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  
adensimo@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 \textit{FFSM}\textsubscript{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 \textit{FFSM}\textsubscript{Diff} is an efficient family-based model learning technique.

CCS CONCEPTS

\textbullet{} Networks \rightarrow{} Formal specifications;  
\textbullet{} Theory of computation \rightarrow{} Query learning;  
\textbullet{} Hardware \rightarrow{} Finite state machines;  
\textbullet{} Software and its engineering \rightarrow{} Software product lines.

KEYWORDS

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

1 INTRODUCTION

Analysis and maintenance of software product lines (SPLs) 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 \textit{FFSM}\textsubscript{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):

\begin{itemize}
  \item \textbf{(RQ1)} Is our automated technique effective in learning succinct family models compared to the total size of the products?
  \item \textbf{(RQ2)} Is the size of learnt family models influenced by the amount of feature reuse?
  \item \textbf{(RQ3)} Is our automated technique effective in learning succinct family models compared to hand-crafted family models?
\end{itemize}
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 $A$ 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.
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 $S = \{\text{Start Game, Bowling Game, Pause Game}\}$, $I = \{\text{Start, Pause, Exit}\}$ and $O = \{0, 1\}$.

![Figure 2: Example of FSM [29]](image)

Transition and output functions are lifted to sequences of input in the standard way. Namely, for the empty input sequence $\epsilon$, $\delta(s, \epsilon) = s$ and $\lambda(s, \epsilon) = \epsilon$. For an input $a \cdot x$ defined in state $s$, we have $\delta(s, a \cdot x) = \delta(\delta(s, a), x)$ and $\lambda(s, a \cdot x) = \lambda(s, a) \lambda(\delta(s, a), x)$. An input sequence $a$ is a prefix of $\beta$, denoted by $a < \beta$, if $\beta = a \cdot \omega$, for some sequence $\omega$. An input sequence $a$ is a proper prefix of $\beta$, denoted by $a < \beta$, if $\beta = a \cdot \omega$, for $\omega \neq \epsilon$. The prefixes of a set $T$ of input sequences are denoted by $\text{prefix}(T) = \{a | \exists \beta \in T, a < \beta\}$. If $T = \text{prefix}(T)$, it is prefix-closed.

An input sequence $a \in I^*$ is a transfer sequence from $s$ to $s'$ if $\delta(s, a) = s'$. An input sequence $\gamma$ is a separating sequence for $s_1, s_2 \in S$ if $\lambda(s_1, \gamma) \neq \lambda(s_2, \gamma)$. Two states $s_1, s_2 \in S$ are equivalent if for all $a \in I$, $\lambda(s_1, a) = \lambda(s_2, a)$, otherwise they are distinguishable. An FSM is complete if $S \times I$, otherwise it is partial.

An FSM is deterministic if, for each state $s_1$ and input $x$, there is at most one possible state $s_2 = \delta(s_1, x)$ and output $y = \lambda(s_1, x)$. 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 FSM is a septuple $(F, A, C, c_0, Y, O, \Gamma)$, where: (i) $F$ is a finite set of features, (ii) $A$ 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:

$$\forall (s, \phi) \in C, \exists \rho \in A | \rho \models \phi$$

(iv) $c_0 = (s_0, \text{true}) \in C$ is the initial conditional state of the FSM, (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) $\Gamma \subseteq C \times X \times Y \times O \times C$ is the set of conditional transitions satisfying the condition:

$$\forall ((s, \phi), (x, \phi'), (s', \phi'')) \in \Gamma, \exists \rho \in A | \rho \models (\phi \land \phi' \land \phi'')$$

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 $\&$, $\lor$, and $\neg$, respectively. An omitted condition means that the condition is true.

Given an FSM $FF = (F, A, C, c_0, Y, O, \Gamma)$ and a configuration $\rho \in A$, the model derivation operator $\Delta_\rho$ [29] can derive a product FSM $\Delta_\rho = (S_0, s_0, I, O, D)$, where: (1) $S = \{s(s, \phi) \in C \land (\rho \models \phi)\}$ is the set of states; (2) $s_0 = s, c_0 = (s, \phi) \in C$ is the initial state; (3) $D = \{(s, x, a, s')(s, \phi), (x, \phi'), a, (s', \phi'')\} \in \Gamma \land \rho \models (\phi \land \phi' \land \phi'')$ 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[5] 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 $\rho = (AGM \land A \land M \land L \lor Y \lor P \land W \lor S \lor B \lor N)$.

![Figure 3: FFSM of the AGM [24]](image)

To make FSMS 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 FSM 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 FSMS also has been improved by grouping up conditional states and transitions into hierarchical entities [25]. Thus, the FSM 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 LTSDiff, an algorithm to compute the precise difference between two state machines. In this section, we discuss the LTSDiff algorithm in terms of FSMS.

