Learning from Difference: An Automated Approach for Learning Family Models from Software Product Lines

Carlos Diego N. Damasceno
damascenodiego@usp.br
University of Sao Paulo, BR and
University of Leicester, UK

Mohammad Reza Mousavi
mm789@leicester.ac.uk
University of Leicester
Leicester, UK

Adenilso Simao
adenilso@icmc.usp.br
University of Sao Paulo (ICMC-USP)
São Carlos, SP, BR

ABSTRACT

Substantial effort has been spent on extending specification notations and their associated reasoning techniques to software product lines (SPLs). Family-based analysis techniques operate on a single artifact, referred to as a family model, that is annotated with variability constraints. This modeling approach paves the way for efficient model-based testing and model checking for SPLs. Albeit reasonably efficient, the creation and maintenance of family models tend to be time consuming and error-prone, especially if there are crosscutting features. To tackle this issue, we introduce $FFSM_{Diff}$, a fully automated technique to learn featured finite state machines (FFSM), a family-based formalism that unifies Mealy Machines from SPLs into a single representation. Our technique incorporates variability to compare and merge Mealy machines and annotate states and transitions with feature constraints. We evaluate our technique using 34 products derived from three different SPLs. Our results support the hypothesis that families of Mealy machines can be effectively merged into succinct FFSMs with fewer states, especially if there is high feature sharing among products. These indicate that $FFSM_{Diff}$ is an efficient family-based model learning technique.

CCS CONCEPTS

• Networks → Formal specifications; **• Theory of computation → Query learning**; **• Hardware → Finite state machines**; **• Software and its engineering → Software product lines**.

KEYWORDS

Software product lines, Model learning, Family model, 150% model

ACM Reference Format:
Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso Simao. 2019. Learning from Difference: An Automated Approach for Learning Family Models from Software Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3336294.3336307

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '19, September 9–13, 2019, Paris, France
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336307

1 INTRODUCTION

Analysis and maintenance of software product lines (SPL) are known to be challenging [58]; they should avoid repetitive analysis of shared assets, and at the same time cater for possible feature interactions [6]. To tackle these issues, substantial effort has been spent on extending specification notations and their associated reasoning techniques to SPLs [13, 20, 28, 29] leading to efficient family-based analysis [58].

Family-based analysis operates on a single specification, referred to as *family model*, that is annotated with feature constraints as propositional logic formulae to express the combination of features involved in the concerned part of the model [58]. Thus, using SAT solvers [42], family models are amenable to family model-based testing [61] and family model checking [10] where redundant analysis of shared assets are avoided or minimised. Moreover, the cost of family-based analysis is mainly determined by the number and size of features and the amount of feature sharing, rather than the number of valid products [58].

Model-based techniques specifically tailored to family models have enabled test generation [9, 14, 29] and model checking [52, 57] at reduced cost. Nevertheless, the creation and maintenance of test models are known to be difficult, time consuming and error-prone [61], and the traceability between the family- and variability models can be complex due to crosscutting features [54]. Added to this, as requirements change and product instances evolve, the lack of maintenance may render family models outdated [68].

Motivated by these issues, in this paper we discuss how the creation and maintenance of family models can be performed by combining techniques for automata learning [62], feature model analysis [11], and automated comparison of state-based models [69]. Thus, we introduce $FFSM_{Diff}$, a fully-automated technique to learn family models by comparing and merging product models.

Our technique extends an approach for comparing labeled transition systems (LTS) [69] to family models, i.e., featured finite state machines (FFSM) [26, 29], by incorporating variability to express product-specific behaviors with feature constraints. An FFSM is a family-based formalism that combines the Mealy Machines [16] of all products of an SPLs into a single model by annotating states and transitions with feature constraints [26]. To evaluate our approach, we designed the following research questions (RQ):

- **(RQ1)** Is our automated technique effective in learning succinct family models compared to the total size of the products?
- **(RQ2)** Is the size of learnt family models influenced by the amount of feature reuse?
- **(RQ3)** Is our automated technique effective in learning succinct family models compared to hand-crafted family models?

In our evaluation, we used 34 Mealy machines derived from three SPLs of previous studies [19, 29]. Although these case studies are abstract representations of SPLs, they comprise many non-trivial aspects, such as the possibility of infinite behaviour and the existence of states with similar or identical behaviour in different products [69].

As a measure of succinctness, we use the average size of learnt FFSMs compared to the average total size of the products under learning. We describe *size* in terms of the *number of states* as it is one of the factors that influences the complexity of model-based testing [16], model checking [10], and model learning [62]. Thus, learning succinct models can lead to efficient analysis.

We used the Mann-Whitney test to check if there was significant difference ($p < 0.01$) between the sizes of the FFSMs and products and used the Vargha-Delaney's $\hat{A}$ effect size [64] to assess the likelihood of the learnt FFSM being more succinct [7]. To evaluate if the amount of feature reuse influenced the size of the learnt FFSMs, we used Pearson's correlation coefficient.

Our results indicate that families of Mealy machines can be effectively combined into a succinct FFSM, i.e., with far fewer states than the total number of states in all products under learning. Moreover, we also show that there is a strong negative correlation between the amount of feature reuse and the size of learnt FFSMs. Thus, FFSMs learnt from similar products tend to have fewer states than those built from drastically different products. These results show that our technique is an efficient family-based technique for automatically learning family models. To our knowledge, this is the first study in learning behavioral models of SPLs. Our approach can be helpful to domain engineering by supporting the inclusion of new application requirements, SPL re-engineering [22], evolution [43], and traceability analysis [63].

Thus, our contributions are threefold: (1) we introduce a technique to automate the process of building family models by means of comparing FSMs and analyzing feature models; (2) we present an experiment evaluating our technique and showing its effectiveness for learning FFSMs; and (3) we show that amount of feature reuse is a factor that affects family model learning.

The rest of this paper is organized as follows: In section 2, we briefly discuss software product lines (SPL), featured finite state machines (FFSM) and an approach to compare state-based models [69]. In section 3, we introduce our $FFSM_{Diff}$ algorithm to learn FFSMs from products specifications. In section 4, we present our experiment design and artifacts, lab package structure[1], the analysis of results and threats to validity. In section 5, we discuss related work. In section 6, we we close this paper with our conclusions and the directions of our future work.

2 PRELIMINARIES

2.1 Software Product Lines

A software product line (SPL) is a family of products sharing a common and managed set of *features* that are developed in a prescribed way to satisfy the specific needs of a particular market segment. Let F be the set of features of an SPL. A product p is defined by a set of features $p \subseteq F$ from a feature model FM [39].

[1] The lab package is available at https://github.com/damascenodiego/learningFFSM

A feature model FM captures all information about common and variant features of an SPL as a hierarchically arranged set of interconnected features. Based on a feature model, 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 [42] can be used to detect valid feature models or feature combinations, core features (i.e., features that are part of all products) and redundancies in feature model [11]. 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.

A product configuration $\rho \in B(F)$ of a product $p \in P$ is a feature constraint that expresses the conjunction of all features in p and the conjunction of negated features that are absent from it, i.e., $\rho = (\bigwedge_{f \in p} f) \wedge (\bigwedge_{f \notin p} \neg f)$. Given a feature constraint $\chi \in B(F)$, a product configuration $\rho \in \Lambda$ satisfies χ, denoted by $\chi \vDash \rho$, iff the feature constraint $\chi \wedge \rho$ is *true*. To illustrate these concepts, we use the Arcade Game Maker SPL.

Example 2.1. (The Arcade Game Maker SPL) The Arcade Game Maker (AGM) SPL has three alternative features (i.e., `Brickle`, `Pong` and `Bowling`) and one optional feature (i.e., `Save`). The feature model depicted in Figure 1 has six valid product configurations, among which three satisfy the feature constraint ¬`S`.

