electronics

MDPI

Article

Reducing Redundant Test Executions in Software Product Line Testing—A Case Study

Pilsu Jung [1], Sungwon Kang [1] and Jihyun Lee [2,*]

[1] School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, Korea; psjung416@gmail.com (P.J.); sungwon.kang@kaist.ac.kr (S.K.)
[2] Department of Software Engineering, Jeonbuk National University, Jeonju 54896, Korea
* Correspondence: jihyun30@jbnu.ac.kr; Tel.: +82-63-270-4860

Abstract: In the context of software product line (SPL) engineering, test cases can be reused for testing a family of products that share common parts of source code. An approach to test the products of a product family is to exhaustively execute each test case on all the products. However, such an approach would be very inefficient because the common parts of source code will be tested multiple times unnecessarily. To reduce unnecessary repetition of testing, we previously proposed a method to avoid equivalent test executions of a product line in the context of regression testing. However, it turns out that the same approach can be used in a broader context than just regression testing of product families. In this paper, we argue the generality of the method in the sense that it can be used for testing of the first version of a product family as well as regression testing of its subsequent versions. In addition, in this paper, in order to make the method practically usable for users, we propose a process for applying it to SPL testing. We demonstrate the generality of our method and the practical applicability of the proposed process for the method by conducting a case study.

Keywords: software product line; software testing; test redundancy; test case execution

Citation: Jung, P.; Kang, S.; Lee, J. Reducing Redundant Test Executions in Software Product Line Testing—A Case Study. *Electronics* **2022**, *11*, 1165. https://doi.org/10.3390/electronics11071165

Academic Editors: George Angelos Papadopoulos and Claus Pahl

Received: 9 February 2022
Accepted: 5 April 2022
Published: 6 April 2022

1. Introduction

Software product line (SPL) engineering is a paradigm of software development for reusing managed artifacts to produce a product family that shares common artifacts [1] (pp. 14–15). In a product family, products are different, yet similar to each other. Due to the commonality shared between products, exhaustively testing all the products of a product family would have much redundancy and result in an unnecessarily high testing cost.

To reduce such redundancy, combinatorial techniques [2–6] have been proposed. They reduce the number of products to test by selecting only a subset of products to increase the coverage of combinatorial feature interactions. However, the redundancy problem still remains with the techniques because the selected products of a product family would have to be individually tested as they do not provide ways of exploiting commonality between selected products. To overcome this drawback, delta-oriented techniques [7–11] have been proposed. They reduce unnecessary testing by reusing test cases of one product of a product family for the other products of the family that share common artifacts. However, when test cases are executed on the products that reuse them, these test executions can include redundant test executions that traverse the same sequence of code statements and produce the same test results (i.e., pass or fail). For this reason, in the context of SPL testing, there is a possibility of reducing test redundancy at the level of test executions, in addition to the levels of products or test cases.

Given a set of test cases T, let E_T be the set of test executions obtained by applying the test cases in T and Faults(E_T) be the set of the faults that are detected by E_T. A test execution e in E_T is said to be a redundant test execution of T if Faults($E_T - \{e\}$) = Faults(E_T) (i.e., e does not help finding faults). Redundant test executions can occur when two products

have common source code [12]. For example, when products of a product family, say P1, P2 and P3, have the same feature that is independent of other features, if the feature has been tested in P1, then testing it again in P2 would be redundant because it does not increase the chance of detecting defects in the feature. The existing variability-aware test execution methods [13–15] reduce redundant test executions for a configurable system by executing it so that the source code common to multiple configurations is covered only once. With some adjustments, the methods can be applied to testing of a product line that produces a set of products which are obtained by assembling a set of code fragments of the product line. However, their methods do not address the execution of existing test cases and thus cannot be applied to regression testing of a product line.

In our previous work [16], we proposed a method of reducing redundant test executions of a product line in the context of regression testing. Given a modified product family and a set of test cases selected for a retest through an existing regression test selection method, our previously proposed method (to be abbreviated as our method in this paper) builds the checksum matrix for the modified product family, which contains checksum values of every code unit (i.e., a section of code written in a programming language that can be executed [16]) of the family. The checksum value of a code unit is computed by applying a hash function to the statements of the code unit. Using the checksum matrix, our method avoids repetitions of equivalent test executions by identifying products where a test case covers the same sequence of identical code units. Through these steps, it reduces the total number of test executions by avoiding test executions that will produce the same execution traces and test results. However, the method focused only on the context of regression testing.

The products of a product family can be viewed as different versions of one product of the family. Then, differences between two products of a family can be viewed as regression deltas. Thus, a regression testing strategy can be applied for testing of a (single version of) software product line [7]. Several techniques [7–11] on SPL testing applied regression testing strategies for SPL testing and reported its validity and effectiveness. Along this direction, our method, which was originally proposed for SPL regression testing in [16], can also be applied for the testing of the initial version of a software product line, not just for its subsequent versions.

In this paper, we argue the generality of our method in the sense that it can be used for the testing of the first version of a software product line as well as the regression testing of its subsequent versions. Also in this paper, we propose an SPL testing process for applying our method and provide an algorithm for reducing redundant test executions, which improves the algorithm of our previous work [16]. Based on the process and the algorithm, we conduct a case study. As the SPL of our case study, we use the VendingMachine SPL because it has been well documented [17] and served as benchmarks in the literature [18].

This paper makes the following contributions:

- We extend our previous method [16] to make it more practicable by providing an improved algorithm that avoids equivalent test executions in order to reduce unnecessary repetition of test executions and unnecessary collection of test execution traces.
- We propose a practical process for applying our method to testing of the first version of a product family as well as regression testing of its subsequent versions.
- Through a case study based on the proposed process, we demonstrate the generality of our method and how it can be used for efficient testing of software product lines. The case study shows that, for the first version and its two subsequent versions of the VendingMachine SPL, our method reduces 42.3% of test executions compared to the exhaustive execution.

The rest of the paper is organized as follows: Section 2 describes how to identify equivalent test executions; Section 3 presents an SPL testing process and an improved version of our previous algorithm for avoiding equivalent test executions; Section 4 demonstrates the generality of our method through a case study; in Section 5, we discuss related work regarding SPL testing; finally, in Section 6, we conclude the paper.

2. Equivalence of Test Executions in SPL

This section provides an example with code snippets to illustrate the notion of equivalence of test executions in SPL that was defined in our previous paper [16]. Figure 1 shows the Lock module of the DoorLock SPL, which has one variation point (i.e., Lock) and three variants (i.e., Fingerprint Scanner, Keypad and Magnetic Card). The Lock variation point has an alternative relationship with its variants, which requires that only one variant among its variants can be selected to produce a product. When a variant is selected to produce a product, the corresponding code is enabled and the code for the other variants is disabled. For example, if the Fingerprint Scanner is selected, line 5 of the Lock class and the code of the FingerPrintAccess class are enabled and lines 6 and 7 of the Lock class and the code of the PINAccess and MagneticAccess classes are disabled. It is assumed that none of the classes in the Lock module contain non-deterministic functions (e.g., the random function). Based on this principle, the DoorLock SPL can produce a set of products as a product family by selecting, for each product, variants of variation points for all the modules (including the modules omitted in this example). We denote the set of products containing the Fingerprint Scanner variant as P_{FS}, the set of products containing the Keypad variant as P_K, the set of products containing the Magnetic Card as P_{MC} and the set of all products that can be produced from the DoorLock SPL as P_{ALL}. Then, P_{ALL} is equal to $P_{FS} \cup P_K \cup P_{MC}$.

