doi:10.1093/comjnl/bxy046

Extending HSI Test Generation Method for Software Product Lines

Vanderson Hafemann Fragal[1*], Adenilso Simao[1], Mohammad Reza Mousavi[2] and Uraz Cengiz Turker[3]

[1]*Institute of Math. and Computer Sciences—ICMC, University of São Paulo, São Paulo, Brazil*
[2]*Department of Informatics, University of Leicester, Leicester, UK*
[3]*Department of Computer Engineering, Gebze Technical University, Gebze, Turkey*
[*]*Corresponding author: vanderson.fragal@gmail.com*

Featured Finite State Machines (FFSMs) were proposed as a modeling formalism that represents the abstract behavior of an entire software product line (SPL). Several model-based testing techniques have been developed to support test case generation for SPL specifications, but none support the full fault coverage criterion for SPLs at the family-wide level. In this paper, we propose an extension of the Harmonized State Identifiers (HSI) method, an FSM-based testing method supporting full fault coverage. By extending the HSI method for FFSMs, we are able to generate a single configurable test suite for groups of SPL products that can be instantiated using feature constraints. We implement a graphical tool named ConFTGen to guide the design, validation, derivation and test case generation for state, transition and full fault coverage of FFSMs. Experimental results indicate a reduction of approximately 50% on the number of test cases required to test 20 random SPL products. Also, we investigate the applicability of our method by applying it to a case study from the automotive domain, namely the Body Comfort System.

Keywords: formal modeling; test case generation; software product line; featured finite state machine

Received 22 August 2017; revised 1 March 2018; editorial decision 13 April 2018
Handling editor: Antonella Santone

1. INTRODUCTION

Software Product Line Engineering (SPLE) is a paradigm to develop software where a family of related products (a Software Product Line—SPL) is built out of a common set of core assets, thus reducing the development cost for each individual product [1]. In SPLE, products are built, step-by-step, by adding or removing functionalities, to alleviate software complexity and improve quality.

Similar to the development of single systems, the SPLE process also has several activities that are executed to ensure software quality. Testing is an example of such activities that is performed to ensure quality and to minimize risks. Testing activities account for a large share of overall project costs and are even more challenging in SPLE than in single systems [2]. In several domains, it is highly non-trivial to follow development standards to efficiently test various product configurations in a systematic manner. For example, the standard ISO 26262[1] for safety-critical automotive software states that each developed product configuration should be tested using model-based techniques with a high degree of test coverage under some test criterion [3].

Several techniques, processes and strategies [4–8] were developed for testing SPLs, but many problems are still open in this area of research. First of all, testing every single product configuration individually by using common testing techniques is not feasible for large SPLs due to the huge number of possible configurations. Second, testing products on-demand is unacceptable, due to the scarce time available for product assembly. Moreover, there are other challenges in SPLE including variability, artifact management, test redundancy, overlay and gaps [9, 10].

In the Model-Based Testing (MBT) approach, we select a test criterion to design test cases using a behavioral test model. The resulting test suite execution must be able to turn as many faults into failures as possible [11]. Thus, good test criteria are the results of trade-offs between testing cost and fault detection capability. The full fault coverage is a test criterion with such a trade-off. There are a number of MBT methods

[1] https://www.iso.org/standard/43464.html

[12–14] that use the full fault coverage to generate test suites from Finite State Machines (FSMs). We focus on the Harmonized State Identifiers (HSI) method [13] as it is an improvement of the well-known W method [12]. We do not use incremental test generation approaches such as the P method [14] as they are substantially more complex and our product-line-centered approach, does not need such a level of complexity. Moreover, the HSI method has better performance than the P method and can generate small test suites compared to other non-incremental methods [15].

In this paper, we propose an extension of the HSI method [13] for Featured Finite State Machines (FFSMs) [16]. We bring the HSI method to the family level, by extending several FSM-based notions of coverage and sequences to the family-level setting. By extending the HSI method, we can generate a single configurable test suite for the SPL. Such test suite can be pruned using feature constraints for a (group of) product configuration(s). A feature constraint can also be used in the derivation of sub-FFSM models for specific sets of SPL product configurations. We implemented a graphical support tool named Configurable Full Test Generator (ConFTGen) to guide the design of FFSMs. The tool performs validation, product model derivation and test case generation for state, transition and full fault coverage. The Z3 SMT tool [17] efficiently process feature constraints.

Moreover, we conducted an experimental study with random FFSMs and feature models, and, for each feature model, we selected 20 random products. We decided to randomize feature models and FFSMs to avoid the bias of specific domains. First, a test suite was generated using the extended HSI method for an FFSM and compared to a second unified test suite generated using the original HSI method for the selected 20 individual configurations. The results indicate a reduction in approximately 50% of the number of new tests required for testing using the first test suite compared to the second. The reduction percentage decreases for less than 20 products and increases for more than 20. Moreover, we illustrate and evaluate our approach and tool by means of a case study from the automotive domain, using the Body Comfort System (BCS) for the VW Golf SPL [18]. The results indicate 25% reduction in the number of new tests required for testing a slice of the BCS for six products.

The main contributions of this paper are summarized below:

(1) Proposing an extension of the HSI test case generation method and proving it to coincide with its product-based counterpart.
(2) Implementing a model-based test generation tool with a graphical interface to support design, validation, derivation and generation of family-based test artifacts.
(3) Experimentally assessing the proposal with both a set of random SPLs and a realistic SPL case study.

The remainder of this paper is organized as follows. Section 2 presents some preliminary notions and concepts regarding the HSI method, feature models and FFSMs. Section 3 presents our approach which extends the MBT concepts for a configurable test design including state, transition, and full fault coverage, followed by the HSI extension. Section 4 presents the implementation details of the ConFTGen tool and usage of SMT Solvers. Section 5 illustrates the experimental study with random models and the analysis of the results. Section 6 provides a case study regarding the Body Comfort System. Section 7 provides an overview of the related work and a comparison of relevant approaches in the literature. Section 8 concludes the paper and presents the directions of our future work.

2. BACKGROUND

This section recapitulates the basic concepts and definitions of the HSI method, feature models and FFSMs (mostly from [16]) that we are going to use throughout the rest of the paper.

2.1. HSI method

The HSI method can generate a test suite which attains the full fault coverage to test the behavior of a system represented by a finite state machine.

2.1.1. *Finite state machines*

The classic Finite State Machine (FSM) formalism, presented below, is often used due to its simplicity and rigor for modeling systems such as communication protocols and reactive systems [19].

DEFINITION 2.1. *An FSM M is defined as a 5-tuple (S, s_0, I, O, T), where S is a finite set of states, $s_0 \in S$ is the initial state, I is the set of inputs, O is the set of outputs and T is the set of transitions in the form of $t = (s, x, o, s') \in T$, where $s \in S$ is the source state, $x \in I$ is the input label, $o \in O$ is the output label and $s' \in S$ is the target state.*

The HSI method requires an FSM with the following three properties: (a) determinism—for each pair of state and input, there is at most one outgoing transition; (b) initially connectedness—there is a sequence of transitions to every state from the initial state; and (c) minimality—all pairs of states must behave differently (be distinguishable) by producing different sequences of outputs for some sequence of inputs.

EXAMPLE 1. Figure 1 presents a deterministic, initially connected and minimal FSM $M = (S, s_0, I, O, T)$, where $S = \{1, 2, 3\}$, $s_0 = 1$, $I = \{a, b, c\}$, $O = \{0, 1\}$ and $T = \{(1, a, 1, 2), (1, b, 0, 1), (1, c, 0, 1), (2, a, 0, 2), (2, b, 1, 3), (2, c, 1, 1), (3, a, 1, 2), (3, b, 0, 3), (3, c, 1, 1)\}$. For determinism, there is only

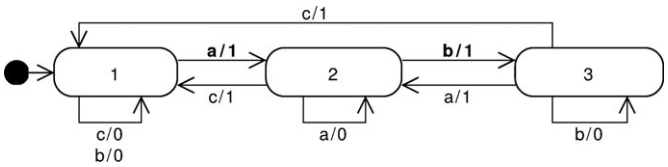


FIGURE 1. Abstract FSM M.

one transition leaving each state for a given input. For initial connectedness, two transitions $(1, a, 1, 2)$ and $(2, b, 1, 3)$ (highlighted) connect the initial state to states 2 and 3. For minimality, the input sequence a results in different output behavior for state pairs 1; 2 and 2; 3, and the input sequence c for 1; 3.

2.1.2. *Full fault coverage criterion*

The full fault coverage criterion is defined using a fault domain.

DEFINITION 2.2. *Consider an FSM $M = (S, s_0, I, O, T)$ with n states and its implementation $N = (Q, q_0, I, O', T')$. The symbol $\Im$ denotes the* fault domain *that is the set of all FSMs that are* (*i*) *deterministic*; (*ii*) *have the same input alphabet as M*; *and* (*iii*) *include all defined input sequences of M. Moreover, $\Im_n$ is the set of FSMs from $\Im$ with n states.*

The following definition summarizes the main results from Simao and Petrenko [14] where the full fault coverage criteria is established based on specific properties.

DEFINITION 2.3. *Given a test-suite T for an FSM M with n states, T is* n-complete *when for all FSMs $N \in \Im_n$ there exist tests in T that distinguish M and N.*

A test suite T satisfies the *full fault coverage criterion* when it contains an n-complete test suite for an FSM M; in that case, by executing T we are capable of detecting any fault (from a failure) in all FSM implementations $N \in \Im_n(T)$.

2.1.3. *Test case generation*

The HSI method is based on the W method [12]; both methods use a characterizing set to distinguish pairs of states in the FSM.

DEFINITION 2.4. *Given an FSM $M = (S, s_0, I, O, T)$ with state set $S = \{s_1, \ldots, s_n\}$, the set $W \subseteq \mathcal{P}(I^*)$ is a* characterizing set *if and only if for all $1 \leq i, j \leq n$ with $i \neq j$ there exists an input sequence $\gamma \in W$ that distinguishes s_i and s_j.*

The HSI method uses the W set to obtain subsets for each state.

DEFINITION 2.5. *Given an FSM $M = (S, s_0, I, O, T)$ with state set $S = \{s_1, \ldots, s_n\}$, the sets $H_1, \ldots, H_n \subseteq \mathcal{P}(I^*)$ are* HSI sets *if and only if for all $1 \leq i, j \leq n$ with $i \neq j$, there exist subsets $H_i \subseteq W$ and $H_j \subseteq W$ for s_i and s_j such that a common prefix γ of $\gamma_i \in H_i$ and $\gamma_j \in H_j$ distinguishes s_i and s_j.*

To generate test cases, we need inputs sequences that reach all transitions (a transition cover set) and the distinguishing set(s). The original W method concatenates each input sequence with the whole W set. The HSI method selects a HSI set for each input sequence, resulting in less test cases.

EXAMPLE 2. The characterizing set W for FSM M presented in Fig. 1 is $W = \{a, c\}$, while the HSI sets are: $H_1 = \{a, c\}$, $H_2 = \{a\}$ and $H_3 = \{a, c\}$. The n-complete test suite is obtained by concatenating a transition cover set $CV = pref(\{b, c, ac, aa, aba, abb, abc\})$ with H_i sets, which results in $TS = pref(\{ba, bc, ca, cc, aca, acc, aaa, abaa, abba, abbc, abca, abcc\})$.

2.2. Feature model

A *feature* is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system [20]. *Feature models* [1] define feature relations based on commonalities and variabilities using graphical models such as feature diagrams. A *feature diagram* [21] uses a notational convention to describe constraint-based feature relations. The basic feature relations are mandatory, optional, inclusive-OR (or), exclusive-OR (alternative), include and exclude [22]. A noteworthy feature modeling method is the Feature-Oriented Domain Analysis (FODA) [20]. Subsequent feature modeling methods extended the FODA to add new dependency relations like the Orthogonal Variability Model (OVM) [23].

EXAMPLE 3. We use in this paper a simplified version of the Car Audio System (CAS) that was presented as a running example in [5]. The CAS SPL can produce different audio systems for cars and provides playbacks and controls. Figure 2 shows the feature diagram of CAS based on FODA. There are two alternative features for the *Player* module (*CD* and *Cassette*) and one optional feature (*USB*) for the *Playback* module. One and only one alternative feature must be selected, and the optional feature may or may not be included.

In general, due to the dependencies and constraints on feature combinations, only some products can be derived. Assume a set of features F of a feature model. The set of all *valid products* P of an SPL is a subset of feature combinations from the power set $\mathcal{P}(F)$ that satisfies the constraints specified by the feature model [24].

A *feature constraint* χ is a propositional formula that interprets elements of the feature set F as propositional variables. The set of all feature constraints is denoted by $B(F)$. Feature relation and constraints of a feature diagram can be extracted

as a feature constraint following a formal semantics [21]. A *product configuration* $\rho \in B(F)$ of a product $p \in P$ is a feature constraint of the form $\rho = (\bigwedge_{f \in p} f) \wedge (\bigwedge_{f \notin p} \neg f)$, i.e. the conjunction of all features present in p and the conjunction of all features absent from p. The set $\Lambda \subseteq B(F)$ denotes all valid product configurations of the SPL. Given a feature constraint $\chi \in B(F)$, a product configuration $\rho \in \Lambda$ *satisfies* χ (denoted by $\rho \vDash \chi$), if and only if the feature constraint $\rho \wedge \chi$ is satisfiable.

