on Reverse Engineering, pp. 308–312. IEEE (2011). DOI 10.1109/WCRE.2011.45. URL https://doi.org/10.1109/WCRE.2011.45
61. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Transactions on Software Engineering **40**(7), 650–670 (2014)
62. Hess, M.R., Kromrey, J.D.: Robust confidence intervals for effect sizes: A comparative study of cohen's d and cliff's delta under non-normality and heterogeneous variances. In: Annual meeting of the American Educational Research Association (2004)
63. Holthusen, S., Wille, D., Legat, C., Beddig, S., Schaefer, I., Vogel-Heuser, B.: Family model mining for function block diagrams in automation software. In: Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14, pp. 36–43. ACM Press. DOI 10.1145/2647908.2655965. URL http://dl.acm.org/citation.cfm?doid=2647908.2655965
64. Huistra, D., Meijer, J., Pol, J.: Adaptive learning for learn-based regression testing. In: F. Howar, J. Barnat (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, pp. 162–177. Springer Publishers, Switzerland (2018). DOI 10.1007/978-3-030-00244-2_11. URL https://doi.org/10.1007/978-3-030-00244-2_11
65. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: W.A. Hunt, F. Somenzi (eds.) Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings, pp. 315–327. Springer, Berlin, Heidelberg (2003). DOI 10.1007/978-3-540-45069-6_31. URL https://doi.org/10.1007/978-3-540-45069-6_31
66. Irfan, M.N., Oriat, C., Groz, R.: Chapter 3 - model inference and testing. In: A. Memon (ed.) Advances in Computers, vol. 89, pp. 89–139. Elsevier (2013). DOI 10.1016/B978-0-12-408094-2.00003-5. URL https://doi.org/10.1016/B978-0-12-408094-2.00003-5
67. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib. In: D. Kroening, C.S. Păsăreanu (eds.) Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, pp. 487–495. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-21690-4_32. URL https://doi.org/10.1007/978-3-319-21690-4_32

68. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of realistic feature models make combinatorial testing of product lines feasible. In: J. Whittle, T. Clark, T. Kühne (eds.) Model Driven Engineering Languages and Systems, pp. 638–652. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-24485-8_47. URL https://doi.org/10.1007/978-3-642-24485-8_47
69. Kampenes, V.B., Dyba, T., Hannay, J.E., Sjoberg, D.I.: A systematic review of effect size in software engineering experiments. Information and Software Technology **49**(11), 1073–1086 (2007). DOI 10.1016/j.infsof.2007.02.015. URL https://doi.org/10.1016/j.infsof.2007.02.015
70. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990). URL https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231
71. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM **19**(7), 371–384 (1976). DOI 10.1145/360248.360251. URL https://doi.org/10.1145/360248.360251
72. Kuhn, D.R., Kacker, R.N., Lei, Y.: Introduction to Combinatorial Testing, 1st edn. Chapman & Hall/CRC (2013)
73. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering **30**(6), 418–421 (2004). DOI 10.1109/TSE.2004.24. URL https://doi.org/10.1109/TSE.2004.24
74. Kunze, S., Mostowski, W., Mousavi, M.R., Varshosaz, M.: Generation of failure models through automata learning. In: 2016 Workshop on Automotive Systems/Software Architectures (WASA), pp. 22–25. IEEE Computer Society, Los Alamitos, CA, USA (2016). DOI 10.1109/WASA.2016.7. URL https://doi.org/10.1109/WASA.2016.7
75. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Science of Computer Programming **78**(8), 1010–1034 (2013). DOI 10.1016/j.scico.2012.05.003. URL https://doi.org/10.1016/j.scico.2012.05.003
76. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In: Proceedings of the 4th International Colloquium on Grammatical Inference, ICGI '98, p. 1–12. Springer-Verlag, Berlin, Heidelberg (1998). DOI 10.1007/BFb0054059. URL https://doi.org/10.1007/BFb0054059
77. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal i/o automata for interface and product line theories. In: R. De Nicola (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, pp. 64–79. Springer. DOI 10.1007/978-3-540-71316-6_6
78. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proceedings of the Third Annual Symposium on Logic in Computer Science, pp. 203–210 (1988). DOI 10.1109/LICS.1988.5119. URL https://doi.org/10.1109/LICS.1988.5119
79. Le Berre, D., Parrain, A.: The SAT4J library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation **7**, 59–64 (2010). URL http://satassociation.org/jsat/index.php/jsat/article/view/82
80. van der Linden, F., Schmid, K., Rommes, E.: Philips consumer electronics software for televisions. In: Software Product Lines in Action, pp. 219–232. Springer Berlin Heidelberg. DOI 10.1007/978-3-540-71437-8_14. URL https://doi.org/10.1007/978-3-540-71437-8_14
81. van der Linden, F., Schmid, K., Rommes, E.: Philips medical systems. In: Software Product Lines in Action, pp. 233–248. Springer Berlin Heidelberg. DOI 10.1007/978-3-540-71437-8_15. URL https://doi.org/10.1007/978-3-540-71437-8_15
82. van der Linden, F., Schmid, K., Rommes, E.: Siemens medical solutions. In: Software Product Lines in Action, pp. 249–263. Springer Berlin Heidelberg. DOI 10.1007/978-3-540-71437-8_16. URL https://doi.org/10.1007/978-3-540-71437-8_16
83. Linden, F.J.v.d., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
84. Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Egyed, A., Alba, E.: Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for