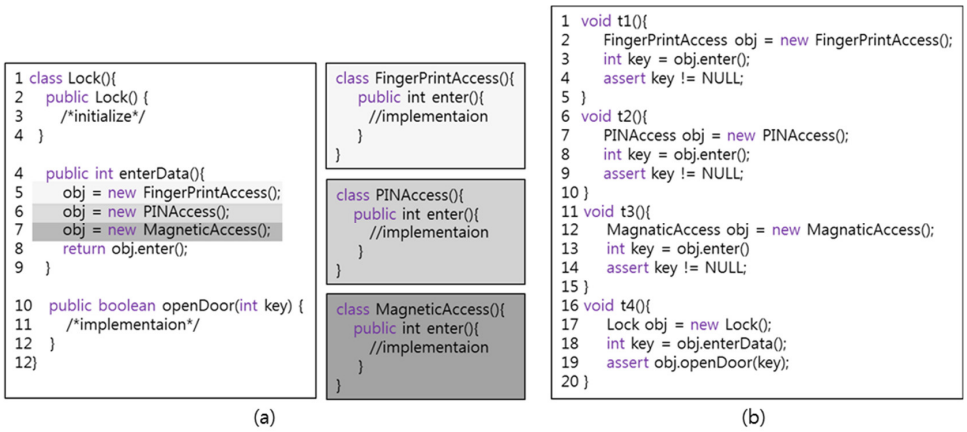

Figure 1. An illustrative example using the DoorLock SPL. (**a**) product line code base. (**b**) test cases.

Let us suppose that test cases t1, t2, t3 and t4 are used to test products in P_{ALL}. In this case, the executions of t1 applied to the products in P_{FS} are equivalent because, for each product $p \in P_{FS}$, t1 traverses only FingerPrintAccess on any $p \in P_{FS}$ and the source code of the FingerPrintAccess class is the same for all the products in P_{FS}. Likewise, the executions of t2 applied to the products in P_K are equivalent and the executions of t3 applied to the products in P_{MC} are equivalent. Therefore, executing each test case on all the relevant products is redundant [16].

3. SPL Testing Process for Avoiding Equivalent Test Executions

In this section, we argue the generality of our method and present a practical process for applying our method to SPL testing in general including regression testing.

In our previous work [16], our method was presented in the context of regression testing where the regression testing is performed by executing only a subset of the existing test cases that is affected by changes. That is, in our method, regression testing is performed by executing a subset of the previous version of test cases on the current version of product family. A special case of this "current version" is the first version. If a testing method requires artifacts (e.g., fault detection history, change information, code dependencies, etc.) from the previous version of product family, the method cannot be used for testing of the first version. However, our method uses only the artifacts for the current version of product family, without relying on the artifacts from the previous version of product family. For this

reason, our method of avoiding redundant test executions for regression testing, which is a testing of the second and later versions, can be used for testing of the very first version of the product family by using the full set of test cases as input. This implies that our method is general enough for testing of any version of a product family including the first version.

Figure 2 shows a process for avoiding equivalent test executions that can be used in the general cases (i.e., the testing of one SPL version and the regression testing of its subsequent versions). The inputs are (1) the code base of a product line and (2) the test cases of the product line. The process first obtains a product family (in Step 1) and constructs its checksum matrix (in Step 2). Then, it selects test cases to run (in Step 3) and executes them (in Step 4). Generation of new test cases and deletion of obsolete test cases are not the scope of this paper. Therefore, we assume that the initial set of test cases is given and that new test cases are generated and obsolete test cases are deleted when a product line is modified. The resulting set of test cases is used as the input, i.e., 'Test cases of a product family' shown in the bottom part of Figure 2.

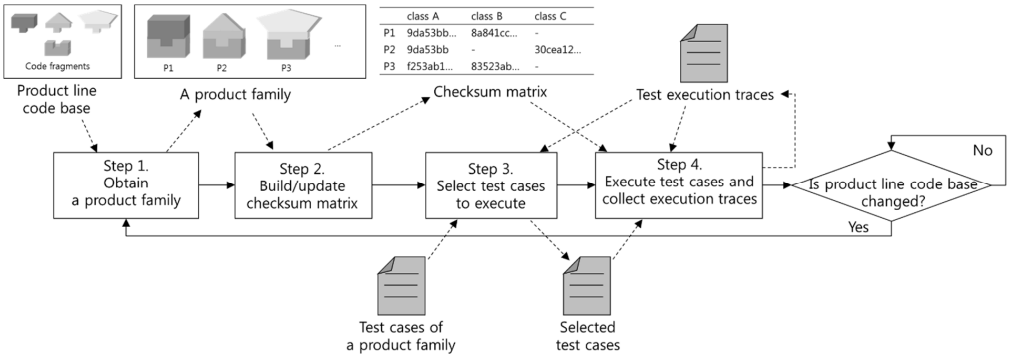

Figure 2. The process for avoiding equivalent test executions.

The proposed application process includes a minimal set of steps necessary to apply our method to SPL testing. Thus, for the full testing of a product line, additional steps for minimization and prioritization of test cases, change impact analysis, etc. can be added to the process.

3.1. Step 1: Obtain a Product Family

This step obtains a product family from the code base of a product line. At the time of testing the first version, all the products of a product family are instantiated from the code base. However, when changes are made to a product line code base, a new version of product family is obtained in this step by propagating changes to the products instantiated from the modified code base. The change propagation technique [19] can be used for this step.

3.2. Step 2: Build/Update Checksum Matrix

To check whether two code units are identical, our method compares their checksums. If the checksums of the two code units are the same, then they are identical, and otherwise, they are not identical. To store the checksum values of code units, this step builds the checksum matrix for a product family. For each product of a product family, it obtains a set of code units from the product. Then, for each code unit, it computes its checksum value and stores it to the checksum matrix. Table 1 shows an example of the checksum matrix for a product family consisting of P1, P2 and P3 implemented in the Java language. In this example, 8-bit checksum is used. This example checksum matrix shows that P1 and P3 contain Foo.class and Bar.class and P2 contains Foo.class and Baz.class. It also shows that Foo.class for P1 is identical to that for P2 as their checksum values are the same. The checksum matrix of a product family is built only once at the time of testing the first version of the family. From the next version of the family, this step updates the existing checksum matrix by modifying only the checksum values of changed classes.

Table 1. An example of checksum matrix.

Class \ Product	Foo.class	Bar.class	Baz.class
P1	9da53bba	8a841cc2	-
P2	9da53bba	-	30cea121
P3	f253ab1c	83523abb	-

3.3. Step 3: Select Test Cases to Execute

This step selects test cases to be executed on the products of a product family. At the time of testing the first version of a product line, the full set of test cases is selected. However, at the time of regression testing of its subsequent versions, this step selects a subset of the existing test cases that is affected by changes. The existing regression test selection approach can be used to select regression test cases. In our case study to be presented in Section 4, we apply the approach of Lity et al. [7], which selects test cases that traverse changes. For example, if a class of a product line has changed and a test case traverses the class, the test case is selected for regression testing of the product line.

3.4. Step 4: Execute Test Cases and Collect Execution Traces

This step executes test cases selected in Step 3 on the products of a product family and collects their test execution traces. For collection of test execution traces, the bytecode instrumentation library such as ASM [20] and BCEL [21] can be used. To reduce redundant test executions, this step uses the testExecution function of Algorithm 1, which is a generalized version of our previous algorithm [16]. Like its previous version, Algorithm 1 avoids unnecessary execution of test cases and unnecessary collection of test execution traces. However, unlike our previous algorithm, Algorithm 1 allows the testing of the first version as well as the testing of its subsequent versions, which makes our method a general SPL testing method.

Algorithm 1 An algorithm for avoiding equivalent test executions

```
Input: TP: target products
Input: SelectedTCs: set of test cases selected from Step 3
Input: chksumMatrix: checksum matrix for a product family
Use: execute(t, p): execute a test case t on a product p
and collect the execution trace of t on p
1.  function testExecution(TP, chksumMatrix)
2.    Map ETrace ← ∅            // key: test case, value: set of traces
3.    for each p ∈ TP do:
4.      T ← SelectedTCs.for(p)
5.      CandidateTCs.addAll(T)
6.      for each t ∈ T do:
7.        TR ← ETrace.getTracesOf(t)
8.        for each tr ∈ TR do: // tr: classes covered by t on products
9.          chksumListA ← chksumMatrix.lookup(p, tr.classes)
10.           chksumListB ← chksumMatrix.lookup(tr.product, tr.classes)
11.         if chksumListA equals chksumListB then:
12.           CandidateTCs.delete(t)
13.         end if
14.       end for
15.     end for
16.     pairs <TestCase, Trace> ← execute(CandidateTCs, p)
17.     ETrace.appendAll(pairs)
18.     CandidateTCs ← ∅
19.   end for
20. end function
```

For each product (p) of target products (TP), it obtains the test cases for p (lines 3–4) and adds these test cases to a set of the candidate test cases (CandidateTCs) (line 5). At the time of testing the first version of a product family, all products of the family are used as TP and at the time of regression testing of a product family, the changed products of the family are used as TP. Then, for each test case (t) for p, if p has classes that are identical to classes that t traversed on the previously tested one product, then the execution of t on p is excluded from CandidateTCs (lines 6–15) because the executions of t on the two products would be equivalent (Cf. Section 2). In this step, the order of selecting products and test cases does not affect the number of repetitions of test case execution. Finally, all the test cases in CandidateTCs are executed on p and their execution traces are collected for the testing of the next product (lines 16–18). This algorithm executes all the test cases of a product if it is the first chosen product of a product family. However, from the next product, the algorithm selectively executes test cases for the product, avoiding equivalent test executions. To identify the code units that are covered by each test case, this algorithm executes all the test cases at least once on a product of the family.