EXAMPLE 4. Given the feature diagram of Fig. 2 the extracted feature set is $F = \{A, B, M, L, N, W, Y, U, C, T\}$, where $O = \{A, B, M, L, N, W, Y\} \subseteq F$ is the subset of mandatory features. The extracted feature constraint that represents all valid products is $\chi = ((\bigwedge_{f \in O} f) \wedge (U \Longrightarrow B) \wedge (C \vee T) \wedge \neg(C \wedge T)) \in B(F)$. There are only four product configurations that satisfy χ

$$\rho_1 = (\bigwedge_{f \in O} f) \wedge C \wedge \neg T \wedge \neg U,$$
$$\rho_2 = (\bigwedge_{f \in O} f) \wedge C \wedge \neg T \wedge U,$$
$$\rho_3 = (\bigwedge_{f \in O} f) \wedge \neg C \wedge T \wedge \neg U,$$
$$\rho_4 = (\bigwedge_{f \in O} f) \wedge \neg C \wedge T \wedge U.$$

The feature diagram graphically represents the feature relation while a feature model can be used to represent relevant information for testing. A feature model is defined as follows.

DEFINITION 2.6. *A feature model FM is a triple* (F, χ, Λ), *where* F *is the set of features,* χ *is the feature constraint of all feature relations and* Λ *is the set of product configurations that satisfies* χ.

Given two feature constraints ω_a and ω_b, and Λ_a, $\Lambda_b \subseteq \Lambda$ satisfying ω_a and ω_b, respectively, we say that ω_a and ω_b are *equivalent* under *FM* if $\Lambda_a \subseteq \Lambda_b$ and $\Lambda_b \subseteq \Lambda_a$.

2.3. Featured Finite State Machines

A FFSM [16] is an extension of a Finite State Machine (FSM) in which states and transitions are annotated with feature constraints. The syntax of an FFSM is defined as follows.

DEFINITION 2.7. *An FFSM is a* 6*-tuple* $(FM, C, c_0, Y, O, \Gamma)$, *where*

(1) $FM = (F, \chi, \Lambda)$ *is a feature model* (*Definition* 2.6),
(2) $C \subseteq S \times B(F)$ *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 following condition*:

$$\forall_{(s,\varphi) \in C} \bullet \exists_{\rho \in \Lambda} \bullet \rho \vDash \varphi$$

(3) $c_0 = (s_0, true) \in C$ *is the* initial conditional state,
(4) $Y \subseteq I \times B(F)$ *is a finite set of* conditional inputs, *where* I *is the set of input labels,*
(5) O *is a finite set of* outputs,
(6) $\Gamma \subseteq C \times Y \times O \times C$ *is the set of conditional transitions satisfying the following condition*:

$$\forall_{((s,\varphi),(x,\varphi''),o,(s',\varphi')) \in \Gamma} \bullet \exists_{\rho \in \Lambda} \bullet \rho \vDash (\varphi \wedge \varphi' \wedge \varphi'')$$

The above-given two conditions ensure that every conditional state and every conditional transition is present in at least one valid product of the SPL. A conditional state $c = (s, \varphi) \in C$ is alternatively denoted by $s(\varphi)$. A conditional transition from conditional state c to c' with conditional input $y = x(\varphi'')$ and output o $t = (c, y, o, c')$ is alternatively denoted by $x(\varphi'')/o$ or $c \xrightarrow[o]{y} c'$. The operators of feature constraints are denoted by && (and), || (or) and ! (not). Omitted feature conditions mean that the condition is *true*, i.e. state s is equivalent to $(s, true) \in C$, and $\xrightarrow[o]{x}$ is equivalent to $\xrightarrow[o]{(x,true)}$.

EXAMPLE 5. Figure 3 shows an FFSM for the CAS SPL for the feature model presented in Fig. 2. The playback behavior begins with radio turned off, and once it is turned on it cycles between playbacks starting with radio, then alternatively CD or cassette, and then USB if it was included, otherwise, it gets back to radio again. This is a simple example in which history is not included, thus, the first command is to execute the radio module. From any conditional state except *Off*, the radio can be turned off that can be noticed by the shutdown output. Alternative modules such as CD (green) and Cassette (dark blue) are combined into a single abstract state independent of the selected feature. Specific transitions can represent the specific behavior of each module, i.e. the transitions from *Radio* to *CD*|*Cassette* for *CD* and *Cassette* features, respectively.

We can use a sequence of conditional transitions to form a conditional path to generate test cases. This concept is formalized in the following definition.

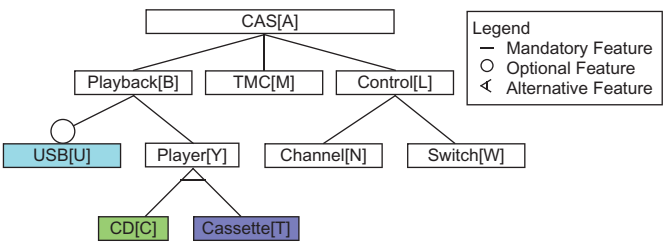


FIGURE 2. CAS Feature Model (adapted from [5]).

DEFINITION 2.8. *Given a conditional input sequence* $\alpha = (y_1, \ldots, y_k) \in Y^*$, *where* $y_i = (x_i, \theta_i) \in Y$ *for* $1 \leq i \leq k$, *a* conditional path *from conditional state* $c_1 = (s_1, \varphi_1)$ *to* c_{k+1} *exists when there are conditional transitions* $t_i = (c_i, y_i, o_i, c_{i+1}) \in \Gamma$, *for* $1 \leq i \leq k$. *A conditional path* σ *is a* 4*-tuple* $(\nu, \delta, \gamma, \omega)$, *where*

(1) $\nu = (c_1, \ldots, c_{k+1}) \in C^*$ *is the conditional state sequence*,
(2) $\delta = (x_i, \ldots, x_k) \in I^*$ *is the input sequence*,
(3) $\gamma = (o_1, \ldots, o_k) \in O^*$ *is the output result*,
(4) $\omega = (\varphi_1 \wedge \cdots \wedge \varphi_{k+1}) \wedge (\theta_1 \wedge \cdots \wedge \theta_k) \in B(F)$ *is the resulting path condition*.

A conditional path is valid *if there is at least a product configuration* $\rho \in \Lambda$ *that can satisfy the path condition* ω, *i.e.* $\exists_{\rho \in \Lambda} \bullet \rho \vDash \omega$. *Notation* $\Theta(c)$ *is used to denote the set of all conditional paths that start at conditional state* $c \in C$. Θ_{FF} *is used to denote* $\Theta(c_0)$.

Further details on model derivation and validation properties for FFSM models can be found in [16].

3. CONFIGURABLE TEST DESIGN

In this section, we extend basic test definitions used in FSM-based test case generation for state coverage, transition coverage, and the HSI method [13] for FFSMs. Only FFSM specifications that are deterministic, initially connected, and minimal (as presented in [16]) are used for test case generation.

3.1. Configurable test suites

To generate conditional test cases, we use sequences of inputs that are valid in at least one product configuration. A configurable test suite, also defined below, is a set of conditional tests.

DEFINITION 3.1. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$ *such that* $FM = (F, \chi, \Lambda)$, *a* conditional test case (*or simply a* conditional test) *of* FF *is a tuple* $(\delta, \omega) \in I^* \times B(F)$, *where* δ *is an input sequence of a valid conditional path* $((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF}$, *and* ω *is the feature constraint of the path. A* configurable test suite $CTS \subseteq \mathcal{P}(I^* \times B(F))$ *of* FF *is a finite set of conditional tests of* FF.

To determine whether a conditional test (δ_a, ω_a) is a conditional prefix of another conditional test (δ_b, ω_b), we use the feature model to compare the configurations satisfied by each feature constraint.

DEFINITION 3.2. *Given an* $FF = (FM, C, c_0, Y, O, \Gamma)$ *such that* $FM = (F, \chi, \Lambda)$, *a conditional test* (δ_a, ω_a) *is a* conditional prefix *of* (δ_b, ω_b) *if*: (*i*) δ_a *is a prefix of* δ_b; (*ii*) *there exist configurations that satisfy both feature constraints, i.e.* $\exists_{\rho \in \Lambda} \bullet \rho \vDash (\omega_a \wedge \omega_b)$; *and* (*iii*) $\Lambda_a \subseteq \Lambda_b$, *where* $\Lambda_a, \Lambda_b \subseteq \Lambda$ *are the subsets of configurations satisfying* ω_a *and* ω_b, *respectively*.

We denote by *cpref* (δ, ω) the set of prefixes of (δ, ω), and *cpref* (CTS) for the prefixes of all tests in a configurable test suite. Moreover, given two conditional tests (δ_a, ω_a) and (δ_b, ω_b), if $\delta_a = \delta_b$, then they can be *merged* into a conditional test $(\delta_a, (\omega_a \vee \omega_b))$. See Section 4.2 for more details about the prefix check using the Z3 tool.

3.2. Test case derivation

The product derivation operator Δ_ϕ induces an FSM from a given FFSM and a given feature constraint ϕ [16]. Similarly, we define the derivation of test suites from a configurable test suite of an FFSM. The feature constraint ϕ is able to filter/prune test cases from the test suite in the same way a configuration model is used for test models.

DEFINITION 3.3. *Let* $FF = (FM, C, c_0, Y, O, \Gamma)$ *be an FFSM, a configurable test suite CTS for* FF, *and a product configuration* $\rho \in \Lambda$. *The derivation operator* Δ_ρ *induces a test suite* $\Delta_\rho(CTS) \subseteq \mathcal{P}(I^*)$ *for an FSM, where*

$$\Delta_\rho(CTS) = \{\delta | (\delta, \omega) \in CTS \wedge \rho \vDash \omega\}$$

.

3.3. State coverage

To define the state coverage criterion for FFSMs, we need to check whether a given FFSM is initially connected. If the given FFSM does not have the property, we cannot generate a configurable state cover set for it. Otherwise, if it is initially connected, then we generate conditional tests from valid conditional paths found for each conditional state. Note that only FFSMs that are deterministic, initially connected, and minimal are used for test design. See [16] for more details about FFSM model validation.

DEFINITION 3.4. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$ *and a conditional state* $c = (s, \varphi) \in C$, *the set* $CTS \subseteq \mathcal{P}(I^* \times B(F))$ covers c *for the state coverage criterion if for all product configurations that satisfy* φ *there is a valid conditional path* $((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF}$ *to reach* c *and* $(\delta, \omega) \in CTS$. *The set CTS is a* configurable state cover *test suite if it covers every conditional state of* FF:

$$\forall_{c=(s,\varphi) \in C} \bullet \forall_{\rho \in \Lambda} \bullet \rho \vDash \varphi \Longrightarrow \exists_{((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF}} \bullet \rho \vDash (\omega \wedge \varphi) \wedge (\delta, \omega) \in CTS$$

EXAMPLE 6. We use the FFSM FF presented in Fig. 3 to identify valid conditional paths for generating a configurable state

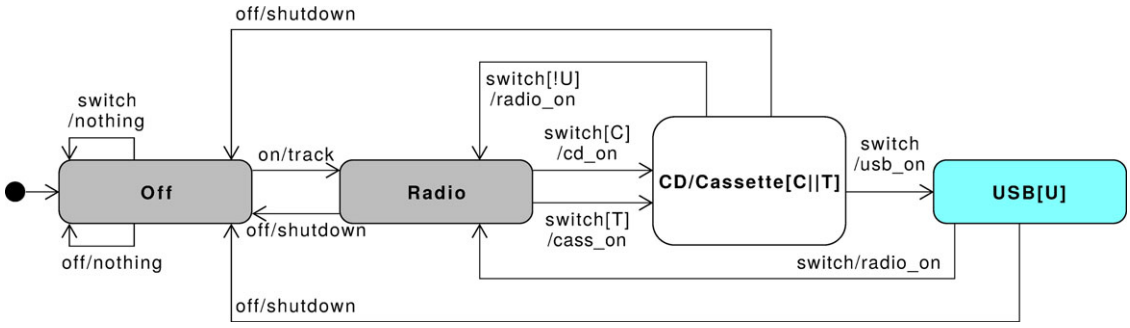


FIGURE 3. FFSM for the CAS SPL.

cover set *CTS*. To cover state *Off*, we need a conditional test with the empty sequence $(\varepsilon, (true))$. For state *Radio*, we can use a single conditional test $(on, (true))$. For state *CD|Cassette*, we use two conditional tests $((on, switch), (C))$ and $((on, switch), (T))$, which can be combined into a single conditional test $((on, switch), (C \vee T))$. Finally, for state *USB*, we need two conditional tests $((on, switch, switch), (C \wedge U))$ and $((on, switch, switch), (T \wedge U))$, which can be combined into a single conditional test $((on, switch, switch), ((C \vee T) \wedge U))$. Thus, the resulting configurable state cover set is $cpref(CTS) = \{((on, switch), (C \vee T)), ((on, switch, switch), ((C \vee T) \wedge U))\}$. Applying the derivation operator for the product configuration $\rho_3 = (\cdots \wedge \neg C \wedge T \wedge \neg U)$ on *CTS*, we derive a test suite $cpref(\Delta_\rho(CTS)) = \{(on, switch)\}$.

