Article

An Experimental Evaluation of Path-Based Product Line Integration Testing and Test Coverage Metrics

Jihyun Lee

Department of Software Engineering, Jeonbuk National University, Jeonju 54896, Republic of Korea; jihyun30@jbnu.ac.kr

Abstract: Product line testing is significant because any faults in a product line platform can lead to widespread impacts on multiple products configured from that platform within a product line. Due to the shared platform, certain testing can be repeatedly performed across different products, leading to unnecessary costs. To enhance quality and reduce costs in product line testing, it is essential to minimize redundant testing of the products in a product line. Because test coverage provides a way to explicitly state the extent to which a software item has been tested, having a clear understanding of test coverage helps avoid unnecessary repetition of tests and ensures that the testing efforts are focused on areas that require attention, ultimately leading to more efficient and effective product line testing. It is necessary to define appropriate test coverage metrics of product line testing that enable testers to identify redundancies in their testing efforts. Path-based integration testing has been proven to be an effective approach to product line integration testing. This paper defines coverage metrics for path-based product line integration testing and demonstrates their effectiveness in preventing redundant testing between platform testing and testing for individual products, while also effectively detecting faults. The experiment results highlight the coverage metrics’ effectiveness in avoiding redundant testing, reducing costs, and covering interfacing across different modules.

Keywords: software product line testing; software product line integration testing; path-based integration testing; test coverage metrics; test coverage measurement

1. Introduction

Software Product Line Testing (SPLT) is significant because any faults in product line platforms can have widespread impacts on multiple products configured based on the platforms. However, the presence of shared platforms may result in repetitive testing across different products within a product line, leading to unnecessary costs in SPLT. Furthermore, variable artifacts shared by only one or a few products are not implemented in platform development, which poses a challenge for integration testing in platform development. For integration testing in this context, derivers and stubs must be developed to simulate the behaviors of the corresponding unimplemented variable artifacts. However, implementing these auxiliary test codes demands a significant amount of effort [1].

Like other SPL lifecycle processes, SPLT should also contribute to achieving the benefits of the product line, i.e., cost reduction, improved productivity, and enhanced quality. This recognition is crucial for optimizing testing efforts and avoiding redundant testing in SPLT. Test coverage provides a way to explicitly state the extent to which an Implementation Under Test (IUT) has been tested by test suites [2,3]; having a clear understanding of test coverage helps avoid unnecessary repetition of tests and ensures that the testing efforts are focused on areas that require attention, ultimately leading to more efficient and effective product line testing. A precise description of the test coverage value of a product line enables test engineers to determine whether the test about to be performed is...
redundant with the test already conducted during platform testing or for other products of a product line [4–6]. Thus, it is essential to measure test coverage using the test coverage metrics of SPLT to find out the extent of redundancy.

However, in SPLT, there has been limited discussion regarding SPL integration testing and test coverage metrics, except SPL system testing, particularly when compared to the extensive discussion about them in single software testing [7]. Recognizing this gap, Lee [8] proposed basic concepts for path-based integration testing and coverage metrics applicable to SPL integration testing. This paper builds upon and refines these concepts, offering comprehensive path-based integration testing and test coverage metrics to address redundant testing issues by providing clear test coverage measurement for both platform testing and product testing. Furthermore, this paper confirms the effectiveness of path-based SPL integration testing and test coverage metrics with respect to test redundancy, test costs, and fault detection.

This paper is structured as follows: Related work is discussed in Section 2. Section 3 presents the path-based SPL integration testing method, along with the associated test coverage metrics and the essential concepts necessary to extend the MM-path-based integration testing method from a single software system to SPLT. Section 4 defines cost and benefit functions used to assess the effectiveness of the proposed method. Section 5 describes experimental evaluation results, including two experiments that provide answers to research questions. Section 6 covers discussions of the experimental evaluation of path-based SPL integration testing, while Section 7 addresses threats to validity. Section 8 concludes this paper.

2. Related Work

In integration testing, it is important to obtain both interaction information between units and the integration order of these units, as not all units interact with each other. Ganesan [9] and Li [10] propose incremental integration testing methods, considering the integration orders, while others propose methods only considering interactions among units [11–20].

Among studies that consider possible interactions only, one category of studies includes studies that see a component as the unit of integration [11–17,20], and the other category of studies regards a feature as the unit of integration [18,19]. SPL integration testing studies use control flows [11], method calls [12], data dependencies [13], deltas [16,17,20], or feature dependencies [18,19,21] to obtain interaction information. Because [18,19] deal with integration at the feature level, it is hard to see that it considers all possible interactions even though integration testing covers all feature interactions. Reis [11], Reuys [12], and Stricker [13] use an activity diagram as a test model tailored to depict interactions between components. They use interactions among components of the system or the subsystem for integration testing. However, the focus of [11] and [13] is on reducing the complexity of test models due to variabilities and [12] deals only with variability in requirements. The fundamental limitation of this kind of specification-based testing is that it is difficult to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of test cases covers an IUT [3].