4. Case Study Using the VendingMachine SPL

In this section, we conduct a case study that applies the process for SPL testing presented in Section 3 (Figure 2). We use the VendingMachine SPL for this case study because its behavioral specifications and evolving scenarios have been well documented [17]. We apply the process to the testing of the first version of the VendingMachine SPL, SPL_{V0}, (in Section 4.1) and the regression testing of its subsequent versions, SPL_{V1} and SPL_{V2} (in Sections 4.2 and 4.3). Then, we discuss the results (in Section 4.4).

Figure 3 shows the feature model for SPL_{V0}. We first implemented features of the feature model using Java language, based on the behavioral specifications of the document [17]. These features were implemented using classes with the same name as the feature name except the VendingMachine feature. The VendingMachine feature was implemented using three classes whose names are VMachine, DrinkOrder and Display. Then, we generated the nine test cases for the first version of the product line, as specified in Table 2. Each test case validates the expected output for a given input, executing a subset of classes. Finally, we defined a product family consisting of four products. Table 3 lists the identifier of each product with the lines of code (LOC) in parentheses, test cases, feature configuration, and class set. Because each product contains a different set of classes, a different set of test cases is assigned to each product.

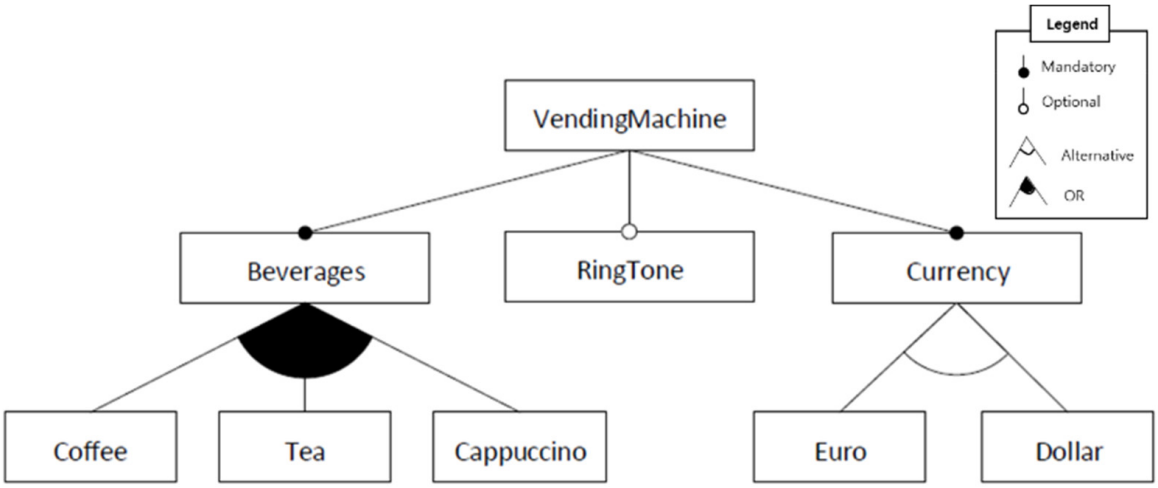

Figure 3. The feature model for SPL_{V0}.

Table 2. Test cases for SPL_{V0}.

Test Case	Input	Expected Output	Description
t1	1	True	Checks whether a cup of tea is provided when 1 is inputted.
t2	2	True	Checks whether a cup of coffee is provided when 2 is inputted.
t3	3	True	Checks whether a cup of cappuccino is provided when 3 is inputted.
t4	"Coffee"	True	Checks whether sugar is poured when a cup of coffee is made.
t5	1	"1 Euro"	Checks whether "1 euro" is displayed when 1 euro is inserted.
t6	5	"5 Dollars"	Checks whether "5 dollars" is displayed when 5 dollars is inserted.
t7	BeberageDelevered(=True)	True	Checks whether a tone is emitted when a beverage has been delivered.
t8	Ring-Off(=False)	False	Checks whether the turn-off switch of the ring correctly works.
t9	ErrNotEnough Sugar(=1)	"not enough sugar"	Checks whether an error message is displayed.

Table 3. A product family SPL_{V0}.

Product (LOC)	Test Case	Feature Configuration	Class Set
P1 (167)	t1, t5, t7, t8, t9	VendingMachine, Beverages, Tea, RingTone, Currency, Euro	VMachine, DrinkOrder, Display, Beverages, Tea, RingTone, Currency
P2 (167)	t3, t6, t7, t8, t9	VendingMachine, Beverages, Cappuccino, RingTone, Currency, Dollar	VMachine, DrinkOrder, Display, Beverages, Cappuccino, RingTone, Currency
P3 (182)	t2, t3, t4, t6, t7, t8, t9	VendingMachine, Beverages, Coffee, Cappuccino, RingTone, Currency, Dollar	VMachine, DrinkOrder, Display, Beverages, Coffee, Cappuccino, RingTone, Currency
P4 (140)	t1, t5, t9	VendingMachine, Beverages, Tea, Currency, Euro	VMachine, DrinkOrder, Display, Beverages, Tea, Currency

4.1. Testing of the Initial Version of a Product Line (Testing of SPL_{V0})

In this section, we present a testing of the first version of the VendingMachine SPL.

Step 1. Obtain a product family. Because this is the first testing of a product family, all the products of the family, i.e., P1, P2, P3 and P4, are instantiated from the code base. Then, the corresponding classes are assembled for each product as presented in Table 3.

Step 2. Build/update checksum matrix. The checksum matrix for the VendingMachine product family is constructed. Table 4 presents the checksum matrix of the family. To simplify the example, we present the checksum value as a single character (e.g., a, b, c and d). Two classes with the same character indicate that their checksum values are the same and so they have an identical source code. The checksum values (i.e., a, b, c and d) of the VMachine class are different for the products of the family because each product provides a different set of beverages and so the source code of the VMachine classes are different for the products. In addition, the checksum value (i.e., i) of the DrinkOrder class for P1 and P4 is different from that for P2 and P3 (i.e., j) because P1 and P4 provide only sugar for serving beverages but P2 and P3 provide milk in addition to sugar. The checksum value (i.e., l) of the Currency class for P1 and P4 is different from that (i.e., m) for P2 and P3 because P1 and P4 use euro for payment while P2 and P3 use dollar.

Step 3. Select test cases to execute. Because this is the testing of the first version of a product family, all the test cases, i.e., 9 test cases, are selected.

Step 4. Execute test cases and collect execution traces. If all the test cases are executed exhaustively on all the products that use the test cases, the total number of test executions is 20. To reduce redundant executions of the test cases, we use Algorithm 1. Because this is the first testing of the product family, all the products of the family are used as the input

of the algorithm, '*TP*'. First, all test cases for P1 are executed and their execution traces are collected as presented in Table 5. For example, t1 is executed on P1 in the sequence of VMachine, Currency, Beverages, DrinkOrder, Tea, RingTone and Display and is executed on P4 in the sequence of VMachine, Currency, Beverages, DrinkOrder, Tea and Display. After that, for each test case of the next product, P2, if P2 has a set of classes identical to a set of classes that the test case traversed on P1, then the test case is not executed on P2. In our case study, as Tables 4 and 5 show, t8 traversed the RingTone class on P1 and the RingTone class for P1 is identical to that for P2 (i.e., the checksum values are the same as 'k'). Thus, t8 is not executed on P2 because the test executions of t8 on P1 and P2 would be equivalent. In the same way, t9 is not executed on P2, too. By the same principle, for the testing of P3, the executions of t6, t7, t8 and t9 are avoided and for the testing of P4, t5 and t9 are avoided because their executions would be equivalent on previously tested products. Consequently, 8 test executions are reduced and so the total number of test executions is 12.