Algorithm 1 generates a configurable state cover set. We identify conditional paths using a breadth-first search looking for different paths' combinations excluding self-loops transitions and those that create a loop in the current path resulting in paths no larger than the number of states minus one.

Next, we state and prove that a configurable state cover set is a state cover set for all its valid product FSMs.

THEOREM 3.1. *If the test suite* $CTS \subseteq \mathcal{P}(I^* \times B(F))$ *is a configurable state cover for an FFSM FF, then CTS contains a state cover set* $\Delta_\rho(CTS) \subseteq \mathcal{P}(I^*)$ *for all derived product FSMs* $\Delta_\rho(FF)$.

Proof. We prove the implication by contradiction. Let the set $CTS \subseteq \mathcal{P}(I^* \times B(F))$ be a configurable state cover for an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$ and assume that the set $\Delta_\rho(CTS) \subseteq \mathcal{P}(I^*)$ does not cover a state $s \in S$ for a product configuration ρ under a derived FSM $\Delta_\rho(FF) = (S, s_0, I, O, T)$. By Definition 3.4 for every product configuration $\rho \in \Lambda$ that can be satisfied by the feature constraint of a conditional state $c = (s, \varphi) \in C$ there exist a valid conditional path $((c_0, \ldots, c), \delta, \gamma, \omega) \in \Theta_{FF}$ that reaches c. By Definition 3.3, the conditional input sequence $(\delta, \omega) \in CTS$ derives an input sequence $\delta \in \Delta_\rho(CTS)$ for ρ that reaches s, which leads to a contradiction as $\delta \notin \Delta_\rho(CTS)$. □

Algorithm 1 Configurable state cover set generation

```
1:  procedure CStateCover(FF)
2:      paths ← findNoLoopPaths(FF)
3:      validPaths ← checkSat(paths)
4:      CTS ← {}
5:      for CState cs : FF.getCStates() do
6:          reach ← {}
7:          for Path path : validPaths do
8:              if cs == path.end() then
9:                  reach ← reach ∪ {path}
10:     if checkCov(cs, reach) == false then
11:         return null
12:     CTS ← CTS ∪ reach.getCSequences()
13:     CTS ← ReduceRedundantPaths(CTS, cs)
14:     return CTS
```

3.4. Transition coverage

To extend the transition coverage from FSMs to FFSMs, first we redefine the transition cover set for FSMs. The *transition coverage criterion* uses an input sequence that can reach the source state of a transition, and then concatenate the transition input to the input sequence. A test suite is a transition cover set for an FSM if it can cover all transitions of the FSM.

DEFINITION 3.5. *Given an FSM* $M = (S, s_0, I, O, T)$, *and a state cover set* $TS \subseteq \mathcal{P}(I^*)$ *for M, the test suite TS covers a transition* $(s, x, o, s') \in T$ *if there exist a path* $((s_0, \ldots, s), \alpha, \beta) \in \Omega_M$ *from state* s_0 *to s, where* $\alpha \in TS$ *is the input sequence to reach s,* β *is the output sequence and* $\alpha x \in TS$. *The set TS is a* transition cover *test suite of M if it covers every transition of M*:

$$\forall_{(s,x,o,s') \in T} \bullet \exists_{\alpha \in TS} \bullet \exists_{((s_0, \ldots, s), \alpha, \beta) \in \Omega_M} \bullet \alpha x \in TS.$$

To define the transition coverage criterion for FFSMs, we use valid conditional paths to reach conditional source states of conditional transitions for all valid products, and then include each and every outgoing transition.

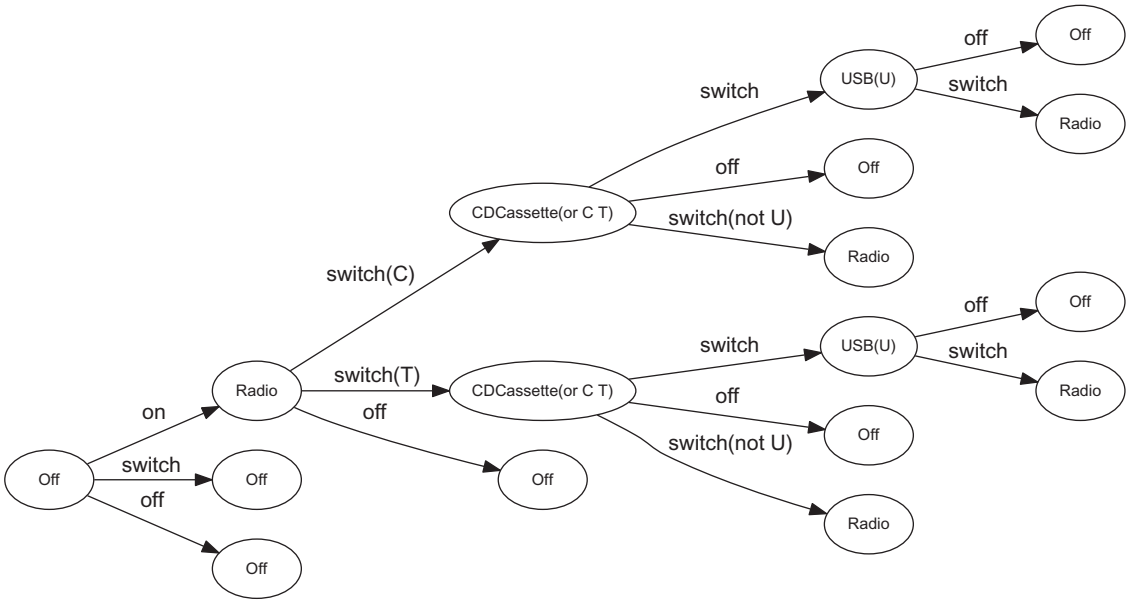


FIGURE 4. Conditional transition tree for AGM.

DEFINITION 3.6. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$, *a configurable state cover* $CTS \subseteq \mathcal{P}(I^* \times B(F))$ *for* FF (*Definition* 3.4), *and a conditional transition* $t = (c_a, (x, \varphi), o, (s_b, \omega_b)) \in \Gamma$, *the test suite* CTS *covers* t *if for all conditional tests* $(\delta, \omega) \in CTS$ *that reach* c_a *such that exists a product configuration* $\rho \in \Lambda$ *that satisfies* $\varphi_t = (\omega \wedge \varphi \wedge \omega_b)$, *then* $(\delta x, \varphi_t) \in CTS$. *The set* CTS *is a* configurable transition cover *test suite if it covers every conditional transition of* FF:

$$\forall_{t \in \Gamma} \bullet \forall_{(\delta,\omega) \in CTS} \bullet \exists_{((c_0,\ldots,c_a),\delta,\gamma,\omega) \in \Theta_{FF}} \bullet \exists_{\rho \in \Lambda} \bullet \rho \vDash \varphi_t \Longrightarrow (\delta x, \varphi_t) \in CTS$$

EXAMPLE 7. Consider the FFSM FF and the configurable state cover set CTS for FF presented in Example 6. Its configurable transition cover set is an extension of $cpref(CTS)$. Figure 4 presents a conditional testing tree generated of CTS. Starting from left to right, the tree shows 11 (the number of leafs in the tree) valid conditional paths. Thus, after merging the conditional tests, substituting long feature constraints with smaller equivalent ones, and removing conditional test prefixes the resulting set is

$(\{(off, true),$
$(switch, true),$
$((on, off), true),$
$((on, switch, off), true),$
$((on, switch, switch), true),$
$((on, switch, switch, off), U),$
$((on, switch, switch, switch), U)\})$

Applying the derivation operator for the product configuration $\rho_3 = (\cdots \wedge \neg C \wedge T \wedge \neg U)$ on CTS, we derive a test suite

$cpref(\Delta_\rho(CTS)) = \{(off), (switch), (on, off),$
$(on, switch, off),$
$(on, switch, switch)\}.$

Algorithm 2 generates a configurable transition cover set. We reuse a subset of the configurable state cover set to reach the source state of a conditional transition, and then we concatenate the conditional transition to each reachable path, check whether if it forms a valid conditional path, and finally add new conditional tests in CTS.

To state and prove that a configurable transition cover set is a transition cover set for all its valid product FSMs, first we redefine a transition cover set for an FSM, then, we move to FFSMs.

THEOREM 3.2. *If the set* $CTS \subseteq \mathcal{P}(I^* \times B(F))$ *is a configurable transition cover for an FFSM* FF, *then* CTS *contains a transition cover set* $\Delta_\rho(CTS) \subseteq \mathcal{P}(I^*)$ *for all derived product FSMs* $\Delta_\rho(FF)$.

Proof. We prove the implication by contradiction. Let the set $CTS \subseteq \mathcal{P}(I^* \times B(F))$ be a configurable transition cover for an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$ and assume that the set $\Delta_\rho(CTS) \subseteq \mathcal{P}(I^*)$ does not cover a transition $t = (s_a, x, o, s_b) \in T$ for a product configuration ρ under a derived FSM $\Delta_\rho(FF) = (S, s_0, I, O, T)$. By Definitions 3.4 and 3.6 for every product configuration $\rho \in \Lambda$ that can be

Algorithm 2 Configurable transition cover set generation

```
1: procedure CTransitionCover(FF)
2:     CTS ← CStateCover(FF)
3:     paths ← recoverValidPaths()
4:     for CState cs : FF.getCStates() do
5:         out ← cs.getOut()
6:         for CTransition t : out do
7:             for Path path : paths.getReach(cs) do
8:                 addNewPath(paths, path, t)
9:         reach ← checkSat(paths.getTReach(t))
10:        if checkTCov(t, reach)==false then
11:            return null
12:        CTS ← CTS ∪ reach.getSequences()
13:        CTS ← ReduceRedundTPaths(CTS, t)
14:    return CTS
```

satisfied by the feature constraint of a conditional state $c_a = (s_a, \varphi_a) \in C$, there exist a valid conditional path $((c_0, \ldots, c_a), \delta, \gamma, \omega) \in \Theta_{FF}$ that reaches c_a, and a transition $t = (c_a, (x, \varphi), o, (s_b, \varphi_b)) \in \Gamma$ such that $(\delta x, (\omega \wedge \varphi \wedge \varphi_b)) \in CTS$. By Definition 3.3, the conditional test $(\delta x, (\omega \wedge \varphi \wedge \varphi_b)) \in CTS$ derives an input sequence $\delta x \in \Delta_\rho(CTS)$ for ρ that reaches s_a concatenated with x, which leads to a contradiction as $\delta x \notin \Delta_\rho(CTS)$. □

3.5. Full fault coverage

To extend the HSI method for FFSMs we are going to use two sets: (i) a configurable transition cover set *CTS*; and (ii) a configurable characterizing set *CW* to distinguish conditional states. We have defined the configurable transition cover set *CTS* in Definition 3.6. Now we need to define parametrized separating sets, define a configurable characterizing set and then build conditional HSI sets for state identification of FFSMs.

3.5.1. *Parametrized separating set*

To distinguish a pair of states of an FSM, we only need one separating sequence. However, to distinguish a pair of conditional states of an FFSM, we need a set of separating sequences.

DEFINITION 3.7. *Let $c_a = (s_a, \varphi_a)$, $c_b = (s_b, \varphi_b)$ be two distinct conditional states of an FFSM FF and also let $\rho \in \Lambda$ be a product configuration for FF under a given SPL. If (i) $\rho \vDash \varphi_a \wedge \varphi_b$; (ii) there exist two valid conditional paths $((c_a, \ldots, c_{a'}), \delta, \gamma_a, \omega_a) \in \Theta_{FF}$ and $((c_b, \ldots, c_{b'}), \delta, \gamma_b, \omega_b) \in \Theta_{FF}$ such that $\rho \vDash \omega_a \wedge \omega_b$ and $\gamma_a \neq \gamma_b$, then a conditional test $(\delta, (\omega_a \wedge \omega_b)) \in I^* \times B(F)$ is called a* separating sequence *for c_a and c_b under the product ρ.*

Note that for a given FFSM, there might be a single separating sequence that can distinguish a pair of states for every satisfiable product configuration. However, we need as many separating sequences as the number of valid product configurations.

EXAMPLE 8. Given the FFSM *FF* presented in Fig. 3 to distinguish the pair of conditional states $(Radio, true)$ and $(CD|Cassette, (C \vee T))$, we use four transitions:

$$(Radio, true) \xrightarrow[cd\_on]{(switch, C)} (CD|Cassette, (C \vee T))$$
$$(Radio, true) \xrightarrow[cass\_on]{(switch, T)} (CD|Cassette, (C \vee T))$$
$$(CD|Cassette, (C \vee T)) \xrightarrow[radio\_on]{(switch, \neg U)} (Radio, true)$$
$$(CD|Cassette, (C \vee T)) \xrightarrow[usb\_on]{(switch, true)} (USB, U)$$

First, the set of product configurations that we need to cover using separating sequences is $true \wedge (C \vee T)$ which is equivalent to all product configurations of Λ. Then, from the transitions, we obtain four separating sequences $s_1 = (switch, (C \wedge \neg U))$, $s_2 = (switch, (C \wedge U))$, $s_3 = (switch, (T \wedge \neg U))$ and $s_4 = (switch, (T \wedge U))$. Next, we can optionally combine/merge those separating sequences into one $s = (switch, true)$. This is the worst case where there are four product configurations and four separating sequences. However, this is unlikely to happen frequently due to the SPL commonalities. For example, for another pair of conditional transitions such as $(Radio, true)$ and $(Off, true)$ we need a single separating sequence $(off, true)$.

