Reducing Concretization Effort in FSM-Based Testing of Software Product Lines

Vanderson Hafemann Fragal[1], Adenilso Simão[1], André Takeshi Endo[2], and Mohammad Reza Mousavi[3]

[1] Institute of Math. and Computer Sciences (ICMC), University of São Paulo, Brazil
[2] Federal Technological University of Paraná (UTFPR-CP), Brazil
[3] Centre for Research on Embedded Systems (CERES), Halmstad University, Sweden

Abstract. In order to test a Software Product Line (SPL), the test artifacts and the test techniques have to be extended to support variability. When new SPL products are developed, new tests are generated to cover new or modified features. According to several case studies, a dominant source of extra effort for such tests is concretization of newly generated test cases. Thus, minimizing the amount of new non-concretized tests required to perform conformance testing on new products reduces the overall test effort significantly. In this paper, we propose a new test reuse strategy for conformance testing of SPL products that aims at reducing test effort. To this end, we use incremental test generation methods based on finite state machine models to maximize test reuse. We combine these methods with our selection algorithm used to select non-redundant concretized tests. We demonstrate our strategy using examples and a case study with an embedded mobile SPL. The results indicate that our strategy can save at least 36% test effort for our case study with typical concretization effort estimated from the literature and compared to current test reuse strategies for the same fault detection capability.

Keywords: Conformance Testing, Test Case Reuse, Model-Based Testing, Finite State Machine.

1 Introduction

Software Product Lines (SPL) address variability in software in terms of its features. For example, developing SPL products using the delta-oriented approach[13] involves designing a core module and a set of delta modules. The core module is a set of features of a basic product and the delta modules add, remove, or modify features from the core module to design new products. Changing the specification of a product or deploying a similar product may require substantial effort for conformance testing. We study this problem in the context of Model-Based Testing (MBT), where models are used to steer the test process effort with the goal of making it more structured and more efficient.

One important step in the MBT test process is concretization [20]. Generated abstract tests are augmented with concrete implementation-specific data turning them executable in the system under test. Checking the conformance of an SPL

product requires extra effort for each new test that needs to be concretized for execution. According to case studies [20, 12] the cost of concretizing a test cases exceeds with two orders of magnitude the cost of executing the same concretized test. To solve this problem adapters are developed to automate the concretization process. The adapters often need to be modified for new products. For example, systems that evolve constantly (e.g. graphical user interface systems) cannot afford updating adapters of each new version of the system, which often take more time than manually testing the system in the first place [7].

In this paper, we propose a new test reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL) that aims at reducing the test effort of newly designed SPL products by reducing the number of new tests that need to be concretized for conformance testing. To this end, we maximize the reuse of tests by processing concretized tests of all old products and incrementing some of them to obtain a small set of tests to concretize. We use finite state machines (FSMs) as test models, which are fundamental semantic models for reactive systems [2].

The contributions of this paper are: (i) the novel test reuse strategy; and (ii) the novel test case selection algorithm. Figure 1 (b) presents an overview of contributions, where our strategy improves the reuse of tests. This is achieved by incrementing existing concretized tests from all old products for the new behavior. On Figure 1 (a) note that on Product 5 only the brown behavior is new compared to other products. Reusing only the last derived product may result in more redundant tests to concretize. Then, we implemented a selection algorithm to obtain non-redundant tests to retest the unchanged parts of the product.
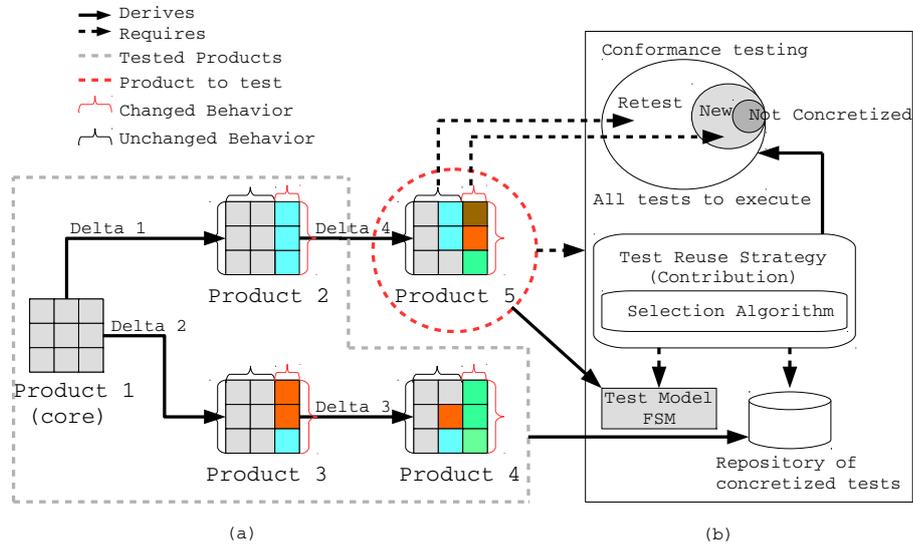

Fig. 1. (a) Derivation of SPL products; and (b) Overview of our contributions.

The effort to test a new product is the sum of the concretization and the execution costs. Reusing all old tests to get a small set of new tests may result in a large accumulated retest set. Thus, selecting non-redundant tests also helps reduce test effort. An experimental evaluation of the proposed strategy was conducted using a case study for the embedded Mobile Media SPL [11]. Initial results show that our strategy can save at least 36% test effort for 24 products with typical concretization effort estimated from the literature.

The remainder of this paper is organized as follows. Section 2 presents some preliminary notions and concepts regarding FSMs, test properties, and effort on SPL testing. Section 3 presents our test reuse strategy, the selection algorithm, and some examples. Section 4 provides results of our experimental study for the embedded Mobile Media SPL. Section 5 discuss the related works, and finally, Section 6 concludes the paper and presents the directions of our future work.

2 Background

In this section, we present basic definitions regarding finite state machines, test properties, and software product line testing.

2.1 Finite State Machine

A finite-state machine (FSM) in our context, is a deterministic Mealy machine, which can be defined as follows.

Definition 1. *An FSM M is a 7-tuple $(S, s_0, I, O, D, \delta, \lambda)$, where S is a finite set of states with the initial state s_0, I is a finite set of inputs, O is a finite set of outputs, $D \subseteq S \times I$ is a specification domain, $\delta : D \to S$ is a transition function, and $\lambda : D \to O$ is an output function.*

If $D = S \times I$, then M is a `complete` FSM; otherwise, it is a `partial` FSM. As M is deterministic, a tuple $(s, x) \in D$ determines uniquely a `defined transition` of M. A transition from state s to s' with input x and output o is represented by quadruple $(s, x, o, s') \in D \to O \times S$, or alternatively by $s \xrightarrow[o]{x} s'$.

