applied sciences

MDPI

Article

Efficient Regression Testing of Software Product Lines by Reducing Redundant Test Executions

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

[1] School of Computing, KAIST, Daejeon 34141, Korea; psjung@kaist.ac.kr (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

Received: 21 October 2020; Accepted: 2 December 2020; Published: 4 December 2020

Abstract: Regression testing for software product lines (SPLs) is challenging because it must ensure that all the products of a product family work correctly whenever changes are made. One approach to reducing the cost of regression testing is the regression test selection (RTS), which selects a subset of regression test cases. However, even when RTS is applied, SPL regression testing can still be expensive because, in the product line context, each test case can be executed on more than one product that reuses the test case, which would typically result in a large number of test executions. A promising direction is to eliminate redundant test executions of test cases. We propose a method that, given a test case, identifies a set of products, on which the test case will cover the same sequence of source code statements and produce the same testing results, and then excludes these products from products to apply the test case to. The evaluation results showed that when the full selection approach and the approach of repetitively applying an RTS method for a single software system are used for test selection, our method reduced, respectively, 59.3% and 40.0% of the numbers of test executions of the approaches.

Keywords: software product lines; regression testing; test redundancy; equivalent test execution

1. Introduction

Regression testing is "the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code" [1]. This is a very expensive testing process as it requires much time and resources. To reduce the cost of regression testing, there have been many studies on regression test selection (RTS) for single software systems [2], which try to reduce the number of test cases for retest by selecting only test cases that are relevant to the change so that they are executed on the target product.

Regression testing for software product lines (SPLs) is conducted in a similar way. As in the case of a single software system, test cases for retest are selected first and then executed on products of the product family. However, unlike regression testing of a single software system, test cases for SPL are reused for products of the product family [3] and so an individual test case can be executed on more than one product, resulting in a potentially large number of test executions when the set of selected test cases are applied. For this reason, even when RTS methods are applied, SPL regression testing can still be very expensive. In short, the cost of regression testing for SPL is affected by the two factors: the number of test cases selected for retest and the number of products to which each selected test case is applied.

The cost from the first factor can be reduced through the existing RTS methods for SPL such as [4–6]. These methods reduce redundancy at the level of test cases by selecting a subset of the existing test cases based on the commonality and the variability [7] of a product line. However, they do not deal

with redundancy at the level of test executions (i.e., the second factor). To deal with the second factor, several SPL testing methods [8–12] have been proposed, which tried to reduce redundancy at the level of test executions by avoiding the unnecessary test executions that do not contribute to finding faults. However, the work of [9] misses faults that would be detected by the exhaustive execution of test cases and the work of [11] requires human intervention. The works of [8,10,12] handle configurable systems that can be executed by themselves without binding of source code and test cases. Moreover, none of these methods addressing the second factor focused on the problem of regression testing.

This paper proposes an automated method for effectively reducing redundant test executions for SPL regression testing. The key insight in our method is to avoid repeating equivalent test executions that cover exactly the same sequence of source code and produce the same testing result over two or more products of a product family. Our method selects a set of products on which each test case selected for retest can be executed with no repetition of equivalent test executions. To identify equivalence of test executions, the execution traces of test cases and the checksum values of source code are used. While reducing the cost of SPL regression testing, our method does not decrease the fault detection effect because it reduces only equivalent test executions. Moreover, because our method does not require any in-depth analysis of source code or test cases, the overhead incurred in identifying test execution equivalence and selecting products is small.

We conducted an experimental evaluation of our method using six SPL systems. The results of our evaluation show that for the test cases selected from, respectively, the full selection and an approach of repetitively applying Ekstazi [13], on average, our method reduces 59.3% and 40.0% of the number of test executions from the exhaustive execution. The overhead was small (less than 10% of the time required for all the phases of regression testing) and there was no loss in fault detection.

This paper contains the following contributions:

- We proposed a method to be used in conjunction with an existing RTS method and reduce its testing cost. The evaluation of our method using six open source product lines shows that our method reduces up to 82.6% of the regression testing cost of existing RTS methods.
- Our method is a fully automated method that requires only source code. Its application scope is wide because our method requires only source code and so it can be applied even to product lines where requirements' specification and architecture are not available or partially eroded.
- Our method reduces not only the redundancy at the level of test cases but also the redundancy at the level of test executions. A method that achieves the latter effect is a novel attempt in the context of SPL.

The rest of the paper is organized as follows. In Section 2, we introduce software product line development. Section 3 shows a motivating example of our approach and Section 4 presents our method to reduce redundant test executions. In Section 5, we evaluate our method. In Section 6, we discuss related work on reducing redundancy in SPL regression testing. In Section 7, we conclude the paper.

2. Background

This section introduces the concept of software product line development and defines related terms necessary to present our method.

2.1. Software Product Lines

Definition 1. (product family) *A product family is a set of products that have a significant amount of commonality*.

Definition 2. (platform artifact) *A platform artifact of a product family is an artifact (e.g., source code, test cases, etc.) that consists of the common parts of the product family and the abstractions of their variable parts*.

Definition 3. (product line) *A product line is a product family that has a set of platform artifacts for its planned products.*

A product line with many features can potentially produce a huge number of products. However, in practice, only a subset of those products is actually planned for production as a product family. So, our method focuses on testing of the products planned as a product family because retesting all the (planned and unplanned) products whenever a change is made would be costly and unnecessary. When unplanned products are added as planned products to a product family, our method to be developed in this paper still can handle such a case by just obtaining information of source code and test cases for the added products.

Definition 4. (product line code base) *Product line code base is the entire set of code fragments that is used to produce products of a product line.*

A product line has its code base partitioned into fragments and certain combinations (or integrations) of them constitute products [3]. Given a product line code base $CB = \{cf_1, cf_2, \ldots, cf_n\}$ where $cf_1, cf_2, \ldots, cf_n$ are code fragments, each product of a product family is produced by combining a subset of *CB*.

Figure 1 presents an SPL project: (a) product line code base, (b) a product family that has three products, P_1, P_2 and P_3 and (c) test cases, t1, t2, ... tn. t1 and t2 are common to all the products and t3 is common to P_2 and P_3. The products of the family are produced by copying code fragments for each product from the product line code base. For example, P_1 is produced by copying the code fragments of @Feature 1 and @Feature 2 and, P_2 and P_3 are produced by copying the code fragments of @Feature 1 and @Feature 3. These products have common code fragments between them. P_1, P_2 and P_3 share the code fragments of @Feature 1, and P_2 and P_3 share the code fragments of @Feature 3. The code fragments of @Feature 2 are unique to P1. In practice, this approach has been used by some major companies that adopted product line development [14,15] and also by well-known existing SPL development tools such as FeatureIDE [16], Pure::variants [17] and CIDE [18].

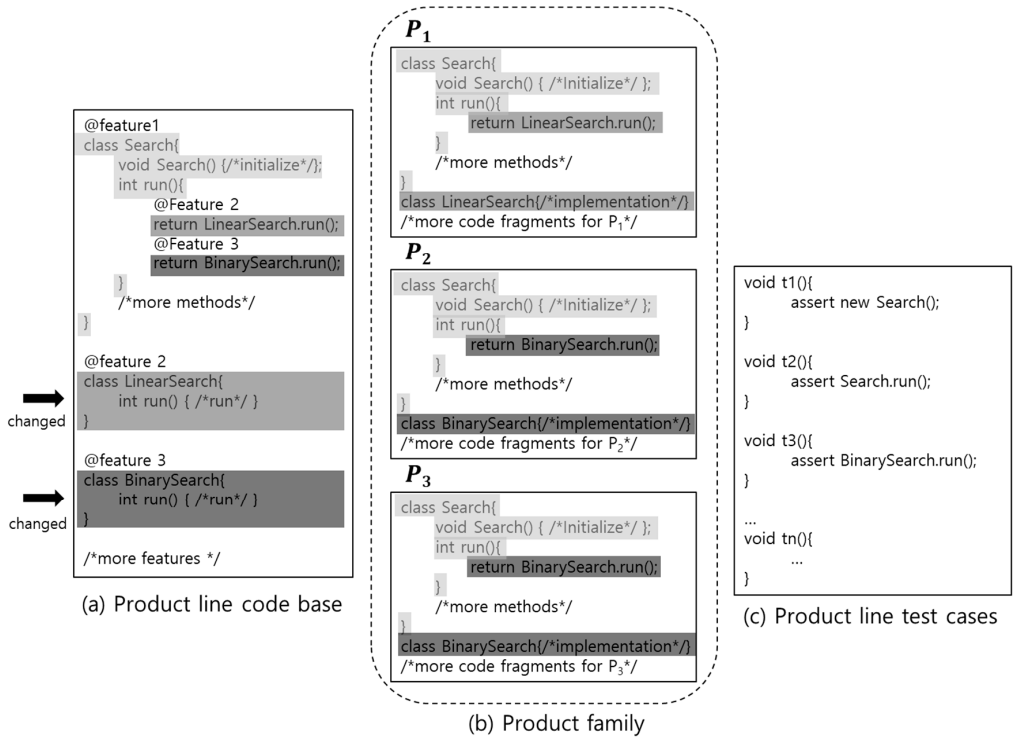

Figure 1. Examples of product line code base, product line test cases and product family.