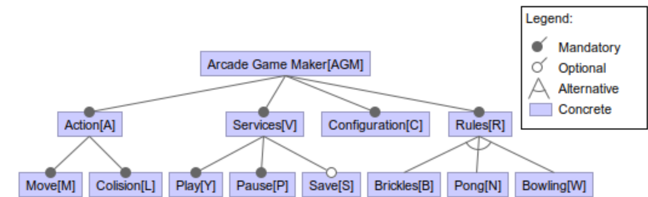

Figure 1: The AGM feature model

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, ω_b are equivalent under FM if $\Lambda_a = \Lambda_b$.

2.2 Featured Finite State Machines

Featured Finite State Machines are extensions of Finite State Machines [27], defined below, with feature constraints.

DEFINITION 2.1. *(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 and O are the sets of inputs and outputs, respectively, $D \subseteq S \times I$ is the specification domain, and $\delta : D \longrightarrow S$ and $\lambda : D \longrightarrow O$ are the transition and output functions.*

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 $\alpha = x_1 \cdot x_2 \cdot \ldots \cdot x_n \in I^*$ is defined in state $s \in S$ if there are states $s_1, s_2, \ldots, s_{n+1}$ 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 tuples (s_i, x, y, s_j) with the origin state, input, output, and destination states, respectively; or by directed edges labeled with input and output symbols, i.e., $s_i \xrightarrow{\text{i/o}} s_j$.

Example 2.2. (Example of Mealy machine) In Figure 2, we show an example of an FSM describing a product from the AGM SPL. In this example, we have $S = \{Start\ Game, Bowling\ Game, Pause\ Game\}$, $I = \{Start, Pause, Exit\}$ and $O = \{0, 1\}$.

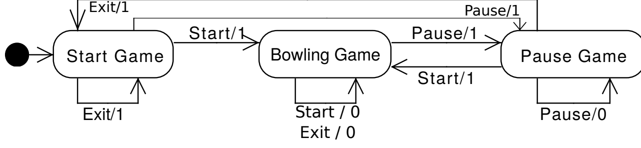

Figure 2: Example of FSM [29]

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 α is a prefix of β, denoted by $\alpha \leqslant \beta$, if $\beta = \alpha \cdot \omega$, for some sequence ω. An input sequence α is a proper prefix of β, denoted by $\alpha < \beta$, if $\beta = \alpha \cdot \omega$, for $\omega \neq \epsilon$. The prefixes of a set T of input sequences are denoted by $pref(T) = \{\alpha | \exists \beta \in T, \alpha < \beta\}$. If $T = pref(T)$, it is *prefix-closed.*

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

An FSM is *deterministic* if, for each state s_i and input x, there is at most one possible state $s_j = \delta(s_i, x)$ and output $y = \lambda(s_i, x)$. If all states of an FSM are pairwise distinguishable, it is *minimal.* If all states of an FSM are reachable from s_0, it is *initially connected.* If every state is reachable from all states, it is *strongly connected.* In this study, we focus on complete, deterministic, minimal, and initially connected FSMs, which are hereafter called finite state machines.

Definition 2.2 (Featured Finite State Machine). *An FFSM is a septuple $\langle F, \Lambda, C, c_0, Y, O, \Gamma\rangle$, where: (i) F is a finite set of features, (ii) Λ is the set of product configurations, (iii) $C \subseteq S \times B(F)$ 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:*

$$\forall (s, \phi) \in C,\ \exists \rho \in \Lambda | \rho \vDash \phi \tag{1}$$

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

$$\forall ((s, \phi), (x, \phi''), o, (s', \phi')) \in \Gamma, \exists \rho \in \Lambda | \rho \vDash (\phi \wedge \phi' \wedge \phi'') \tag{2}$$

The conditions (1) and (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 $x[\phi]/o$. 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 $\rho \in \Lambda$, the model derivation operator Δ_ρ [29] can derive a product FSM $\Delta_\rho = (S, s_0, I, O, D)$, where: (1) $S = \{s | (s, \phi) \in C \wedge (\phi \vDash \rho)\}$ is the set of states; (2) $s_0 = s, c_0 = (s, \phi) \in C$ is the initial state; (3) $D = \{(s, x, o, s') | ((s, \phi), (x, \phi'), o, (s', \phi'')) \in \Gamma \wedge \rho \vDash (\phi \wedge \phi' \wedge \phi'')\}$ is the set of transitions.

Example 2.3. (The Arcade Game Maker FFSM) Figure 3 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 2, the FSM is an example of product derived using the configuration $\rho = (AGM \wedge A \wedge M \wedge L \wedge V \wedge Y \wedge P \wedge W \wedge \neg S \wedge \neg B \wedge \neg N)$.

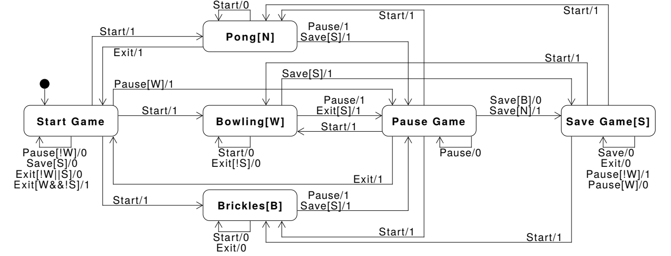

Figure 3: FFSM of the AGM [24]

To make FFSMs suitable for model-based testing, Fragal, Simao and Mousavi [29] propose validation techniques to check if basic properties hold, such as determinism, completeness, initially connectedness, and minimality. Added to this, they also show that any FSM derived from it satisfies the aforementioned properties.

Recently, the FFSM formalism has been extended to generate configurable test suites that can be pruned using feature constraints for groups of product configuration [26]. The readability of FFSMs also has been improved by grouping up conditional states and transitions into hierarchical entities [25]. Thus, the FFSM formalism has the prospect of serving as a suitable basis for family-based testing.

2.3 Comparison of state-based models

Structurally comparing two state machines is a difficult task which involves establishing equivalence relationships between states and transitions. To achieve this goal, Walkinshaw and Bogdanov [69] proposed LTS_{Diff}, an algorithm to compute the precise difference between two state machines. In this section, we discuss the LTS_{Diff} algorithm in terms of FSMs.

2.3.1 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:

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

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 $(a,b) \in S_r \times S_u$.

Example 2.4. (Illustration of a system of linear equations) In Table 1, we depict the system of equations resulting from the comparison of the alternative products from the AGM SPL in Figures 2 and 4. State pairs are represented by the first two letters of their respective names.

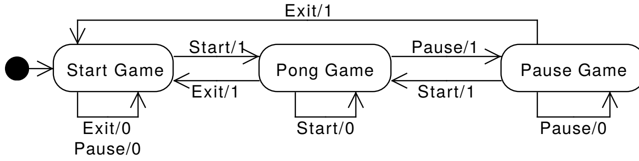

Figure 4: FSM of an alternative product from the AGM SPL

Table 1: Ilustration of a system of linear equations

Pair	(St,St)	(St,Po)	(St,Pa)	(Bo,St)	(Bo,Po)	(Bo,Pa)	(Pa,St)	(Pa,Po)	(Pa,Pa)	
(St,St)	5.0	0.0	0.0	0.0	-0.5	0.0	0.0	0.0	0.0	3
(St,Po)	0.0	12.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
(St,Pa)	0.0	0.0	7.5	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

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 each pair (a,b) are averaged as follows:

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

Algorithm 1: The LTS_{Diff} algorithm

```
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 = ∅ and S(s0_r, s0_u) > 0 then
5  |   KPairs ← (s0_r, s0_u);
6  end
7  NPairs ← ∪_(a,b)∈KPairs Surr(a, b) − KPairs;
8  while NPairs ≠ ∅ do
9  |   while NPairs ≠ ∅ do
10 |   |   (a, b) ← pickHighest(NPairs, PairsToScore);
11 |   |   KPairs ← KPairs ∪ (a, b);
12 |   |   NPairs ← removeConflicts(NPairs, (a, b));
13 |   end
14 |   NPairs ← ∪_(a,b)∈KPairs Surr(a, b) − KPairs;
15 end
16 return(KPairs);
```

2.3.2 The LTS_{Diff} algorithm. Given the averaged scores, the comparison of two models follows a similar process to how humans navigate through an unfamiliar landscape with a map [69]. This process is shown in Algorithm 1.