Table 4. The checksum matrix for SPL_{V0}.

Class \ Product	VMachine	Beverages	Coffee	Tea	Cappuccino	DrinkOrder	RingTone	Currency	Display
P1	a	e	-	g	-	i	k	l	n
P2	b	e	-	-	h	j	k	m	n
P3	c	e	f	-	h	j	k	m	n
P4	d	e	-	g	-	i	-	l	n

Table 5. Execution traces of test cases and their checksum values for SPL_{V0}.

Test Case	Used Products	Test Execution Trace	Checksum Values
t1	P1, P4	{P1}: [VMachine, Currency, Beverages, DrinkOrder, Tea, RingTone, Display] {P4}: [VMachine, Currency, Beverages, DrinkOrder, Tea, Display]	P1: aleigkn, P4: dleign
t2	P3	{P3}: [VMachine, Currency, Beverages, DrinkOrder, Coffee, RingTone, Display]	P3: cmejfkn
t3	P2, P3	{P2, P3}: [VMachine, Currency, Beverages, DrinkOrder, Cappuccino, RingTone, Display]	P2: bmejhkn, P3: cmejhkn
t4	P3	{P3}: [DrinkOrder]	P3: j
t5	P1, P4	{P1, P4}: [Currency, Display]	P1: ln, P4: ln
t6	P2, P3	{P2, P3}: [Currency, Display]	P2: mn, P3: mn
t7	P1, P2, P3	{P1, P2, P3}: [DrinkOrder, RingTone]	P1: ik, P2: jk, P3: jk
t8	P1, P2, P3	{P1, P2, P3}: [RingTone]	P1: k, P2: k, P3: k
t9	P1, P2, P3, P4	{P1, P2, P3, P4}: [Display]	P1: n, P2: n, P3: n, P4: n

4.2. Modification for Adding New Test Cases (Testing of SPL_{V1})

When a product line is modified, new features can be added to the product line and new test cases can be generated to verify these features. As the modified product line in Figure 4 shows, features for beverage size are added and the Size feature has an 'exclude' relationship with the Dollar feature. Thus, only when the Euro feature is selected, different choices for the beverage size are allowed. We marked the changed parts using a dotted line. Based on the evolving scenario, we modified the DrinkOrder class of the product line code base so that choosing features of different beverage sizes is allowed. Then, we generated new test cases for new features, as specified in Table 6. For test case generation, we used the method of Lity et al. [7]. Finally, we modified the VendingMachine product family by adding new features for beverage size, presented as bold type in Table 7. Only the feature configurations for P1 and P4 have been modified because P2 and P3 have the Dollar feature, which is mutually exclusive with the Size feature.

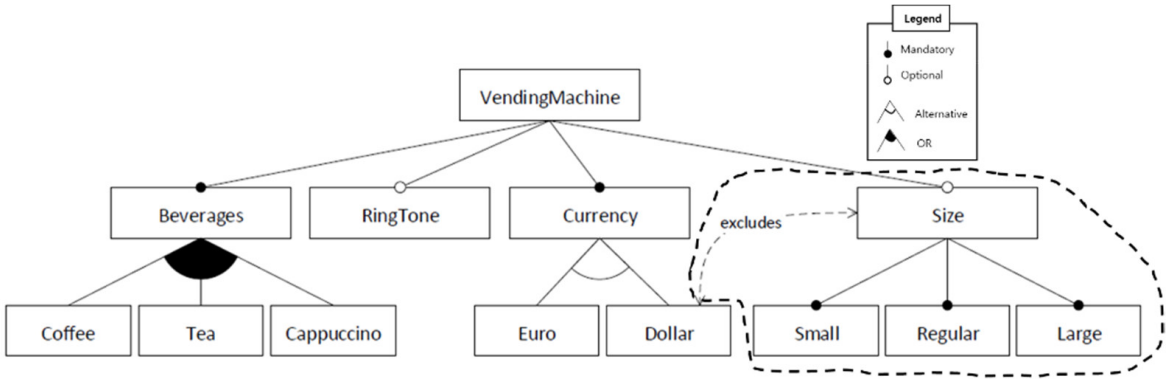

Figure 4. The feature model for SPL_{V1}.

Table 6. New test cases for SPL_{V1}.

Test Case	Input	Expected Output	Description
t10	1	"Small"	Checks whether a small size beverage is chosen.
t11	2	"Regular"	Checks whether a regular size beverage is chosen.
t12	3	"Large"	Checks whether a large size beverage is chosen.

Table 7. A product family SPL_{V1}. Features or classes changed in SPL_{V0} are presented as bold type.

Product (LOC)	Test Case	Feature Configuration	Class Set
P1 (194)	t1, t5, t7, t8, t9, t10, t11, t12	VendingMachine, Beverages, Tea, RingTone, Currency, Euro, **Size**, **Small**, **Regular**, **Large**	VMachine, **DrinkOrder**, Display, Beverages, Tea, RingTone, Currency
P2 (167)	t3, t6, t7, t8, t9	VendingMachine, Beverages, Cappuccino, RingTone, Currency, Dollar	VMachine, DrinkOrder, Display, Beverages, Cappuccino, RingTone, Currency
P3 (182)	t2, t3, t4, t6, t7, t8, t9	VendingMachine, Beverages, Coffee, Cappuccino, RingTone, Currency, Dollar	VMachine, DrinkOrder, Display, Beverages, Coffee, Cappuccino, RingTone, Currency
P4 (167)	t1, t5, t9, t10, t11, t12	VendingMachine, Beverages, Tea, Currency, Euro, **Size, Small, Regular, Large**	VMachine, **DrinkOrder**, Display, Beverages, Tea, Currency

Step 1. Obtain a product family. Changes of the DrinkOrder class are propagated to P1 and P4, and P2 and P3 remain the same as their previous versions.

Step 2. Build/update checksum matrix. This step updates the existing checksum matrix by re-computing the checksum of the modified class (i.e., the DrinkOrder class). Table 8 presents the updated checksum matrix of the VendingMachine product family. Because the evolving scenario affects only the DrinkOrder class, only the checksum values of the DrinkOrder class change and the other values remain the same. We presented the changed values in bold in Table 7.

Step 3. Select test cases to execute. All the new test cases, t10, t11 and t12, are selected to test new features relevant to beverage size. Additionally, because the DrinkOrder class changed according to the evolving scenario, the test cases that traversed the DrinkOrder class in the previous version are selected. In our case, t1, t2, t3, t4 and t7 are selected since as presented in Table 5, they traversed the DrinkOrder in the previous version.

Step 4. Execute test cases and collect execution traces. If the test cases selected in Step 3 (i.e., t1, t2, t3, t4, t7, t10, t11 and t12) are executed exhaustively, the total number of test executions is 15. However, these test executions include redundant test executions, and our algorithm can reduce them in the context of regression testing as well. Because P1 and P4 have been affected by the changes, only these products are targets for regression

testing and used as the input '*TP*' of Algorithm 1. The algorithm first executes the test cases applicable to P1 (i.e., t1, t7, t10, t11 and t12). Table 9 shows the test execution traces and their checksum values. Checksum values of test execution traces that contain a changed class (i.e., the DrinkOrder class for P1 and P4) change. We presented the values changed from the values in the previous version in bold. Then, when the test cases applicable to P4 (i.e., t1, t10, t11 and t12) are executed, the executions of t10, t11 and t12 are excluded because P4 has a set of classes identical to a set of classes that these test cases traversed on P1, which indicates that their test executions would be equivalent. For example, t10 traversed the DrinkOrder class and the Display class on P1 and those classes for P1 are identical to those for P4 (i.e., the checksum values are the same as 'on'). For this reason, the execution of t10 on P4 is redundant. Even though t2, t3 and t4 are selected in Step 3, they are not executed on any products because they have not been used for the changed products, P1 or P4. Consequently, 9 test executions are reduced and so the total number of test executions is 6.