A sequence $\alpha = x_1, ..., x_k, \alpha \in I^*$ is a `defined input sequence` at state $s \in S$, if there exist states $s_1, ..., s_{k+1} \in S$, where $s = s_1$ such that $(s_i, x_i) \in D$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \le i \le k$. Notation $\Omega(s)$ is used to denote all defined input sequences for state $s \in S$ and Ω_M denotes $\Omega(s_0)$. We extend the transition and output functions from input symbols to defined input sequences, including the `empty sequence` ε, as usual, assuming $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$ for $s \in S$.

Given sequences $\alpha, \beta, \gamma \in I^*$, a sequence α is `prefix` of a sequence β, denoted by $\alpha \le \beta$, if $\beta = \alpha\gamma$, for some sequence γ, and γ is a `suffix` of β. A sequence α is `proper prefix` of $\beta, \alpha < \beta$, if $\beta = \alpha\omega$ for some $\omega \ne \varepsilon$. We denote by $pref(\beta)$ the set of prefixes of β, i.e. $pref(\beta) = \{\alpha | \alpha \le \beta\}$. For a set of sequences A, $pref(A)$ is the union of $pref(\beta)$ for all $\beta \in A$. If $A = pref(A)$, then we say that A is `prefix-closed`. Moreover, we say that a sequence $\alpha \in A$ is `maximal` in A if there is no sequence $\beta \in A$ such that α is a proper prefix of β. Given a sequence α and $k \ge 0$, we define α^k recursively as follows: $\alpha^0 = \varepsilon$; $\alpha^k = \alpha\alpha^{k-1}$,

if $k > 0$. The `common extensions` of two sequences are the sequences obtained by appending a common sequence to them.

An FSM M is said to be `initially connected`, if for each state $s \in S$, there exists an input sequence $\alpha \in \Omega_M$, such that $\delta(s_0, \alpha) = s$, called a `transfer sequence` for state s. Given a set $C \subseteq \Omega(s) \cap \Omega(s')$, states s and s' are `C-equivalent` if $\lambda(s, \gamma) = \lambda(s', \gamma)$ for all $\gamma \in C$. Otherwise, if there exists a $\gamma \in C$ such that $\lambda(s, \gamma) \neq \lambda(s', \gamma)$, then s and s' are `C-distinguishable`. An FSM M is `minimal` (or `reduced`), if every pair of states $s, s' \in S$ are C-distinguishable. In this paper, only minimal and initially connected machines are considered, since it is a pre-requisite for the P test case generation method[19] used to execute our strategy.

A set $C \subseteq \Omega_M$ is a `state cover` for an FSM M if, for each state $s \in S$, there exists $\alpha \in C$ such that $\delta(s_0, \alpha) = s$. The set $C \subseteq \Omega_M$ `covers` a transition (s, x) if there exists $\alpha \in C$ such that $\delta(s_0, \alpha) = s$ and $\alpha x \in C$. The set C is a `transition cover` (for M) if it covers every defined transition of M. A set of sequences is `initialized` if it contains the empty sequence.

2.2 Test Properties

In this paper, we use the full fault coverage criteria for FSMs from the P method[19]. We use the notion of test suite completeness with respect to a given fault domain and sufficiency conditions based on convergence and divergence properties introduced in [19].

Throughout this paper, we assume that $M = (S, s_0, I, O, D, \delta, \lambda)$ and $N = (Q, q_0, I, O', D', \Delta, \Lambda)$ are a specification FSM and an implementation FSM, respectively. Moreover, n is the number of states of M. We denote by $\Im$ the set of all deterministic FSMs with the same input alphabet as M for which all sequences in Ω_M are defined, i.e. for each $N \in \Im$ it holds that $\Omega_M \subseteq \Omega_N$. The set $\Im$ is called a `fault domain` for M and $\Im_n$ is the set of FSMs with n states. Faults can be detected by tests, which are input sequences defined in the specification FSM M.

Definition 2 ([19]). *A defined input sequence of FSM M is called a `test case` (or simply a `test`) of M. A `test suite` of M is a finite prefix-closed set of tests of M.*

The size of a test $\alpha \in I^*$ denoted by $|\alpha|$ is calculated by the number of inputs that it contains, i.e., $|\alpha| = k, \alpha = (x_1 ... x_k)$. Similarly, $|T|$ is the size of a test suite T calculated by the sum of all tests plus the reset operation for each maximal test, i.e., $|T| = \sum(|\alpha| + 1), \alpha \in T, \nexists_{\beta \in T} \bullet \beta = \alpha\gamma \wedge \gamma \neq \varepsilon$.

The distinguishability of FSMs is defined as the corresponding relation of their initial states, thus, tests are assumed to be applied in the initial state. Given a test suite T, FSMs are `T-equivalent` if their initial states are T-equivalent. Similarly, FSMs are `T-distinguishable` if their initial states are T-distinguishable.

Given two tests $\alpha, \beta \in \Omega_M$ they `converge` if when applied to the initial state they take the FSM into the same state, and they `diverge` if they take the FSM from the initial state to different states. Given a non-empty set of FSMs

$\Sigma \subseteq \Im$ and two tests $\alpha, \beta \in \Omega_M$, we say that α and β are Σ-convergent if they converge in each FSM of the set Σ. Similarly, we say that α and β are Σ-divergent if they diverge in each FSM of Σ.

Two tests α and β in a given test suite T are T-separated if there exist common extensions $\alpha\gamma, \beta\gamma \in T$, such that $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$. T-separated tests are divergent in all FSMs that are T-equivalent to M. Given a test suite T, let $\Im(T)$ be the set of all $N \in \Im$, such that N and M are T-equivalent.

Lemma 1 ([19]). *Given a test suite T of an FSM M, T-separated tests are $\Im(T)$-divergent.*

We refer to [19] for detailed proofs of the results presented in this section.

Divergence of two tests can be identified by different outputs produced by the tests, while the convergence of two tests cannot be directly ascertained. However, it can be shown that if the maximal number of states of FSMs in the fault domain is known, and the two tests are $\Im(T)$-divergent with tests reaching all but one state of the FSM M, these two tests must also converge in the same state in any FSM in the fault domain that is T-equivalent to M. Given a test suite T, let $\Im_n(T) = \Im_n \cap \Im(T)$, i.e. the set of FSMs in which are T-equivalent to M and have at most n states.

Lemma 2 ([19]). *Given a test suite T and $\alpha \in T$, let K be an $\Im_n(T)$-divergent set with n tests and $\beta \in K$ be a test M-convergent with α. If α is $\Im_n(T)$-divergent with each test in $K \setminus \{\beta\}$, then α and β are $\Im_n(T)$-convergent.*

The condition for n-completeness of a test suite T uses the notion of convergence-preserving set, for which the M-convergence implies the $\Im_n(T)$-convergence.

Definition 3 ([19]). *Given a test suite T of an FSM M, a set of tests is $\Im_n(T)$-convergence-preserving (or, simply, convergence-preserving) if all its M-convergent tests are $\Im_n(T)$-convergent.*

Any M-divergent set is by definition convergence-preserving, and $M \in \Im_n$, then $\Im_n(T)$ is by definition not empty.