2.2. Changes to Software Product Lines

In a product line, changes can be made not only to the product line code base but also to the source code of products produced from the product line code base. However, the latter case should lead to the change propagation to the product line code base to make the product family consistent with the changed version of the product line [19]. Consequently, both cases make changes to the product line code base.

When the code base of a product line is changed, the change is propagated to products of a product family by reproducing them from the changed code base, which makes products of the family up to date. For example, in Figure 1, if the code fragment of @feature 3 is changed, then the change is propagated to P_2 and P_3 by reproducing them. In this case, regression testing for the changed product family should be performed to check if new errors have been introduced into the previously tested product family. Our method proposed in this paper addresses the challenge of reducing redundancy of test executions when the code base of a product line has been changed.

3. A Motivating Example

In this section, we describe our key insight using the example in Figure 1. Let us suppose that by some changes to features 2 and 3 in the product line code base, the modified product family, P_1', P_2' and P_3' has been produced and the test cases t1, t2 and t3 were selected for regression testing of the modified product family. In this case, the selected test cases should be rerun on the product family to check whether the current revision still works well. If the selected test cases are executed exhaustively (i.e., t1 and t2 are run on all the products and t3 is run on P_2' and P_3'), the total number of test executions is 8.

A naïve approach that reduces the number of test executions is to run all the test cases selected for retest on only one product of a product family. For example, if t1, t2 and t3 are run only on P_2' or P_3', then five test executions are reduced and so the total number of test executions is three. However, this approach can miss faults that would be detected by the exhaustive execution of test cases because a test case can be designed to have variability that allows testing of different parts depending on the product to be tested or because, by modifications of a product family, an existing test case may provide different code coverage for the modified products of the product family. In the case of our example, when t2 is applied to P_1', it covers the LinearSearch class and when t2 is applied to P_2' and P_3', it covers the BinarySearch class. In this case, if t2 is applied only to P_1', the faults contained in the BinarySearch are not exposed and if t2 is applied only to P_2' or P_3', the faults contained in the LinearSearch class are not exposed.

To overcome this problem, the execution trace of a test case can be used. When t1 is applied to P_1', P_2' and P_3', it covers the same sequence of classes (in this case, Search class only) and the code of the class is identical for the products. So, it suffices to apply t1 to just one of P_1', P_2' and P_3'. Likewise, it suffices to apply t2 to P_1' and one of P_2' and P_3'. Because t3 is not executable on P_1', it suffices to apply t3 to just one of P_2' and P_3'. Therefore, four redundant test executions can be eliminated and the reduced number of test executions is four. This approach is more efficient than the exhaustive execution and will not miss faults that the exhaustive execution detects. Although a simple example was shown here to introduce our approach clearly, the problem exhibited through this example arises very frequently in the testing of product families.

To develop our approach, two problems should be handled. The first one is how to identify redundancy of test executions. The second one is how to select a set of products on which each test case selected for retest can be executed with no loss in fault detection. In Section 4, we explain our method that solves these two problems.

4. Test Execution Reduction Method for Efficient SPL Regression Testing

In this section, we present our method together with definitions, an assumption and a lemma necessary to present our method.

4.1. Definitions, Assumption and Lemma

Test execution is an application of a test case to the system under test (SUT). If *tc* is a test case and *p* is an SUT, then we will write *TE*(*p*,*tc*) to denote the test execution of *tc* applied to *p*.

Test executions can become redundant for various reasons such as that they are unnecessarily repeated executions of a test case; that they are executions of duplicated test cases; and that they test already covered code, not contributing to test coverage increase. Redundant test executions are useless in finding faults of a software program and just increase the testing cost because they do not detect any faults other than the faults already detected by the previous test executions. Redundant test execution can be defined as follows:

Definition 5. (Redundant test execution) *For any test execution e in a set of test executions E, let Faults(E) be the set of the faults that are detected by E. If Faults(E − {e}) = Faults(E), then e is said to be a redundant test execution.*

In Figure 1, let us suppose that a fault is located in the constructor of the Search class and t1 detects the fault. In this case, when t1 is applied to P_1′, P_2′ and P_3′ sequentially, the test execution of t1 applied to P_1′ is not redundant because t1 detects the fault. However, the test executions of t1 applied to P_2′ and P_3′ are redundant because they do not detect any fault other than the fault already detected when t1 is applied to P_1′. The test executions of t2 and t3 applied to P_1′, P_2′ and P_3′ are redundant, too, because they do not test the constructor of the Search class and so do not detect the fault.

In this paper, we consider only the case of unnecessarily repeated executions of test cases selected for retest in the context of an SPL. To reduce such redundant test executions, our method utilizes the notion of equivalence of test executions. To define it precisely, we use the notion of code unit.

Definition 6. (Code unit) *A code unit is a section of code written in a programming language that can be executed.*

For example, in the case of object-oriented programming languages, code units are classes, methods and statements and in the case of procedural programming languages, code units are files, functions and statements.

Now the equivalence of test executions is defined as follows.

Definition 7. (Equivalence of test executions) *Two test executions are equivalent if they cover exactly the same sequence of code units and produce the same testing result. Formally, when P_1 and P_2 are products, tc_1 and tc_2 are test cases, the coverage of a test execution e is denoted as Cov(e) and the test result of a text execution e is denoted as Res(e), if $Cov(TE(P_1,tc_1)) = Cov(TE(P_2,tc_2))$ and $Res(TE(P_1,tc_1)) = Res(TE(P_2,tc_2))$ then we say that the two test executions $TE(P_1,tc_1)$ and $TE(P_2,tc_2)$ are equivalent and write $TE(P_1,tc_1) \equiv TE(P_2,tc_2)$.*

If one of two equivalent test executions detects a fault, the other will also detect the fault and if one does not detect a fault, the other will not detect the fault either.

In Figure 1, the test executions of t1 applied to P_1′ and P_2′, respectively, are equivalent because they cover exactly the same sequence of code statements of the Search constructor and produce the same testing result. However, the test executions of t2 applied to P_1′ and P_2′, respectively, are not equivalent because the test execution of t1 applied to P_1′ covers the code of the function for the LinearSearch class but the test execution of t1 applied to P_2′ covers the code of the function for the BinarySearch class.

A testing result can be affected by various factors such as operating system, network connection, hardware settings, etc. For example, if memory available for dynamic allocation is changed, then a test case that passed on a program can fail on the same program. For another example, as a case of nondeterministic test execution, if a test case tests a random function, it can produce different results in different runs. Considering all influencing factors would make regression testing complicated. To ensure that a testing result is affected only from changes of source code, we use the following assumption that has been adopted in many regression testing studies, e.g., [6,20,21].

> Controlled Regression Testing Assumption. *When the modified program P′ is tested with a test case t, all factors that might influence the testing result of P′ except for the code in P′ are constant with respect to their states when the current program P was tested with t.*

In general, the equivalence of test executions does not imply the equality of test execution traces. However, if this assumption holds, then the execution of a test case is deterministic, i.e., the test coverage and the testing result are not affected by any factors except source code. Therefore, the equivalence of test executions implies the equality of test execution traces. More formally, for products P_1 and P_2 and a test case *tc*, when two sequences of code units that *tc* traverses on P_1 and P_2 are the same, that is, $TR(TE(P_1,tc))) = [x_1, x_2, \ldots, x_n]$ and $TR(TE(P_2,tc)) = [y_1, y_2, \ldots, y_m]$ and $[x_1, x_2, \ldots, x_n] = [y_1, y_2, \ldots, y_m]$, i.e., $x_i = yi$, $1 \leq i \leq n$, and $m = n$, then, under this assumption, $Cov(TE(P_1,tc)) = Cov(TE(P_2,tc))$ and $Res(TE(P_1,tc)) = Res(TE(P_2,tc))$ because $TE(P_1,tc)$ and $TE(P_2,tc)$ are deterministic and there is no difference between the source code that is executed by *tc* on P_1 and P_2.