First, a filtering method denoted by *identifyLandmarks* selects the top t% most equivalent pairs, and, 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 is selected, then the initial states are selected as initial landmarks. Second, the algorithm proceeds from the initial landmarks using the $Surr(a,b)$ function to reach the surrounding states through matching incoming and outgoing transitions and adds them to a set of candidate matched state pairs *NPairs*. Third, the set *NPairs* is iterated in the order of similarity scores. Once a pair (a,b) is selected, it is added to a set of confirmed matches *KPairs* and all elements in *NPairs* that include either a or b are discarded. This process iterates until *NPairs* becomes empty.

3 THE $FFSM_{Diff}$ ALGORITHM

In this section, we introduce the $FFSM_{Diff}$ algorithm, a fully automated technique that combines FSMs into one succinct FFSM [26, 29] and annotates states and transitions with feature constraints. Although our technique is discussed in terms of FFSMs, it can be extended to other family-based notations [12], such as featured transition systems (FTS) [13, 20].

Essentially, $FFSM_{Diff}$ allows to learn an FFSM model (i) from two FSMs, or (ii) including an 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 $\rho_u \notin \Lambda_r$ not included in an FFSM FF_r specifying a set of configurations Λ_r, respectively. In both cases, we assume that the feature model of the SPL and the configurations of the products under learning are known.

3.1 Learning a fresh FFSM

Consider two FSM models $M_r = \langle S_r, s_{0_r}, I_r, O_r, D_r, \delta_r, \lambda_r \rangle$ and $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$ specifying products p_r and p_u that implement configurations $\rho_r = (\bigwedge_{f \in p_r} f) \wedge (\bigwedge_{f \notin p_r} \neg f)$ and $\rho_u = (\bigwedge_{f \in p_u} f) \wedge (\bigwedge_{f \notin p_u} \neg f)$. To learn a fresh FFSM from M_r and M_u, three assumptions are required: (i) M_r, M_u are complete, deterministic, initially connected and minimal FSMs built a priori (e.g., using automata learning [62]), and (ii) their respective configurations ρ_r, ρ_u, and (iii) their feature model are known a priori. Thus, we leverage the comparison of state-based models to conditional states and describe how fresh FFSMs can be learnt from two product FSMs.

Definition 3.1 (FFSM learnt from two configurations). *An FFSM learnt from $\langle M_r, M_u \rangle$ is a tuple $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where (i) $F = (p_r \cup p_u)$, (ii) $\Lambda = \{\rho_r, \rho_u\}$, (iii) $C \subseteq S \times B(F)$ is the set of conditional states satisfying conditions (1) and*

$$\forall (a,b) \in KPairs,\ \exists (c_m, \rho_r | \rho_u) \in C |\ a,b \mapsto c_m, \tag{3}$$

$$\forall s_i \in S_r, (s_i, \cdot) \notin KPairs,\ \exists (c_r, \rho_r) \in C \mid s_i \mapsto c_r, \tag{4}$$

$$\forall s_j \in S_u, (\cdot, s_j) \notin KPairs,\ \exists (c_u, \rho_u) \in C \mid s_j \mapsto c_u \tag{5}$$

(iv) $(c_0, true) \in C$ is the initial conditional state such that $s_{0_r}, s_{0_u} \mapsto c_0$, (v) $Y \subseteq (I_r \cup I_u) \times B(F)$ is a finite set of conditional input symbols, (vi) $O = (O_r \cup O_u)$ is the finite set of output symbols, and (vii) $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying conditions (1-5) and

$$\forall (a_1, b_1), (a_2, b_2) \in KPairs | (a_1, x) \in D_r \wedge (b_1, x) \in D_u \tag{6}$$

$$\wedge\ \lambda_r(a_1, x) = \lambda_u(b_1, x) = o \tag{7}$$

$$\wedge\ \delta_r(a_1, x) = a_2\ \wedge\ \delta_u(b_1, x) = b_2 \tag{8}$$

$$\exists ((c,\phi), (x, \phi'), o, (c', \phi'')) \in \Gamma \text{ where} \tag{9}$$

$$a_1, b_1 \mapsto c\ \wedge\ \phi' = (\rho_u | \rho_r)\ \wedge\ a_2, b_2 \mapsto c' \tag{10}$$

to guarantee that transitions of M_r and M_u that match origin, input, output and destination (6-8) are combined into one conditional transition (9) annotated with the disjunction of their configurations (10); otherwise, for each defined transition of M_r and M_u (11,14), there is a conditional transition (12,15) annotated with their configurations (13,16) as follows:

$$\forall (a_1, x) \in D_r | \lambda_r(a_1, x) = o_r \wedge \delta_r(a_1, x) = a_2 \tag{11}$$

$$\exists ((c_r, \phi),(x, \phi'_r), o_r, (c'_r, \phi'')) \in \Gamma \text{ where} \tag{12}$$

$$a_1 \mapsto c_r\ \wedge\ \phi'_r = \rho_r\ \wedge\ a_2 \mapsto c'_r \tag{13}$$

$$\forall (b_1, y) \in D_u | \lambda_u(b_1, y) = o_u \wedge \delta_u(b_1, y) = b_2 \tag{14}$$

$$\exists ((c_u, \phi),(y, \phi'_u), o_u, (c'_u, \phi'')) \in \Gamma \text{ where} \tag{15}$$

$$b_1 \mapsto c_u\ \wedge\ \phi'_u = \rho_u\ \wedge\ b_2 \mapsto c'_u \tag{16}$$

To identify common states, the $FFSM_{Diff}$ algorithm uses a mapping $\mapsto$ between states to conditional states of the learnt FFSM. This function ensures that the learnt initial conditional state maps to both initial states, i.e., $s_{0_r}, s_{0_u} \mapsto c_0$. Added to this, we replace the lines 4-6 from Algorithm 1 by $KPairs \longleftarrow (s_{0_r}, s_{0_u}) \cup KPairs$. Reduce the complexity of feature constraints, states and transitions are annotated with *simplified configurations* where core features [11] are discarded from their associated formulae.

Example 3.1 (FFSM learnt from two product configurations)**.** In Figure 5, we depict a fragment of the FFSM resulting from the comparison of the FSMs in Figures 2 and 4. 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 constraint $(W \wedge \neg S \wedge \neg B \wedge \neg N)$ is an example of simplified configuration for the product in Figure 2.

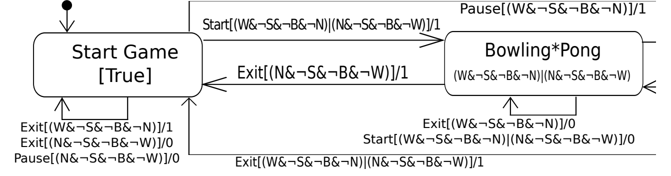

Figure 5: Fragment of the FFSM learnt from the AGM SPL

3.2 Including new product behavior into an existing FFSM

Consider the FFSM $FF_r = \langle F_r, \Lambda_r, C_r, c_{0_r}, Y_r, O_r, \Gamma_r \rangle$ built from a set of configurations Λ_r. If an FFSM FF_r does not include the behavior of an FSM $M_u = \langle S_u, s_{0_u}, I_u, O_u, D_u, \delta_u, \lambda_u \rangle$ specifying a configuration $\rho_u \notin \Lambda_r$, a new FFSM FF can be learnt by comparing and merging $\langle FF_r, M_u \rangle$ using the following definition.

To include a new product into an existing FFSM, three assumptions are required: (iv) FF_r, M_u are complete, deterministic, initially connected, and minimal built a priori, and (v) configurations ρ_u is known in advance, and (vi) the FSM and FFSM under learning share a feature model that is known a priori. Thus, on top of Definition 3.1, we define how an FFSM can be enriched with novel behavior.