Given an FFSM and an SPL, there may be exponentially many number of valid product configurations, hence it may not be practical to derive a separating sequence for each valid product configuration. Instead, we compute separating sequences for sets of product configurations and put in a *parametrized separating set*.

DEFINITION 3.8. *Let $c_a = (s_a, \varphi_a)$, $c_b = (s_b, \varphi_b)$ be two conditional states of an FFSM FF such that there exists at least one product configuration $\rho \in \Lambda$ that satisfying both feature constraints, i.e. $\rho \vDash \varphi_a \wedge \varphi_b$. The set $PS_{ab} \in \mathcal{P}(I^* \times B(F))$ is called a* parametrized separating set *for c_a and c_b if the disjunction of conditions of every separating sequence $(\delta, \omega) \in PS_{ab}$ (Definition 3.7) is equivalent to $\varphi_a \wedge \varphi_b$. Thus, PS_{ab} can distinguish c_a and c_b.*

EXAMPLE 9. From Example 8, we know that $c_a = (Radio, true)$, $c_b = (CD|Cassette, (C \vee T))$ which results in $PS_{ab} = \{(switch, (C \wedge \neg U)), (switch, (C \wedge U)), (switch, (T \wedge \neg U)), (switch, (T \wedge U))\}$, or simply $PS_{ab} = \{(switch, true)\}$.

3.5.2. State identification

The configurable characterizing set CW for state identification of FFSMs is built using parametrized separating sets for every conditional state pair of the FFSM.

DEFINITION 3.9. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$, *with conditional state set* $C = \{c_1, \ldots, c_n\}$, *the set* $CW \in \mathcal{P}(I^* \times B(F))$ *is a* configurable characterizing set, *if and only if, for all* $1 \leq i, j \leq n$ *with* $i \neq j$ *and* $\Lambda_i \cap \Lambda_j \neq \varnothing$, *there exists a parametrized separating set* $PS_{ij} \subseteq CW$ (*Definition* 3.8).

EXAMPLE 10. In Example 9, we obtained the parametrized separating set for conditional states $c_a = (Radio, true)$, $c_b = (CD|Cassette, (C \vee T))$. Combining other pairs of conditional states $s_c = (Off, true)$ and $s_d = (USB, U)$, we have

$$PS_{ab} = \{(switch, true)\}, \quad PS_{ac} = \{(off, true)\},$$
$$PS_{ad} = \{(switch, U)\}, \quad PS_{bc} = \{(off, true)\},$$
$$PS_{bd} = \{(switch, U)\}, \quad PS_{cd} = \{(off, true)\}.$$

The resulting configurable characterizing set (after removing conditional prefixes—Definition 3.2) is $cpref(CW) = \{(switch, true), (off, true)\}$.

The conditional HSI sets for state identification of FFSMs are built using the configurable characterizing set CW.

DEFINITION 3.10. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$, *with conditional state set* $C = \{c_1, \ldots, c_n\}$, *the sets* $CH_1, \ldots, CH_n \in \mathcal{P}(I^* \times B(F))$ *are conditional HSI, if and only if for all* $1 \leq i, j \leq n$ *with* $i \neq j$ *and* $\Lambda_i \cap \Lambda_j \neq \varnothing$, *there exists a common parametrized separating subset* $PS_{ij} \subseteq CW_i \cap CW_j$ (*Definition* 3.8) *with conditional prefixes from* CW *that distinguishes* c_i *and* c_j *using separating sequences*.

EXAMPLE 11. The conditional characterizing set CW of Example 10 for the FFSM FF presented in Fig. 3 is: $CW = \{(off, true), (switch, true)\}$. The conditional HSI sets for the FFSM FF are

$$CH_{Off} = \{(off, true)\},$$
$$CH_{Radio} = \{(off, true), (switch, true)\},$$
$$CH_{CD|Cassette} = \{(off, true), (switch, true)\}, \text{ and}$$
$$CH_{USB} = \{(off, true), (switch, U\}.$$

3.5.3. Extended HSI method

Now we have sets CTC for transition coverage and conditional HSI sets for state identification. Then, the final configurable test suite CTS is the concatenation of CTC with every CH_i.

DEFINITION 3.11. *Given an FFSM* $FF = (FM, C, c_0, Y, O, \Gamma)$, *the configurable transition cover set CTC and the conditional HSI* CH_i *sets, the extended conditional HSI method defines a* complete configurable test suite CTS *for* FF *by concatenating every tuple of CTC with every* CH_i *set for each* $c_i \in C$ *such that only conditional tests of CTC that reach* c_i *are concatenated and the conjunction of constraints is* satisfied

$$\forall_{c_i \in C} \bullet \forall_{(\delta,\omega) \in CTC} \bullet \exists_{((c_0,\ldots,c_i),\delta,\gamma,\omega) \in \Theta_{FF}} \bullet \forall_{(\beta,\omega') \in CH_i} \bullet \exists_{\rho \in \Lambda} \bullet \rho \vDash (\omega \wedge \omega') \Longrightarrow (\delta\beta, (\omega \wedge \omega')) \in CTS$$

EXAMPLE 12. The complete configurable test suite is obtained by concatenating CTC of Example 7 with the conditional HSI sets presented in Example 11. The following set is a complete configurable test suite for FF

$cpref(CTS) = ((off, off), true),$
$((switch, off), true),$
$((on, off, off), true),$
$((on, switch, off, off), true),$
$((on, switch, switch, off), true),$
$((on, switch, switch, switch), true),$
$((on, switch, switch, off, off), U),$
$((on, switch, switch, switch, off), U),$
$((on, switch, switch, switch, switch), U)\})$

Applying the derivation operator for the product configuration $\rho_3 = (\cdots \wedge \neg C \wedge T \wedge \neg U)$ on CTC, we derive a test suite

$cpref(\Delta_\rho(CTC)) = \{(switch, off)(on, off, off),$
$(on, switch, off, off),$
$(on, switch, switch, off),$
$(on, switch, switch, switch)\}.$

Next, we state and prove that a complete configurable test suite is a complete test suite for all its valid product FSMs.

THEOREM 3.3. *If the set* $CTS \subseteq \mathcal{P}(I^* \times B(F))$ *is an n-complete configurable test suite for an FFSM* FF, *then* CTS *contains an n-complete test suite* $TS \subseteq \mathcal{P}(I^*)$ *for all derived product FSMs* $\Delta_\rho(FF)$.

Proof. We prove the implication by contradiction. Let the set $TS \subseteq \mathcal{P}(I^* \times B(F))$ be an n-complete configurable test suite for an FFSM $FF = (F, \Lambda, C, c_0, Y, O, \Gamma)$ and the set $\Delta_\rho(TS) \subseteq \mathcal{P}(I^*)$ be not an n-complete configurable test suite derived from a product configuration ρ under a derived FSM $\Delta_\rho(FF) = (S, s_0, I, O, T)$. As $\Delta_\rho(TS)$ is not an n-complete configurable test suite for state $\Delta_\rho(FF)$, then does not exist a test $\delta h \in \Delta_\rho(TS)$ such δ exists in a path to reach a state s and h distinguishes s with another state s'. By Definitions 3.10 and 3.11 for each conditional state $c_i \in C$, we have at least one conditional test $(\delta h, (\omega \wedge \omega')) \in TS$, $\rho \vDash (\omega \wedge \omega')$ where $(\delta, \omega) \in CTC$ reaches c_i and $(h, \omega') \in CH_i$ distinguishes c_i with another conditional state $c_j \in C$, $i \neq j$. By

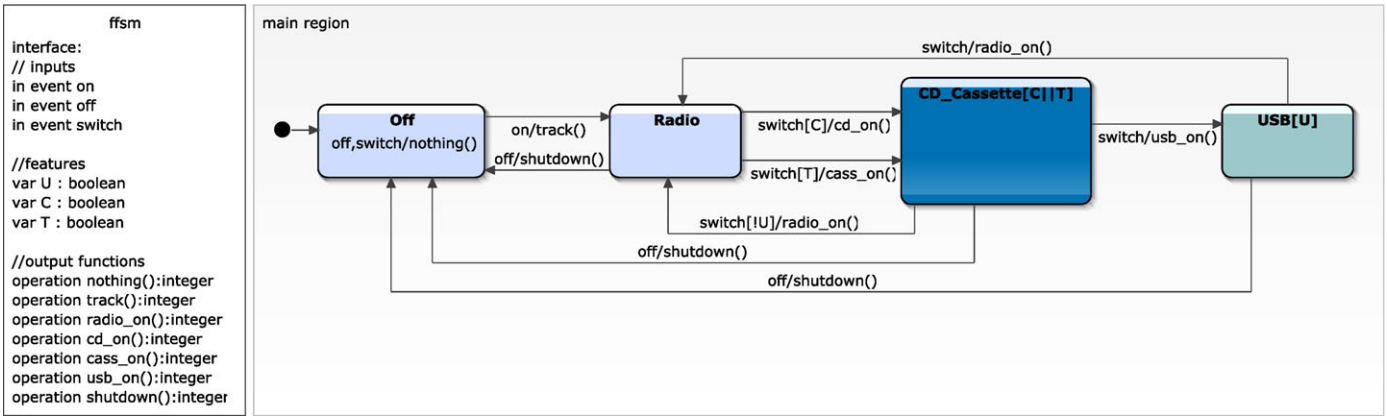


FIGURE 5. ConFTGen tool graphical interface for CAS SPL.

Definition 3.3, the conditional input sequence $(\delta h, (\omega \wedge \omega')) \in TS$ derives an input sequence $\delta h \in \Delta_\rho(TS)$ for ρ, which is a contradiction as $\delta h \notin \Delta_\rho(TS)$. □

4. TOOL SUPPORT

This paper reports on the design, implementation and application of a test design tool Configurable Full Test Generator (ConFTGen).[2] The ConFTGen has a graphical editor based on the Eclipse platform that was extended from the Yakindu GitHub Project[3] (Eclipse Public Licence) and integrated with FeatureIDE [25] (Lesser General Public Licence—LGPL) and Z3 SMT Solver [17] (MIT licence) for constructing feature models and analyzing feature constraints, respectively. ConFTGen supports the automatic validation and derivation of FFSM models, and automatic generation of test suites for state, transition, and full fault coverage. Figure 5 shows the FFSM for CAS SPL in the ConFTGen tool. Inputs, outputs and some features are declared on the left, and behavior represented inside conditional states are self-loop transitions.

4.1. Eclipse platform

The Eclipse Platform is a framework that provides an open source software development environment. Eclipse has a core architecture that supports the integration of tools and other development environments. Moreover, Eclipse projects are implemented in Java language and can be run in several operating systems including Linux, Mac OS X and Windows.

The functional unit of Eclipse is plug-ins. Plug-ins are combined with the core architecture and can be integrated to build complex tools. As stated before, the ConFTGen tool extends plug-ins from the Yakindu Project. The core plug-ins used to develop the ConFTGen tool were *org.eclipse.core, org.eclipse.ui, org.eclipse.gef, org.eclipse.emf, org.eclipse.gmf.*

4.2. Satisfiability Modulo theories solvers

The ConFTGen tool has been implemented on the Eclipse platform, and we used the Z3 Satisfiability Modulo Theory (SMT) Solver [17] to process feature constraints. It is well-known that feature models can be translated into propositional formulas; see, e.g. [24, 26]. This translation enables mechanizing the analysis of feature-based specifications using existing logic-based tools, such as SMT solvers. We present how SMT solvers are used for test design by checking prefix and equivalence relations based on feature constraints. We used the Java language to parse and process FFSMs and subsequently generate assertions in the SMT file format.

The Z3 tool is used for checking conditional prefixes and equivalence relations for a pair of conditional tests (δ_a, ω_a), $(\delta_b, \omega_b) \in I^* \times B(F)$, where $\Lambda_a \subseteq \Lambda$ is the subset of configurations that the constraint ω_a can satisfy, and $\Lambda_b \subseteq \Lambda$ for ω_b. To check whether (δ_a, ω_a) is a conditional prefix of (δ_b, ω_b), we perform the following steps:

(1) If δ_a is a prefix of δ_b, then we assert the feature constraint χ of the feature model $FM = (F, \chi, \Lambda)$ and the logical conjunction of feature constraints $(\omega_a \wedge \omega_b)$. If it returns unsatisfiable, then there is no common configuration between both constraints $\Lambda_a \cap \Lambda_b = \varnothing$ and we cannot derive/deduce the prefix relation.
(2) If the first check returns satisfiable, then, another check identify the prefix relation $\Lambda_a \subseteq \Lambda_b$. We assert the first condition ω_a followed by the negation of the second $\neg\omega_b$ to check whether $\Lambda_a \cap (\Lambda \backslash \Lambda_b) = \varnothing$ is unsatisfiable. If the proposition turns out to be

[2]Open Source Yakindu Project https://github.com/vhfragal/ConFTGen-tool
[3]Open Source Yakindu Project https://github.com/Yakindu/statecharts

satisfiable, then there are configurations for ω_a which $\neg\omega_b$ was not able to invalidate, i.e. $\Lambda_a \nsubseteq \Lambda_b$. Otherwise, when the result is unsatisfiable, then, $\neg\omega_b$ was able to invalidate all configurations of ω_a, i.e., $\Lambda_a \subseteq \Lambda_b$.