Therefore, under the Controlled Regression Testing assumption, the following lemma holds:

Lemma 1. *For a test case tc and products P_1 and P_2 to which tc is commonly applied, let the execution trace of a test case tc applied toP_1be $[x_1, x_2, \ldots, x_n]$, i.e.,$TR(TE(P_1,tc)) = [x_1, x_2, \ldots, x_n]$. Then, if, for $1 \leq i \leq n$, P_2 contains x_i, then theexecution of tc applied to $P_2$is equivalent to the executionof tc applied to P_1, i.e., $TE(P_2,tc) \equiv TE(P_1,tc)$.*

Proof. Let P_1 be the set of code units $\{w_1, w_2, \ldots, w_k, y_1, y_2, \ldots, y_l\}$ and P_2 be the set of code units $\{w_1, w_2, \ldots, w_{k,}\, z1, \ldots, z_m\}$. So the code units that are common to P_1 and P_2 are $\{w_1, w_2, \ldots, w_k\}$. Furthermore, let the result of executing the test case tc against P_1 be $[x_1, x_2, \ldots, x_n]$, i.e., $TR(TE(P_1,tc))) = [x_1, x_2, \ldots, x_n]$ and $\{x_1, x_2, \ldots, x_n\} \subseteq P_1$. Now suppose that $\{x_1, x_2, \ldots, x_n\} \subseteq P_2$. Then under the Controlled Regression Testing assumption, $Cov(TE(P_2,tc)) = Cov(TE(P_1,tc))$ and $Res(TE(P_2,tc)) = Res(TE(P_1,tc))$ hold because by the assumption, the execution of a test case is deterministic and so there would be no difference between the source code of code units that *tc* traverses on P_1 and P_2. Therefore, $TE(P_2,tc) \equiv TE(P_1,tc)$. □

4.2. Overview of our Method

A typical RTS method is conducted in three phases [13]: the Analysis (*A*) phase for identifying the changes and selecting test cases for retest, the Execution (*E*) phase for executing the selected test cases and the Collection (*C*) phase for collecting information (e.g., test execution traces) from the current version to enable analysis for the next version. Our method includes all these phases and performs, additionally, the Family Checksum (*FC*) phase for obtaining checksum values of source code for the product family.

Figure 2 is an overview of our method. The inputs are (1) the source code of the original product family, (2) the test cases of the original product family and (3) the source code of the modified product family. Using these inputs, our method identifies and eliminates redundant executions and collects test execution traces. We presented the *E* phase and *C* phase as a single step because test execution traces can be obtained by executing the test cases (i.e., the *C* phase is always performed together with the *E* phase).

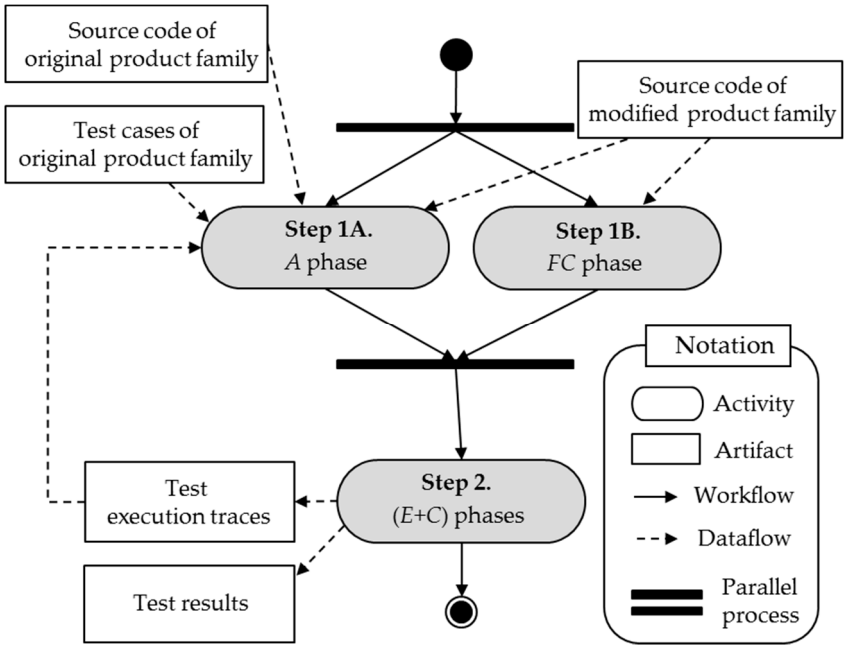

Figure 2. Overview of our method.

Our method requires the same set of inputs and produces the same output as those of the typical code-based RTS methods [13,22,23]. However, our method performs additional actions in the *FC* and (*E* + *C*) phases to identify equivalent test execution. Figure 3 shows the detailed process of our method. In the following subsections, we explain each step of our method in detail.

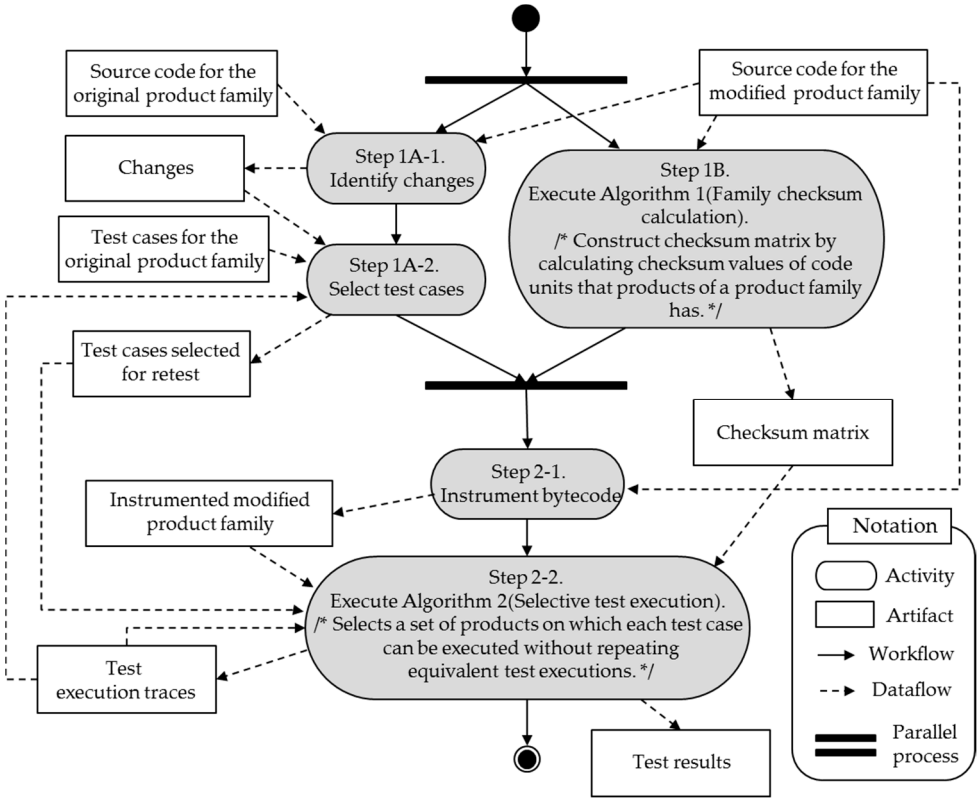

Figure 3. The detailed process of our method.

4.3. Step 1A. Identify Changes and Select Test Cases

For Step 1A, our method uses an existing code coverage-based RTS method that outputs test cases at the code level. However, so far, no code coverage-based RTS method for SPL has been proposed, as exposed in the recent survey papers listed in a tertiary study [24]. To address this problem, we can use an existing RTS method for a single software system such as the Ekstazi [13] and the SPIRITuS [23] by repeating it to each product of a product family. Even though this approach does not provide an efficient solution, it helps to assess the effectiveness of our method. This approach identifies changes using the original source code and the modified source code (Step 1A-1) and then select test cases using these changes (Step 1A-2). The output of this step is a set of test cases selected for retest and is used for Step 2-2.