Table 8. The checksum matrix for SPL_{V1}. Checksum values changed in SPL_{V0} are presented as bold type.

Class \ Product	VMachine	Beverages	Coffee	Tea	Cappuccino	DrinkOrder	RingTone	Currency	Display
P1	a	e	-	g	-	**o**	k	l	n
P2	b	e	-	-	h	j	k	m	n
P3	c	e	f	-	h	j	k	m	n
P4	d	e	-	g	-	**o**	-	l	n

Table 9. Execution traces of test cases and their checksum values for SPL_{V1}. Checksum values changed in SPL_{V0} are presented as bold type.

Test Case	Used Products	Test Execution Trace	Checksum Values
t1	P1, P4	{P1}: [VMachine, Currency, Beverages, DrinkOrder, Tea, RingTone, Display] {P4}: [VMachine, Currency, Beverages, DrinkOrder, Tea, Display]	**P1: aleogkn, P4: dleogn**
t2	P3	{P3}: [VMachine, Currency, Beverages, DrinkOrder, Coffee, RingTone, Display]	P3: cmejfkn
t3	P2, P3	{P2, P3}: [VMachine, Currency, Beverages, DrinkOrder, Cappuccino, RingTone, Display]	P2: bmejhkn, P3: cmejhkn
t4	P3	{P3}: [DrinkOrder]	P3: j
t5	P1, P4	{P1, P4}: [Currency, Display]	P1: ln, P4: ln
t6	P2, P3	{P2, P3}: [Currency, Display]	P2: mn, P3: mn
t7	P1, P2, P3	{P1, P2, P3}: [DrinkOrder, RingTone]	**P1: ok**, P2: jk, P3: jk
t8	P1, P2, P3	{P1, P2, P3}: [RingTone]	P1: k, P2: k, P3: k
t9	P1, P2, P3, P4	{P1, P2, P3, P4}: [Display]	P1: n, P2: n, P3: n, P4: n
t10	**P1, P4**	**{P1, P4}: [DrinkOrder, Display]**	**P1: on, P4: on**
t11	**P1, P4**	**{P1, P4}: [DrinkOrder, Display]**	**P1: on, P4: on**
t12	**P1, P4**	**{P1, P4}: [DrinkOrder, Display]**	**P1: on, P4: on**

4.3. Modification Affecting Existing Test Cases (Testing of SPL_{V2})

Some modifications in a product line can affect the existing test cases by causing them to traverse different classes depending on the products to be tested. As the modified product line in Figure 5 shows, the feature for small beverage size is deleted and the Milk feature is newly added. The Milk feature has the 'require' relationships with the 'Cappuccino' feature. So only a product that offers cappuccino has the Milk feature. According to this evolving scenario, we first added the Milk class to the product line code base and implemented its functions that when a cup of cappuccino is made, a cup of milk is supplied and when the amount of milk is not enough, the VendingMachine displays an error message. Next, to support the milk supply and stop serving the small size of beverages, we modified the DrinkOrder class. Due to this modification, the code coverages of the existing test cases are

changed. Then, as specified in Table 10, we generated new test cases for the Milk feature. Finally, as presented in Table 11, we modified the feature configurations for all the products of the VendingMachine product family by adding the Milk feature or deleting the Small feature for the relevant products.

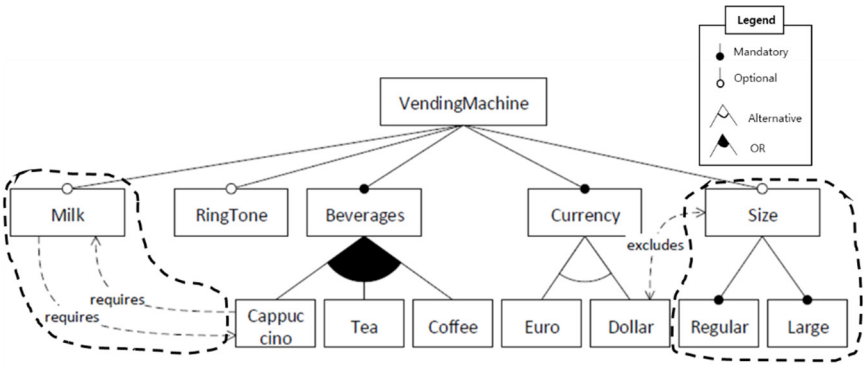

Figure 5. The feature model for SPL_{V2}.

Table 10. New test cases for SPL_{V2}.

Test Case	Input	Expected Output	Description
t13	0	ErrNotEnoughMilk	Checks whether the ErrNotEnoughMilk event is created when the amount of milk is not enough.

Table 11. A product family SPL_{V2}. Features or classes changed in SPL_{V1} are presented as bold type.

Product (LOC)	Test Case	Feature Configuration	Class Set
P1 (189)	t1, t5, t7, t8, t9, t10, t11, t12	VendingMachine, Beverages, Tea, RingTone, Currency, Euro, Size, Regular, Large	VMachine, **DrinkOrder**, Display, Beverages, Tea, RingTone, Currency
P2 (185)	t3, t6, t7, t8, t9, t13	VendingMachine, Beverages, Cappuccino, RingTone, Currency, Dollar, **Milk**	VMachine, **DrinkOrder**, Display, Beverages, Cappuccino, RingTone, Currency, **Milk**
P3 (200)	t2, t3, t4, t6, t7, t8, t9, t13	VendingMachine, Beverages, Coffee, Cappuccino, RingTone, Currency, Dollar, **Milk**	VMachine, **DrinkOrder**, Display, Beverages, Coffee, Cappuccino, RingTone, Currency, **Milk**
P4 (162)	t1, t5, t9, t10, t11, t12	VendingMachine, Beverages, Tea, Currency, Euro, Size, Regular, Large	VMachine, **DrinkOrder**, Display, Beverages, Tea, Currency

Step 1. Obtain a product family. Changes of the DrinkOrder class are propagated to all the products and addition of the Milk feature is propagated to P2 and P3.

Step 2. Build/update checksum matrix. This step updates again the previous version of the checksum matrix by computing the checksum value of the DrinkOrder class and the Milk class, as presented in Table 12. Thus, the checksum values of the DrinkOrder class change for all the products and the checksum values for the Milk class are added to columns for P2 and P3. The other values remain the same.

Step 3. Select test cases to execute. t13 is selected for retest because it is a new test case. Additionally, t1, t2, t3, t4, t7, t10, t11 and t12 are selected since they traversed the changed class (i.e., DrinkOrder class) in the previous version, as presented in Table 9.

Step 4. Execute test cases and collect execution traces. Because all the products have been affected by the changes, all the products of the family are targets for regression testing. The algorithm first executes the test cases applicable to P1 (i.e., t1, t7, t10, t11 and t12). Table 13 shows the test execution traces and their checksum values. Next, the test cases applicable to P2 (i.e., t3, t7 and t13) are executed. At this time, no test execution is avoided because

t3 and t13 were not previously executed on any products and t7 were executed on P1 but P2 does not have the identical set of classes that t7 traversed on P1. In the same way, for the test cases applicable to P3 (i.e., t2, t3, t4, t7 and t13), the algorithm does not execute t7 and t13 on P3, and for the test cases applicable to P4 (i.e., t1, t10, t11 and t12), the algorithm does not execute t10, t11 and t12 on P4. In summary, 5 test executions are reduced and so the total number of test executions is 12.

Table 12. The checksum matrix for SPL_{V2}. Checksum values changed in SPL_{V1} are presented as bold type.

Class \ Product	VMachine	Beverages	Coffee	Tea	Cappuccino	DrinkOrder	RingTone	Currency	Display	Milk
P1	a	e	-	g	-	**p**	k	l	n	-
P2	b	e	-	-	h	**q**	k	m	n	**r**
P3	c	e	f	-	h	**q**	k	m	n	**r**
P4	d	e	-	g	-	**p**	-	l	n	-

Table 13. Execution traces of test cases and their checksum values for SPL_{V2}. Checksum values changed in SPL_{V1} are presented as bold type.