The following theorem summarize the main results from [19] where the full fault coverage criteria is established based on convergence and divergence properties.

Theorem 1 ([19]). *Let T be a test suite for an FSM M with n states. We have that T is an n-complete test suite for M if T contains an $\Im_n(T)$-convergence-preserving initialized transition cover set for M.*

When a test suite T is n-complete for an FSM M (attend the full fault coverage criteria), then, by executing T we are capable of detecting any fault in all FSM implementations $N \in \Im_n(T)$.

There exist several methods to generate n-complete test suites [6, 15, 19]. For example, the P method [19] uses two input parameters: a deterministic, initially

connected, and minimal FSM M; and an initial test suite T. The initial set T can be empty, and new tests are added/incremented (if necessary) until an n-complete test suite for M is produced. Therefore, the P method checks if all implementations $N \in \Im_n$ can be distinguished from M using T, and decides if more sequences need to be added to T. Experimental evaluation indicates that the P method often results in smaller n-complete test suites compared with other methods [9].

The reuse of test cases is important to save test effort in several domains that develop similar systems. In this paper, we demonstrate how this can be exploited in the testing of software product lines.

2.3 Concretization Effort

Our approach to exploiting old test cases in conformance testing for new SPL products is inspired by regression testing. Given a new product, we first check the changed behavior to ensure that it conforms to the intended behavior by concretizing and executing a set of new test cases. Then, to ensure that the unchanged behavior was not affected by modifications we execute a set of tests to retest the old behavior.

Without a proper reuse strategy, new tests may be generated not considering old products and increasing the number of new tests to concretize, or even concretize redundant tests. For example, Figure 1 (a) shows the evolution of products and their components. Note that the colored boxes denote the new features added to products.

We calculate the effort to test a new product by the sum of tests that have to be executed plus those that have to be concretized times a value x (concretization value over execution), i.e., *effort*= (*concrete* $* x$) + *execution*. Execution cost is calculated based on the number of tests that have to be executed for both changed and unchanged behavior. For example, to execute a prefix-closed test suite T, the execution cost is equivalent to its size $|T|$, i.e., *execution* $= |T|$. Given a set of new tests $NT \subseteq T$, the concretization cost is calculated for each and every new tests $\alpha \in NT$ that have to be concretized. Namely, if a prefix $\beta \in T \backslash \{NT\}$ of a new test $\alpha = \beta\gamma, \gamma \neq \varepsilon$ was already concretized before, then we can reuse β and the cost is the sum of the size of all suffixes γ, i.e., *concrete* $= \sum |\gamma|$.

In the next section, we present our test reuse strategy to obtain a small number of new tests cases for concretization and to remove redundant tests of an n-complete test suite.

3 Testing Products Incrementally

In this section, we present our test reuse strategy, the selection algorithm, and some examples.

3.1 Test Reuse Strategy

The Incremental Regression-based Testing for Software Product Lines (IRT-SPL) strategy is inspired by earlier approaches in this domain [19, 21, 3, 14] and

developed to improve the reuse of tests case prefixes to reduce concretization effort. We use incremental test generation methods (to increment concretized tests) for full fault coverage criteria explained in Section 2.2. Figure 3 presents the main abstract steps of IRT-SPL.[1] Given a new product to check conformance we design the test model as an FSM M, identify all defined tests $D \subseteq \Omega_M$ that were concretized in old products, and execute the following sequence of steps and conditions:

Step 1 Process all defined tests of D to find divergent, convergent, and convergence-preserving tests from Lemmas 1, 2, and Definition 3, respectively. Also, initialize the set of new tests that need to be concretized $NT = \{\emptyset\}$.

Condition 1 Is the D set n-complete for M? When the answer is true, move to Step 2; otherwise, copy D to set T and move to Step 3.

Step 2 Increment tests using a test generation method. Incremental test generation methods have a local cost calculation that decides which new test gives a small increment based on tests of D. Thus, we used the P method [19] for this step. New tests are incremented from D and put in NT, resulting in an n-complete test suite $T = D \cup NT$.

Step 3 Select tests using our algorithm (described next, depicted in Figure 3.(b)). Execute the selection algorithm using M and T as parameters, obtain the n-complete test suite S and return $R = S - NT$ as the set of selected concretized tests and NT as the set of non-concretized tests.
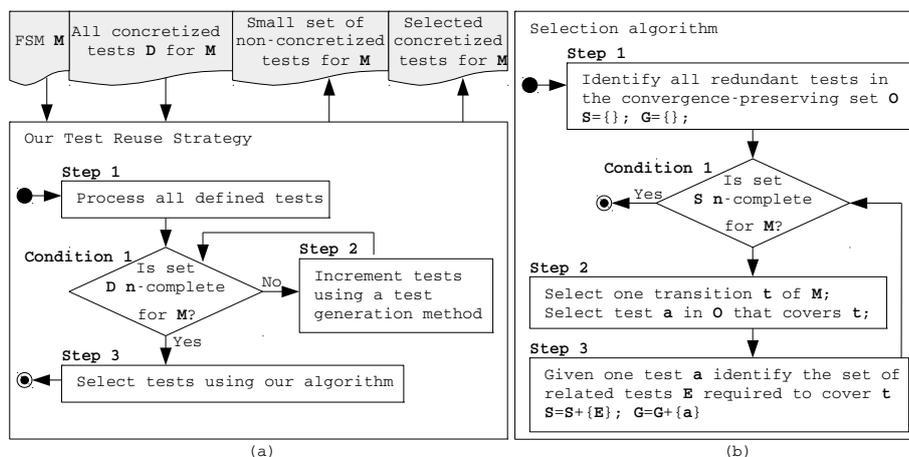

Fig. 2. (a) IRT-SPL test reuse strategy, and (b) selection algorithm.

This strategy can be adapted to other non-incremental test generation methods and other test coverage criteria, however, it may have weaker reuse efficiency. In

---

[1] The detailed algorithm can be found in `http://ceres.hh.se/mediawiki/Vanderson_Hafemann`

Section 4, we provide results of experimental evaluation for IRT-SPL compared to other reuse strategies.

3.2 Selection Algorithm

The proposed selection algorithm is developed as the last step of our IRT-SPL strategy. Given an FSM M, an n-complete test suite T, and an initialized convergence-preserving transition cover set $O \subseteq T$ for M, we select non-redundant tests of O resulting in an n-complete test suite $S \subseteq T$. The main steps are:

Step 1 Identify all redundant tests. All tests of O that cover each transition of M are identified. Also, the resulting set S and $G \subseteq O$ (coverage check) are initialized.

Condition 1 Is the S set n-complete for M? The set $G \subseteq S$ has to be convergence-preserving for M according to Theorem 1. When the answer is true, return S; otherwise, move to Step 2.