EXAMPLE 13. Figure 6 presents parts of the generated SMT file used to check whether a conditional test $(\delta_a, \omega_a) = (a, (B \wedge S))$ is a conditional prefix of $(\delta_b, \omega_b) = (aa, (B))$, where (a) all features are declared as Boolean variables, (b) the feature constraint of the feature model is asserted and (c) the prefix relation checks are asserted. In Z3, `push` and `pop` commands are used to temporarily set the context (e.g. with assertions), and once a verification goal is discharged the context can be reset. The $(check - sat)$ command is used to evaluate the assertions which return *sat* or *unsat*. First, we notice that a is a prefix of aa, then, checking the conjunction of both conditions $((B \wedge S) \wedge B) \equiv (B \wedge S)$ results in *sat*. Next, we assert the first constraint $(B \wedge S)$ and also assert the negation of the second constraint $\neg(B)$ to eliminate configurations of the first, which results in *unsat*. Thus, $(a, (B \wedge S))$ is a conditional prefix of $(aa, (B))$.

The Z3 tool is also used for checking feature constraints in most definitions in this paper. The usage is a variation of Example 13 where we change the (c) part of the SMT file. Given a feature constraint $\varphi \in B(F)$ that requires checking, we use $(assert\ \varphi)$ in (c) to check if it returns *sat* or *unsat*.

5. EXPERIMENTAL STUDY

To evaluate the applicability and the efficiency of our approach, we conducted an experiment to evaluate random FFSMs and random feature models. We focus on the characteristics of the models, the comparison of test suite size and test generation time against the product-by-product approach.

Focusing on the full fault coverage, we are aware of some incremental test case generators for FSMs [14, 27, 28]. These methods can process an existing test set and increment few tests for a new product configuration. Despite the good results on the few number of new tests, these methods have to process the test set for every new product configuration. Moreover, they are quite sensitive to the size of the test set used as input. Hence, they may not scale well to increment large test sets for large specifications.

We aimed at answering the following research questions:

- Q1—Is there a difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM?
- Q2—In which scenario do we reduce the number of test cases using an FFSM instead of FSMs?
- Q3—In which scenario do we have smaller test generation times using an FFSM instead of FSMs?
- Q4—Is there a relation between the feature model and the configurable test suite size?

(a)

```
(define-sort Feature () Bool)
(declare-const G Feature)
(declare-const A Feature)
(declare-const M Feature)
(declare-const L Feature)
(declare-const C Feature)
(declare-const R Feature)
(declare-const B Feature)
(declare-const N Feature)
(declare-const W Feature)
(declare-const V Feature)
(declare-const Y Feature)
(declare-const P Feature)
(declare-const S Feature)
```

(b)

```
(assert G)
(assert (and
   (= A G)
   (= M A)
   (= L A)
   (= C G)
   (= R G)
   (= (or B N W) R)
   (not (and B N))
   (not (and B W))
   (not (and N W))
   (= V G)
   (= Y V)
   (= P V)
   (=> S V)
))
```

(c)

```
(push)
(assert (and (and B S) B) )
(check-sat)
(pop)

(push)
(assert (and B S) )
(assert (not B) )
(check-sat)
(pop)



RESULT
sat
unsat
```

FIGURE 6. SMT parts for checking a conditional prefix relation.

5.1. Experimental setup

Our experiment aims at generating test suites (full fault coverage) for random FFSMs and random feature models. We choose to use random FFSMs to avoid bias of specific domains. Different feature models also were randomized to use in combination with a respective FFSM. Different feature structures may result in different numbers of product configurations which we may check whether it affects validation and generation or not. The adaptable parameters (independent variables) include random feature models that are generated with the following parameters required for the BeTTy tool [29]:

- Number of features (20).
- Percentage of custom constraints (0).
- Probability of a feature being mandatory (0).
- Probability of a feature being optional (33).
- Probability of a feature being in an or-relation (33).
- Probability of a feature being in an alternative-relation (33).
- Maximal branching factor (10).
- Maximal number of children in a set relationship (10).

The probability parameters use numbers from 0 to 100 (in percentage) and their sum must not exceed 100. We decided to use no custom constraints and mandatory features. Custom constraints may be redundant to constraints defined by the feature structure, and they are not used in order to simplify the analysis. About mandatory features, we have only one that is the root feature which represents the core specification. Using more mandatory features may result in SPLs with fewer product configurations as we are using a fixed number of 20 features. Moreover, we only use feature constraints with non-mandatory features in the FFSM model. Thus, all mandatory behavior is implicitly represented in the core specification.

The maximal branching factor limits the number of branches in the feature model structure. The number of children in a set relationship limits the number of features that may be in an alternative/or/optional subset relation.

To visualize some characteristics for the relation/influence of feature models and FFSMs, we divided 20 random feature models into two groups. The first group has feature models with less than the median of independent features. The second group has feature models with more than the median of independent features. An independent feature is placed in an or-relation or set in parallel with other optional features, which makes the number of product configurations grow exponentially.

To answer our research questions, we executed three groups of FFSMs. Each group has 20 random FFSMs and 20 random feature models. The FFSM groups are

(1) FFSMs with 12 conditional states, 10–15 inputs/outputs and 100–450 transitions.
(2) FFSMs with 18 conditional states, 15–20 inputs/outputs and 225–800 transitions.
(3) FFSMs with 24 conditional states, 20–25 inputs/outputs and 400–1250 transitions.

Each random FFSM is generated by randomizing the target state of conditional transitions. We assume that only one-third of the conditional states and transitions have random feature constraints. Thus, two-third of the behavior is part of the core specification. We use FFSMs that are deterministic, initially connected and minimal.

The fixed parameters (controlled variables) are

- Each random FFSM is linked (for feature mapping) to one random feature model.
- The number of inputs and outputs are the same as the number of FFSM states.
- One derived FSM (from the FFSM) represents the core specification of the SPL.
- The test generation method HSI used for FSM and FFSMs.

Then, we measure (dependable variables):

(1) The number of new different tests: We measure the number of new tests that are required to test all products using an FFSM-based test suite against 20 random products with individual FSM-based test suites.
(2) Test suite size for the FSM core: We measure the size of the test suite pruned for the FSM core specification from an FFSM-based test suite against generating the test suite directly from the FSM core specification.
(3) Checking and generation time: We measure the amount of time it takes to generate tests for an entire FFSM against one FSM and a set of FSMs.

The running environment used Ubuntu 15.04 (64 bit) operating system on an Intel processor i7-5500U at 2.40 GHz with 12 GB of RAM.

5.2. Analysis and threats to validity

To answer our questions, we generated complete configurable test suites for all three groups of FFSMs. As stated before, each group has 20 random FFSMs, such that each group has larger FFSM specifications than the previous group. For each FFSM, we derived an FSM which represents the SPL core specification (core product). The collected data in our experiments are analyzed below.

5.2.1. *Q1—Is there a difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM?*

Results: To answer this question, we used a one sided test assuming that the null hypothesis is true. Our null hypothesis (H0) is: ($mu = a$) the true mean of complete test suites for the core product using the HSI method. Our alternative hypothesis is: ($mu > a$) the mean of complete test suites for the core product pruned from a configurable complete test suite generated using an FFSM is larger than a. Figure 7 shows the results on the test suite size comparing our H0 and H1 for all three FFSM groups. The P-value of a normal distribution of all three groups (12, 18 and 24 states) is respectively, 0.149, 0.182 and 0.140.

Analysis: We noticed that the results from the first FFSM group continue in the other two groups as it increases the FFSM size. There is a small difference favoring the direct application of the HSI on the core product over pruning a configurable test suite from the extended HSI method for the same core product. Thus, our experiment does not indicate a statistically meaningful difference between generating a test suite for an FSM and pruning a configurable test suite for the same FSM.

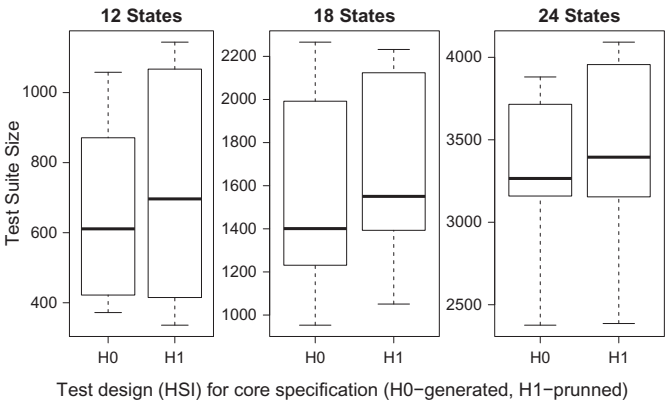


FIGURE 7. Test suite size of the core specification.

Threats: The number of FFSMs in each group. Larger samples may reduce the *p*-value.

5.2.2. *Q2—In which scenario do we reduce the number of test cases using an FFSM instead of FSMs?*

Results: To answer this question, we selected 20 random valid configurations of each FFSM. For every selected configuration, an FSM was derived and a complete test suite was generated using the HSI method. Figure 8 shows the results on the accumulated number of new tests comparing the (FFSM) configurable test suite size with three cases: (FSM.core) complete test suite size for the core product; (10FSMs) merged set of complete test suites from the first 10 selected configurations; (20FSMs) merged set of complete test suites from the all selected configurations. In the first FFSM group, the means are (FFSM) 1422, (FSM.core) 660, (10FSMs) 2199 and (20FSMs) 3782. In the second FFSM group, the means are (FFSM) 3810, (FSM.core) 1757, (10FSMs) 6819 and (20FSMs) 8946. In the third FFSM group, the means are (FFSM) 9191, (FSM.core) 3002, (10FSMs) 13 924 and (20FSMs) 16 588.

Analysis: We noticed that the results from the first FFSM group continue in the other two groups as it increases the FFSM size. There is a significant difference comparing the first and third cases as we need to test more FSMs. An FFSM is linked to a feature model ranging from 21 to 2^{20} valid configurations depending on the feature structure. Selecting up to the minimum number of valid configurations shows us how powerful is to exploit commonalities between products compared to the product-by-product approach. For example, an FFSM of the first group with 12 conditional states may represent 2^{10} valid configurations and may have a configurable test suite of size 1422 (the mean), and to test 20 out of 2^{10} configurations individually we require an average set of tests with size 3782. When we create a test case without checking test suites of similar products we may end up generating a different but equivalent test case. In our approach, one test case may be reused across all valid products minimizing redundant test cases. In all three groups, the difference is above 50%. Thus, our experiment indicates a statistically meaningful difference of approximately 50% in the number of new test cases when we have more than 20 random valid configurations to test individually compared to an entire SPL using a configurable test suite generated from an FFSM.

Threats: Very similar specifications may not accumulate so many different test cases with so few products.

5.2.3. *Q3—In which scenario do we have smaller test generation times using an FFSM instead of FSMs?*

Results: To answer this question, we selected 100 random valid configurations of each FFSM. For every selected configuration, an FSM was derived and a complete test suite was generated using the HSI method. Figure 9 shows the results on the time required to generate complete test suites comparing the (FFSM) time to generate a configurable test suite with three cases: (FSM.core) time to generate a complete test suite for the core product; (50FSMs) time to generate complete test suites from the first 50 selected configurations; (100FSMs) time to generate complete test suites from the all selected configurations. In the first FFSM group, the means are (in minutes): (FFSM) 0.66, (FSM.core) 0.00017, (50FSMs) 0.008 and (100FSMs) 0.017. In the second FFSM group, the means are (in minutes): (FFSM) 1.92, (FSM.core) 0.012, (50FSMs) 0.63 and (100FSMs) 1.27. In the third FFSM group, the means are (in minutes): (FFSM) 4.99, (FSM.core) 0.11, (50FSMs) 5.96 and (100FSMs) 11.92.

Analysis: We noticed that the results vary as it increases the FFSM size. There is a significant difference comparing the first and third cases in the last FFSM group. Assume that we have to generate a configurable complete test suite for an FFSM which may have 2^{10} valid configurations or individually 100 valid configurations. If our FFSM have more than 24 conditional states and 450 conditional transitions and we need to test more than 100 valid configurations, we may consider a better generation time using our approach.