Test Case	Used Products	Test Execution Trace	Checksum Values
t1	P1, P4	{P1}: [VMachine, Currency, Beverages, DrinkOrder, Tea, RingTone, Display] {P4}: [VMachine, Currency, Beverages, DrinkOrder, Tea, Display]	**P1: alepgkn, P4: dlepgn**
t2	P3	{P3}: [VMachine, Currency, Beverages, DrinkOrder, Coffee, RingTone, Display]	**P3: cmeqfkn**
t3	P2, P3	{P2, P3}: [VMachine, Currency, Beverages, DrinkOrder, Cappuccino, RingTone, Display]	**P2: bmeqhkn, P3: cmeqhkn**
t4	P3	{P3}: [DrinkOrder]	**P3: q**
t5	P1, P4	{P1, P4}: [Currency, Display]	P1: ln, P4: ln
t6	P2, P3	{P2, P3}: [Currency, Display]	P2: mn, P3: mn
t7	P1, P2, P3	{P1, P2, P3}: [DrinkOrder, RingTone]	**P1: pk, P2: qk, P3: qk**
t8	P1, P2, P3	{P1, P2, P3}: [RingTone]	P1: k, P2: k, P3: k
t9	P1, P2, P3, P4	{P1, P2, P3, P4}: [Display]	P1: n, P2: n, P3: n, P4: n
t10	P1, P4	{P1, P4}: [DrinkOrder, Display]	**P1: pn, P4: pn**
t11	P1, P4	{P1, P4}: [DrinkOrder, Display]	**P1: pn, P4: pn**
t12	P1, P4	{P1, P4}: [DrinkOrder, Display]	**P1: pn, P4: pn**
t13	**P2, P3**	**{P2, P3}: [DrinkOrder, Milk, Display]**	**P2: qrn, P3: qrn**

4.4. Threats to Validity

In the case study, the reliability of the target product line can be a threat to the validity of the case study. To mitigate this problem, we selected the vending machine product line, which has been well documented [17] and evaluated [7]. Moreover, we chose from the document evolving scenarios that can show various application cases of our method.

The overhead of our method more or less depends on the programming language. In the case of the Java language, even though multiple classes are implemented in a single file, separate class files are produced. In this case, checksum comparison between two classes is not costly. However, in the case of the C++ language, the overhead in parsing each class from a file occurs in order to conduct checksum comparison between two classes. This overhead can be reduced by using a code unit of a coarser level of granularity. However, the effect in reducing repetitions of test execution may decrease.

4.5. Result and Discussion

Scope of this work. In our previous paper [16], we proposed a method of reducing redundant test executions of a product line. However, it focused on constructing technical

theory for our method and it did not provide a practical guidance on how to apply our method. Moreover, the previous paper focused only on the context of regression testing even if with some adjustments it can be generalized to the initial version of a product line. For these reasons, as a follow-up work of our previous paper, this paper proposed a generalized process for applying our method to testing of the initial version of a product family as well as regression testing of its subsequent versions, an improved algorithm that avoids equivalent test executions, and demonstrated the generality under the proposed process. Scalable evaluation is not the scope of this work, and we will perform such extended evaluation in future work.

Result of the case study. In this case study, we tested three versions of the Vending-Machine SPL under the application process presented in Figure 2. For the first version, SPL_{V0}, and its two subsequent versions, SPL_{V1} and SPL_{V2}, overall, our method reduced test executions by 42.3% (={12 + 6 + 12}/{20 + 15 + 17}) compared to the exhaustive execution. This case study shows how our method works to reduce repetitions of equivalent test executions, and its result demonstrates the practical applicability of the proposed process for the method and the generality of our method.

Impacts of commonality and size of a product family. This effect can increase when a product family with a lot of commonality is used, because the more commonality a product family has, the more the products of a product family will reuse test cases and so the more our method will reduce (equivalent) test executions. Moreover, our method is more effective when it is applied to a product family that contains many products because, as test cases are reused for more products, our method reduces equivalent test executions more.

Cost reduction and fault detection effect. Our method can be used when the first version of a product family has been developed and whenever the product family has changed. Thus, it can reduce the significant amount of SPL testing cost during the testing phase and the maintenance phase of the software development lifecycle. Nonetheless, our method does not miss faults that would be detected by the exhaustive execution of test cases because it avoids only equivalent test executions that do not contribute to detecting faults.

Overhead of our method. To avoid equivalent test executions, our method builds the checksum matrix in Step 2 and determines the equivalence of test executions in Step 4. However, as shown in Sections 4.2 and 4.4, because these activities do not include any in-depth analysis of source code and test cases, the overhead incurred in avoiding test execution equivalence is small. To measure the overhead incurred in our method, we obtained the BerkeleyDB SPL from the SPL2go repository [22] and produced a product family that contains 10 products, which are implemented by, on average, 263 files and 40,078 lines. Then, we measured the time required for avoiding equivalent test executions. As a result, it took 2.64 s on Windows 7 with an Intel Core i7 CPU (3.40 GHz) and 12 GB RAM and this accounts only for 7.7% of the end-to-end time (i.e., Steps 1~4) for SPL testing.

Impacts of N-to-1 relationships between features and source code. Our method requires additional work in the case that multiple features are implemented in a single class. For example, when features A and B are implemented in a different part of class C, executions of the two test cases t1 and t2 for features A and B, respectively, produce the same checksum value because they commonly traverse class C. In this case, our method selects only one test case for testing of two features because the execution trace of t1 is the same as that of t2. To avoid this problem, class C should be separated into two different classes that implement features A and B, respectively. This practice is reasonable because it makes loose coupling between features.

5. Related Work

This section discusses related work regarding testing of a single SPL version (in Section 5.1), regression testing of its subsequent versions (in Section 5.2) and reduction of test redundancy at the level of test executions (in Section 5.3).

5.1. Studies on Testing of a Single SPL Version

In the product line testing community, two different interests have gained main attention: selection of products to test and actual testing of products [23].

The first one is the problem of selecting a representative set of products to test in order to verify a product line. To address this problem, combinatorial interaction testing techniques select a minimal set of products in which at least one product covers t-wise interaction, based on feature model [2–6,24–27], source code [25,26], dependency graph [27], etc. Their common insight is that most faults are caused by interactions among a fixed number of at most t features. Even though they do not verify all the products that can be produced from a product line, they are promising in maximizing the fault detection capability under limited resources. As another solution, search-based techniques randomly select an initial set of products and then gradually optimize the product set under a certain set of constraints [28]. To find the optimized set of products, they use an evolutionary algorithm that optimizes coverage [29–31], number of products [30,31], similarity [30,32], mutation score [30,33], complexity [29], etc. The other approaches include the greedy approach and the manual approach [28]. The techniques discussed above reduce test redundancy at the level of products. However, they still have test redundancy because they do not avoid the duplicate testing of common parts between products. To avoid this redundancy, these techniques should be used together with a technique like our method that reduces unnecessary test repetition by exploiting the commonality and variability of a product family.

The second one is the problem of how to check the correctness of the products that are produced from a product line. There have been extensive techniques to address this problem. Incremental delta-oriented techniques [7,8,10,34] incrementally construct test artifacts for every product by exploiting the commonality and variability between two consecutive products. Stepping from one product to the next product, they reuse test artifacts of the previous products for the next product. Search-based techniques [35–38] address a multi-objective optimization problem for testing a product line using a meta-heuristic algorithm. These techniques generate a test suite optimized with multi-objective functions under time constraints. Similarity-based techniques [32,38,39] increase feature interaction coverage by maximizing the diversity of products to test or test cases to run. They generate dissimilar test cases [38] or prioritize products [32,39] in the order in which the next product is dissimilar with the previous products. Other techniques include the coverage-based technique [40], the fault-history based technique [41], the risk-based technique [42], etc. These techniques discussed above reduce test redundancy at the level of test cases. However, they can have test redundancy at the level of test executions. When a product line test case is executed on the products that reuse it, its executions can be equivalent on these products. Such test executions should be avoided because they do not contribute to finding new faults.

The existing SPL testing techniques reduce test redundancy at the level of products and test cases, but not at the level of test executions. Our technique proposed in this paper reduces test redundancy at the level of test executions by avoiding equivalent executions of a given set of test cases. For this reason, it can be used with the existing techniques discussed above in a complementary way and can further save cost in addition to the cost saved by using them.

5.2. Studies on SPL Regression Testing

Techniques on SPL regression testing have been proposed for three research topics: selection, test prioritization and minimization of test cases.