4.4. Step 1B. Family Checksum Calculation

This step constructs the key artifact for reducing test execution. According to Lemma 1, two test executions are equivalent if every pair of code units they traverse are identical. To check whether two code units are identical, our method compares their MD5 checksums [25]. The MD5 checksum of a code unit is computed by applying a hashing function to statements of the code unit. So, if one statement of two code units is different, their checksum values are different. Furthermore, if the checksums of two code units are the same, then they are identical. This step builds a checksum matrix that contains the checksum values of every code unit of the modified family. Using the checksum matrix, our method determines whether two code units are identical and furthermore, whether two test executions are equivalent (cf. Section 4.6). As shown in the Family Checksum function of Algorithm 1, for each product of the modified product family, it extracts all the code units from the product (lines 3, 4). Then, it builds a checksum matrix by assigning the checksum value of each code unit to the corresponding matrix index (lines 5–7). The checksum matrix is used as an input of Step 2-2.

Algorithm 1 Family checksum

```
Input: MPF: modified product family
Output: chksumMatrix: checksum matrix
Use: extractCUs(p): input a product p and return a set of code units in p
Use: checksum(c): input a code unit c and return the checksum value of c
1. function FamilyChecksum(MPF)
2.     CUs ← ∅
3.     foreach p ∈ MPF do:
4.         CUs ← extractCUs(p)
5.         foreach c ∈ CUs do:
6.             chksumMatrix[p][c] ← checksum(c)
7.         endfor
8.     endfor
9.     return chksumMatrix
10. end function
```

Table 1 shows an example of a checksum matrix for a modified product family consisting of P_1', P_2' and P_3'. In this example, we use a class as a code unit. P_1 has Classes A and B; P_2' has Classes A and C; P_3' has Classes A, B and C. To simplify the example, we presented a 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 the identical source code. In Table 1, Class A is common to all the products of the product family and its source code is identical over all the products. Class B is used for P_1' and P_3' but not for P_2' and its source code for P_1' is identical to that for P_3', too. Class C is used for P_2' and P_3' but not for P_1' and its source code for P_2' is not identical to that for P_3'.

Table 1. Example of checksum matrix.

	Class A	Class B	Class C
P_1'	a	b	-
P_2'	a	-	c
P_3'	a	b	d

4.5. Step 2-1. Instrument Bytecode

In this step, the bytecode that collects the execution traces to products of the modified product family is instrumented. For each product of the family, our method instruments the bytecode to the start of a constructor and the start of a static initializer using the BCEL bytecode manipulation framework [26]. Test execution traces are obtained when the test cases are executed on the instrumented products in Step 2-2.

4.6. Step 2-2. Selective Test Execution

This step applies the test cases selected in Step 1A to the instrumented products created in Step 2-1, collecting their execution traces. To reduce redundant test executions and reduce redundancy in the test execution traces, our method selects a set of products on which each test case can be executed with no repetition of equivalent test executions, based on Lemma 1. As shown in the SelectiveTestExecution function of Algorithm 2, for each test case (t) that is selected from Step 1A, the function first obtains an execution trace (tr) by applying t to a certain product (p) out of the products (P) that reuse t (lines 2–5). Next, it obtains a list (*chksumListA*) of checksum values of the code units in tr (line 6). Finally, if another product (q) in P has code units whose list of checksum values is the same as *chksumListA*, then the algorithm excludes q from the set of products to which t would be applied (line 8–15) because, by Lemma 1, executions of t applied to p and q would be equivalent.

Algorithm 2 Selective Test Execution

Input: *IMPF*: instrumented modified product family
Input: *TCs*: test cases selected from the A phase
Input: *chksumMatrix:* checksum matrix for a product family
Use: execute(t, p): input a test case t and an instrumented product p, and return the execution trace of t for p

```
1. function SelectiveTestExecution(IMPF, TCs, chksumMatrix)
2.     foreach t ∈ TCs do:
3.         P ← products of IMPF for which t is reused
4.         foreach p ∈ P do:
5.             tr ← apply(t, p)
6.             chksumList ← chksumMatrix.lookup(p, tr.CodeUnits)
7.             Q ← P − {p}
8.             foreach q ∈ Q do:
9.                 chksumListB ← chksumMatrix.lookup(q, tr.CodeUnits)
10.                if chksumListA equals chksumListB then:
11.                    P ← P − {q}
12.                endif
13.            endfor
14.        endfor
15.    endfor
16. end function
```

Table 2 shows examples of selective execution for the three test cases listed in the first column. The second column presents products that use the test cases. The third and fourth columns present the products to which the test case is applied and the test execution traces for the products, respectively.

The checksum matrix presented in Table 1 is used in this example. As shown in Table 2, t1 is reused for P_1', P_2', and P_3'. So, first, t1 is applied to certain one (and exactly one) product out of the products and its execution trace is obtained. We assume that P_1' was selected for the first test execution and t1 traversed only Class A on P_1'. Next, the algorithm obtains the checksum values of Class A for P_1' and the other products that reused t1. As shown in Table 1, their checksum values are the same, i.e., "a". In this case, the algorithm does not apply t1 to P_2' and P_3' because, by Lemma 1, executions of t1 applied to P_1', P_2' and P_3' would be equivalent. t2 is reused for P_2' and P_3'. We assume that P_2' was selected for the first test execution and t2 traversed only Class C on P_2'. In this case, t2 is applied to P_3' too because Class C of P_3' is not identical to Class C of P_2', as shown in Table 1. t3 is used for P_1' only. In this case, t3 must be applied to P_1' because there are no equivalent test executions.

Table 2. Example of selective test execution.

Test Case	Target Products	Selected Products	Test Execution Traces Obtained from the Selected Products
t1	P_1', P_2', P_3'	P_1'	{P_1'}:[Class A]
t2	P_2', P_3'	P_2', P_3'	{P_2', P_3'}:[Class C]
t3	P_1'	P_1'	{P_1'}:[Class A, Class B]

Units of code changes, code units in test execution traces and code units in the checksum matrix can have various granularities such as file, class, method, statement, etc. However, if their granularities are different, test cases affected by a change cannot be determined directly from the test execution traces and the equivalence of test executions cannot be determined directly from the checksum matrix. Therefore, we need to align them at the same granularity level. Using a coarser granularity leads to the low test execution reduction effect, but it makes the *A* phase and the *C* phase faster. On the other hand, using a finer granularity leads to the high test execution reduction effect, but it makes the *A* phase and the *C* phase slower.

5. Evaluation

We developed a prototype tool that implements our method, which called Checksum-based Product Line Test Execution (CPLTE). Using the CPLTE tool, we evaluate our method using six subject product lines. The main objective of our evaluation is to show that, in the context of SPL, CPLTE is effective in reducing the total number of test executions with no loss in fault detection compared to the exhaustive execution. For this objective, we answer the following research questions:

- RQ1. How effective is CPLTE in reducing the number of test executions?
- RQ2. Does the reduction of test executions resulting from CPLTE affect the fault detection effect?
- RQ3. How effective is CPLTE in reducing the cost of SPL regression testing?

In Section 5.1, we describe our subject product lines (in Section 5.1.1), test selection strategy to be used for the *A* phase of RTS (in Section 5.1.2) and a base line to be compared for assessment of CPLTE (in Section 5.1.3). Section 5.2 shows our experimental process. Section 5.3 presents the experimental results and answers the research questions and Section 5.4 discusses the threats that can affect the validity of the results.

5.1. The Experimental Setup

5.1.1. Subject Product Lines, Tests and Faults

Selection of subject product lines. SPL2go [27] is an open-source product line repository that has been recently used in SPL research [28–30]. It has 40 SPL products (accessed December 2019). However, most of the projects are rather small (i.e., less than 3000 lines of source code). Excluding such small

products and some projects that did not work due to errors, we selected six SPL projects, which are all implemented in Java. Table 3 lists the subject product lines. In our experiment, we use a java class file as a code unit because all of our subject product lines are Java systems and in Java, a class file can be used to compute checksum values without a parse of code. Because the test coverage of a product line code base cannot be directly measured, we measured the test coverage of each product of a product family and averaged them. For TankWar, their product families have been defined by their developers, but for the others, product families have not been defined and so we created product families with different numbers of products by randomly selecting their features. We confirmed that all the created product families had widely different sets of features. Invalid products that cannot be executed at the level of source code were excluded from the experimental objects.

Table 3. Subject product lines used in the experiment.