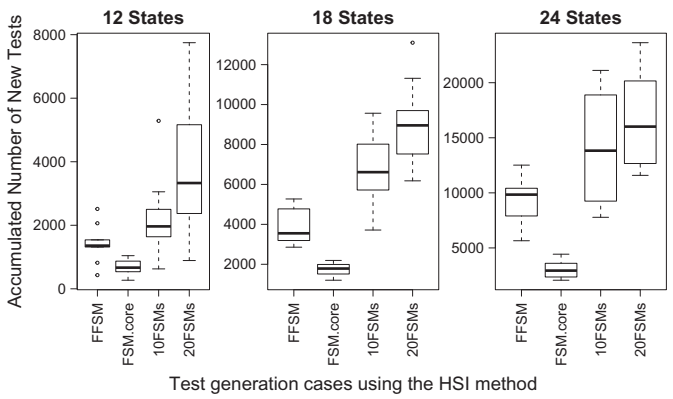


FIGURE 8. Number of new tests for an FFSM and FSMs.

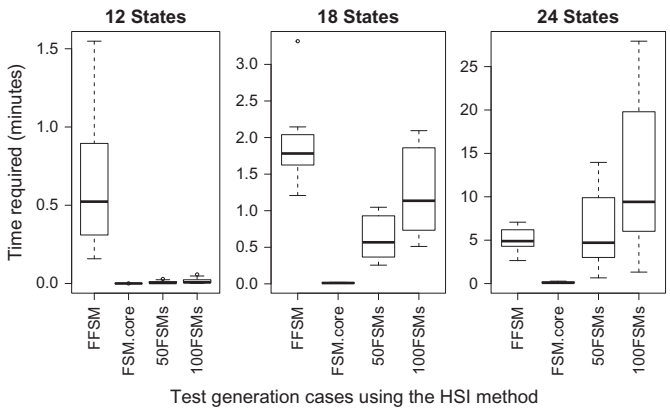


FIGURE 9. Time required to execute the HSI method for one FFSM and some FSMs.

Threats: The number of required paths for test generation due to complex feature models may be different in specific FFSMs.

5.2.4. *Q4—Is there a relation between the feature model and the configurable test suite size?*

Results: To answer this question, we divided each FFSM group in two subgroups of 10 FFSMs. The median of the valid feature configurations was used to separate the FFSMs into two types of feature models. In the first group of FFSMs with 12 conditional states, the median of product configurations found from all feature models was 6276, then 12 808 for the second group, and 25 608 for the third group. Figure 10 shows the results for the relation between the configurable test suite size and types of features models.

Analysis: We noticed that the size of complete configurable test suites is larger in those FFSMs which have feature models with more than the median of valid configurations in each group. As we stated before, each random FFSM is generated by randomizing the target state of conditional transitions. We assume that only one-third of the conditional states and transitions have random feature constraints. Thus, two-third of the behavior is part of the core specification. Also, only one feature is used on each conditional state/transition. The first FFSM group (12 states) has smaller specifications than other FFSM groups, and we first believe that less features are used (out of 20) resulting in a smaller feature configuration median. To check that belief, we also checked the number of conditional transitions of each subgroup. Figure 11 shows the results for the relation between the number of transitions of FFSMs and types of features models.

We noticed that FFSMs linked to feature models with more configurations than the median has fewer conditional transitions. When an FFSM has fewer conditional transitions the extended HSI algorithm has fewer options to find common conditional paths, resulting in more different conditional tests.

We also checked the relation between the feature model and the extended HSI test generation time. Figure 12 shows the results for the relation between the time required to generate a configurable test suite size and types of features models.

We noticed that the time required to generate configurable test suites does not have a pattern from the three FFSM groups. Despite the larger configurable test suites for those FFSMs with more than the median of valid configurations presented in Fig. 10, the time required to generate them do not follow. Also, it does not have a direct relation to the FFSM size as presented in Fig. 11. We believe that this is caused by the irregular number of required conditional paths and separating sequences. Thus, our experiment indicates no influence of the feature model for the time required to generate configurable test suites.

Threats: Our experiment indicates a tendency to have larger configurable test suites for FFSMs when the feature model has many valid configurations but possibly affected by the number of fewer transitions. This indication may be a threat to validity. The irregular distribution of valid configurations may be a coincidence or a consequence of the FFSM random generator.

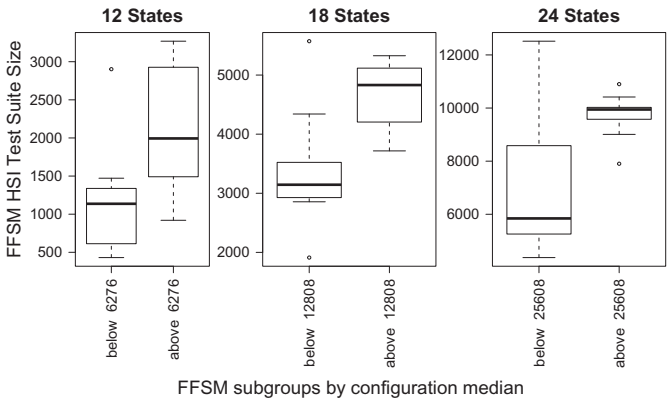


FIGURE 10. Configurable test suite size per kind of feature model.

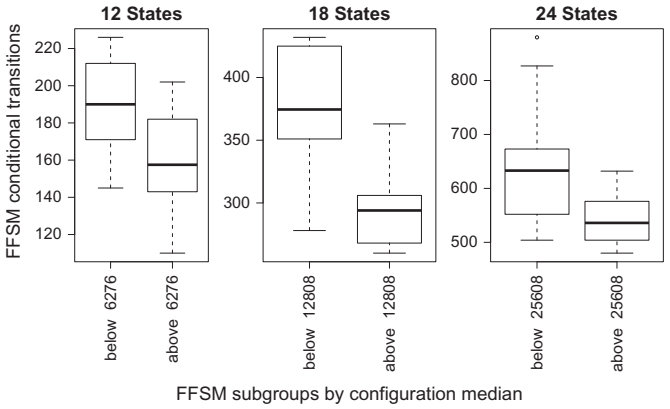


FIGURE 11. Number of FFSM conditional transitions per kind of feature model.

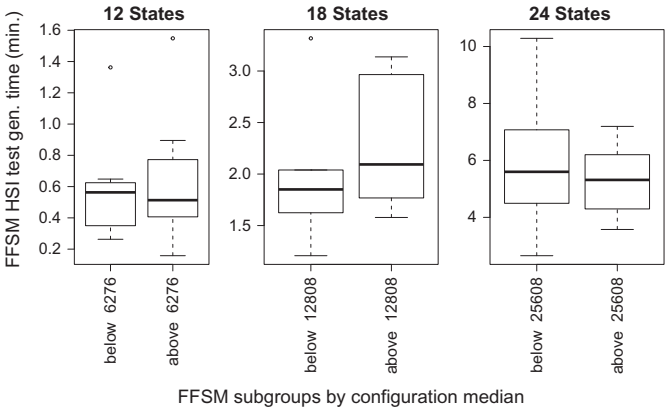


FIGURE 12. Time required to execute the extended HSI method per kind of feature model.

6. BODY COMFORT SYSTEM CASE STUDY

We illustrate and evaluate our approach in a prototypical implementation using a case study from the automotive domain, namely, Body Comfort System (BCS) for the VW Golf SPL [18]. We use the BCS to reduce the threats to validity and contrast the results from randomly generated FFSMs with their real-world counterparts. The FeatureIDE tool [25] was used to elaborate feature models and their configurations. The original BCS system has 19 non-mandatory features and 11 616 configurations.

Our first observation was that flattening the whole system into an FFSM was infeasible due to a large number of conditional states (in theory more than 50 000 states). Thus, we used a simplified version (slice) which integrates some components, and we performed a manual flattening of the slice. We selected a part of the feature model with four non-mandatory features and six possible configurations for four components: *Finger Protection* (*FP*) blocking the window movement when a finger is clamped in a window, *Manual Power Window* (*ManPW*) or alternatively *Automatic Power Window* (*AutPW*) and *Central Locking System* (*CLS*) with optional *Automatic Locking* (*AL*) when the car is driving. Figure 13 shows the selection of the Automatic Power Window component while leaving the Central Locking System and Automatic Locking features open which means that this model still accommodates three out of six possible configurations.

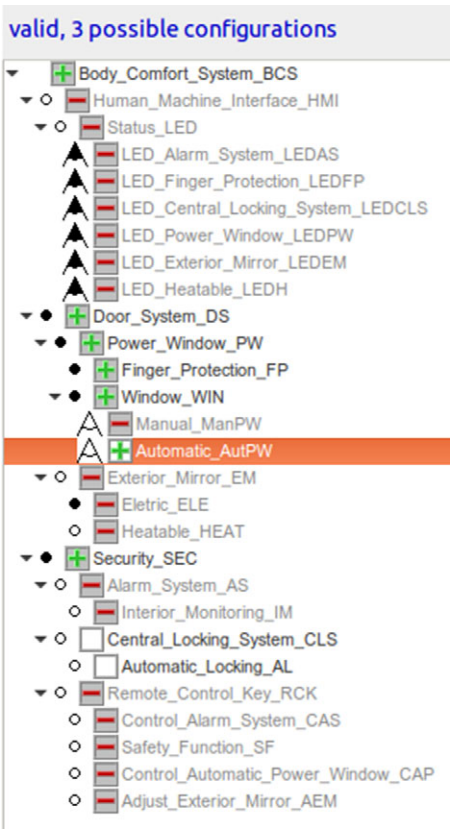


FIGURE 13. Feature model configuration selection for BCS.

To help design FFSM models, we used the ConFTGen tool proposed in Section 4. Figure 14 presents the flat composition of four selected components of BCS and abstract inputs and outputs described below. The original 150% behavioral model of each component can be found in [18]. States 1, 2 and 3 represent the behavior of Manual and Automatic Power Window alternative components. The Finger Protection component has two states, and hence, the same behavior is repeated in States 4, 5 and 6. The last component Central Locking System also has two states which lead to the same behavior repeated in States 7–12.

The ConFTGen tool has two types of model derivation. The first type of model derivation (FFSM) ignores the open features and derives an FFSM for a subset of valid configurations using a feature constraint without the open features. The second type of model derivation (FSM) uses a feature constraint with all features of the model and negates the open features to derive an FFSM model for a single configuration that corresponds to an FSM. Figure 15 shows the first FFSM model derivation type for three product configurations. Figure 16 shows the second FFSM model derivation type for a single product configuration which excludes Central Locking System and Automatic Locking features.

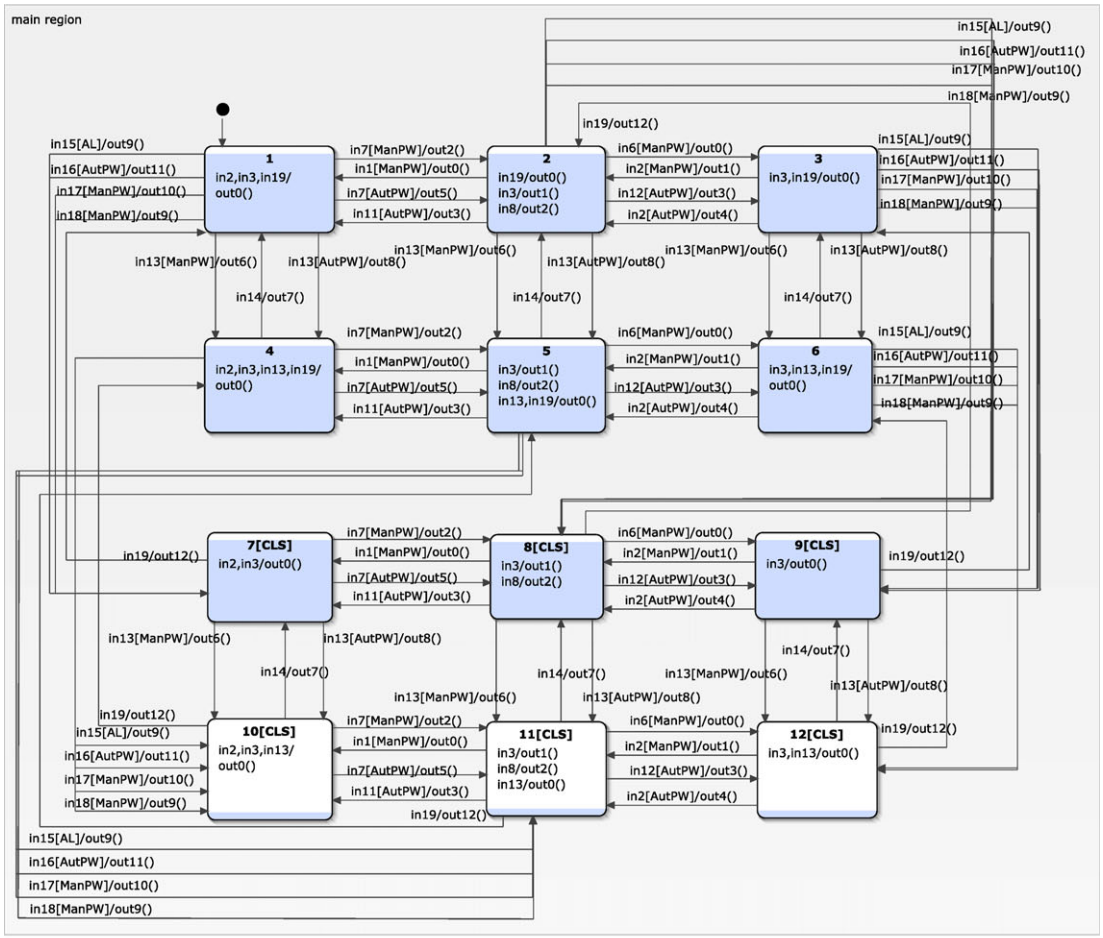
Following our product line-centered approach, the configurable test suite obtained for the state coverage has 14 tests and size 40. The configurable test suite obtained for the transition coverage has 124 tests and size 715. The configurable test suite obtained for the full fault coverage has 433 tests and size 2311. The validation time takes approximately 10 s while the configurable test suite generation 1 min.