2.3.1 Similarity score. In the LTSDiff algorithm, the differences between two FSM models $M_r = (S_r, s_0, I_r, O_r, D_r, \delta_r, \lambda_r)$ and $M_d = (S_d, s_0, I_d, O_d, D_d, \delta_d, \lambda_d)$ 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 $a \in S_r, b \in S_d$ using the individual number of pairs of states that can be reached by matching transitions, as follows:

$$\text{Succ}_{a,b} = \{ (c, d, i, o) \in S_r \times S_d \times (I_r \lor I_d) \times (O_r \lor O_d) \text{ such that } \delta_r(a, i) = c, \delta_d(b, i) = d, \text{ and } \lambda_r(a, i) = \lambda_d(b, i) = o \}$$
Second, a global similarity score is calculated by aggregating the scores of states connected to the original pair as follows:

\[ S_{Succ}^G(a, b) = \frac{1}{2} \left( \sum_{(c, d) \in Succ_{a,b}} (1 + k \times S_{Succ}^G(c, d)) \right) \]

An attenuation ratio \(k\) is used to give precedence to state pairs that are closer to the original pair of states and the notation \(\sum_{r}^{out}(a)\) refers to the set of labels of outgoing transitions for state \(a\) of \(M_r\). Thus, the expression \(\left| \sum_{r}^{out}(a) - \sum_{r}^{out}(b) \right| + \left| \sum_{r}^{out}(b) - \sum_{r}^{out}(a) \right|\) 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 \(S_{Succ}^G(a, b)\) is used to build a system of linear equations, such that each equation corresponds to the \(S_{Succ}^G(a, b)\) 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.

### Table 1: Illustration of a system of linear equations

<table>
<thead>
<tr>
<th>Pair</th>
<th>(St,St)</th>
<th>(St,Po)</th>
<th>(St,Pa)</th>
<th>(Bo,St)</th>
<th>(Bo,Pa)</th>
<th>(Pa,St)</th>
<th>(Pa,Pa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(St,St)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(St,Po)</td>
<td>0.0</td>
<td>12.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(St,Pa)</td>
<td>0.0</td>
<td>0.0</td>
<td>12.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(Bo,St)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(Bo,Pa)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(Pa,St)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(Pa,Pa)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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 \(S_{Prev}^G(a, b)\) is calculated in a similar manner.

Consider the systems of equations for \(S_{Succ}^G(a, b)\) and \(S_{Prev}^G(a, b)\), the similarity scores for each pair \((a, b)\) are averaged as follows:

\[ S(a, b) = \frac{S_{Succ}^G(a, b) + S_{Prev}^G(a, b)}{2} \]

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 \textit{identifyLandmarks} selects the top \(f\%\) 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 \(\text{Surr}(a, b)\) 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.

### Algorithm 1: The \(LTS_{Diff}\) algorithm

1. Input: FSM \(M_r\), FSM \(M_u\), \(k\), \(r\);
2. \(PairsToScore \leftarrow \text{computeScores}(M_r, M_u, k)\);
3. \(KPairs \leftarrow \text{identifyLandmarks}(PairsToScore, r)\);
4. if \(|KPairs| = 0\) and \(S(s_0, s_0) > 0\) then
   5. \(KPairs \leftarrow (s_0, s_0)\);
6. end
7. \(NPairs \leftarrow \cup_{(a, b) \in KPairs} \text{Surr}(a, b) - KPairs\);
8. while \(NPairs \neq \emptyset\) do
   9. while \(NPairs \neq \emptyset\) do
      10. \((a, b) \leftarrow \text{pickHighest}(NPairs, PairsToScore)\);
      11. \(KPairs \leftarrow KPairs \cup (a, b)\);
      12. \(NPairs \leftarrow \text{removeConflicts}(NPairs, (a, b))\);
   end
14. \(NPairs \leftarrow \cup_{(a, b) \in KPairs} \text{Surr}(a, b) - KPairs\);
15. end
16. return\((KPairs)\);

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 FSMs, 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 FSM existing a priori and the latter if there is a new configuration \(p_o \notin \Lambda_r\) not included in an FFSM \(FF_r\) specifying a set of configurations \(\Lambda_r\), 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 $M_F = (S_F, q_0, I_F, O_F, D_F, \delta_F, \lambda_F)$ and $M_B = (S_B, q_{0_B}, I_B, O_B, D_B, \delta_B, \lambda_B)$ specifying products $p_F$ and $p_B$ that implement configurations $p_F = (\land \, \delta_F, \Gamma) \land (\lor \, \delta_F, \lnot \Gamma)$ and $p_B = (\lor \, \delta_B, \Gamma) \land (\land \, \delta_B, \lnot \Gamma)$. To learn a fresh FSM from $M_F$ and $M_B$, three assumptions are required: (i) $M_F$ and $M_B$ are complete, deterministic, initially connected, and minimal FSMS built a priori (e.g., using automata learning [62]), and (ii) their respective configurations $p_F$ and $p_B$ (and their feature model are known). To further refine the comparison of state-based models to conditional states and describe how fresh FSMS can be learnt from two product FSMS.