Definition 3.2 (FFSM learnt from FF_r and configuration ρ_u). *An FFSM learnt from $\langle FF_r,\ M_u \rangle$ is a tuple $FF = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where (i) $F = F_r \cup \{p_u\}$, (ii) $\Lambda = \Lambda_r \cup \{\rho_u\}$, (iii) $C \subseteq S_r \times B(F)$ is the set of states satisfying conditions (1) and*

$$\forall (a,b) \in KPairs, \exists (a, \phi_a) \in C_r \wedge (c_m, \phi_a | \rho_u) \in C | a,b \mapsto c_m, \tag{17}$$

$$\forall (s_i, \phi_i) \in C_r, (s_i, \cdot) \notin KPairs,\ \exists (c_r, \phi_i) \in C \mid s_i \mapsto c_r, \tag{18}$$

$$\forall s_j \in S_u, (\cdot, s_j) \notin KPairs,\ \exists (c_u, \rho_u) \in C \mid s_j \mapsto c_u \tag{19}$$

(iv) $(c_0, true) \in C$ is the initial conditional state such that the pair $c_{0_r}, s_{0_u} \mapsto c_0$, (v) $Y \subseteq Y_r \cup I_u \times B(F)$ is a finite set of conditional input symbols, (vi) $O = (O_r \cup O_u)$ is the finite set of output symbols, and (vii) $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying conditions (1), (2), (17-19) and

$$\forall (a_1, b_1),(a_2, b_2) \in KPairs \mid (b_1, x) \in D_u \tag{20}$$

$$\wedge\ \lambda_u(b_1, x) = o\ \wedge\ \delta_u(b_1, x) = b_2 \tag{21}$$

$$\wedge\ ((a_1, \phi_{a_1}), (x, \phi_r), o, (a_2, \phi_{a_2})) \in \Gamma_r \tag{22}$$

$$\exists ((c, \phi), (x, \phi'_u), o, (c'_u, \phi'')) \in \Gamma \text{ where} \tag{23}$$

$$a_1, b_1 \mapsto c\ \wedge\ \phi'_u = (\phi_r | \rho_u)\ \wedge\ a_2, b_2 \mapsto c' \tag{24}$$

to guarantee that transitions of FF_r and M_u that match origin, input, output and destination (20-22) are combined into conditional transitions (23) annotated with the disjunction of their constraint and configuration (24); otherwise, for each defined transition of FF_r and M_u (25,28), there is one conditional transitions (26,29) annotated with their configuration/constraint (27,30) as follows:

$$\forall((a_1, \phi_{a_1}), (x, \phi_r), o_r, (a_2, \phi_{a_2})) \in \Gamma_r \quad (25)$$
$$\exists((c_r, \phi), (x, \phi'_r), o_r, (c'_r, \phi'')) \in \Gamma \text{ } where \quad (26)$$
$$a_1 \mapsto c_r \ \wedge \ \phi'_r = \phi_r \ \wedge \ a_2 \mapsto c'_r \quad (27)$$

$$\forall(b_1, y) \in D_u | \lambda_u(b_1, y) = o_u \wedge \delta_u(b_1, y) = b_2 \quad (28)$$
$$\exists((c_u, \phi), (y, \phi'_u), o_u, (c'_u, \phi'')) \in \Gamma \text{ } where \quad (29)$$
$$b_1 \mapsto c_u \ \wedge \ \phi'_u = \rho_u \ \wedge \ b_2 \mapsto c'_u \quad (30)$$

To include a new product into an existing FFSM, lines 4-6 are replaced by $KPairs \leftarrow (c_{0_r}, s_{0_u}) \cup KPairs$. Thus, we guarantee that the initial states of M_u and FF_r map to the same learnt initial state.

4 EMPIRICAL EVALUATION

A family-based analysis technique is effective when its complexity is determined by the number and size of features and the amount of reuse among configurations, rather than the number of valid configurations [58]. Thus, 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* product configurations. By succinct, we mean that the FFSMs learnt have far fewer states than the total number of states in all products under learning, especially if there is high feature sharing. By simplified, we mean that product configuratons are modified by removing core features found using SAT solvers [42].

4.1 Research questions

To evaluate the $FFSM_{Diff}$, we defined three research questions (RQ). In Table 2, we present our hypotheses about the RQs.

Table 2: Hypotheses

RQ	Hypotheses	Description
RQ1	H_0^{RQ1}	The size of learnt FFSMs is larger than the total size of the products analyzed
	H_1^{RQ1}	The size of learnt FFSMs is at most equal to the total size of the products analyzed
RQ2	H_0^{RQ2}	The size of learnt FFSMs is not influenced by the amount of feature sharing
	H_1^{RQ2}	The size of learnt FFSMs is influenced by the amount of feature sharing
RQ3	H_0^{RQ3}	The learnt FFSMs are larger than the hand-crafted FFSMs
	H_1^{RQ3}	The learnt FFSMs have at most the same size as hand-crafted FFSMs

We implemented a tool to compare and combine product FSMs into an FFSM using our algorithm explained in Section 3 and, for feature model analysis, we used the FeatureIDE [59] library and the SAT4J solver [42]. To solve the system of linear equations, we used the Apache Commons Mathematics Library [5]. Details about the experiment design, subject systems and experimental and coding artifacts are presented in Sections 4.2, 4.3 and 4.4, respectively.

As a measure of succinctness, we use the average size of the learnt FFSMs compared to the total size of the products under learning (RQ1) and the size of the hand-crafted FFSMs (RQ3). We describe *size* in terms of *number of states* as it is one of the factors that influences the complexity of model-based techniques [10, 16, 62]. To measure the *statistical significance*, we used the Mann-Whitney test to check if there was significant difference ($p < 0.01$) between the sizes of the learnt FFSMs and products under learning.

To measure the *scientific significance* [38], we used the Vargha-Delaney's $\hat{A}$ effect size [64] to assess the probability of the learnt FFSM being more succinct [7]. If $\hat{A} < 0.5$, then the learnt FFSM is smaller than the total size of the FSMs of products under learning. If $\hat{A} = 0.5$, they have equivalent sizes. To categorize the magnitude of $\hat{A}$, we used the intervals between $\hat{A}$ and 0.5 [33, 60]: $negligible < 0.147 \leq small < 0.33 \leq medium < 0.474 \leq large$.

Finally, we used Pearson's correlation coefficient to evaluate the relationship between succinctness and reuse (RQ2). Thus, we measured the correlation between the ratio of the size of learnt FFSM to the total size of products analyzed on one hand and the ratio of common features to the total of features implemented by the SULs, on the other hand.

4.2 Experiment Design

Let $\{\rho_0, \rho_1, \ldots, \rho_m\} \subseteq B(F)$ be a set of valid configurations, such that the product derived from ρ_i has at most the same number of states as ρ_{i+1}, i.e., they are sorted by their FSM size.

For **RQ1** and **RQ2**, we ran the $FFSM_{Diff}$ for all pairs of products to find common states by solving the system of linear equations and measured the size of the learnt FFSMs. Then, we checked if there were significant and relevant differences between the size of learnt FFSMs and the products analyzed and the correlation between the size of FFSMs and feature reuse. In Figure 6, we illustrate the experiment design.

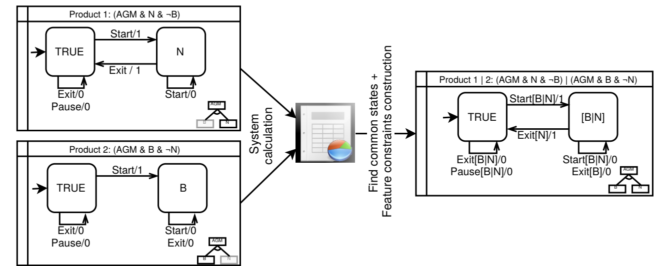

Figure 6: Experiment design

For **RQ3**, added to the FFSMs learnt from pairs of products, we incrementally built FFSMs by merging the FSMs resulting from all products $\bigcup_{i=0}^{j-1}(\rho_i)$ with the FSM of the next product ρ_j, and compared the sizes of the learnt and hand-crafted FFSMs.