Step 2 Select a transition t of M that is not yet covered. Select a test $a \in O$ that covers t as follows: Among the redundant tests in O that cover t, select the test that gives the smallest increment of tests for S.

Step 3 Given a test a identify the set of related tests E required to cover t. Every test $a \in O$ may be linked to several tests that were used to build the convergence-preserving property. Update S and G by incrementing them with E and a, respectively.

Test sets The relations C and D represent subsets of $\Im_n(T)$-convergent and $\Im_n(T)$-divergent, respectively. The relation D is initially the set of all pairs of T-separated tests according to Definition 1. Next, a M-divergent state cover set K with n tests is identified. The relation C is the set of all pairs of $\Im_n(T)$-convergent tests according to Definition 2 (including the identity set where $(\alpha, \alpha) \in C$). We used 10 rules that were introduced in [19] to identify extra pairs of convergent and divergent test pairs in C and D, respectively. The ten rules are [19]:

1. If (α, β) is added to C, for each $(\alpha, \chi) \in C$, add (β, χ) to C.
2. If (α, β) is added to C, then, for all their common extensions $\alpha\varphi, \beta\varphi \in T$, add $(\alpha\varphi, \beta\varphi)$ to C.
3. If (α, β) is added to D, and they are common extensions of tests α' and β', then add (α', β') to D.
4. If (α, β) is added to C, then, for each $\chi \in T$ if $(\alpha, \chi) \in D$, add (β, χ) to D; if $(\beta, \chi) \in D$, add (α, χ) to D.
5. If (α, β) is added to D, then, for each $\chi \in T$ if $(\alpha, \chi) \in C$, add (β, χ) to D; if $(\beta, \chi) \in C$, add (α, χ) to D.
6. If (α, β), with $\alpha \leq \beta$, is added to D and there exists sequence φ and $k > 1$, such that $\beta = \alpha\varphi^k$, then add $(\alpha, \alpha\varphi)$ to D.
7. If $(\alpha, \alpha\beta\gamma)$ is added to C, and $(\alpha, \alpha\gamma) \in D$, then add $(\alpha, \alpha\beta)$ to D.
8. If $(\alpha, \alpha\gamma)$ is added to D, then, for each sequence β such that $(\alpha, \alpha\beta\gamma) \in C$, add $(\alpha, \alpha\beta)$ to D.

9.   If $(\alpha, \alpha\gamma)$ is added to C, then, for each sequence β such that $(\beta, \beta\gamma) \in D$, add (α, β) to D.
10.   If $(\beta, \beta\gamma)$ is added to D, then, for each sequence α such that $(\alpha, \alpha\gamma) \in C$, add (α, β) to D.

The relation $C_{\cup}(K) = \{\beta | (\alpha, \beta) \in C, \alpha \in K\}$ is an $\Im_n(T)$-convergence-preserving set for M according to Definition 3. Moreover, we define sets $V \subseteq C \cup D$ for verified pairs of tests of C and D, $G \subseteq C_{\cup}(K)$ for goal coverage, and $S \subseteq T$ to store the selected tests of T. To identify convergent and divergent pairs that were added to C and D by rules 1-10, we use designed inverse rules to trace tests back to T-separated pairs where they originally came from.

11.   If $(\beta, \chi) \in C$ was added by rule 1, then check $(\alpha, \beta), (\alpha, \chi) \in C$
12.   If $(\alpha\varphi, \beta\varphi) \in C$ was added by rule 2, then check $(\alpha, \beta) \in C$
13.   If $(\alpha, \beta) \in D$ was added by rule 3, then check $(\alpha\gamma, \beta\gamma) \in D$.
14.   If $(\alpha, \chi) \in D$ was added by rule 4, then check $(\beta, \chi) \in D, (\alpha, \beta) \in C$.
15.   If $(\alpha, \chi) \in D$ was added by rule 5, then check $(\beta, \chi) \in C, (\alpha, \beta) \in D$.
16.   If $(\alpha, \alpha\varphi) \in D$ was added by rule 6, then check $(\alpha, \beta) \in D$, with $\alpha \leq \beta, \beta = \alpha\varphi^k, k > 1$.
17.   If $(\alpha, \alpha\beta) \in D$ was added by rule 7, then check $(\alpha, \alpha\gamma) \in D, (\alpha, \alpha\beta\gamma) \in C$.
18.   If $(\alpha, \alpha\beta) \in D$ was added by rule 8, then check $(\alpha, \alpha\gamma) \in D, (\alpha, \alpha\beta\gamma) \in C$.
19.   If $(\alpha, \beta) \in D$ was added by rule 9, then check $(\beta, \beta\gamma) \in D, (\alpha, \alpha\gamma) \in C$.
20.   If $(\alpha, \beta) \in D$ was added by rule 10, then check $(\beta, \beta\gamma) \in D, (\alpha, \alpha\gamma) \in C$.

Detailed Selection Algorithm The main detailed steps of the selection algorithm are presented in Figure 3.

Given an FSM M and n-complete test suite T for M, initially the algorithm processes T and identify sets D, K, C, and $C_{\cup}(K)$. Some tests may be added in those sets (except K) by rules 1-10. The verification set V start empty, coverage set G initialized, and the resulting selected set S empty.

Condition 1 checks whether G meets the condition of a n-complete test suite, and since it is initially empty Step 2 is executed. To populate G first select a transition t not covered by G and select a test $\alpha \in C_{\cup}(K)$ such that α covers t, $\chi \in K$, $(\alpha, \chi) \in C$, and $(\alpha, \chi) \notin V$. Then, we check whether the pair (α, χ) was added to C by some rule or by Lemma 2. Let Condition 2 be true for (α, χ), then on Condition 3 no pair (α, υ) is true as V is empty at this point. Moving to Condition 4, let it be true, then tests $\alpha\gamma, \upsilon\gamma$ are added to S and (α, υ) added to V marking this pair as visited and checked. Back to Condition 3, assuming that every other pair (α, υ) make Condition 4 true, then after verifying them Condition 3 turns true, (α, χ) is added to V, α is added to S and G finishing the basic cycle for t.