Subject SPL	*SLOC	*$\|P_{SPL}\|$	*Com.	*CC.	*$\|T_{SPL}\|$ (*Cov.)
MobileMedia	3196	4	44.2%	57.7%	913 (85.8%)
TankWar	4845	5	62.9%	26.4%	735 (71.5%)
Prevayler	5109	3	72.5%	93.1%	1157 (67.8%)
		5	72.5%	91.6%	2144 (67.9%)
MobileRSSReader	16,664	3	93.1%	86.3%	2764 (66.5%)
		5	93.1%	89.2%	4258 (66.3%)
Lampiro	30,158	4	98.8%	72.4%	5265 (48.1%)
		6	98.6%	72.4%	7492 (49.3%)
BerkeleyDB	44,994	3	69.3%	58.9%	9027 (57.2%)
		5	69.3%	53.9%	15,161 (56.0%)
		7	69.3%	56.0%	20,858 (59.3%)
		10	69.3%	56.8%	24,952 (59.3%)

*SLOC: Source line of code counted by LocMetrics [31]. *$|P_{SPL}|$: the number of products. *Com.: the percentage of the source code common to all products. *CC.: the percentage of the code equivalent units. *$|T_{SPL}|$: the number of test cases. *Cov.: the percentage of the methods covered by test cases.

Test case generation. Our subject product lines had no test cases. For this reason, we generated product line test cases for them using the EvoSuite [32] (with default settings), which is a search-based automatic test case generation tool. To use EvoSuite, the source code must be compiled. However, because the code bases of our subject product lines could not be compiled before products are instantiated, we could not directly generate the product line test cases from the product line code bases. Moreover, according to the principle of SPL development [3], product line test cases should be reused for products that share common parts of the source code. To address these two concerns, first we produced, for each subject, the products of the product family from its code base and generated a set of test cases for each product using EvoSuite. Then for each product in this set of test cases we also included test cases generated for the other products if they are applicable to that product. In this process, we removed duplicate test cases because they unnecessarily increase test redundancy, creating a bias on experimental results regarding the number of redundant test executions. Through these steps, we obtained product line test cases that include both test cases common to two or more products and test cases unique to a single product, with the result that test cases have been reused for the products that share the same code units.

Fault injection. For each subject product line, we created 50 mutants by injecting mutation faults generated by the PIT [33] mutation analysis tool to the code base of the product line. The mutations include addition/deletion of classes, methods and statements, replacing arithmetic/logical operators, negating conditional decisions and so on. The mutated code bases produced the product families described in Table 3. Therefore, for each subject product family, 50 faulty versions have been produced. The mutants created by the PIT tool can include mutants that are equivalent to other mutants. Because

automatically detecting all equivalent mutants is impossible [34], we manually inspected the equivalent mutants based on the work of Grün et al. [35] and then excluded them. The mutants created in this way were used as the input, i.e., "Source code of modified product family" shown at the top part of Figure 2.

5.1.2. Test Selection Strategy

We experiment with two different test selection strategies, which are described below.

Full selection. This strategy selects all the product line test cases for retest. This method causes a significant amount of regression testing costs.

Ekstazi_SPL selection. Ekstazi [13] is the state-of-the-art RTS method for a single software system that supports an automated tool. To compare an old version and its new version, Ekstazi computes the checksum of the new file versions. If the checksum is different from the checksum of the old version file, Ekstazi considers the file has been changed. To select test cases for retest, Ekstazi computes the file dependencies of test units (i.e., test method or test class). If file dependency remains the same in the new version, then no test unit is selected. However, if file dependency has been changed, then the test units that address the dependency on the file are selected. For regression testing of SPL, Ekstazi can be applied to products of a product family iteratively. We call this version of Ekstazi *Ekstazi_SPL selection*. The Ekstazi_SPL selection selects test cases by iteratively applying Ekstazi to each product of a product family. Ekstazi defined two kinds of granularity: the dependency granularity is the level where changes are determined and the selection granularity is the level where test cases are tracked and selected. Ekstazi uses classes for the dependency granularity and either methods or classes for the selection granularity. In our experiment, we use method granularity for the selection granularity because CPLTE collects execution traces at the level of test method.

Jung et al. [36] proposed a code-based RTS method for product lines. However, CPLTE cannot be used with the RTS method of Jung et al. [36] because the RTS method [36] does not collect parts of test execution traces but CPLTE requires the complete set of test execution traces. So, the RTS method of Jung et al. [36] is excluded from test selection strategies to use for the evaluation of our method.

Lity et al. [4] proposed a model-based RTS method for product lines. However, CPLTE cannot be used with the RTS method of Jung et al. [36] because their method requires the state machines of each product of a product family and our subject product lines do not have them. State machines of the subject product lines can be generated manually but it may produce a bias on experimental results. Moreover, CPLTE is not compatible with their method because the output of their method is a set of paths (i.e., alternating sequences of states and transitions) through state machines and CPLTE accepts only test cases at the code level.

5.1.3. Assessment of CPLTE

Unfortunately, the existing methods for SPL test execution have critical limitations regarding fault detection and automation except for those on configuration systems, which are systems that can be executed without binding of source code and test cases (cf. Section 6.2). For this reason, a fair comparison among CPLTE and the existing methods in terms of reduction of test executions is not feasible. Instead, we compare our method with the *exhaustive execution*, which will help answer the research questions for the following reason. The exhaustive execution executes the test cases selected from the A phase of RTS on all of their applicable products of a product family. This approach is effective for finding as many faults as possible when there are no known better approaches for executing SPL test cases. To show the difference in the number of detected faults, between the case when test cases are executed on all the products necessary for finding faults (our method) and the case when test cases are not executed on all the products necessary for finding faults, we use an additional SPL test execution method called *single-test-single-execution*. This latter method executes each selected test case only on one product among the products that reuse the test case, and thus results in a minimal

number of test executions (i.e., the number of test executions is equal to the number of test cases) but may miss faults.

5.2. Experimental Design

We show the effectiveness of our method using the exhaustive execution and the single-test-single-execution introduced in Section 5.1.3. Our experimental process is depicted in Figure 4. First, three sets of regression test cases are selected from the full selection and the Ekstazi_SPL selection. After that, CPLTE, the exhaustive execution and the single-test-single-execution are applied to the selected test cases. This process is conducted for each mutant of each subject product family. For CPLTE, we used a Byte Code Engineering Library (BCEL) [26] to instrument products of each product family. Finally, we assess the results in terms of the total number of test executions, the number of faults detected and the end-to-end time. The end-to-end time is the total time to perform SPL regression testing (i.e., the sum of time to perform, for CPLTE, *A*, *FC*, *E* and *C* phases and for the exhaustive execution and the single-test-single-execution, *A*, *E* and *C* phases without applying Algorithms 1 and 2).

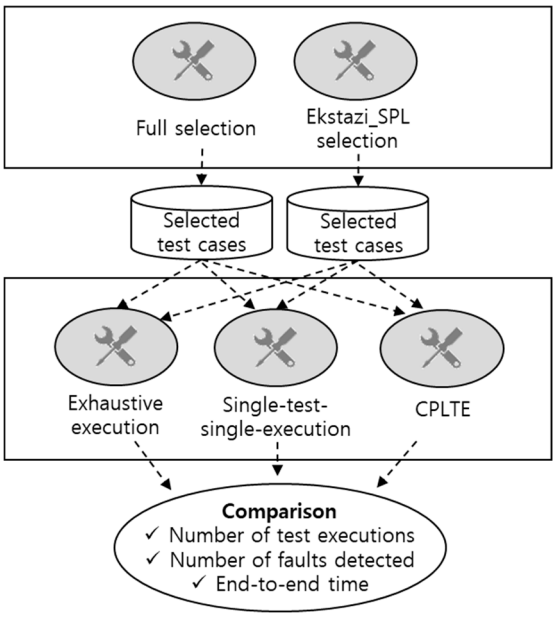

Figure 4. Process of our experiment.

5.3. Results and Discussion

In this section, we show the experimental results that answer our research questions and discuss them.

5.3.1. RQ1. How Effective Is CPLTE in Reducing the Number of Test Executions?

To answer RQ1, we compared the number of test executions resulting from CPLTE and the exhaustive execution. Table 4 presents the average number of test cases selected (#Sel in columns 2 and 6), the average number of test executions resulting from CPLTE and the exhaustive execution (columns 3, 4, 7, and 8) and the percentage of reduced test executions (Savings in columns 5 and 9). Table 4 shows the measurements obtained when the full selection and the Ekstazi_SPL were used for the *A* phase of RTS. From the table, the following observations can be made.