4.3 Subject systems

In our evaluation, we used 34 FSMs derived from three SPLs of previous studies [19, 29]. Table 3 depicts the SPLs in terms of the number of features, valid configurations, size of FFSMs and the total sum of the number of states in all valid products analyzed. The AGM SPL is an example from [29] and the Vending Machine (VM) and the Wiper System (WS) are FFSMs hand-crafted by the authors based on FTS [20] from a collection of examples [19].

Table 3: Description of the SPLs under learning

SPL		Number of		Sum total of states in	
ID	Name	Features	Valid config.	FFSM	All products
AGM	Arcade Game Maker	13	6	6	21
VM	Vending Machine	9	20	14	207
WS	Wiper System	8	8	13	56

All the FSMs used in this study were derived from FFSMs using the model derivation operator Δ_ρ [29]. Thus, we had reference FSMs and FFSMs to assess the learnt models, namely, the product-line FFSMs were handcrafted and the intermediate FFSMs and the product FSMs were automatically generated. The VM and WS SPLs are described in the following sections.

4.3.1 Vending Machine. The Vending Machine (VM) is an SPL that we hand-crafted based on featured transition systems from a collection of illustrative SPLs [19]. In Figure 7, we depict the VM feature model.

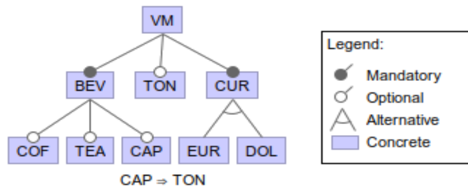

Figure 7: The VM feature model

In our VM SPL, we support three beverages, (i.e., *Coffee*, *Tea*, and *Cappuccino*), one optional *RingTone* played when the beverage is completed, and two alternative currencies (i.e., *Dollar* or *Euro*). These features composed interesting case as they resulted on FSMs with distinct structures and languages. Among the FSMs derived, we highlight two main differences: (i) the addition of states for each of beverage; (ii) changes in the initial state for each currency. This is our largest SPL in terms of number of products and states.

4.3.2 Wiper System. The Wiper System (WS) is another SPL that we hand-crafted based on the aforementioned collection of examples [19]. In Figure 8, we depict the feature model of the WS SPL.

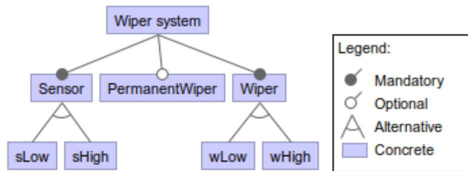

Figure 8: The WS feature model

Our WS SPL has two subsystems – a sensor to detect rain, and the wiper itself; both features are available in two qualities, i.e., *high* and *low*, and one optional feature for permanent movement. A high quality sensor can discriminate between heavy and light rain, whereas a low quality sensor can only distinguish between rain and no rain. Similarly, the high quality wipers can operate at two speeds, and the low quality wiper operates at one single speed.

4.4 Experiment and coding artifacts

For the sake of reproducibility, we have included a lab package with a variety of artifacts (e.g., source-code, test scripts, FFSMs, FSMs, feature models). The lab package is available on GitHub at https://github.com/damascenodiego/learningFFSM.

In our lab package, we have included the subject systems , i.e., `agm`, `vm`, and `ws`; and their respective models (i.e., FSMs, FFSMs), feature models, visual representations and test scripts. For the analysis of results, we have included an RStudio [49] project with R scripts for calculating and plotting statistics. To reproduce our experiments, we have included two sets of Python scripts, i.e., `run_‹ID›_pairs.py` and `run_‹ID›.py`, that execute our tool for each SPL and construct FFSMs from all pairs of products and incrementally learn an FFSM by merging all products.

The source-code of our $FFSM_{Diff}$ implementation is organized as a Java project where two packages are included: i.e., `br.usp.icmc` and `uk.le.ac`. The former includes code artifacts developed by Fragal et al. [29] that we used in the first phase of our study to validate FFSMs and derive FSMs. The latter includes code artifacts that we developed to (i) read/write FSMs; (ii) solve the systems of linear equations to compare the FSMs and FFSMs under learning; and (iii) merge FSM models into annotated FFSMs. The project can be opened using the Eclipse IDE [36] and compiled using the JDK version 1.8.

For reading and writing FSMs, we designed the `ProductMealy` class using the `CompactMealy` class from LearnLib [48]. This class extends basic operations over FSMs (e.g., reset, transition/output functions) with methods to maintain product configurations, as specified by our `IConfigurableFSM` interface.

Using the `Apache Commons Math` library [5], we calculate the state pairs most likely to be equivalent. Based on empirical observations by [69], we have set the attenuation ratio to $k = 0.5$ and, to guarantee the mapping between initial states, we have implemented the *identifyLandmarks*() function to return the pair of initial states (s_{0_r}, s_{0_u}) to *KPairs*.

To represent FFSMs, we designed the `FeaturedMealy` class using the `FastNFA` class, one of the LearnLib building blocks to represent non-deterministic models. We opted for a non-deterministic representation because, if we ignore presence conditions, FFSMs can be seen as a non-deterministic FSM. To represent conditional states/transitions, we designed the classes `ConditionalState` and `ConditionalTransition` with collections of `Node` objects. The `Node` class is a `FeatureIDE` building block to represent feature constraints.

4.5 Analysis of Results

In this section, we discuss the results of our experiments in terms of the RQs we defined to this study and the Hypotheses we have shown in Table 2.

4.5.1 Is the $FFSM_{Diff}$ algorithm effective in learning succinct family models compared to the total size of the products?

In Figure 9, we show the boxplots for the size of learnt FFSMs and total size of all products analyzed. The boxplots indicate that all FFSMs learnt from AGM and WS presented fewer states than the average total size of products. For the VM SPL, on the other hand, there were cases where the learnt FFSMs had more states than the hand-crafted FFSMs.

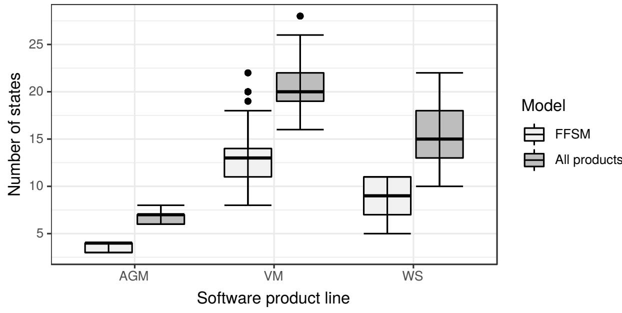

Figure 9: Boxplots of the size of the learnt FFSMs (white) compared to the total size of all products analyzed (gray)

We analyzed the FSMs from the VM SPL and found that modifications in the input/output symbols of early transitions and the addition of extra states lead to significant changes in their structure and language. These changes related to features about the currency and drinks supported by the vending machine, e.g., *Dollar*, *Euro*, and played a key role in outgoing transitions from initial states. As result, these changes masqueraded the similarity of surrounding states and hence difficulted their combination. The outliers above the VM boxplot depict these unsuccessful comparisons.

By analyzing the Mann-Whitney test, we found statistically significant differences ($p < 0.01$) between the size of learnt FFSMs and total size of products. The effect size also indicated differences of *large* magnitude with the size of learnt FFSMs as smaller than the total size of all products analyzed. Thus, our results support the hypothesis H_1^{RQ1} that the size of learnt FFSMs is at most equal to the total size of products analyzed.

4.5.2 *Is the size of learnt family models influenced by the amount of feature reuse?*

In Figure 10, we show the scatter plot for the amount of feature sharing and size of learnt FFSMs for all pairs of products. We have normalized *the size of learnt FFSMs* using the ratio between the number of states of the learnt FFSM to the total number of states of the products analyzed and *the amount of feature sharing* as the ratio of common features to the total of features.