To test individually all six product configurations, we derived six FSMs and we generated six test suites using the original HSI method. Then, to calculate the number of new tests required for all six products we unified all six test suites and counted the number of tests that would be concretized and executed. In the end, we found out that our unified test suite has 463 tests and size 3071. Using our approach we had 433 tests and size 2311 which is a reduction of 25%.

Comparing the results with the random FFSMs with 12 conditional states, we found out that the number of new test cases is in accordance with Question 2 and test suite generation is in accordance with Question 3.

7. RELATED WORK

Usually, an SPL can generate several similar products where only a few features vary from one to another. One challenge in SPL testing is the verification of products using a simplified behavioral model that takes advantage of the similarity between products. There are proposals [3, 30] that provide a concise formalism for representing SPL behaviors in one model. However,




```
model
interface:
//states 1=(pw_up, FP_off,cls_unlock) //states 2=(pw_pend, FP_off,cls_unlock) //states 3=(pw_dn, FP_off,cls_unlock) //states 4=(pw_up, FP_on,cls_unlock)
//states 5=(pw_pend, FP_on,cls_unlock) //states 6=(pw_dn, FP_on,cls_unlock) //states 7=(pw_up, FP_off,cls_lock) //states 8=(pw_pend, FP_off,cls_lock)
//states 9=(pw_dn, FP_off,cls_lock) //states 10=(pw_up, FP_on,cls_lock) //states 11=(pw_pend, FP_on,cls_lock) //states 12=(pw_dn, FP_on,cls_lock)
//INPUTS
in event in1  //pw_but_up[pw_pos==1]
in event in2  //pw_but_up[fp_on==false]
in event in3  //pw_but_up[pw_pos==0 && fp_on==false]
in event in4  //pw_but_up[pw_auto_mv==-1]
in event in5  //pw_but_up[pw_auto_mv==0 && fp_on==false]
in event in6  //pw_but_dn[pw_pos==-1]
in event in7  //pw_but_dn[pw_enabled==true]
in event in8  //pw_but_dn[pw_pos==0 && pw_enabled==true]
in event in9  //pw_but_dn[pw_auto_mv==0 && pw_enabled==true]
in event in10 //pw_but_dn[pw_auto_mv==1 && pw_enabled==true]
in event in11 //[pw_pos==1 && pw_auto_mv==1]
in event in12 //[pw_pos==-1 && pw_auto_mv==-1]
in event in13 //fp_finger_detected
in event in14 //pw_but_dn
in event in15 //car_drives
in event in16 //key_pos_lock
in event in17 //key_pos_lock[pw_pos==1]
in event in18 //key_pos_lock[pw_pos!=1]
in event in19 //key_pos_unlock
//FEATURES
var ManPW : boolean  var AutPW : boolean  var FP : boolean  var CLS : boolean  var AL : boolean
//OUTPUT FUNCTIONS
operation out0():integer  //nothing
operation out1():integer  //GEN(pw_mv_up)
operation out2():integer  //GEN(pw_mv_dn)
operation out3():integer  //pw_auto_mv=0
operation out4():integer  //pw_auto_mv=1
operation out5():integer  //pw_auto_mv=-1
operation out6():integer  //fp_on=true
operation out7():integer  //fp_of=false
operation out8():integer  //pw_auto_mv=0;fp_on=true
operation out9():integer  //cls_locked=true
operation out10():integer  //cls_locked=true;pw_enabled=false;
operation out11():integer  //cls_locked=true;pw_enabled=false;GEN(pw_but_up)
operation out12():integer  //cls_locked=false;pw_enabled=true
```

FIGURE 14. FFSM of four composed components of BCS.

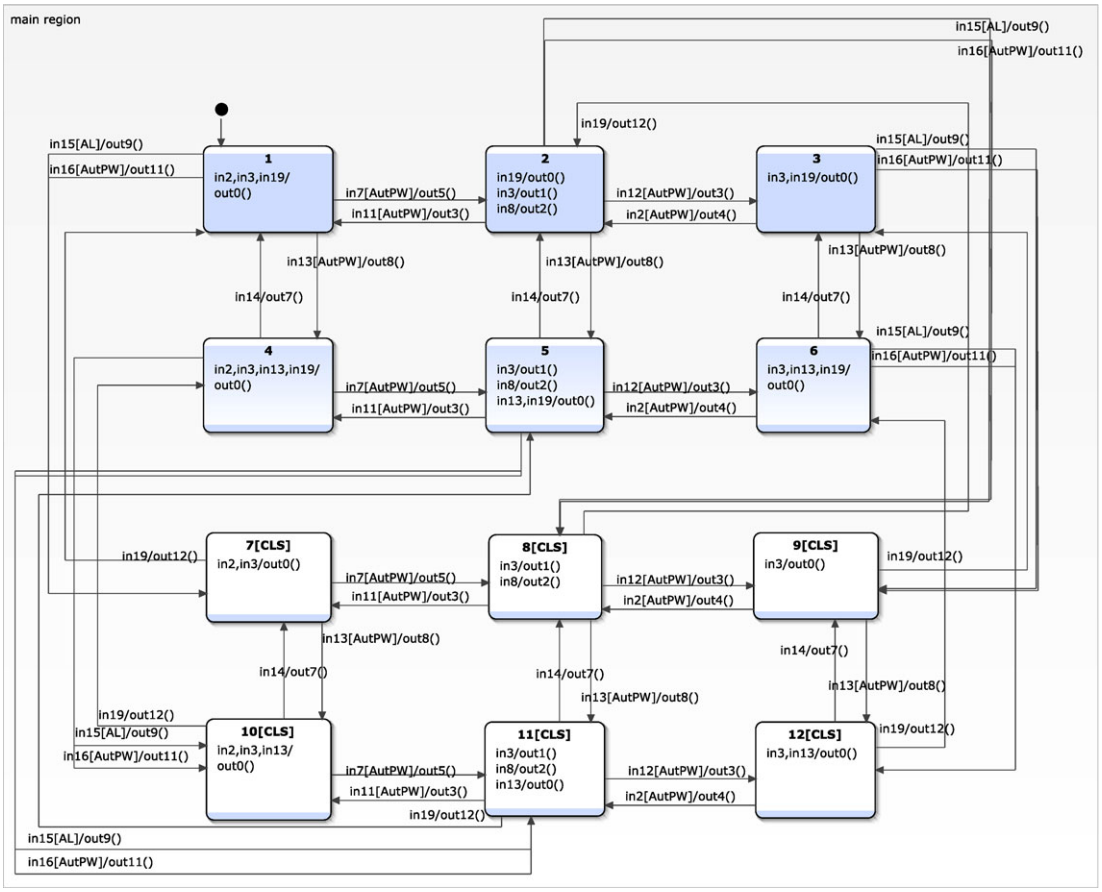


FIGURE 15. FFSM derived for three configurations with three composed components.

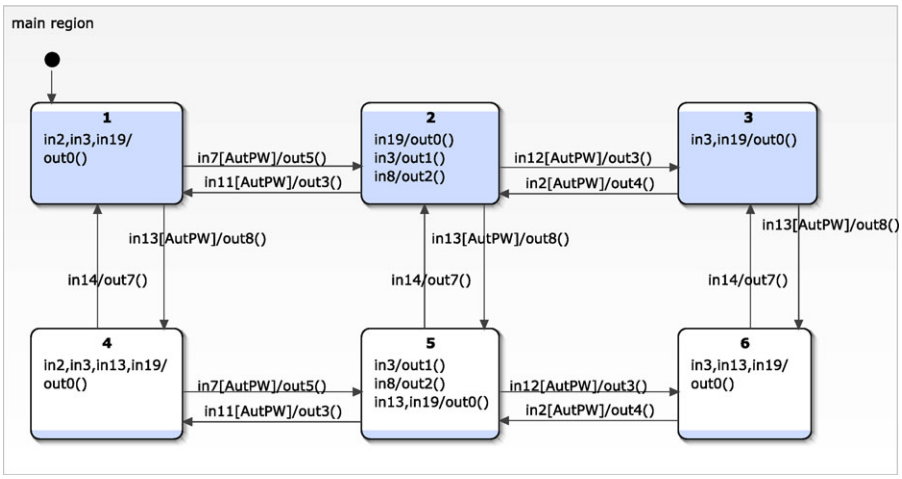


FIGURE 16. FFSM derived for one configuration and two composed components.

they are focused on model checking [30] or simple test criteria like boundary tests [3]. In this paper, we take a step forward by extending the FSM-based formalism for SPLs. The main purpose of this extension is to enable test case generation methods that use family-based FSM models to achieve comprehensive, configurable test suites using the full fault coverage.

Regarding regression-based approaches for SPLs, there are several incremental test approaches [27, 28, 31–33] devoted

to generating, reusing and optimizing test suites for SPLs. El-Fakih *et al.* [27] adapted FSM-based test generation methods for conformance testing. Their approach allows for the generation of test cases only for the modified parts of an evolving specification. Pap *et al.* [28] extended their work and designed a bounded incremental algorithm that maintains two sets based on the HSI method [13]. They utilize existing test cases of the previous version of the system to generate test cases for the modified version. Similarly, Capellari *et al.* [32] explored the FSM-based Testing of SPLs (FSM-TSPL) testing strategy where the P method is used to design new test cases based on the last product derived. Uzuncaova *et al.* [31] also developed an incremental test generation approach that uses SAT-based analysis to develop tests suites for every product of an SPL, while Baller and Lochau [33] focused on test suite optimization. Moreover, recent delta-oriented approaches [7, 34, 35] developed regression-based SPL approaches to design and reuse test artifacts.

Regarding configurable test artifacts, most modeling concepts for variability can be distinguished into three main approaches: annotative, compositional and transformational variability modeling [36]. Compositional approaches for modeling variability capture variation by selecting specific component variants. Compositional variability modeling [37] allows for a modular description of variability but limits the impact of changes to the applied composition technique. Transformational approaches represent variability by transformation of a base architectural model. Model transformation rules guide the derivation of products by performing additions, modifications or removals using variability. For example, delta modeling [38] can represent variability in the model transformation in which a core system is developed, and subsequent products are derived by executing such transformations rules. Annotative approaches use variant annotations (also called 150%-models), e.g. UML stereotypes in UML models to define which model elements belong to specific product variants. In the orthogonal variability model (OVM) [39], a separate variability representation with links to the architecture model replaces direct annotations. Some approaches [26, 40] propose a pruning-based approach to UML 150% test model for SPLs, separating variability from the base models using mapping models.

Using an annotative 150% statechart test model and transition coverage criterion, Cichos *et al.*'s approach [3] presents SPL test design for complete test model coverage with subsequent product subset selection for test suite execution. Weissleder *et al.* [5] propose an approach for automatic test suite derivation based on reusable UML state machine test models and OCL expressions. Similarly, Liu *et al.* [41] use statecharts to model reusable components and also through pruning, derive instances syntactically, and may be combined with Wasowski's approach [42] to flattening statecharts without the state explosion problem. In Devroey *et al.*'s approach [43], they use mutation testing applied to annotate Featured Transition Systems (FTS) [30]. Moreover, Luthmann *et al.*'s approach [44] uses the Featured Time Automata formalism (a variation of FTS) to check real-time properties of SPLs.

Model-based testing can be used in SPL testing. We refer the reader to Oster *et al.* [8] for a summary of model-based SPL testing approaches, to [36, 45–47] for recent surveys, and Thum *et al.*'s recent survey [48] for a classification of different SPL analysis techniques. Some behavioral models proposed in the literature, e.g. those in [41, 49, 50] are based on Finite State Machines or Labeled Transition Systems. They are mainly used to provide a formal specification for SPLs and enable their formal verification using model checking.

Our proposed approach for configurable test artifacts can be classified as a family-based and feature-oriented specification. To our knowledge, however, there only a few pieces of research that extend test models, test case generation and test case execution to the family-based level; examples of such work include earlier delta-oriented techniques such as [34, 35, 51] and feature-oriented approaches [52, 53] using FTS-based formalisms. However, the approach proposed in [52, 54] exploits a non-deterministic test case generation algorithm (with no fault model or finite test suite) and hence, semantic validation of test models is not an issue in their approach. We are not aware of any prior study on extending the FSM-based test-model validation and test case generation techniques to the family-based setting that is based on the notion of full fault coverage.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an extension of the HSI test case generation method for FFSMs. An FFSM test model represents the abstract behavior of SPL components and its compositions. The HSI test generation method was originally designed to generate tests using FSMs for the full fault coverage criterion. However, FSMs used as inputs to HSI require semantic properties such as determinism, initially connected and minimal. In our previous work [16], we presented an extension of the FSM for SPLs named FFSMs, where such semantic properties were extended to FFSMs, and we showed that they coincide with their corresponding properties for the product FSM models. In this paper, the HSI method was extended from FSMs to FFSMs.