Let Step 2 select a test pair not verified $(\beta, \chi) \in C$ that was added to C by rule 1. If Condition 2 turn false, then Step 5 identify pairs e.g., $(\alpha, \beta), (\alpha, \chi) \in C$ by rule 11 that were used to add (β, χ) in C, then, add β and χ to S and put (β, χ) in S to mark as verified. First, if $(\alpha, \beta) \notin V$ then the execution continues on Condition 2 for (α, β) instead of (β, χ) and (α, χ) enters in a waiting state. If $(\alpha, \beta) \in V$, then (α, β) is ignored, and if $(\alpha, \chi) \notin V$ then the execution continues
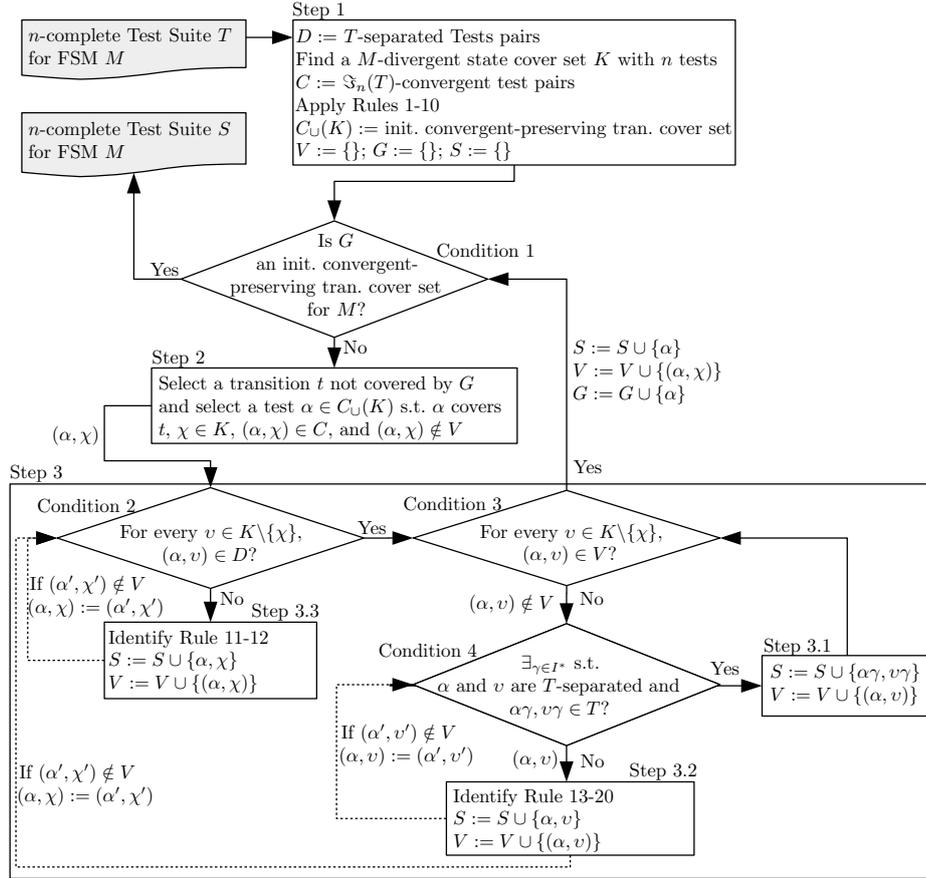
Fig. 3. Algorithm for reducing an n-complete test suite.

on Condition 2 for (α, χ) instead of (β, χ). However, if $(\alpha, \beta), (\alpha, \chi) \in V$ then the execution returns to the last waiting state. If there is no waiting point to return, then the execution stops with a failure, and T is not n-complete.

On Condition 3 for every pair $(\alpha, \upsilon) \notin V$ select one and use as input for Condition 4 and put the rest in a waiting state. If Condition 4 is false, then identify which pairs of tests were responsible on the addition of (α, υ) to D, then add α and υ to S, and (α, υ) to V to mark as verified. For example, if (α, υ) was triggered by Rule 14, then $(\beta, \upsilon) \in D, (\alpha, \beta) \in C$. If $(\beta, \upsilon) \notin V$ then the execution continues on Condition 4 for (β, υ) and (α, β) enters in a waiting state. If $(\beta, \upsilon) \in V$, then (β, υ) is ignored, and if $(\alpha, \beta) \notin V$ then the execution continues on Condition 2 for (α, β). However, if $(\beta, \upsilon), (\alpha, \beta) \in V$ then the execution returns to the last waiting state. If there is no waiting point to return, then the execution stops with a failure, and T is not n-complete.

Analysis In the remainder of this section, we prove that the algorithm terminates and the obtained test suite S is indeed n-complete.

Theorem 2. *Given an n-complete test suite T for an FSM M, the algorithm terminates with an n-complete test suite $S \subseteq T$.*

Proof. The algorithm contains four cycles. We show that each cycle can be executed a finite number of times and, thus, the algorithm indeed terminates. Then, we prove that the resulting test suite is n-complete.

Cycle 1 corresponds to the executions where Condition 1 does not hold, but Conditions 2-3 do, i.e. select tests to populate G. As the selected pair (α, χ) cannot be in the set of verified pairs V and the number of pairs in C is finite, at the end of the cycle (α, χ) is put in V and cannot be selected again, thus, the number of executions of Cycle 1 is bounded.

Cycle 2 corresponds to the executions where Condition 3 does not hold, but Condition 4 does, i.e. the pair (α, υ) is T-separated. On Step 3 (α, υ) is put in V to mark as verified. As the number of pairs in D is finite, the Cycle is bounded to D size.

Cycle 3 corresponds to the executions where Condition 2 does not hold, i.e. the pair (α, χ) was added to C by rules. On Step 5 (α, χ) is put in V to mark as verified. The cycle continues only if one of the responsible pairs for (α, χ) is not in V, otherwise, the execution returns to the last waiting state. As the number of pairs in C is finite, the Cycle is bounded to C size.

Cycle 4 corresponds to the executions where Condition 4 does not hold, i.e. the pair (α, υ) was added to D by rules. On Step 4 (α, υ) is put in V to mark as verified. The cycle continues only if one of the responsible pairs for (α, υ) is not in V, otherwise, the execution returns to the last waiting state. As the number of pairs in D is finite, the Cycle is bounded to D size.

Given an FSM M, we now show that the obtained test suite S is n-complete. When the algorithm terminates, Condition 1 holds and G is an initialized $\Im_n(S)$-convergent-preserving transition cover set for M, by Definition 1. For every test α added to G, a subset of tests that made $\alpha \in C_\cup(K)$ were added to S resulting in a n-complete test suite for M. □

The time required to execute the algorithm is similar to the P method, since several steps have the same complexity executed in inverse order. However there are two exceptions: the state cover set K that was identified by simple breadth-search instead of a clique; and we only apply the first 10 rules on the first step instead of each step of P method. We refer to [19] for more details about the complexity on each step.

3.3 Example

In this section we present an example of the IRT-SPL strategy to test new SPL products.