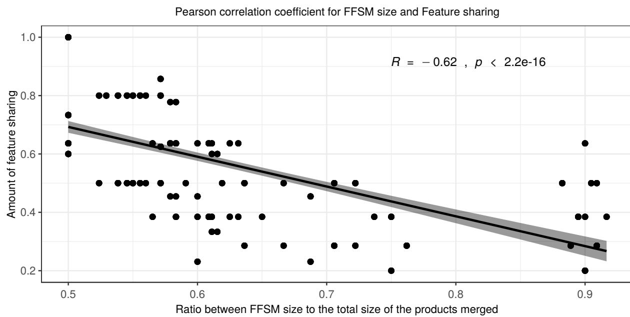

Figure 10: Pearson correlation coefficient between FFSM size and amount of feature sharing

An amount of feature sharing equal to 1.0 means that both products have the same feature configuration. A ratio between size of learnt FFSM and total size of products equal to 0.5 means that the products analyzed implement equivalent FSMs, otherwise the learnt FFSM includes states depicting variability.

By calculating the Pearson correlation coefficient, we found a strong negative correlation between FFSM size and amount of feature sharing. Thus, FFSMs learnt from products implementing a similar set of features tend to be more succinct than those built from products implementing fewer common features. These findings support our hypothesis H_1^{RQ2} that the size of learnt FFSMs is influenced by the amount of feature sharing.

4.5.3 *Is the FFSM$_{Diff}$ algorithm effective in learning succinct family models compared to hand-crafted family models?*

To evaluate the succinctness of the FFSMs learnt using our approach, we also compared the size of hand-crafted models to the FFSMs learnt in two settings: (i) FFSMs incrementally learnt from all products and (ii) FFSMs learnt from pairs of products. In Figure 11, we show the size of learnt FFSMs from all products of the SPLs.

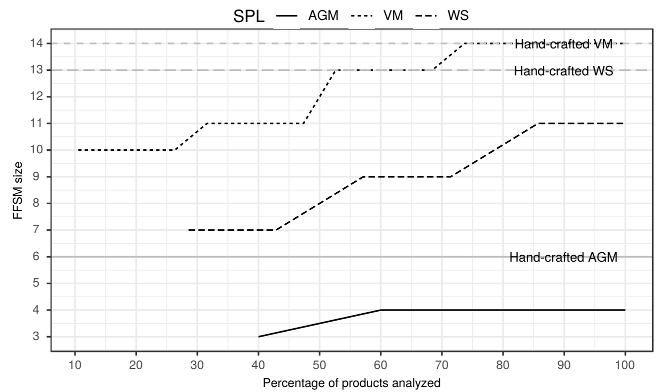

Figure 11: Size of the FFSMs recovered

It turns out that two FFSMs were learnt with fewer states than their original models, i.e., AGM and WS. We inspected the learnt and hand-crafted FFSMs models and found that two states presented similar behaviors and could be merged without side-effects. As shown in Table 3, the hand-crafted FFSMs for the AGM and WS SPLs presented 6 and 13 conditional states, respectively. The FFSMs learnt from AGM and WS, in their turn, included 4 and 11 states, respectively.

Moreover, we found that the FFSMs learnt from the AGM SPL coincided with the alternative representation in Figure 12. This representation was shown by Fragal [24] in his thesis as an alternative representation to the AGM SPL with fewer states. In this alternative representation, all three conditional states are composed into one state annotated with the disjunction of the alternative features.

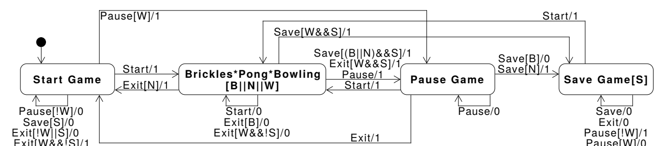

Figure 12: Alternative FFSM for AGM with fewer states [24]

On the other hand, the hand-crafted and learnt FFSMs for the VM SPL presented the same size. In this case, the order that the products were analyzed favoured the $FFSM_{Diff}$ algolrithm to learn an FFSMs with the same size as the original FFSM of VM SPL. These findings indicate that selecting "good" configurations could be helpful for learning more succinct FFSMs. To achieve this, configuration selection [65] and prioritization [32] can be used to improve the overall t-wise coverage each time a configuration is incorporated into an existing FFSM.

To compare the sizes of FFSMs hand-crafted and learnt from product pairs, we calculated the Mann-Whitney test and $\hat{A}$ effect size. In Figure 13, we depict boxplots for the sizes of FFSMs learnt from product pairs. The size of hand-crafted FFSMs is shown as gray lines.

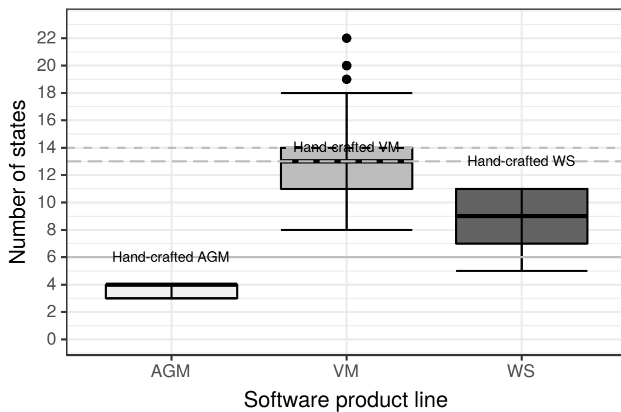

Figure 13: Average size of the learnt FFSMs compared to the size of hand-crafted FFSMs

By analyzing the results of the Mann-Whitney test, we found statistically significant differences ($p < 0.01$) between the size of FFSMs learnt from AGM and WS and their respective hand-crafted versions. For these SPLs, the effect size indicated differences of *large* magnitude with the learnt FFSMs as the smaller models. For the VM SPL, on the other hand, we did not find statistically significant differences ($p > 0.01$) between the size of the learnt and hand-crafted FFSMs. Thus, our results support the hypothesis H_1^{RQ3} that learnt FFSMs have at most the same size as hand-crafted FFSMs.

4.6 Discussion

In this section, we discuss some of the practical implications and limitations of our study.

What are the implications for practitioners? Our work complements traditional reverse engineering techniques on helping practitioners to create family models from product specifications. These specifications can be either created manually or extracted using model learning [62]. As result, our approach can leverage family model-based testing [61] and family model checking [10] to projects where there are no family models a priori.

What types of models may it work/not work? In our current implementation, we fixed *identifyLandmarks*() to the mapping between the initial states (s_{0_r}, s_{0_u}). As a result, some types of changes (e.g., those added to initial states of FSMs) hurdled the process of building succinct models. To improve our approach, more parameters can be added to *identifyLandmarks*(), as in its original design where a threshold t and a ration r are used to find state pairs most likely to be equivalent [69], or either by allowing engineers to set mappings between state pairs as user-defined assumptions.

How are the different notions of variability represented? Currently, our approach annotates state and transitions using the disjunction of simplified configurations. As result, the representation of feature constraints is limited to one single format (i.e., OR with ANDs). To overcome this limitation, more sophisticated presence-condition simplification techniques [67] could be used to reduce the complexity of feature constraints.

4.7 Threats to validity

In this section, we discuss the threats to validity of the methods used in this research paper.

Conclusion validity: These threats concern the relationship between treatment and outcome. To ensure the reliability of our measures and treatment implementation, we have a setup in place based on widely used tools for automata learning [48], SAT solving [42], and SPL analysis [59].

External validity: These concern with the generalization of our results to industrial subject states. Our results are based on three small product lines as subject systems; the small number of product lines and their small sizes pose a threat to external validity. We plan to remedy this by extending our study to more and larger product lines. The fact that the intermediate FFSMs and the product FSMs are generated from the product line FFSM pose another threat to external validity. To remedy this external threat, we need to use learned FSMs along with handcrafted FFSMs from different subsets of larger software product lines. Thus, we could come up with subject systems with more irregularities arising from learning and handcrafting models.