Table 4. Number of test executions and savings of Checksum-based Product Line Test Execution (CPLTE).

Subject Product Family	Full Selection				Ekstazi_SPL Selection			
	#Sel	Exhaustive Execution	CPLTE	Savings	#Sel	Exhaustive Execution	CPLTE	Savings
MobileMedia (4)	913	2711	1602	40.9%	303.4	813.9	617.2	24.2%
TankWar (5)	735	2968	1941	34.6%	178.8	669.3	608.7	9.0%
Prevayler (3)	1157	2940	1231	58.1%	139.8	311.8	176.1	43.5%
Prevayler (5)	2144	7499	2314	69.1%	324.8	799.1	399.4	50.0%
MobileRSSReader (3)	2764	8227	3036	63.1%	204.9	605.7	261.5	56.8%
MobileRSSReader (5)	4258	21,090	4707	77.7%	317.0	1559.6	411.5	73.6%
Lampiro (4)	5265	20,810	5330	74.3%	468.2	1729.5	503.7	70.9%
Lampiro (6)	7492	44,386	7725	82.6%	692.3	3818.3	833.0	78.2%
BerkeleyDB (3)	9027	24,504	16,390	33.1%	789.4	1925.1	1732.1	10.0%
BerkeleyDB (5)	15,161	66,525	34,659	47.9%	1764.5	7025.1	6038.3	14.0%
BerkeleyDB (7)	20,858	126,314	50,231	60.2%	2473.0	13,339.2	10,436.0	21.8%
BerkeleyDB (10)	24,352	200,534	59,960	70.1%	3346.5	24,634.5	17,712.2	28.1%
Average				59.3%				40.0%

#Sel: average number of test cases selected.

First, CPLTE identifies the considerable number of redundant test executions and reduces them. When the full selection and the Ekstazi_SPL selection are used for the *A* phase of RTS, CPLTE reduced, respectively, 59.3% and 40.0%, on average, of the number of test executions resulting from the exhaustive execution. This result indicates that the exhaustive execution repeats a significant number of equivalent test executions and CPLTE reduces them.

Second, CPLTE is much effective for the product families that have high Com. and CC. values. When a product family has a high Com. value, its test cases are likely to be reused for many products of the product family because a high portion of source code was shared for many products of the product family. When a product family has a high CC. value, a test case common to more than one product of the family is likely to cover the equivalent set of code units over the products. For example, the product families of Prevayler, MobileRSSReader and Lampiro have high values of both Com. and CC. The average of the two values is more than 82%. For the same product families, when the full selection and the Ekstazi_SPL selection were used for the *A* phase of RTS, CPLTE reduced up to 82.6% and 78.2%, respectively. On the other hand, the product families of MobileMediea, TankWar and BerkeleyDB have a moderate or low value of Com. or CC. So, for these product families, CPLTE reduced relatively low percentage of test executions as shown in Table 4. To analyze the correlation between the Savings value and the values of Com. and CC., we computed the Spearman's ρ correlation coefficient [37]. As a result, we obtained a value of 0.706 (p value < 0.01) for the full selection and 0.853 (p value < 0.01) for the Ekstazi_SPL selection, which indicates that the Savings value and the values of Com. and CC. are statistically significantly correlated.

Third, CPLTE reduces more test executions for the product families that have more products. The more products a product family contains, the more products would share test cases. For this reason, as the size of a product family grows, the number of redundant test executions that CPLTE can reduce increases significantly. Table 4 shows that CPLTE achieves more test execution savings for larger size product families for each of the product lines.

These observations indicate that CPLTE effectively reduces the total number of test executions by avoiding repeated equivalent executions of test cases selected for retest. For this reason, CPLTE can be used to extend the existing RTS method. Furthermore, CPLTE would be very effective for large product families that have significant amounts of commonality among the products (in our experiments, Com. and CC. values of more than 70%), which makes CPLTE practical.

5.3.2. RQ2. Does the Reduction of Test Executions Resulting from CPLTE Affect the Fault Detection Effect?

To answer RQ2, we compared the number of faults detected by the single-test-single-execution, the exhaustive execution and CPLTE. Table 5 presents the total number of faults injected (column 2) and the total number of faults detected (columns 3–5) for each test execution method. Because the RTS methods presented in Section 5.1.2 selected all the test cases that expose faults under the Controlled Regression Testing assumption (i.e., their fault detection effects were the same), they do not affect the result of this comparison.

Table 5. Number of faults detected by the single-test-single-execution, the exhaustive execution and CPLTE.

Subject SPL	#f	Number of Faults Detected		
		Single-Test-Single-Execution	Exhaustive Execution	CPLTE
MobileMedia (4)	301	187	213	213
TankWar (5)	250	122	154	154
Prevayler (3)	329	207	210	210
Prevayler (5)		204	211	211
MobileRSSReader (3)	406	281	287	287
MobileRSSReader (5)		282	287	287
Lampiro (4)	498	255	264	264
Lampiro (6)		252	266	266
BerkeleyDB (3)	607	282	329	329
BerkeleyDB (5)		280	329	329
BerkeleyDB (7)		284	334	334
BerkeleyDB (10)		284	334	334

#f: number of faults injected.

Table 5 shows that CPLTE detected more faults than the single-test-single-execution for all the subject product families. This result is caused by the difference in code coverage resulting from the two methods. The single-test-single-execution executes test cases only on a single product whereas CPLTE executes test cases on one or more products. For this reason, in the case of the test cases that cover different parts of source code depending on the product to apply, CPLTE has a higher code coverage than the single-test-single-execution and so CPLTE can detect faults that could not be detected by the single-test-single-execution. On the other hand, CPLTE showed the same fault detection effect as the exhaustive execution, which means that all the faults detected by the exhaustive execution were detected by CPLTE. This indicates that, under the Controlled Regression Testing assumption, CPLTE effectively avoids repeating equivalent test executions without losing fault detection effect.

Table 5 also shows that, for the product families of the Prevayler, MobileRSSReader and Lampiro product lines, the fault detection effect of CPLTE is slightly better than that of the single-test-single-execution. This indicates that test executions on these product families contain high proportions of equivalent test executions. Despite the small amount of benefit in fault detection in such cases, applying CPLTE to such product families is still preferable. This is because, as presented in Table 4, CPLTE requires only a small number of additional test executions (i.e., on top of the number of test executions from the single-test-single-execution which is equal to the number of test cases) that can be executed within a few seconds and detects faults that the single-test-single-execution may miss.

5.3.3. RQ3. How Effective Is CPLTE in Reducing the Cost of SPL Regression Testing?

To answer RQ3, we compared CPLTE and the exhaustive execution with regard to the end-to-end time. In this comparison, only the Ekstazi_SPL selection is used for the *A* phase because the full

selection is a naïve approach. Table 6 presents the average end-to-end time (in seconds) resulting from CPLTE and the exhaustive execution (columns 2 and 3) and the average percentage of the reduced end-to-end time (Savings in column 4). From Table 6, the following observations can be made.

Table 6. Average end-to-end time savings (in seconds) of CPLTE when the Ekstazi_SPL selection is used.

Subject SPL	Ekstazi_SPL Selection		
	Exhaustive Execution	CPLTE	Savings
MobileMedia (4)	9.2	7.5	18.5%
TankWar (5)	4.7	4.5	4.3%
Prevayler (3)	14.7	8.7	40.8%
Prevayler (5)	35.5	18.9	46.8%
MobileRSSReader (3)	13.5	6.5	51.9%
MobileRSSReader (5)	35.3	12.5	64.6%
Lampiro (4)	150.2	52.3	65.2%
Lampiro (6)	302.7	93.9	69.0%
BerkeleyDB (3)	37.9	35.0	7.7%
BerkeleyDB (5)	137.1	125.3	8.6%
BerkeleyDB (7)	257.1	221.2	14.0%
BerkeleyDB (10)	342.4	272.6	20.4%
Average			34.2%

First, CPLTE effectively reduces the time to perform SPL regression testing. For all the subject product families, CPLTE reduced the end-to-end time ranging from 4.3% to 69.0%. This indicates that CPLTE effectively reduces the time to execute test cases and the time to collect their execution traces compared to the exhaustive execution of test cases. This reduction of the end-to-end time is proportional to the amount of reduction of test executions because the cost of executing test cases and the cost of collecting their execution traces are the most significant part of the overall regression testing cost.