Example 1. The Arcade Game Maker (AGM) [18] is able to produce up to six arcade games with different game rules. All games are played by a single player, aiming to get more points. Figure 4 (a) shows the feature model of AGM. There are three alternative features for the game rule (`Brickles`, `Pong` and `Bowling`) and one optional feature (`Save`) to save the game. A new product p_3 is designed for configuration `Pong` rule without the `Save` option that is represented by a deterministic, complete, initially connected and minimal FSM M_3 illustrated by Figure 4 (b).
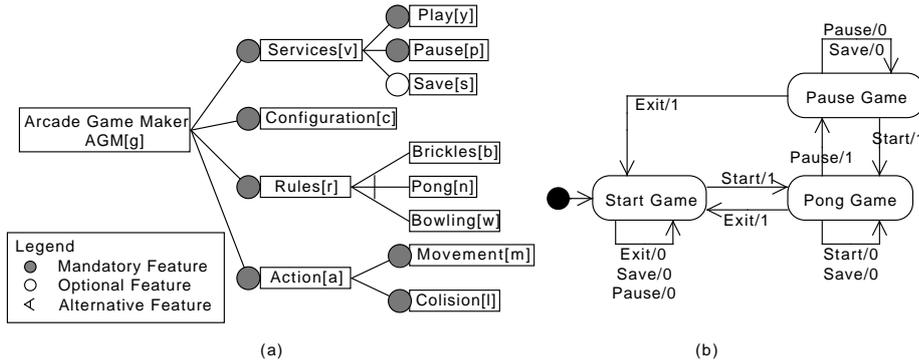

Fig. 4. (a) AGM Feature Model (adapted from [18]) and (b) FSM of the new product for `Pong` rule.

Assume that two products were already tested for `Brickles` with and without the `Save` option. Figure 5 shows four test sets generated by IRT-SPL: (a) defined tests D for M_3 that were already concretized before; (b) n-complete test suite T for M_3 generated by P method by incrementing D; (c) a selected n-complete test suite $S \subseteq T$ for M_3 generated by our selection algorithm; and (d) test set R for retesting the unchanged behavior. Test cases were simplified for readability and each input is abbreviated as follows: (i) SG - Start, (ii) PS - Pause, (iii) EX - Exit, and (iv) SV - Save.

```
 1  SG,PS,PS,EX              1  SG,PS,PS,EX           1  EX,PS,EX            1  EX,PS,EX
 2  PS,EX                    2  PS,EX                 2  SG,PS,PS,EX         2  SG,PS,PS,EX
 3  EX,PS,EX                 3  EX,PS,EX              3  SV,PS,EX            3  SV,PS,EX
 4  SV,PS,EX                 4  SV,PS,EX              4  SG,SG,PS            4  SG,SG,PS
 5  PS,PS,EX                 5  PS,PS,EX              5  PS,EX               5  PS,EX
 6  SG,PS,EX,PS,EX           6  SG,PS,EX,PS,EX        6  SG,EX,PS,EX         6  SG,EX,PS,EX
 7  SG,PS,SV,EX              7  SG,PS,SV,EX           7  SG,SV,PS            7  SG,SV,PS
 8  SG,SV,PS,EX              8  SG,SV,PS,EX           8  SG,PS,SG,PS         8  SG,PS,SG,PS
 9  SG,SG,PS,EX              9  SG,SG,PS,EX           9  SG,PS,EX,PS,EX      9  SG,PS,EX,PS,EX
10  SG,EX,PS,EX             10  SG,EX,PS,EX          10  SG,PS,SV,EX        10  SG,PS,SV,EX
11  SG,PS,SG,PS,EX          11  SG,PS,SG,PS,EX       11  SG,PS,SV,PS,SG
12  SG,SV,SV,SV,PS,EX       12  SG,SV,SV,SV,PS,EX
13  SG,SV,SV,PS,EX,PS,EX,SG,PS,EX 13 SG,SV,SV,PS,EX,PS,EX,SG,PS,EX      (c)                 (d)
                            14  SG,PS,SV,PS,SG
          (a)                         (b)
```

Fig. 5. Test sets: (a) defined tests D for M_3; (b) n-complete test suite T for M_3; (c) selected n-complete test suite S for M_3; and (d) test set R for retest unchanged behavior.

Note that the difference between (a) and (b) is the addition of Line 14 on (b), as well as the inputs designated in bold. As explained in Section 2.3, we only count the concretization cost using suffixes of tests that were incremented from another test already present in D. Since all four sets are prefix-closed, every prefix is also a test to be counted. Notice that the prefix (SG, PS, SV) of Line 14 (b) is already present on Line 7 as a prefix that can be reused. In column (c), the algorithm to select concretized tests is executed using as input M_3 and column (b), such that it keeps new tests; subsequently, it reduces tests that were originally from column (a), because some of them are redundant to cover the unchanged behavior. Then, the set of new tests is defined in Line 11 (c) and the retest set is given in column (d).

The effort to test p_3 using M_3 and IRT-SPL is $effort = (2 * x) + 51$. Assuming that $x = 10$, the suffix PS, SG count 2 and $|TR|$ (c) 51, resulting in $effort = 71$.

4 Experimental Study

To evaluate the applicability of IRT-SPL and measure the reuse efficiency of our selection algorithm, we conducted an experiment. We measured the efficiency of our algorithm and compared it to a few alternative approaches reported in the literature. Our research question is: *How much test effort can be saved using IRT-SPL to test a set of new SPL products compared to existing test case reuse strategies?*

4.1 Experimental Setup

The setup of our experiment consists of designing several SPL products in different orders and compare the total effort required to test all products. We compare the effort of IRT-SPL to other reuse strategies found in the literature. A survey on some approaches [8, 17, 4, 5, 21, 3, 13, 14, 1, 22] indicated two reuse strategies, which are described bellow:

1. The first reuse strategy (henceforth called TSPL) was proposed by Capellari et al. [3]. In their approach, only test cases from the last product are used to increment tests in conformance testing of a new product.

2. The second reuse strategy (henceforce referred to as DIATP) was a delta-oriented incremental testing process proposed by Lochau et al. [14]. They reuse test cases of all previous products to test the unchanged behavior of the new product, but they generate new tests for the changed behavior without incrementing based on the (test prefixes from the) old tests.

To obtain a fair comparison among our strategy, TSPL, and DIATP, we setup similar environments also using the P method for TSPL and DIATP. Figure 6 (a) shows the environment for TSPL to get the required sets for conformance testing and calculate the effort required. Defined tests for a new product are reused from the last derived product and incremented without any selection/reduction. Figure 6 (b) shows the environment for DIATP to get required sets for conformance testing and calculate the effort required. In the existing approaches (TSPL and DIAIP), all defined tests for a new product are reused without increment.
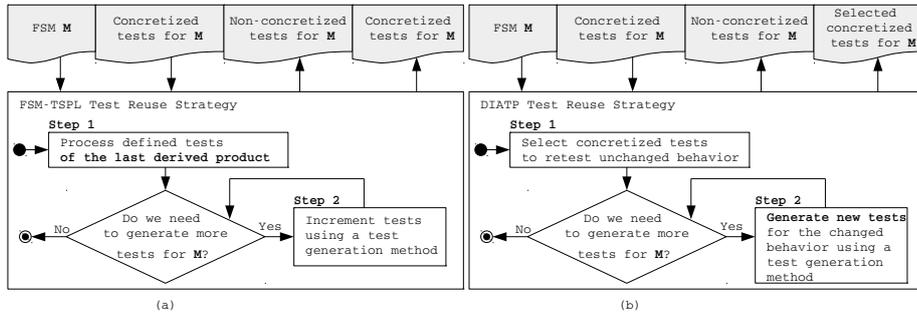

Fig. 6. Adapted reuse strategies for (a) TSPL and (b) DIATP.