pairwise testing of software product lines. In: 2014 IEEE Congress on Evolutionary Computation (CEC), pp. 387–396 (2014). DOI 10.1109/CEC.2014.6900473. URL `https://doi.org/10.1109/CEC.2014.6900473`

85. Luthmann, L., Gerecht, T., Stephan, A., Bürdek, J., Lochau, M.: Minimum/maximum delay testing of product lines with unbounded parametric real-time constraints **149**, 535 – 553. DOI https://doi.org/10.1016/j.jss.2018.12.028. URL `http://www.sciencedirect.com/science/article/pii/S0164121218302851`
86. Luthmann, L., Stephan, A., Bürdek, J., Lochau, M.: Modeling and testing product lines with unbounded parametric real-time constraints. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A on - SPLC '17, pp. 104–113. ACM Press. DOI 10.1145/3106195.3106204. URL `https://doi.org/3106195.3106204`
87. Mariani, L., Pezzè, M., Zuddas, D.: Chapter four - recent advances in automatic black-box testing. In: A. Memon (ed.) Advances in Computers, vol. 99, pp. 157–193. Elsevier (2015). DOI 10.1016/bs.adcom.2015.04.002. URL `https://doi.org/10.1016/bs.adcom.2015.04.002`
88. Marques, M., Simmonds, J., Rossel, P.O., Bastarrica, M.C.: Software product line evolution: A systematic literature review. Information and Software Technology **105**, 190–208 (2019). DOI 10.1016/j.infsof.2018.08.014. URL `https://doi.org/10.1016/j.infsof.2018.08.014`
89. Mendez, D., Graziotin, D., Wagner, S., Seibold, H.: Open science in software engineering. In: M. Felderer, G.H. Travassos (eds.) Contemporary Empirical Methods in Software Engineering, pp. 477–501. Springer International Publishing. DOI 10.1007/978-3-030-32489-6_17. URL `http://link.springer.com/10.1007/978-3-030-32489-6_17`
90. Neider, D., Smetsers, R., Vaandrager, F., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: T. Margaria, S. Graf, K.G. Larsen (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, pp. 390–416. Springer International Publishing. DOI 10.1007/978-3-030-22348-9_23. URL `https://doi.org/10.1007/978-3-030-22348-9_23`
91. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of variant feature specifications. IEEE Transactions on Software Engineering **38**(6), 1355–1375 (2012). DOI 10.1109/TSE.2011.112. URL `https://doi.org/10.1109/TSE.2011.112`
92. Neubauer, J., Steffen, B., Bauer, O., Windmuller, S., Merten, M., Margaria, T., Howar, F.: Automated continuous quality assurance. In: 2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), pp. 37–43. IEEE (2012). DOI 10.1109/FormSERA.2012.6229787. URL `https://doi.org/10.1109/FormSERA.2012.6229787`
93. Oster, S.: Feature model-based software product line testing. Ph.D. thesis, Technische Universität (2012). [Online] `http://tuprints.ulb.tu-darmstadt.de/2881/`
94. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: J. Wu, S.T. Chanson, Q. Gao (eds.) Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China, pp. 225–240. Springer US, Boston, MA (1999). DOI 10.1007/978-0-387-35578-8_13. URL `https://doi.org/10.1007/978-0-387-35578-8_13`
95. Perrouin, G., Sen, S., Klein, J., Baudry, B., l. Traon, Y.: Automated and scalable t-wise test case generation strategies for software product lines. In: 2010 Third International Conference on Software Testing, Verification and Validation, pp. 459–468 (2010). DOI 10.1109/ICST.2010.43. URL `http://doi.org/10.1109/ICST.2010.43`
96. Petke, J., Yoo, S., Cohen, M.B., Harman, M.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 26–36. ACM, New York, NY, USA (2013). DOI 10.1145/2491411.2491436. URL `http://doi.acm.org/10.1145/2491411.2491436`

97. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
98. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. International Journal on Software Tools for Technology Transfer **11**(4), 307 (2009). DOI 10.1007/s10009-009-0120-7. URL http://doi.org/10.1007/s10009-009-0120-7
99. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: L. Baresi, R. Heckel (eds.) Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings, pp. 377–380. Springer, Berlin, Heidelberg (2006). DOI 10.1007/11693017_28. URL http://doi.org/10.1007/11693017_28
100. von Rhein, A., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., Berger, T.: Presence-condition simplification in highly configurable systems. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pp. 178–188. IEEE, Piscataway, NJ, USA (2015). DOI 10.5555/2818754.2818779. URL http://doi.org/10.5555/2818754.2818779
101. RStudio: RStudio: Open source and enterprise-ready professional software for data science. https://www.rstudio.com/ (2019). [Online; accessed 19-May-2019]
102. Runeson, P., Engström, E.: Chapter 7 - regression testing in software product line engineering. In: A. Hurson, A. Memon (eds.) Advances in Computers, vol. 86, pp. 223–263. Elsevier (2012). DOI 10.1016/B978-0-12-396535-6.00007-7. URL https://doi.org/10.1016/B978-0-12-396535-6.00007-7
103. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Extraction of feature models from formal contexts. In: Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11, pp. 1–8. ACM, New York, NY, USA (2011). DOI 10.1145/2019136.2019141. URL http://doi.org/10.1145/2019136.2019141
104. Sabouri, H., Khosravi, R.: Delta modeling and model checking of product families. In: F. Arbab, M. Sirjani (eds.) Fundamentals of Software Engineering: 5th International Conference, FSEN 2013, Tehran, Iran, April 24-26, 2013, Revised Selected Papers, pp. 51–65. Springer, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-40213-5_4. URL http://doi.org/10.1007/978-3-642-40213-5_4
105. Samih, H., Guen, H.L., Bogusch, R., Acher, M., Baudry, B.: Deriving usage model variants for model-based testing: An industrial case study. In: 2014 19th International Conference on Engineering of Complex Computer Systems, pp. 77–80 (2014). DOI 10.1109/ICECCS.2014.19
106. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: J. Bosch, J. Lee (eds.) Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings, pp. 77–91. Springer, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-15579-6_6. URL https://doi.org/10.1007/978-3-642-15579-6_6
107. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer **14**(5), 477–495 (2012). DOI 10.1007/s10009-012-0253-y. URL http://doi.org/10.1007/s10009-012-0253-y
108. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In: E. Ábrahám, M. Huisman (eds.) Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings, pp. 311–325. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-33693-0_20. URL http://dx.doi.org/10.1007/978-3-319-33693-0_20
109. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking. In: H. Chockler, D. Kroening, L. Mariani, N. Sharygina (eds.) Validation of Evolving Software, pp. 55–72. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-10623-6_6. URL http://doi.org/10.1007/978-3-319-10623-6_6
110. Shahbaz, M., Groz, R.: Inferring mealy machines. In: A. Cavalcanti, D.R. Dams (eds.) FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings, pp. 207–222. Springer, Berlin, Heidelberg

(2009). DOI 10.1007/978-3-642-05089-3_14. URL http://dx.doi.org/10.1007/978-3-642-05089-3_14