Definition 3.1 (FFSM learnt from two configurations). An FFSM learnt from $(M_F, M_B)$ is a tuple $\langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where (i) $F = (p_F \cup p_B)$, (ii) $\Lambda = \{p_F, p_B\}$, (iii) $C \subseteq S \times B(F)$ is the set of conditional states satisfying conditions (1) and

$$
\forall (a, b) \in KPairs, \exists (cm, pr | p_B) \in C | a, b \mapsto cm.
$$

(3)

$$
\forall s_t \in S_F, (s_t, \cdot) \notin KPairs, \exists (cr, pr | p_B) \in C | s_t \mapsto cr.
$$

(4)

$$
\forall s_j \in S_B, (\cdot, s_j) \notin KPairs, \exists (c_B, p_B) \in C | s_j \mapsto c_B
$$

(5)

(iv) $(c_0, true) \in C$ is the initial conditional state such that $s_0, s_0 \mapsto c_0$, (v) $Y \subseteq (I \cup I_B) \times B(F)$ is a finite set of conditional input symbols, (vi) $O = (O_F \cup O_B)$ 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).

$$
\forall (a_1, b_1), (a_2, b_2) \in KPairs((a_1, x) \in D_F \land (b_1, x) \in D_B)
$$

(6)

$$
\land \lambda_F(a_1, x) = \lambda_B(b_1, x) = o \land \delta_F(a_1, x) = a_2 \land \delta_B(b_1, x) = b_2
$$

(7)

$$
\exists (c, \phi), (x, \phi'), o, (c', \phi'') \in \Gamma
$$

(9)

$$
\exists (a_1, b_1) \mapsto c \land \phi' \Rightarrow (p_B | p_F) \land a_2, b_2 \mapsto c'
$$

(10)

to guarantee that transitions of $M_F$ and $M_B$ 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 $M_F$ and $M_B$ (11,14), there is a conditional transition (12,15) annotated with their configurations (13,16) as follows:

$$
\forall (a, b) \in KPairs, \exists (a, \phi) \in C \land (c, \phi) \in \Gamma | a, b \mapsto cm.
$$

(17)

$$
\forall (s_t, \phi_t) \in C, (s_t, \cdot) \notin KPairs, \exists (o, \phi) \in C | s_t \mapsto cr.
$$

(18)

$$
\forall s_j \in S_B, (\cdot, s_j) \notin KPairs, \exists (c_B, \phi) \in C | s_j \mapsto c_B
$$

(19)

(iv) $(c_0, true) \in C$ is the initial conditional state such that the pair $c_0, s_0 \mapsto c_0$, (v) $Y \subseteq Y_F \cup Y_B \times B(F)$ is a finite set of conditional input symbols, (vi) $O = (O_F \cup O_B)$ 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 (11), (2), (17-19) and

$$
\forall (a_1, b_1), (a_2, b_2) \in KPairs | (b_1, x) \in D_B
$$

(20)

$$
\land \lambda_B(b_1, x) = o \land \delta_B(b_1, x) = b_2
$$

(21)

$$
\exists (a_1, \phi, (x, \phi'), o, (c', \phi'')) \in \Gamma
$$

(22)

$$
\exists (c, \phi), (x, \phi'), o, (c', \phi'') \in \Gamma
$$

(23)

$$
\exists (a_1, b_1) \mapsto c \land \phi \Rightarrow (p_B | p_F) \land a_2, b_2 \mapsto c'
$$

(24)

to guarantee that transitions of $FF_F$ and $M_B$ 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 $FF_F$ and $M_B$ (25,28), there is one conditional transition (26,29) annotated with their configuration/constraint (27,30) as follows:

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 Bowling Game and Bowling Pong were merged into one state BowlingPong where there is one conditional transition with input symbol Exit for each configuration. The constraint $(W \land \lnot S \land \lnot B \land \lnot N)$ is an example of simplified configuration for the product in Figure 2.

3.2 Including new product behavior into an existing FFSM

Consider the FFSM $FF_F = \langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$ built from a set of configurations $\Lambda$. If an FFSM $FF_F$ does not include the behavior of an FSM $M_B = (S_B, q_0, I_B, O_B, D_B, \delta_B, \lambda_B)$ specifying a configuration $p_B \in \Lambda$, a new FFSM FF can be learnt by comparing and merging $(FF_F, M_B)$ using the following definition.