The implementation of our experiments was executed in Java. The running environment used Ubuntu 14.04 LTS (64 bit) on an Intel processor i5-5300U at 2.30GHz[2].

4.2   Case Study

The embedded camera Mobile Media SPL [11] was used to compare all three reuse strategies. The Mobile Media SPL contains several features, such as photo manipulation, music, and videos on mobile devices. Figure 7 (a) presents the feature diagram with three alternative features (`Photo(MP)`, `Music(MM)`, and `Video(MV)`) and three optional (`Favourites(F)`, `Copy Media(CM)`, and `SMS Transfer(SMS)`) used to characterize all possible configurations of the SPL. Figure 7 (b) presents

---

[2] The experiment package for Eclipse IDE can be found in `http://ceres.hh.se/mediawiki/Vanderson_Hafemann`

24 configurations of Mobile Media used to design corresponding products. Note that each product of the order 1 to 24 increases the number of features compared to the previous products. Thus, we designed those products in three orders: (i) increasing features (1 to 24); (ii) decreasing features (24 to 1); and (iii) random derivation.
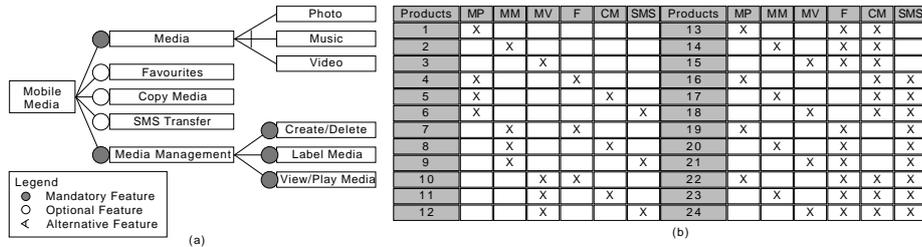

Products	MP	MM	MV	F	CM	SMS	Products	MP	MM	MV	F	CM	SMS
1	X						13	X			X	X	
2		X					14		X		X	X	
3			X				15			X	X	X	
4	X			X			16	X				X	X
5	X				X		17		X			X	X
6	X					X	18			X		X	X
7		X		X			19	X			X		X
8		X			X		20		X		X		X
9		X				X	21			X	X		X
10			X	X			22	X			X	X	X
11			X		X		23		X		X	X	X
12			X			X	24			X	X	X	X

Fig. 7. (a) Mobile Media feature model; and (b) derived products from Mobile Media SPL.

For each product, an FSM was modeled with varying number of states from three to six, and with fixed eight inputs and two outputs. All FSMs share the same basic properties for test case generation, namely, they are complete, deterministic, reduced, and initially connected.

4.3 Analysis and Threats to Validity

The collected data after running our experiments is visualized in Figure 8. The variable x is the concretization value from the effort formula $effort= (concrete * x) + execution$ presented in Section 2.3. On (a) and (b) we have the derivation of products with an increasing number of features when $x = 10$ and $x = 100$, respectively. On (c) and (d) we have the derivation of products with decreasing number of features when $x = 10$ and $x = 100$, respectively. Finally, on (e) and (f) we have the derivation of products with a random number of features when $x = 10$ and $x = 100$, respectively.

As an immediate observation, we noticed that the total effort required to test the Mobile Media SPL in any order vary according to the value of x. Also, the number of newly designed products should be considered as for few products there is no significant difference of effort.

To summarize, we conclude that if the concretization cost is over 100 times the execution, then for our case study IRT-SPL can save at least 36% of the total effort required to test all SPL products in any order compared to other test reuse strategies. (Currently available case studies [20, 12] suggest that concretization time for each test case is about 200 times its execution time.) However, the time to execute every strategy depends on the complexity of the P method. Traditional test generation methods for the full fault coverage that use FSMs as test models (e.g. W [6], HSI [15]) are not incremental. Thus, they cannot increment test
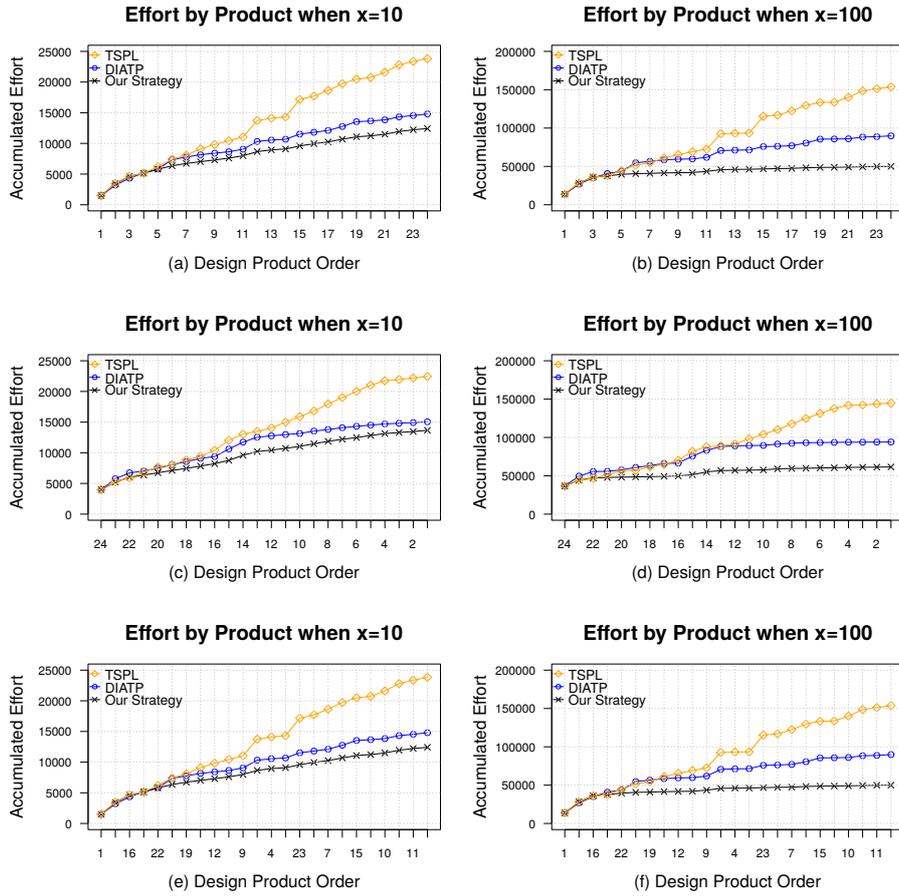
Fig. 8. Accumulated effort per designed product when concretization cost is x times the cost of execution. (a) the increment of features for $x = 10$ (b) the increment of features for $x = 100$; (c) the decrement of features for $x = 10$; (d) the decrement of features for $x = 100$; (e) random features for $x = 10$; and (f) random features for $x = 100$.

cases for new specifications based on old test cases to improve reuse. However, generated new tests can be compared to old tests for reuse resulting in a weaker reuse strategy as some of these new tests may be equivalent to existing old tests.