We conducted an experimental study comparing the number of new tests required to test SPL products using the extended HSI method with a random set of products using individual test suites. Random FFSMs and feature models were generated, and our implemented method was applied on them. The results indicate a significant decrease in the number of new tests when compared to the traditional product-by-product approach. The experiments showed that in general, we have more new tests from only 20 products than the whole SPL by using our approach with an FFSM. Also, the

case study shows that only with six products we still have more new tests using the product-by-product approach. Moreover, we checked the relation between FFSMs and feature models in respect to configurable test suite size and test case generation time. We observed no strong influence on different kinds of feature models for test case generation using FFSMs.

A prototyping tool named ConFTGen was implemented to guide the design of FFSMs. The tool also performs validation, derivation and test case generation for the state, transition and full fault coverage. A case study for the Body Comfort System was used to present the tool and show some issues related to the current FFSM specification.

In a parallel line of work, we plan to extend the FFSM model to Hierarchical FFSMs (using concepts from Statecharts and UML State Machines) to handle the state explosion problem identified in the case study. We then apply validation (and test case generation) on hierarchical models. Another possible line of work is to improve test case generation using new concepts from regression-based incremental methods.

FUNDING

This research was partially supported by the Science Without Borders project number 201694/2015-8, the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation project number 20140312.

REFERENCES

[1] Pohl, K., Böckle, G. and van der Linden, F. (2005) Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, Berlin.

[2] Tevanlinna, A., Taina, J. and Kauppinen, R. (2004) Product family testing. *ACM SIGSOFT Softw. Eng. Notes*, **29**, 1–12.

[3] Cichos, H., Oster, S., Lochau, M. and Schürr, A. (2011) Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. *14th Int. Conf. Model Driven Engineering Languages and Systems (MODELS)*, Wellington, New Zealand, October 16–21, pp. 425–439. Springer-Verlag, Berlin.

[4] Uzuncaova, E., Garcia, D., Khurshid, S. and Batory, D. (2008) Testing Software Product Lines Using Incremental Test Generation. *19th Int. Symp. Software Reliability Engineering (ISSRE)*, Seattle, WA, USA, November 10–14, pp. 249–258. IEEE.

[5] Weißleder, S., Sokenou, D. and Schlingloff, B.-H. (2008) Reusing State Machines for Automatic Test Generation in Product Lines. *Model-Based Testing in Practice (MoTiP)*, Berlin, Germany, pp. 1–10. Fraunhofer IRB Verlag.

[6] Oster, S., Markert, F. and Ritter, P. (2010) Automated Incremental Pairwise Testing of Software Product Lines. *Int. Conf. Software Product Lines (SPLC)*, Jeju Island, South Korea, September 13–17, pp. 196–210. Springer-Verlag, Berlin.

[7] Lochau, M., Schaefer, I., Kamischke, J. and Lity, S. (2012) Incremental Model-Based Testing of Delta-Oriented Software Product Lines. *Int. Conf. Tests and Proofs (TAP)*, Prague, Czech Republic, May 31–June 1, pp. 67–82. Springer-Verlag, Berlin.

[8] Oster, S., Wubbeke, A., Engels, G. and Schurr, A. (2012) A Survey of Model-Based Software Product Lines Testing. In Zander, J., Schieferdecker, I. and Mosterman, P.J. (eds.) *Model-Based Testing for Embedded Systems*, pp. 338–381. CRC Press. Chapter 13.

[9] Runeson, P. and Engstrom, E. (2012) Software Product Line Testing: A 3D Regression Testing Problem. *5th Int. Conf. Software Testing, Verification and Validation (ICST)*, Montreal, QC, Canada, April 17–21, pp. 742–746. IEEE.

[10] Engstrom, E. and Runeson, P. (2013) Test overlay in an emerging software product line—an industrial case study. *Inf. Softw. Technol.*, **55**, 581–594.

[11] 610.12-1990, I. S. (1990) *IEEE Standard Glossary of Software Engineering Terminology*. IEEE.

[12] Chow, T. (1978) Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, **SE-4**, 178–187.

[13] Luo, G., Petrenko, A., Petrenko, R. and Bochmann, G.V. (1994) Selecting Test Sequences For Partially-Specified Nondeterministic Finite State Machines. *Int. Feder. Information Processing (IFIP)*, Boston, MA, pp. 91–106. Springer US.

[14] Simao, A. and Petrenko, A. (2010) Fault coverage-driven incremental test generation. *Comput. J.*, **53**, 1508–1522.

[15] Endo, A.T. and Simao, A. (2013) Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. *Inf. Softw. Technol.*, **55**, 1045–1062.

[16] Fragal, V.H., Simao, A. and Mousavi, M.R. (2016) Validated Test Models for Software Product Lines: Featured Finite State Machines. *13th Int. Conf. Formal Aspects of Component Software (FACS)*, Besançon, France, October 19–21, pp. 210–227. Springer International Publishing.

[17] de Moura, L. and Bjørner, N. (2008) Z3: An Efficient SMT Solver. *14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, March 29–April 6, pp. 337–340. Springer-Verlag, Berlin.

[18] 2012-07 (2012) *Delta-Oriented Software Product Line Test Models—The Body Comfort System Case Study*. TU Braunschweig.

[19] Broy, M., Jonsson, B., Katoen, J., Leucker, M. and Pretschner, A. (2005) *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer-Verlag, Berlin.

[20] CMU/SEI-90-TR-021 (1990) *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Carnegie-Mellon University Software Engineering Institute.

[21] Schobbens, P.Y., Heymans, P. and Trigaux, J.C. (2006) Feature Diagrams: A Survey and a Formal Semantics. *14th Int. Conf. Requirements Engineering (RE)*, Minneapolis, MN, USA, September 11–15, pp. 139–148. IEEE.

[22] Kang, S., Lee, J., Kim, M. and Lee, W. (2007) Towards a Formal Framework for Product Line Test Development. *7th IEEE Int. Conf. Computer and Information Technology (CIT)*, Aizu-Wakamatsu, Fukushima, Japan, October 16–19, pp. 921–926. IEEE.

[23] Linden, F., Schmif, K. and Rommes, E. (2007) *Software Product Lines in Action*. Springer-Verlag, New York.

[24] Batory, D. (2005) Feature Models, Grammars, and Propositional Formulas. *9th Int. Conf. Software Product Lines (SPLC)*, Rennes, France, September 26–29, pp. 7–20. Springer-Verlag, Berlin.

[25] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G. and Leich, T. (2014) FeatureIDE: an extensible framework for feature-oriented software development. *Sci. Comput. Prog.*, **79**, 70–85.

[26] Czarnecki, K. and Antkiewicz, M. (2005) Mapping Features to Models: A Template Approach Based on Superimposed Variants. *4th Int. Conf. Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 29–October 1, pp. 422–437. Springer-Verlag, Berlin.

[27] EI-Fakih, K., Yevtushenko, N. and Bochmann, G. (2004) FSM-based incremental conformance testing methods. *IEEE Trans. Softw. Eng.*, **30**, 425–436.

[28] Pap, Z., Subramaniam, M., Kovács, G. and Németh, G.Á. (2007) A Bounded Incremental Test Generation Algorithm for Finite State Machines. *Testing of Software and Communicating Systems*, Tallinn, Estonia, June 26–29, pp. 244–259. Springer-Verlag, Berlin.

[29] Segura, S., Galindo, J.A., Benavides, D., Parejo, J.A. and Ruiz-Cortés, A. (2012) BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. *6th Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Leipzig, Germany, January 25–27, pp. 63–71. ACM, New York.

[30] Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A. and Raskin, J.-F. (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking . *IEEE Trans. Softw. Eng.*, **39**, 1069–1089.

[31] Uzuncaova, E., Khurshid, S. and Batory, D. (2010) Incremental test generation for software product lines. *IEEE Trans. Softw. Eng.*, **36**, 309–322.

[32] Capellari, M.L., Gimenes, I.M.S., Simao, A. and Endo, A.T. (2012) Towards Incremental FSM-based Testing of Software Product Lines. *11th Brazilian Symp. Software Quality (SBQS)*, Fortaleza, Brazil, 11–15 June, pp. 9–23. SBC, RS.

[33] Baller, H. and Lochau, M. (2014) Towards Incremental Test Suite Optimization for Software Product Lines. *6th Int. Workshop on Feature-Oriented Software Development (FOSD)*, Vasteras, Sweden, September 14–14, pp. 30–36. ACM, New York.

[34] Lochau, M., Lity, S., Lachmann, R., Schaefer, I. and Goltz, U. (2014) Delta-oriented model-based integration testing of large-scale systems. *J. Syst. Softw.*, **91**, 63–84.

[35] Varshosaz, M., Beohar, H. and Mousavi, M.R. (2015) Delta-Oriented FSM-Based Testing. *Int. Conf. Formal Engineering Methods (ICFEM)*, Paris, France, November 3–5, pp. 366–381. Springer International Publishing.

[36] Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S. and Villela, K. (2012) Software diversity: state of the art and perspectives. *Int. J. Softw. Tools Technol. Transf.*, **14**, 477–495.

[37] Haber, A., Rendel, H., Rumpe, B., Schaefer, I. and van der Linden, F. (2011) Hierarchical Variability Modeling for Software Architectures. *15th Int. Software Product Line Conf. (SPLC)*, Munich, Germany, August 22–26, pp. 150–159. IEEE.

[38] Clarke, D., Helvensteijn, M. and Schaefer, I. (2011) Abstract Delta Modeling. *9th Int. Conf. Generative Programming and Component Engineering (GPCE)*, Eindhoven, The Netherlands, October 10–13, pp. 13–22. ACM, New York.

[39] Pohl, K. and Metzger, A. (2006) Software product line testing. *Commun. ACM*, **49**, 78–81.

[40] Grönninger, H., Krahn, H., Pinkernell, C. and Rumpe, B. (2008) Modeling Variants of Automotive Systems using Views. *Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Berlin, Germany, pp. 1–14. TU Braunschweig.

[41] Liu, J., Dehlinger, J. and Lutz, R. (2007) Safety analysis of software product lines using state-based modeling. *J. Syst. Softw.*, **80**, 1879–1892.

[42] Wasowski, A. (2004) Flattening Statecharts Without Explosions. *ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Washington, DC, USA, June 11–13, pp. 257–266. ACM, New York.

[43] Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.-Y. and Heymans, P. (2016) Featured model-based mutation analysis. *38th Int. Conf. Software Engineering (ICSE)*, Austin, Texas, May 14–22, pp. 655–666. ACM, New York.

[44] Luthmann, L., Stephan, A., Bürdek, J. and Lochau, M. (2017) Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints. *21st Int. Systems and Software Product Line Conf. (SPLC)*, Sevilla, Spain, September 25–29, pp. 104–113. ACM, New York.

[45] Beohar, H., Varshosaz, M. and Mousavi, M.R. (2016) Basic behavioral models for software product lines: expressiveness and testing pre-orders. *Sci. Comput. Prog.*, **123**, 42–60.

[46] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K. and Wasowski, A. (2012) Cool Features and Tough Decisions. *6th Int. Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, Leipzig, Germany, January 25–27, pp. 173–182. ACM, New York.

[47] Benduhn, F., Thüm, T., Lochau, M., Leich, T. and Saake, G. (2015) A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. *9th Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Hildesheim, Germany, January 21–23, pp. 80–87. ACM, New York.

[48] Thüm, T., Apel, S., Kästner, C., Schaefer, I. and Saake, G. (2014) A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv. (CSUR)*, **47**, 1–45.

[49] Asirelli, P., ter Beek, M.H., Gnesi, S. and Fantechi, A. (2011) Formal Description of Variability in Product Families. *15th Int. Software Product Line Conf. (SPLC)*, Munich, Germany, August 22–26, pp. 130–139. IEEE.

[50] Classen, A., Heymans, P., Schobbens, P.-Y. and Legay, A. (2011) Symbolic Model Checking of Software Product Lines. *33rd Int. Conf. Software Engineering (ICSE)*, Honolulu, HI, USA, May 21–28, pp. 321–330. IEEE.

[51] Lochau, M. and Kamischke, J. (2012) Parameterized Preorder Relations for Model-Based Testing of Software Product Lines. *5th Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLA)*, Heraklion, Crete, Greece, October 15–18, pp. 223–237. Springer-Verlag, Berlin.

[52] Beohar, H. and Mousavi, M.R. (2014) Input–Output Conformance Testing Based on Featured Transition Systems. *29th Annu. ACM Symp. Applied Computing (SAC)*, Gyeongju, Republic of Korea, March 24–28, pp. 1272–1278. ACM, New York.

[53] Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P.-Y. and Heymans, P. (2014) Coverage Criteria for Behavioural Testing of Software Product Lines. *6th Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Imperial, Corfu, Greece, October 8–11, pp. 336–350. Springer-Verlag, Berlin.

[54] Beohar, H. and Mousavi, M. (2014) Spinal Test Suites for Software Product Lines. *9th Workshop on Model-Based Testing (MBT)*, Grenoble, France, April 6, pp. 44–55. ACM, New York.