To include a new product into an existing FFSM, three assumptions are required: (iv) $FF_F, M_B$ are complete, deterministic, initially connected, and minimal built a priori, and (v) configurations $p_B$ 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 $FF_F$ and configuration $p_B$). An FFSM learnt from $(FF_F, M_B)$ is a tuple $\langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where (i) $F = F_B \cup \{p_B\}$, (ii) $\Lambda = \Lambda_F \cup \{p_B\}$, (iii) $C \subseteq S \times B(F)$ is the set of states satisfying conditions (1) and

$$
\forall (a, b) \in KPairs, \exists (a, \phi) \in C \land (c, \phi) \in \Gamma | a, b \mapsto cm.
$$

(17)

$$
\forall (s_t, \phi_t) \in C, (s_t, \cdot) \notin KPairs, \exists (o, \phi) \in C | s_t \mapsto cr.
$$

(18)

$$
\forall s_j \in S_B, (\cdot, s_j) \notin KPairs, \exists (c_B, \phi) \in C | s_j \mapsto c_B
$$

(19)

(iv) $(c_0, true) \in C$ is the initial conditional state such that the pair $c_0, s_0 \mapsto c_0$, (v) $Y \subseteq Y_F \cup Y_B \times B(F)$ is a finite set of conditional input symbols, (vi) $O = (O_F \cup O_B)$ 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

$$
\forall (a_1, b_1), (a_2, b_2) \in KPairs | (b_1, x) \in D_B
$$

(20)

$$
\land \lambda_B(b_1, x) = o \land \delta_B(b_1, x) = b_2
$$

(21)

$$
\exists (a_1, \phi, (x, \phi'), o, (c', \phi'')) \in \Gamma
$$

(22)

$$
\exists (c, \phi), (x, \phi'), o, (c', \phi'') \in \Gamma
$$

(23)

$$
\exists (a_1, b_1) \mapsto c \land \phi \Rightarrow (p_B | p_F) \land a_2, b_2 \mapsto c'
$$

(24)

to guarantee that transitions of $FF_F$ and $M_B$ 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 $FF_F$ and $M_B$ (25,28), there is one conditional transition (26,29) annotated with their configuration/constraint (27,30) as follows:
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 $< 0.147 \leq \text{small} < 0.33 \leq \text{medium} < 0.474 \leq \text{large}$.

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 $\rho_i$ has at most the same number of states as $\rho_{i+1}$, i.e., they are sorted by their FSM size.

For RQ1 and RQ2, we ran the FFSMDiff 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](image)

For RQ3, added to the FFSMs learnt from pairs of products, we incrementally built FFSMs by merging the FSMs resulting from all products $\bigcup_{i=0}^{n-1}(\rho_i)$ with the FSM of the next product $\rho_j$, 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

<table>
<thead>
<tr>
<th>SPL</th>
<th>Number of features</th>
<th>Valid config.</th>
<th>Sum total of states in FSMs</th>
<th>All products</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGM</td>
<td>Arcade Game Maker</td>
<td>13</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>VM</td>
<td>Vending Machine</td>
<td>9</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>WS</td>
<td>Wiper System</td>
<td>8</td>
<td>8</td>
<td>13</td>
</tr>
</tbody>
</table>

All the FSMs used in this study were derived from FFSMs using the model derivation operator $\Delta_p$ [29]. Thus, we had reference FSMs and FFSMs to assess the learnt models, namely, the product-line FSMs 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](image)

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](image)

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_diff_pairs.py and run_ID_pairs.py, that execute our tool for each SPL and construct FFSMs from all pairs of products and incrementally learn an FSM by merging all products.

The source-code of our FFSMDiff implementation is organized as a Java project where two packages are included: i.e., br.usp.icmc and uk.ie.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 $(s_0, s_{0b})$ to KPairs.

To represent FFSMs, we designed the FeatureMealy class using the FastSNAF 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 FFSMDiff 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 FSMs.
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 a 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\(_{\text{Diff}}\) 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 favored the FFSM\textsubscript{DiFF} algorithm to learn an FFSM 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 \(^4\text{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 \(H_0: QSS\) 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 \(\text{identifyLandmarks()}\) to the mapping between the initial states \((s_0, s_0_u)\). 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 \(\text{identifyLandmarks()}\), 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 FSM 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 has 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 paves 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-based 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 FFSMs and annotate states and transitions with feature constraints. We designed our approach to cope with (i) learning fresh FFSMs from two FFSMs, 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 FFSMs 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 investigate 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 consists 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 2013/07357-0). The authors are also grateful to the anonymous reviewers; the VA Lidation and Verification (VALVE) research group at the University of Leicester; and the Laboratory of Software Engineering (LaBES) at the University of Sao Paolo (ICMC-USP) for their useful comments and suggestions.

REFERENCES