Lity et al. [7] proposed a model-based test case selection technique for delta-oriented SPLs. They applied incremental model slicing to capture changes between the products of a product family and between different versions of the family. When changes are made to a product family, regression test suites are selected by applying retest coverage criteria to model slices, which consist of states and transitions affected by the change. We previously

proposed a code-based SPL regression test selection technique [43]. To reduce the overall testing efforts, this technique handles changes to the common parts of source code and changes to the variable parts of source code, separately. For the changes to the common parts of code, it minimizes unnecessary collections of test execution traces and unnecessary executions of test cases based on the commonality between the products of a product family. For the changes to the variable parts of code, it identifies the test cases that can never be affected by the changes and filters them out without any in-depth analysis of source code and test cases.

Al-Hajjaji et al. [39] and Henard et al. [32] proposed techniques which prioritize products of a product line based on similarity of their selected features. Their common insight is that two dissimilar products are likely to contain different sets of faults in a product line and so a diverse set of products should be tested first for early fault detection. For this reason, the products that have little similarity with the previously tested products have high priorities. Lachmann et al. [9] proposed an approach that computes regression deltas, and then ranks test cases in accordance with their capability to cover changes. Ensan et al. [44] proposed an approach which first assigns priorities to features and their relevant test cases according to the degree of the stakeholder's goal satisfaction. Wang et al. [45] and Arrieta et al. [46] proposed a test prioritization approach that uses search-based algorithms to address a multi-objective optimization problem for SPL test prioritization.

To minimize the size of test suites and the test execution cost, Baller et al. [47] uses a multi-objective technique and Wang et al. [48] uses a random-weighted generic algorithm.

When a set of test cases that are selected, prioritized and/or minimized using the techniques discussed above is executed on the products of a product family, their redundant executions can be reduced if these techniques are used in conjunction with our method. In addition, when time constraints exist due to a resource limit, our method allows regression test cases to be executed more under the time constraints, which makes a product line more reliable. For this reason, regression testing techniques can also be used with our method in a complementary way.

5.3. Studies on Reducing Test Redundancy at the Level of Test Executions

The existing techniques presented in Sections 5.1 and 5.2 reduce test redundancy only at the level of products or test cases. Thus, they still have test redundancy at the level of test executions. To address this problem, Kim et al. [13], Nguyen et al. [14] and Wong et al. [15] suggested variability-aware execution methods for testing of a configurable system. All of these methods execute common code only once, not executing it on each and every configuration. These methods share the same research direction with our method in that they reduce redundant test executions without decreasing the chance of detecting new faults. However, the methods do not address the execution of existing test cases and therefore cannot be applied to regression testing of a product line.

For testing of a product line system that our method targets, Stricker et al. [12] proposed a data flow-based technique called ScenTED-DF. ScenTED-DF avoids redundant testing of common parts of a product line using an annotated variable activity diagram. However, ScenTED-DF requires the intervention of human experts when modeling the annotated variable activity diagram [12]. In contrast, our technique is an automated one and requires only source code and test cases of a product line.

Li et al. [49] proposed a method of reusing test execution traces for an SPL by recording the name, version, inputs and outputs of modules that the executed test case traverses over a product. If another test case traverses a module with the same name, the same version and the same inputs, then the recorded output is used without re-executing the test case. However, this method does not consider that even when two modules of two different products have the same name, the same version or the same inputs, the two modules can still produce different outputs when implementations of the modules are not equivalent. For this reason, some part of source code may not be tested and so, due to the decreased

test coverage, faults that would be detected by the exhaustive test execution can be missed. In addition, it conducts only a small case study.

In summary, the methods discussed above reduce redundancy at the level of test execution. However, they either have limitations in fault detection and automation or have different application scopes.

6. Conclusions

In this paper, we showed that our method proposed in our previous work can be applied not just to SPL regression testing but to SPL testing in general. To that end, we first argued that our method is applicable for testing of both the first version of a software product line and its subsequent versions. Then, we proposed a practical process for applying the method to SPL testing and an algorithm that improves our previous algorithm in order to reduce unnecessary repetition of test execution and unnecessary collection of test execution traces. The case study result that we conducted based on the proposed process showed that our method reduced test executions by 42.3% (22/52) compared to those of the exhaustive execution approach.

Our method is easy to apply for most product lines because it requires only source code and does not require other artifacts such as requirements and architecture that are often hard to obtain and reducing repetitions of equivalent test executions through checksum comparison is fully automated. For this reason, with the use of our method for SPL testing can be conducted more efficiently.

For future work, we plan to extend our method to make it applicable at a finer granularity level. Moreover, we considered in this paper test redundancy only at the test execution level, but we plan to extend our method by applying product sampling techniques that reduce test redundancy at the product level. Finally, we plan to demonstrate the scalability of our method by applying it to larger size software product lines from the industry.

Author Contributions: Conceptualization, P.J. and S.K.; Funding acquisition, J.L.; Supervision, S.K.; Writing—review and editing, P.J., S.K. and J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by a grant from the National Research Foundation of Korea funded by the Korean government (MSIT) (NRF-2020R1F1A1071650).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pohl, K.; Böckle, G.; van Der Linden, F.J. *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed.; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2005.
2. Johansen, M.F.; Haugen, Ø.; Fleurey, F. An algorithm for generating t-wise covering arrays from large feature models. In Proceedings of the 16th International Software Product Line Conference, Salvador, Brazil, 2–7 September 2012; Volume 1, pp. 46–55.
3. Al-Hajjaji, M.; Krieter, S.; Thüm, T.; Lochau, M.; Saake, G. IncLing: Efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Not.* **2016**, *52*, 144–155. [CrossRef]
4. Hervieu, A.; Marijan, D.; Gotlieb, A.; Baudry, B. Practical minimization of pairwise-covering test configurations using constraint programming. *Inf. Softw. Technol.* **2016**, *71*, 129–146. [CrossRef]
5. Reuling, D.; Bürdek, J.; Rotärmel, S.; Lochau, M.; Kelter, U. Fault-based product-line testing: Effective sample generation based on feature-diagram mutation. In Proceedings of the 19th International Conference on Software Product Line, Nashville, TN, USA, 20–24 July 2015; pp. 131–140.
6. Marijan, D.; Gotlieb, A.; Sen, S.; Hervieu, A. Practical pairwise testing for software product lines. In Proceedings of the 17th International Software Product Line Conference, Tokyo, Japan, 26–30 August 2013; pp. 227–235.
7. Lity, S.; Nieke, M.; Thüm, T.; Schaefer, I. Retest test selection for product-line regression testing of variants and versions of variants. *J. Syst. Softw.* **2019**, *147*, 46–63. [CrossRef]
8. Lochau, M.; Lity, S.; Lachmann, R.; Schaefer, I.; Goltz, U. Delta-oriented model-based integration testing of large-scale systems. *J. Syst. Softw.* **2014**, *91*, 63–84. [CrossRef]