Internal validity: These concern the phenomena that can affect the casual relationship between the treatment and outcomes. One variable that will form a threat to internal validity of our results for larger product lines is the *order* of incorporating product FSMs into product-line FFSMs. In our study, we considered one single order for incorporating products into an FFSM. We plan to incorporate this variable into our experimental setup for larger product lines and complement our study with various configuration selection [65] and prioritization [32] techniques.

Construct validity: These are concerned with 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 and the engineers expertise. Highly specialized engineers are more likely to come up with optimal models (i.e., minimal number of states) than those with less experience. This may hamper the construction of more succinct models. However, optimal representations are interesting artifacts to show that our technique can also recover models *as succinct as* those hand-crafted by experts.

5 RELATED WORK

In this section, we discuss our approach in terms of related work and how it can be helpful in their contexts.

5.1 Automata learning

Automata learning [4] has been a popular approach to automatically derive behavioral models. For an overview of related work and an introduction to automata learning and applications, we refer to [2, 37, 56]. Automata learning have been harnessed for black box model checking [46], real-world protocols [1, 23], software evolution [21, 35], automatic test generation [47], and generalization of failure models [17, 40]. The problem of learning models from SPLs becomes more complex as it has to cope with products that may have their own models, requirements and code. Our approach pave the way for family model learning techniques, which are still understudied.

5.2 Family-based analysis of SPLs

For an overview on techniques for family-model analysis, testing and modeling, we refer the interested readers to recent surveys [12, 15, 58]. Family models have been exploited as theoretical foundation to perform efficient test case generation [9, 14], family model checking [52, 57], to automatically generate specifications of individual products [8], to support the automatic validation of families of products [29], and to specify fine-grained differences among product variants [53]. Thus, we believe that our approach is complementary to the aforementioned techniques in the sense that it can leverage family model-based techniques to cases where family models are non-existent or outdated.

5.3 Comparison of state-based models

The ability to compare FSMs is important for software engineering tasks [69] such as conformance testing [16], and for the sake of evaluating the accuracy of automata learning techniques [62]. Studies related to ours are by Walkinshaw and Bogdanov [69] and Nejati et al. [44].

Walkinshaw and Bogdanov [69] compared two approaches for computing the precise difference between labeled transition systems (LTS) in terms of their language and structure. For comparing the language of state-based models, the authors proposed an approach based on the proportion of test sequences generated by a well-known FSM-badsed testing method, called the W-method [18, 66], that are classified in the same way by two models M_r and M_u. Thus, measurements such as, precision, recall, and F-measure can be used to compare the language of two LTS.

A major issue of comparing FSMs by their language is the fact that minor differences can mask structural similarities. To solve this problem, they proposed the LTS_{Diff} algorithm, presented in Section 2.3.2. These two approaches are said to be *complementary* as two models may have similar state transition structure, but completely different languages, or vice-versa. In the setting of SPLs, the FSMs specifying products can be compared using the LTS_{Diff} algorithm. However, the algorithm lacks a step to incorporate feature constraints. Thus, we designed $FFSM_{Diff}$ to fill this gap.

Nejati et al. [44] presented an approach for matching and merging Statecharts [30]. 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 that relying on both operators produces higher precision than relying on them independently.

In our study, we aimed at the problem of matching and merging FSMs to build FFSMs. Recently, the FFSM formalism has been extended to support hierarchy [25]. The hierarchical featured finite state machine (HFSM) model improves FFSM readability by grouping up FFSM conditional states and transitions into abstracted entities [25]. The results indicate that HFSMs can be used to succinctly represent and efficiently validate the behavior of parallel components in SPLs. The problem of analyzing Statecharts and learning HFSMs provides interesting possibilities to extend our approach.

5.4 Reverse engineering feature models

Feature models play a central role in the variability management for SPLs [11]. Unfortunately, companies often develop software variants in an unstructured manner and may lack feature models as their construction is time-consuming and error prone [31].

In this context, several approaches have been proposed to automatically build feature models from sets of product configurations [3, 31, 51]. Approaches based on Formal Concept Analysis (FCA) show promising possibilities on reverse engineering feature models as they can detect interdependencies and hierarchies between features [3]. Our proposal aims at one similar problem, which can be described as "reverse engineering" family models from product specifications. 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 is required.

5.5 Software product line evolution

In the context of SPLs, the tasks of re-engineering and refactoring are vital to the maintenance and evolution of their software products. For an overview on SPL evolution, refactoring and re-engineering, we refer the interested readers to [22, 41, 43].

A large variety of artifacts have been considered in SPL evolution, but feature models are by far the most researched ones [43]. Moreover, recent studies have shown that there is a need for re-engineering approaches specifically tailored for agile processes [43], and migration of SPL paradigms [41].

Several studies have investigated model learning techniques to cope with traditional software evolution and regression testing [34, 55]. However, to the best of our knowledge, there are no works investigating model learning in the setting of SPLs. Combined with automata learning techniques [4], we believe that our algorithm can support model-based regression testing in SPLs [50] and family model checking [52, 57] in agile processes [45].

6 CONCLUSION

The problem of outdated and deprecated models can arise in the setting of SPLs and hamper the application of family-based analysis. Family-based analysis techniques operate on a single artifact, referred to as a family model, annotated with feature constraints to express variability in terms of states and transitions specific to product configurations. To tackle this issue, we introduce $FFSM_{Diff}$, an automated technique to learn succinct FFSMs from sets of FSMs specifying products.

Our technique incorporates variability to compare and merge FSMs and annotate states and transitions with feature constraints. We designed our approach to cope with (i) learning fresh FFSMs from two FSMs, and (ii) including an FSM into an existing FFSM. To evaluate our technique, we used 34 products derived from three SPLs and measured its effectiveness in terms of the size of learnt FFSMs and amount of feature reuse.

Our results supported the hypothesis that families of FSMs can be effectively merged into succinct FFSMs, especially if there is high feature reuse among products. These indicate that $FFSM_{Diff}$ is an efficient family-based model learning technique that can pave the way to several family-based analysis techniques without family models specified *a priori*.

As future work, we would like to invesitgate further how family models can be used to steer the process model learning. Adaptive learning is a variant of model learning that attempts to reuse sequences from existing models to speed up state discovery [34]. We believe that FFSMs can be useful to derive queries and improve the process of reverse engineering family models. Another possible branch of this research consist of evaluating how configuration prioritization [32] may affect the size of the family models. Our results indicated that some combinations may lead to FFSMs larger than hand-crafted versions and we believe that similarity functions [32] may be useful to accelerate and support family model learning. The problem of learning HFSMs from Statecharts is left as future work.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and University of Leicester, 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 also grateful to the anonymous reviewers; the *VALidation and Verification* (VALVE) research group at the University of Leicester; and the *Laboratory of Software Engineering* (LabES) at the University of Sao Paulo (ICMC-USP) for their useful comments and suggestions.

REFERENCES

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. 2012. *Automata Learning through Counterexample Guided Abstraction Refinement.* Springer, Berlin, Heidelberg, 10–27.

[2] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. 2018. Model Learning and Model-Based Testing. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). Springer International Publishing, Cham, 74–100.

[3] Ra'Fat Ahmad Al-Msie'Deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. 2014. Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In *CLA: Concept Lattices and their Applications*, Karell Bertet and Sebastian Rudolph (Eds.), Vol. 1252. 95–106.

[4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.

[5] Apache. 2016. Commons Math: The Apache Commons Mathematics Library. http://commons.apache.org/proper/commons-math/. [Online; accessed 28-Mar-2019].

[6] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development.* ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2528265.2528267

[7] Andrea Arcuri and Lionel Briand. 2011. 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).* ACM, New York, NY, USA, 1–10.

[8] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2012. *A Compositional Framework to Derive Product Line Behavioural Descriptions.* Springer, Berlin, Heidelberg, 146–161.

[9] Joanne M. Atlee, Sandy Beidu, Uli Fahrenberg, and Axel Legay. 2015. 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. 38–43.