Second, the overhead of CPLTE is small. One source of overhead in CPLTE is the *FC* phase (i.e., Algorithm 1 in Section 4.4). However, the overhead from the *FC* phase is negligible. In the case of BerkeleyDB (10), which is the largest system used in our experiment, it took only 0.11 s to perform the *FC* phase. Another source of overhead in CPLTE is the selective test executions (i.e., Algorithm 2 in Section 4.6). However, this overhead is insignificant, too. As Table 4; Table 6 show, the savings in terms of the end-to-end time are 2.3~9.2% lower than savings in terms of the number of test executions. Because the overhead from the *FC* phase is negligible and the Ekstazi_SPL selection was commonly used for the *A* phase in both comparison test execution methods, this result indicates that the selective test execution of CPLTE requires 2.3~9.2% more of the time to perform all the phases of regression testing. Consequently, CPLTE incurs only a small amount of overhead in identifying equivalent test execution and conducting the selective test execution and so we expect CPLTE to work well even with larger SPL systems.

5.4. Threats to Validity

5.4.1. Internal validity

Validity of prototype tool. Our prototype tool for the experiment may contain defects. To overcome this threat, we reviewed its code, tested it on several product lines and manually inspected the results.

Validity of assumption. Our experiments were conducted on Windows 10 and so Windows background processes can be a threat to the controlled regression testing assumption because they can be noise for the end-to-end time measure. To mitigate this threat, before starting the experiments, we inspected them and stopped background processes that are unnecessary in conducting our experiments.

5.4.2. Construct validity

Validity of comparisons. To evaluate our method, we used the conventional methods, i.e., the exhaustive execution and the single-test-single-execution and this can be a threat to evaluating the effectiveness of our method. However, they are methods that can be used when there is no better practice.

5.4.3. External validity

Validity of subject product lines. Our subject product lines may not be representative of real product lines. To mitigate this threat, we selected open-source product lines that were recently used in other SPL research as experimental objects [28–30]. They vary in sizes, number of features and application domain. In addition, the product families produced from the subject product lines vary in Com. and CC. values, which significantly affects the proportion of equivalent test executions.

Validity of test case and fault. The test cases and faults used in our experiments may not be representative of real ones. Moreover, if the generated test cases are highly redundantly executed, the proposed method can produce better result. To address this threat, in terms of test case generation, we used EvoSuite, which is the state-of-the-art tool that obtained the highest overall score in coverage and fault detection in a competition with other tools [38]. Moreover, we manually excluded duplicated test cases to reduce test redundancy. The generated test cases are more suitable for our experiment than hand-made test cases. In terms of fault generation, a mutation tool has been commonly used for generating faults as in previous testing research [22,23], so we used a PIT mutation tool to inject faults because, as reported in [39], mutation faults are suitable for software testing experiments.

Validity of product line test cases. To generate a set of product line test cases, we produced a product family and then generated test cases for each product of the family because tools for automated SPL test case generation do not exist and the test cases of our subject product lines could not be directly generated by EvoSuite. However, the test cases generated for each product may not include test cases common to two or more products. Then they cannot be used as product line test cases. To overcome this threat, we carefully classified the test cases generated for each product into test cases that can be reused for other products and test cases unique to a single product, and afterwards we manually assigned each test case to the relevant products. Through this process, we obtained the test cases that are common to two or more products and specific to a single product and also produced test cases for each product of a product family.

6. Related Work

In this section, we discuss related work regarding reduction of redundant test cases and redundant test executions in the context of SPL regression testing, respectively in Section 6.1 and 6.2.

6.1. Studies on Reducing Redundant Test Cases for SPL

The RTS methods [4,5,36,40] for SPL have been proposed to reduce redundancy of test cases for retest of a modified product family. Jung et al. [36] propose an automated code-based RTS method, which reduces unnecessary testing cost by reusing the testing artifacts of one product for the other products and by avoiding the in-depth analysis of source code and test cases. Lity et al. [4] selects regression test cases using a delta modelling technique and an incremental model slicing technique. Neto et al. [5] use the architectural similarity between products of a product family to select and prioritize test cases for a modified product family. Ensan et al. [40] propose a goal-oriented test selection approach that selects test cases based on the priorities of features.

Several other approaches [41–44] have been proposed to minimize the size of test suite by removing obsolete or redundant test cases. Baller et al. [41] use an integer linear programming to minimize a test suite for a product family. Their method obtains an optimal test suite that minimizes the testing costs and maximizes the testing profits. Wang et al. [44] suggest a weight-based genetic

algorithm that optimizes the total execution time of a test suite and its fault detection capability. Marijan et al. [43] propose TITAN, which is a test suite optimization technique for highly configurable systems. Their method optimizes a test suite using both a test minimization technique and a test prioritization technique so as to achieve a high fault detection rate and a low test execution time. Another work of Marijan et al. [42] uses the coverage matrix of test cases and the fault detection history to identify redundant test cases that are not likely to detect faults. Their method minimizes a test suite by excluding redundant test cases from a test suite.

The approaches discussed above reduce the cost of SPL regression testing by selecting test cases for retest or eliminating redundant test cases. However, they do not address how to efficiently execute test cases for retest. So even though these approaches are applied, SPL regression testing still may suffer from redundancy at the level of test execution. On the other hand, CPLTE addresses the challenge of reducing redundancy of test executions for individual SPL test cases.

6.2. Studies on Reducing Redundant Test Executions for SPL

Kim et al. [8] suggest a shared execution approach, which executes instructions common to multiple program executions just once. They execute all the configurations together instead of executing each configuration separately by using a common call stack. The experiment showed that their approach can run test cases of a product line up to 50% faster than the exhaustive brute-force execution. Nguyen et al. [10] also address the challenge of executing a test case exhaustively over all configurations of a software product. They propose a variability-aware execution, which executes common code only once and executes variable code separately on each of the configurations. However, Wong et al. [12] say that because the variability-aware execution modifies existing JVM, it suffers from a conflict between functionalities. To address this problem, Wong et al. [12] propose a variational execution approach using automatic bytecode transformation. Their approach transforms individual instructions and modifies the control flow of methods to share common executions across configurations without modification of existing JVMs. The evaluation showed their approach is 2 to 46 times faster than the state-of-the-art variational execution interpreter for Java, VarexJ [30]. The above approaches have the same research direction as our work in this paper in that they reduce redundant test executions that do not contribute to finding faults. However, they do not address the problem of regression testing. Moreover, their methods and our method handle different types of systems. Their target system is a single software system that contains configurable options at the source code whereas, our target is, as we described in Section 2.1, a product line that produces products planned as a product family by assembling code fragments. Due to this difference, their methods are not applicable to the systems our method targets and cannot be compared with our method on an even basis.

ScenTED-DF [11] is a data flow-based product line testing technique. To describe the variability of a product line, ScenTED-DF uses an annotated variable activity diagram. Using this diagram, it avoids redundant testing of common parts of a product line by reusing partial traces of the existing test cases. However, modeling the variable activity diagram is "a creative modeling activity which cannot be automated" [11]. Therefore, the intervention by human experts is required. In addition, the annotated variable activity diagram is not a typical product line artifact and annotating the variable activity diagram requires significant efforts.

Li et al. [9] propose reusing a test execution trace to reduce efforts of SPL testing. When executing a test case on a product, their method records name, version, inputs and outputs of modules that the test case traverses over the product. If another test case traverses a module with the same name, version and inputs, then the output is retrieved from the recorded trace without recomputing it. However, they do not provide how to record names, versions and inputs of modules during test executions. Moreover, because these data are not typically used in regression testing, their method is not compatible with the existing regression testing techniques such as test case selection, prioritization and minimization that are conducted earlier to apply the method. So, much additional overhead would occur. On the other hand, our method does not require additional efforts for collecting test

execution traces because our method uses data (i.e., a sequence of code units covered by a test case) that are typically collected in the earlier steps of code-based regression testing.

In summary, although the approaches discussed above will be effective in reducing redundancy at the level of test executions, they have limitations in fault detection and automation and, moreover, handle systems that are different from those that we handle.

7. Conclusions

In this paper, we proposed an effective method of reducing redundant test executions for SPL regression testing. When a set of test cases is selected for retest, our method reduces redundant test executions by selecting in stages a set of products on which each test case can be executed with no repetition of equivalent test executions. Because our method does not require an in-depth analysis of source code and test cases, it does not incur much overhead in reducing redundant test executions. Moreover, because our method reduces only equivalent test executions, it has the same fault detection effect as the exhaustive execution of test cases. Our experimental evaluation showed that when the full selection and the Ekstazi_SPL selection are used for the *A* phase of RTS, our method reduced on average 59.3% and 40.0%, respectively, of the number of test executions resulting from the exhaustive execution, showing the same fault detection effect. Moreover, the overhead of our method was only 2.3~9.2% of the time required for all the phases of regression testing.