9. Lachmann, R.; Lity, S.; Lischke, S.; Beddig, S.; Schulze, S.; Schaefer, I. Delta-oriented test case prioritization for integration testing of software product lines. In Proceedings of the 19th International Conference on Software Product Line, Nashville, TN, USA, 20–24 July 2015; pp. 81–90.
10. Varshosaz, M.; Beohar, H.; Mousavi, M.R. Delta-oriented FSM-based testing. In *Formal Methods and Software Engineering, Proceedings of the International Conference on Formal Engineering Methods, Paris, France, 3–5 November 2015*; Springer: Cham, Switzerland, 2015; pp. 366–381.
11. Xu, Z.; Cohen, M.B.; Motycka, W.; Rothermel, G. Continuous test suite augmentation in software product lines. In Proceedings of the 17th International Software Product Line Conference, Tokyo, Japan, 26–30 August 2013; pp. 52–61.
12. Stricker, V.; Metzger, A.; Pohl, K. Avoiding redundant testing in application engineering. In *Software Product Lines: Going Beyond, Proceedings of the International Conference on Software Product Lines, Jeju Island, South Korea, 13–17 September 2010*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 226–240.
13. Kim, C.H.P.; Khurshid, S.; Batory, D. Shared execution for efficiently testing product lines. In Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering, Dallas, TX, USA, 27–30 November 2012; pp. 221–230.
14. Nguyen, H.V.; Kästner, C.; Nguyen, T.N. Exploring variability-aware execution for testing plugin-based web applications. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 907–918.
15. Wong, C.P.; Meinicke, J.; Lazarek, L.; Kästner, C. Faster variational execution with transparent bytecode transformation. *Proc. ACM Program. Lang.* **2018**, *2*, 1–30. [CrossRef]
16. Jung, P.; Kang, S.; Lee, J. Efficient Regression Testing of Software Product Lines by Reducing Redundant Test Executions. *Appl. Sci.* **2020**, *10*, 8686. [CrossRef]
17. Nahrendorf, S.; Lity, S.; Schaefer, I. Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines. In *TU Braunschweig-Institute of Software Engineering and Automotive Informatics*; Technical Report; Institute for Software Engineering and Automotive Informatics: Braunschweig, Germany, 2018.
18. Classen, A. *Modelling with FTS: A Collection of Illustrative Examples*; PReCISE Research Center, University of Namur: Namur, Belgium, 2010.
19. Thao, C. A Configuration Management System for Software Product Lines. Ph.D. Thesis, University of Wisconsin, Milwaukee, WI, USA, 2012.
20. ASM. Available online: https://asm.ow2.io/ (accessed on 23 March 2022).
21. BCEL. Available online: http://commons.apache.org/proper/commons-bcel/ (accessed on 23 March 2022).
22. SPL2go. Available online: http://spl2go.cs.ovgu.de/projects/ (accessed on 23 March 2022).
23. Do Carmo Machado, I.; McGregor, J.D.; Cavalcanti, Y.C.; De Almeida, E.S. On strategies for testing software product lines: A systematic literature review. *Inf. Softw. Technol.* **2014**, *56*, 1183–1199. [CrossRef]
24. Arcaini, P.; Gargantini, A.; Vavassori, P. Generating tests for detecting faults in feature models. In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015; pp. 1–10.
25. Kim, C.H.P.; Batory, D.S.; Khurshid, S. Reducing combinatorics in testing product lines. In Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, Porto de Galinhas, Brazil, 21–25 March 2011; pp. 57–68.
26. Tartler, R.; Lohmann, D.; Dietrich, C.; Egger, C.; Sincero, J. Configuration coverage in the analysis of large-scale system software. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems, Cascais, Portugal, 23 October 2011; pp. 1–5.
27. Shi, J.; Cohen, M.B.; Dwyer, M.B. Integration testing of software product lines using compositional symbolic execution. In *Fundamental Approaches to Software Engineering, Proceedings of the International Conference on Fundamental Approaches to Software Engineering, Tallinn, Estonia, 24 March–1 April 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 270–284.
28. Varshosaz, M.; Al-Hajjaji, M.; Thüm, T.; Runge, T.; Mousavi, M.R.; Schaefer, I. A classification of product sampling for software product lines. In Proceedings of the 22nd International Systems and Software Product Line Conference, Gothenburg, Sweden, 10–14 September 2018; Volume 1, pp. 1–13.
29. Ensan, F.; Bagheri, E.; Gašević, D. Evolutionary search-based test generation for software product line feature models. In *Advanced Information Systems Engineering, Proceedings of the International Conference on Advanced Information Systems Engineering, Gdansk, Poland, 25–29 June 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 613–628.
30. Ferreira, T.N.; Lima, J.A.P.; Strickler, A.; Kuk, J.N.; Vergilio, S.R.; Pozo, A. Hyper-heuristic based product selection for software product line testing. *IEEE Comput. Intell. Mag.* **2017**, *12*, 34–45. [CrossRef]
31. Henard, C.; Papadakis, M.; Perrouin, G.; Klein, J.; Traon, Y.L. Multi-objective test generation for software product lines. In Proceedings of the 17th International Software Product Line Conference, Tokyo, Japan, 26–30 August 2013; pp. 62–71.
32. Henard, C.; Papadakis, M.; Perrouin, G.; Klein, J.; Heymans, P.; Le Traon, Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Softw. Eng.* **2014**, *40*, 650–670. [CrossRef]
33. Gregg, S.P.; Albert, D.M.; Clements, P. Product Line Engineering on the Right Side of the V. In Proceedings of the 21st International Systems and Software Product Line Conference, Sevilla, Spain, 25–29 September 2017; Volume A, pp. 165–174.
34. Damiani, F.; Faitelson, D.; Gladisch, C.; Tyszberowicz, S. A novel model-based testing approach for software product lines. *Softw. Syst. Modeling* **2017**, *16*, 1223–1251. [CrossRef]

35. Markiegi, U.; Arrieta, A.; Sagardui, G.; Etxeberria, L. Search-based product line fault detection allocating test cases iteratively. In Proceedings of the 21st International Systems and Software Product Line Conference, Sevilla, Spain, 25–29 September 2017; Volume A, pp. 123–132.
36. Li, X.; Wong, W.E.; Gao, R.; Hu, L.; Hosono, S. Genetic algorithm-based test generation for software product line with the integration of fault localization techniques. *Empir. Softw. Eng.* **2018**, *23*, 1–51. [CrossRef]
37. Hierons, R.M.; Li, M.; Liu, X.; Parejo, J.A.; Segura, S.; Yao, X. Many-objective test suite generation for software product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2020**, *29*, 1–46. [CrossRef]
38. Devroey, X.; Perrouin, G.; Legay, A.; Schobbens, P.Y.; Heymans, P. Search-based similarity-driven behavioural SPL testing. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems, Salvador, Brazil, 27–29 January 2016; pp. 89–96.
39. Al-Hajjaji, M.; Thüm, T.; Lochau, M.; Meinicke, J.; Saake, G. Effective product-line testing using similarity-based product prioritization. *Softw. Syst. Modeling* **2019**, *18*, 499–521. [CrossRef]
40. Markiegi, U.; Arrieta, A.; Etxeberria, L.; Sagardui, G. Test case selection using structural coverage in software product lines for time-budget constrained scenarios. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; pp. 2362–2371.
41. Arrieta, A.; Segura, S.; Markiegi, U.; Sagardui, G.; Etxeberria, L. Spectrum-based fault localization in software product lines. *Inf. Softw. Technol.* **2018**, *100*, 18–31. [CrossRef]
42. Lachmann, R.; Beddig, S.; Lity, S.; Schulze, S.; Schaefer, I. Risk-based integration testing of software product lines. In Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems, Eindhoven, The Netherlands, 1–3 February 2017; pp. 52–59.
43. Jung, P.; Kang, S.; Lee, J. Automated code-based test selection for software product line regression testing. *J. Syst. Softw.* **2019**, *158*, 110419. [CrossRef]
44. Ensan, A.; Bagheri, E.; Asadi, M.; Gasevic, D.; Biletskiy, Y. Goal-oriented test case selection and prioritization for product line feature models. In Proceedings of the Information Technology: New Generations (ITNG), Las Vegas, NV, USA, 11–13 April 2011; pp. 291–298.
45. Wang, S.; Buchmann, D.; Ali, S.; Gotlieb, A.; Pradhan, D.; Liaaen, M. Multi-objective test prioritization in software product line testing: An industrial case study. In Proceedings of the 18th International Software Product Line Conference, Florence, Italy, 15–19 September 2014; Volume 1, pp. 32–41.
46. Arrieta, A.; Wang, S.; Sagardui, G.; Etxeberria, L. Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines. *J. Syst. Softw.* **2019**, *149*, 1–34. [CrossRef]
47. Baller, H.; Lity, S.; Lochau, M.; Schaefer, I. Multi-objective test suite optimization for incremental product family testing. In Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, OH, USA, 31 March–4 April 2014; pp. 303–312.
48. Wang, S.; Ali, S.; Gotlieb, A. Cost-effective test suite minimization in product lines using search techniques. *J. Syst. Softw.* **2015**, *103*, 370–391. [CrossRef]
49. Li, J.J.; Geppert, B.; Rößler, F.; Weiss, D.M. Reuse Execution Traces to Reduce Testing of Product Lines. In Proceedings of the 11th International Software Product Line Conference, Kyoto, Japan, 10–14 September 2007; Volume 2, pp. 65–72.