Our experiment has been limited mostly to non-hierarchical FSMs with few states and few new products. One of the issues regarding the P method is the increasing time required to generate tests based on the number of states, inputs and input test set. Both of these issues (few subjects and limiting time) are threats to the validity of our results for real-world cases. We plan to mitigate these threats by analyzing a number of realistic case studies as a benchmark for our future research. Realistic FSMs use hierarchy to sustain scalability. Hence, an extension of FSM-based test generation methods for full fault coverage to hierarchical FSMs is required. We plan to investigate this further in the near future.

5   Related Work

Much recent research has been devoted to developing efficient testing techniques for SPLs by exploiting variability in a systematic manner; we refer to [10, 16] for recent surveys of the field.

There are several incremental test approaches [8, 17, 21, 3, 1] devoted to generating, reusing, and optimizing test suites for SPLs. El-Fakih et al. [8] presented an incremental algorithm to automatically re-generate tests in response to changes to a system specification. Pap et al. [17] extended their work and designed an algorithm that maintains two sets incrementally based on the HSI method [15]. Capellari et al. [3] explored the FSM-based Testing of SPLs (FSM-TSPL) testing strategy where the P method is used to design new tests based on the last product derived. Uzuncaova et al. [21] developed an incremental test generation approach, while Baller and Lochau [1] focused on test suite optimization. Moreover, recent delta-oriented approaches [13, 14, 22] developed regression-based SPL approaches to design and reuse test artifacts.

In contrast to current approaches, our work introduces a test reuse strategy focused on reducing concretization effort for the set of new SPL products. We analyze the already concretized tests of derived products to generate a small set of new tests to-be-concretize for conformance testing. To our knowledge, there is no proposal that reuse tests from all previous products to reduce concretization effort for new SPL products using incremental test generation methods.

6   Conclusions

This paper proposes a test reuse strategy named Incremental Regression-based Testing for Software Product Lines (IRT-SPL) that aims at reducing test effort on checking conformance of several SPL products. Tests of previously designed products can be efficiently reused for a newly designed product using incremental test generation methods to reduce the number of required tests for concretization. We assume that concretization of tests (as seen in some case studies [20, 12])

is several times more expensive than executing the same test. Thus, the effort required to test such new product is directly related to concretization costs.

Finite State Machines were used to represent the abstract behavior of the products as test models. To maximize reuse of tests all concretized tests are analyzed and some of them are selected to retest the unchanged behavior of the new product under test. Thus, our strategy also contains a selection algorithm to perform the selection of non-redundant concretized tests.

To illustrate our strategy, we used examples and a case study of an embedded Mobile Media SPL [11]. The results indicate that our approach can save at least 36% test effort for 24 products when the concretization cost is 100 times more expensive than execution compared to current test reuse strategies for the same fault detection capability.

The problem and the approach described above are very much inspired by a similar problem and approach in regression testing. Regression testing concerns testing software evolution in time (in versions) while SPL testing is about testing software evolution in space (in features) [10]. Hence, we believe our approaches and the obtained results will be also applicable to the regression testing setting.

As future work, we plan to investigate test models with hierarchy and adapt test generation methods for such models to handle scalability problems. Also, we intend to investigate more studies regarding formal representations of SPLs in order to perform incremental reuse of test artifacts.

References

1. Baller, H., Lochau, M.: Towards incremental test suite optimization for software product lines. In: Proc. of FOSD 2014. pp. 30–36. ACM (2014)
2. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems, Advanced Lectures, Lecture Notes in Computer Science, vol. 3472. Springer-Verlag, Berlin (2005)
3. Capellari, M.L., Gimenes, I.M.S., Simao, A., Endo, A.T.: Towards Incremental FSM-based Testing of Software Product Lines. In: Proc. of SBQS 2012. pp. 9–23 (2012)
4. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: Proc. of A-MOST 2007. pp. 54–62. ACM Press (2007)
5. Chen, Y., Probert, R.L., Ural, H.: Regression test suite reduction based on SDL models of system requirements. Journal of Software Maintenance and Evolution: Research and Practice 21(6), 379–405 (2009)
6. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering SE-4(3), 178–187 (1978)
7. Dev, R., Jääskeläinen, A., M., K.: Advances in Computers, vol. 85. Elsevier (2012)
8. EI-Fakih, K., Yevtushenko, N., Bochmann, G.: FSM-based incremental conformance testing methods. IEEE Transactions on Software Engineering 30(7), 425–436 (2004)
9. Endo, A.T., Simao, A.: Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. Information and Software Technology 55(6), 1045–1062 (2013)
10. Engström, E.: Exploring Regression Testing and Software Product Line Testing -Research and State of Practice. Ph.D. thesis, Lund University (2010)

11. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving software product lines with aspects. ACM/IEEE 30th International Conference on Software Engineering pp. 261–270 (2008)
12. Graham D., F.M.: Experiences of Test Automation: Case Studies of Software Test Automation, vol. 1. Addison-Wesley Professional (2012)
13. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In: Proc. of TAP 2012. pp. 67–82 (2012)
14. Lochau, M., Lity, S., Lachmann, R., Schaefer, I., Goltz, U.: Delta-oriented model-based integration testing of large-scale systems. Journal of Systems and Software 91, 63–84 (may 2014)
15. Luo, G., Petrenko, A., Petrenko, R., Bochmann, G.V.: Selecting Test Sequences For Partially-Specified Nondeterministic Finite State Machines. In: Proc. of IFIP 1994. pp. 91–106 (1994)
16. Oster, S., Wubbeke, A., Engels, G., Schurr, A.: A Survey of Model-Based Software Product Lines Testing. In: Model-Based Testing for Embedded Systems, pp. 338–381. CRC Press (2012)
17. Pap, Z., Subramaniam, M., Kovács, G., Németh, G.Á.: A Bounded Incremental Test Generation Algorithm for Finite State Machines. In: Proc. of IFIP 2007. pp. 244–259. Springer (2007)
18. SEI: A framework for software product line practice (2011), `http://www.sei.cmu.edu/productlines/tools/framework/`
19. Simao, A., Petrenko, A.: Fault Coverage-Driven Incremental Test Generation. The Computer Journal 53(9), 1508–1522 (2010)
20. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach, vol. 1. Morgan Kaufmann (2006)
21. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental Test Generation for Software Product Lines. IEEE Transactions on Software Engineering 36(3), 309–322 (2010)
22. Varshosaz, M., Beohar, H., Mousavi, M.: Delta-oriented fsm-based testing. In: Proc. of ICFEM 2015 LNCS. pp. 366–381. Springer (2015)