111. Sharp, D.C.: Component-based product line development of avionics software. In: P. Donohoe (ed.) Software Product Lines, pp. 353–369. Springer US. DOI 10.1007/978-1-4615-4339-8_19. URL http://link.springer.com/10.1007/978-1-4615-4339-8_19
112. Sofia Cassel Falk Howar, B.J.: Ralib: A learnlib extension for inferring efsms. DIFTS 2015 (2015). URL http://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf
113. Sokolova, M., Lapalme, G.: A systematic analysis of performance measures for classification tasks. Information Processing & Management **45**(4), 427 – 437 (2009). DOI https://doi.org/10.1016/j.ipm.2009.03.002. URL http://www.sciencedirect.com/science/article/pii/S0306457309000259
114. SPLC: Hall of fame – SPLC. https://splc.net/fame.html (2020). [Online; accessed 29-May-2020]
115. Steffens, M., Oster, S., Lochau, M., Fogdal, T.: Industrial evaluation of pairwise spl testing with moso-polite. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, pp. 55–62. ACM, New York, NY, USA (2012). DOI 10.1145/2110147.2110154. URL http://doi.acm.org/10.1145/2110147.2110154
116. Stevenson, A., Cordy, J.R.: A survey of grammatical inference in software engineering. Sci. Comput. Program. **96**(P4), 444–459 (2014). DOI 10.1016/j.scico.2014.05.008. URL http://doi.org/10.1016/j.scico.2014.05.008
117. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a software product line for train control: A case study of CVL. In: J. Bosch, J. Lee (eds.) Software Product Lines: Going Beyond, vol. 6287, pp. 106–120. Springer Berlin Heidelberg. DOI 10.1007/978-3-642-15579-6_8. URL http://link.springer.com/10.1007/978-3-642-15579-6_8. Series Title: Lecture Notes in Computer Science
118. ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L.: Static analysis of featured transition systems: [research]. In: Proceedings of the 23rd International Systems and Software Product Line Conference - volume A - SPLC '19, pp. 1–13. ACM Press. DOI 10.1145/3336294.3336295. URL http://dl.acm.org/citation.cfm?doid=3336294.3336295
119. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mcrl2. In: M. Huisman, J. Rubin (eds.) Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, pp. 387–405. Springer, Berlin, Heidelberg (2017). DOI 10.1007/978-3-662-54494-5_23. URL http://doi.org/10.1007/978-3-662-54494-5_23
120. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 1–45 (2014). DOI 10.1145/2580950. URL http://doi.org/10.1145/2580950
121. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming **79**(Supplement C), 70–85 (2014). DOI 10.1016/j.scico.2012.06.002. URL http://doi.org/10.1016/j.scico.2012.06.002
122. Thum, T., Kastner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In: 2011 15th International Software Product Line Conference, pp. 191–200. IEEE. DOI 10.1109/SPLC.2011.53. URL http://doi.org/10.1109/SPLC.2011.53
123. Torchiano, M.: effsize: Efficient Effect Size Computation (v. 0.7.1). CRAN package repository (2017). https://cran.r-project.org/web/packages/effsize/effsize.pdf [Online; accessed 20-Nov-2017]
124. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability **22**(5), 297–312 (2012). DOI 10.1002/stvr.456. URL http://doi.org/10.1002/stvr.456
125. Vaandrager, F.: Model learning. Communications of the ACM **60**(2), 86–95 (2017). DOI 10.1145/2967606. URL https://doi.org/10.1145/2967606

126. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000). DOI 10.3102/10769986025002101. URL `https://doi.org/10.3102/10769986025002101`
127. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1, SPLC '18, pp. 1–13. ACM, New York, NY, USA (2018). DOI 10.1145/3233027.3233035. URL `http://doi.org/10.1145/3233027.3233035`
128. Varshosaz, M., Khosravi, R.: Discrete time markov chain families: modeling and verification of probabilistic software product lines. In: Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops, p. 34. ACM Press. DOI 10.1145/2499777.2500725. URL `http://doi.org/10.1145/2499777.2500725`
129. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics **9**(4), 653–665 (1973). DOI 10.1007/BF01068590. URL `https://doi.org/10.1007/BF01068590`
130. VIBeS: VIBeS - Variability Intensive system Behavioural teSting - Case study: Aero UC5. `https://projects.info.unamur.be/vibes/case-study-aerouc5.html` (2016). [Online; accessed 23-Mar-2020]
131. VIBeS: VIBeS - Variability Intensive system Behavioural teSting - Case study: Card payement terminal. `https://projects.info.unamur.be/vibes/case-study-cpterminal.html` (2016). [Online; accessed 23-Mar-2020]
132. VIBeS: VIBeS - Variability Intensive system Behavioural teSting - Case study: Minepump. `https://projects.info.unamur.be/vibes/case-study-aerouc5.html` (2016). [Online; accessed 23-Mar-2020]
133. Walkinshaw, N.: Chapter 1 - reverse-engineering software behavior. In: A. Memon (ed.) Advances in Computers, vol. 91, pp. 1–58. Elsevier (2013). DOI 10.1016/B978-0-12-408089-8.00001-X. URL `http://doi.org/10.1016/B978-0-12-408089-8.00001-X`
134. Walkinshaw, N., Bogdanov, K.: Automated comparison of state-based software models in terms of their language and structure. ACM Transactions on Software Engineering and Methodology **22**(2), 1–37 (2013). DOI 10.1145/2430545.2430549. URL `http://doi.org/10.1145/2430545.2430549`
135. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Systematic literature reviews. In: Experimentation in Software Engineering, pp. 45–54. Springer, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-29044-2_4. URL `https://doi.org/10.1007/978-3-642-29044-2_4`
136. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer Publishing Company, Incorporated. DOI 10.1007/978-3-642-29044-2. URL `https://doi.org/10.1007/978-3-642-29044-2`
137. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification & Reliability **22**(2), 67–120 (2012). DOI 10.1002/stv.430. URL `https://dl.acm.org/doi/10.1002/stv.430`