Our method uses test execution traces to identify the repetitions of equivalent test executions. However, our method does not require additional efforts for collecting test execution traces because, in coverage-based regression testing, test execution traces are typically managed for regression testing of the next version. For this reason, our method is suitable for regression testing rather than the general testing.

For future work, we are investigating the following directions. First, we are implementing our method by extending FeatureIDE [16], which is an open-source framework for feature-oriented software development based on Eclipse. Second, the Controlled Regression Testing assumption in Section 4.1 limits the application scope of our method. We are currently extending our method to make it work without this assumption. Finally, we will apply our method to larger scale SPLs from the industry.

Author Contributions: Conceptualization, P.J.; methodology, P.J.; validation, P.J.; investigation, P.J.; writing—original draft preparation, P.J.; writing—review and editing, S.K. and J.L.; supervision, S.K. All authors have read and agreed to the final version of the manuscript.

Funding: This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (NRF-2017M3C4A7066210) and by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2020R1F1A1071650).

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

References

1. Aggarwal, K.K.; Yogesh, S. *Software Engineering Programs Documentation, Operating Procedures*, 2nd ed.; Revised; New Age International Publishers: New Delhi, India, 2005.
2. Kazmi, R.; Jawawi, D.N.; Mohamad, R.; Ghani, I. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.* **2017**, *50*, 29. [CrossRef]
3. Pohl, K.; Böckle, G.; Van der Linden, F.J. *Software Product Line Engineering: Foundations, Principles and Techniques*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2005; p. 359.
4. 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]

5. Neto, P.A.D.M.S.; Do Carmo Machado, I.; Cavalcanti, Y.C.; De Almeida, E.S.; Garcia, V.C.; De Lemos Meira, S.R. A regression testing approach for software product lines architectures. In Proceedings of the 4th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), Bahia, Brazil, 27–29 September 2010; pp. 41–50.
6. Rothermel, G.; Harrold, M.J. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.* **1996**, *22*, 529–551. [CrossRef]
7. Weiss, D.M.; Lai, C.T.R. *Software Product-Line Engineering: A Family-Based Software Development Process*; Addison-Wesley: Boston, MA, USA, 1999; Volume 1.
8. Kim, C.H.P.; Khurshid, S.; Batory, D. Shared execution for efficiently testing product lines. In Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering, Dallas, TX, USA, 27–30 November 2012; pp. 221–230.
9. 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 Systems and Software Product Line Conference, Kyoto, Japan, 10–14 September 2007; pp. 65–72.
10. 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.
11. Stricker, V.; Metzger, A.; Pohl, K. Avoiding redundant testing in application engineering. In Proceedings of the 14th International Conference on Software Product Lines, Jejudo, Korea, 13–17 September 2010; pp. 226–240.
12. Wong, C.P.; Meinicke, J.; Lazarek, L.; Kästner, C. Faster variational execution with transparent bytecode transformation. *Proc. ACM Program. Lang.* **2018**, *2*, 117. [CrossRef]
13. Gligoric, M.; Eloussi, L.; Marinov, D. Practical regression test selection with dynamic file dependencies. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 12–17 July 2015; pp. 211–222.
14. 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; pp. 165–174.
15. Young, B.; Flores, R.; Cheatwood, J.; Peterson, T.; Clements, P. Product line engineering meets model based engineering in the defense and automotive industries. In Proceedings of the 21st International Systems and Software Product Line Conference, Sevilla, Spain, 25–29 September 2017; Volume A, pp. 175–179.
16. FeatureIDE. Available online: http://www.featureide.com/ (accessed on 3 December 2020).
17. Pure:variants. Available online: https://www.pure-systems.com/ (accessed on 3 December 2020).
18. CIDE. Available online: http://ckaestne.github.io/CIDE/ (accessed on 3 December 2020).
19. Farahani, E.D.; Habibi, J. Configuration management model in evolutionary software product line. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26*, 433–455. [CrossRef]
20. Felderer, M.; Fourneret, E. A systematic classification of security regression testing approaches. *Int. J. Softw. Tools Technol. Transf.* **2015**, *17*, 305–319. [CrossRef]
21. Wang, H.; Xing, J.; Yang, Q.; Wang, P.; Zhang, X.; Han, D. Optimal control-based regression test selection for service-oriented workflow applications. *J. Syst. Softw.* **2017**, *124*, 274–288. [CrossRef]
22. Qu, X.; Acharya, M.; Robinson, B. Configuration selection using code change impact analysis for regression testing. In Proceedings of the 28th IEEE International Conference on Software Maintenance, Tento, Italy, 23–28 September 2012; pp. 129–138.
23. Romano, S.; Scanniello, G.; Antoniol, G.; Marchetto, A. SPIRITuS: A simple information retrieval regression test selection approach. *Inf. Softw. Technol.* **2018**, *99*, 62–80. [CrossRef]
24. Marimuthu, C.; Chandrasekaran, K. Systematic studies in software product lines: A tertiary study. In Proceedings of the 21st International Systems and Software Product Line Conference, Sevilla, Spain, 25–29 September 2017; pp. 143–152.
25. Rivest, R.; Dusse, S. *The MD5 Message-Digest Algorithm*; MIT Laboratory for Computer Science and RSA Data Security, Inc.: New York, NY, USA, 1992.
26. BCEL. Available online: http://commons.apache.org/proper/commons-bcel/ (accessed on 3 December 2020).
27. SPL2go. Available online: http://spl2go.cs.ovgu.de/projects/ (accessed on 3 December 2020).

28. Kim, J.; Batory, D.; Dig, D. Refactoring java software product lines. In Proceedings of the 21st International Systems and Software Product Line Conference, Sevilla, Spain, 25–29 September 2017; pp. 59–68.
29. Krüger, J.; Al-Hajjaji, M.; Leich, T.; Saake, G. Mutation operators for feature-oriented software product lines. *Softw. Test. Verif. Reliab.* **2018**, *29*, 1676. [CrossRef]
30. Meinicke, J.; Wong, C.P.; Kästner, C.; Thüm, T.; Saake, G. On essential configuration complexity: Measuring interactions in highly-configurable systems. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 483–494.
31. LocMetrics. Available online: http://www.locmetrics.com/ (accessed on 3 December 2020).
32. Fraser, G.; Arcuri, A. EvoSuite: Automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 416–419.
33. PIT. Available online: http://pitest.org/ (accessed on 3 December 2020).
34. Jia, Y.; Harman, M. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **2011**, *37*, 649–678. [CrossRef]
35. Grün, B.J.; Schuler, D.; Zeller, A. The impact of equivalent mutants. In Proceedings of the 2009 International Conference on Software Testing, Verification, and Validation Workshops, Denver, Colorado, 1–4 April 2009; pp. 192–199.
36. Jung, P.; Kang, S.; Lee, J. Automated code-based test selection for software product line regression testing. *J. Syst. Softw.* **2019**, *158*, 110419. [CrossRef]
37. Spearman, C. The proof and measurement of association between two things. *Am. J. Psychol.* **1904**, *15*, 72–101. [CrossRef]
38. Campos, J.; Panichella, A.; Fraser, G. EvoSuiTE at the SBST 2019 tool competition. In Proceedings of the 12th International Workshop on Search-Based Software Testing, Montreal, QC, Canada, 26–27 May 2019; pp. 29–32.
39. Just, R.; Jalali, D.; Inozemtseva, L.; Ernst, M.D.; Holmes, R.; Fraser, G. Are mutants a valid substitute for real faults in software testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 654–665.
40. 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.
41. Baller, H.; Lity, S.; Lochau, M.; Schaefer, I. Multi-objective test suite optimization for incremental product family testing. In Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation, Cleveland, OH, USA, 31 March–4 April 2014; pp. 303–312.
42. Marijan, D.; Gotlieb, A.; Liaaen, M. A learning algorithm for optimizing continuous integration development and testing practice. *Softw. Pract. Exp.* **2019**, *49*, 192–213. [CrossRef]
43. Marijan, D.; Liaaen, M.; Gotlieb, A.; Sen, S.; Ieva, C. Titan: Test suite optimization for highly configurable software. In Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 13–17 March 2017; pp. 524–531.
44. 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]