[10] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[12] Fabian Benduhn, Thomas Thüm, Malte Lochau, Thomas Leich, and Gunter Saake. 2015. 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).* ACM, New York, NY, USA, 80–87.

[13] Harsh Beohar and Mohammad Reza Mousavi. 2014. Input-output conformance testing based on featured transition systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14.* ACM Press, New York, New York, USA, 1272–1278. https://doi.org/10.1145/2554850.2554949

[14] Harsh Beohar and Mohammad Reza Mousavi. 2016. InputâĂŞoutput conformance testing for software product lines. *Journal of Logical and Algebraic Methods in Programming* 85, 6 (2016), 1131–1153.

[15] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. 2016. Basic behavioral models for software product lines: Expressiveness and testing preorders. *Science of Computer Programming* 123 (2016), 42–60.

[16] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures.* Springer, Berlin, Heidelberg. 557–603 pages.

[17] Martin Chapman, Hana Chockler, Pascal Kesseli, Daniel Kroening, Ofer Strichman, and Michael Tautschnig. 2015. *Learning the Language of Error.* Springer International Publishing, Cham, 114–130.

[18] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* 4, 3 (May 1978), 178–187.

[19] Andreas Classen. 2010. Modelling with FTS: a Collection of Illustrative Examples. https://researchportal.unamur.be/en/publications/modelling-with-fts-a-collection-of-illustrative-examples

[20] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (Aug 2013), 1069–1089.

[21] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15).* USENIX Association, Berkeley, CA, USA, 193–206.

[22] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2013. A Taxonomy of Software Product Line Reengineering. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14).* ACM, New York, NY, USA, 1–8.

[23] Paul Fiterău-Broştean and Falk Howar. 2017. *Learning-Based Testing the Sliding Window Behavior of TCP Implementations.* Springer International Publishing, Cham, 185–200.

[24] Vanderson Hafemann Fragal. 2017. *Automatic generation of configurable test-suites for software product lines.* Ph.D. Dissertation. Universidade de São Paulo. [Online] http://www.teses.usp.br/teses/disponiveis/55/55134/tde-10012019-085746/.

[25] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. 2019. Hierarchical featured state machines. *Science of Computer Programming* 171 (2019), 67–88.

[26] Vanderson Hafemann Fragal, Adenilso Simao, Mohammad Reza Mousavi, and Uraz Cengiz Turker. 2018. Extending HSI Test Generation Method for Software Product Lines. *Comput. J.* 62, 1 (05 2018), 109–129.

[27] Arthur Gill. 1962. *Introduction to the Theory of Finite State Machines.* McGraw-Hill, New York.

[28] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. *Modeling and Model Checking Software Product Lines.* Springer, Berlin, Heidelberg, 113–131.

[29] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. 2017. *Validated Test Models for Software Product Lines: Featured Finite State Machines.* Springer International Publishing, Cham, 210–227.

[30] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274.

[31] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Reverse Engineering Feature Models from Programs' Feature Sets. In *2011 18th Working Conference on Reverse Engineering*. IEEE, 308–312.

[32] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. 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 (2014), 650–670.

[33] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's d and CliffâĂŹs Delta Under Non-normality and Heterogeneous Variances. In *Annual meeting of the American Educational Research Association*.

[34] David Huistra, Jeroen Meijer, and Jaco van de Pol. 2018. Adaptive Learning for Learn-Based Regression Testing. In *Formal Methods for Industrial Critical Systems (Lecture Notes in Computer Science)*, Falk Howar and Jiri Barnat (Eds.). Springer Publishers, Switzerland, 162–177.

[35] Hardi Hungar, Oliver Niese, and Bernhard Steffen. 2003. *Domain-Specific Optimization in Automata Learning.* Springer, Berlin, Heidelberg, 315–327.

[36] Eclipse IDE. 2019. Eclipse desktop and Web IDEs. https://www.eclipse.org/ide/. [Online; accessed 19-May-19].

[37] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. 2013. Chapter 3 - Model Inference and Testing. Advances in Computers, Vol. 89. Elsevier, 89–139.

[38] Vigdis By Kampenes, Tore Dyba, Jo E. Hannay, and Dag I.K. Sjoberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11 (2007), 1073–1086.

[39] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[40] Sebastian Kunze, Wojciech Mostowski, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2016. Generation of Failure Models through Automata Learning. In *2016 Workshop on Automotive Systems/Software Architectures (WASA).* 22–25.

[41] Miguel A. Laguna and Yania Crespo. 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.

[42] Daniel Le Berre and Anne Parrain. 2010. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

[43] MaÃŋra Marques, Jocelyn Simmonds, Pedro O. Rossel, and MarÃŋa Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Information and Software Technology* 105 (2019), 190–208.

[44] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. 2012. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering* 38, 6 (Nov 2012), 1355–1375.

[45] Johannes Neubauer, Bernhard Steffen, Oliver Bauer, Stephan Windmuller, Maik Merten, Tiziana Margaria, and Falk Howar. 2012. Automated continuous quality assurance. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE, 37–43.

[46] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. 1999. *Black Box Checking.* Springer US, Boston, MA, 225–240.

[47] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. 2009. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer* 11, 4 (2009), 307.

[48] Harald Raffelt and Bernhard Steffen. 2006. *LearnLib: A Library for Automata Learning and Experimentation.* Springer, Berlin, Heidelberg, 377–380.

[49] RStudio. 2019. RStudio: Open source and enterprise-ready professional software for data science. https://www.rstudio.com/. [Online; accessed 19-May-19].

[50] Per Runeson and Emelie Engström. 2012. Chapter 7 - Regression Testing in Software Product Line Engineering. Advances in Computers, Vol. 86. Elsevier, 223–263.

[51] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2011. Extraction of Feature Models from Formal Contexts. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11).* ACM, New York, NY, USA, 1–8.

[52] Hamideh Sabouri and Ramtin Khosravi. 2013. *Delta Modeling and Model Checking of Product Families.* Springer, Berlin, Heidelberg, 51–65.

[53] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. *Delta-Oriented Programming of Software Product Lines.* Springer, Berlin, Heidelberg, 77–91.

[54] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.

[55] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2015. *Incremental Upgrade Checking.* Springer International Publishing, Cham, 55–72.

[56] Andrew Stevenson and James R. Cordy. 2014. A Survey of Grammatical Inference in Software Engineering. *Sci. Comput. Program.* 96, P4 (Dec. 2014), 444–459.

[57] Maurice H. ter Beek, Erik P. de Vink, and Tim A. C. Willemse. 2017. *Family-Based Model Checking with mCRL2.* Springer, Berlin, Heidelberg, 387–405.

[58] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (June 2014), 1–45.

[59] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79, Supplement C (2014), 70–85.

[60] Marco Torchiano. 2017. *effsize: Efficient Effect Size Computation (v. 0.7.1).* CRAN package repository. https://cran.r-project.org/web/packages/effsize/effsize.pdf [Online; accessed 20-November-2017].

[61] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

[62] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95.

[63] Tassio Vale, Eduardo Santana de Almeida, Vander Alves, UirÃą Kulesza, Nan Niu, and Ricardo de Lima. 2017. Software product lines traceability: A systematic mapping study. *Information and Software Technology* 84 (2017), 1–18.

[64] AndrÃąs Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[65] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. 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).* ACM, New York, NY, USA, 1–13.

[66] M. P. Vasilevskii. 1973. Failure diagnosis of automata. *Cybernetics* 9, 4 (1973), 653–665.

[67] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15).* IEEE, Piscataway, NJ, USA, 178–188.

[68] Neil Walkinshaw. 2013. Chapter 1 - Reverse-Engineering Software Behavior. In *Advances in Computers*, Atif Memon (Ed.). Advances in Computers, Vol. 91. Elsevier, 1–58.

[69] Neil Walkinshaw and Kirill Bogdanov. 2013. Automated Comparison of State-Based Software Models in Terms of Their Language and Structure. *ACM Transactions on Software Engineering and Methodology* 22, 2 (March 2013), 1–37.