These studies either do not measure test coverage or they primarily rely on feature coverage or feature combination coverage. Feature coverage and feature combination coverage, commonly utilized in existing SPLT studies, are types of coverage related to sample product testing. Most of them are about a selection of products to test [18,22]. Jung [23] tracks execution traces for the purpose of avoiding redundant testing by checking whether each test case covers the same code units. However, the method of this study is applicable to system testing rather than integration testing. Thus far, test coverage of integration testing has rarely been studied. SPL Integration testing is typically performed in both platform testing (testing of common components shared by products in a product line) and product testing (testing of individual products of a product line). It is essential to define
clear test coverage for these two testing processes to avoid redundant testing while still achieving quality goals.

3. Path-Based SPL Integration Testing and Test Coverage Metrics

3.1. Path-Based SPL Integration Testing

In single software testing, the path-based integration method offers the advantage of not needing stub and drive development, which typically demand a significant amount of test effort during integration testing. The path-based integration avoids the need for developing stubs and drivers by directly utilizing the original development code. Jorgensen [24] proposed the MM-path method, a technique based on method–message paths. An MM-path is represented as a series of module execution paths (MEPs), which are sequences of statements starting from a source node and ending at a sink node, without any intermediate sink nodes. In this context, the source node refers to a state fragment or executed lines of a program. Figure 1 illustrates an example of an MM-path graph, describing the message interaction between the two methods belonging to different modules.

![Figure 1](image1.png)

**Figure 1.** An MM-path between the two methods belonging to different modules.

In this example, there are several possible MEPs, including MEP-A1<a1, a2, a3, a4>, MEP-A2<a4, a5, a7, a8>, MEP-A3<a4, a6, a7, a8>, MEP-B1<b1, b2, b3>, and MEP-B2<b4, b5>. One particular MM-path comprises MEP-A1, MEP-B1, and MEP-A3. Each MM-path serves as a test case covering the integration path, as shown in Figure 2. In Figure 2, the dotted arrow represents the return path.

![Figure 2](image2.png)

**Figure 2.** An MM-path graph using MEPs.

The MM-path represents a path that focuses on interaction between different units. MM-path-based integration testing in single software testing can be employed for SPL integration testing because interactions among units in SPL are fundamentally performed through method calls between different units. The primary challenge in using MM-path-
based integration testing for SPLT is to effectively handle variabilities that distinguish different products of a product line. The variabilities of a unit are tested during unit testing, much like unit testing of single software testing does. Consequently, integration testing concentrates solely on interactions between units, making it essential to consider the variability involved in these interactions. Regarding commonalities, once they have been tested, redundant testing should be avoided. In this context, types of MM-paths in SPLT depend on the commonality and variability of modules. Thus, this paper divides the statement fragment used by [3] as a basic concept for extracting the nodes constituting the path into the following two types [8]:

- Common statement fragment: A sequence of statements, methods, or modules commonly shared by all products of a product line. A statement can consist of a single node.
- Variable statement fragment: A sequence of statements, methods, or modules that are either shared by two or more products of a product line or used by a single product of a product line. A statement can consist of a single node.

Based on the two types of MM-paths, the starting and ending source and sink nodes of each MEP can be categorized as common and variable types, respectively:

- Common source node: a common node where program execution starts or resumes.
- Variable source node: a variable node where program execution starts or resumes.
- Common sink node: a common statement fragment at which program execution terminates or transfers control to other units.
- Variable sink node: a variable statement fragment at which program execution terminates or transfers control to other units.

In SPLT, the MEP is identical to single software testing in that it represents a series of paths from the source node to the sink node within the same unit. However, in contrast to earlier work [3], I leverage the MEP as a coverage metric consisting of common and/or variable nodes as its elementary units, with the goal of reducing redundant testing in SPLT. The MEP can take various forms, considering the combination of common or variable nodes. This study defines the following two paths: an MEP and an MM-path formed based on the MEPs.

**Definition 1 (xx-MEP).** A sequence of nodes that begins with a common/variable source node and ends with a common/variable sink node, with no intervening sink nodes.

**Definition 2 (xx-MM-path).** A sequence of xx-MEPs linked by messages.

Based on the two definitions, the path-based integration testing for SPL classifies types of both the SPL-specific MEP and MM-path for use in test coverage measurements. In contrast to the path-based integration testing of single software testing, path-based SPL integration testing requires the handling of variability. In other words, the IUT in SPL can incorporate variability. Therefore, the MEP should be defined by considering this. However, in path-based SPL integration testing, it is not necessary to consider whether the statement fragments between source and sink nodes are common or variable statement fragments. This is because, in integration testing, only the statement fragment interacting with another unit (method or class) is a significant test target. The statement fragments located between the source and sink nodes belong to the same unit and are consequently tested during unit testing. Table 1 describes four types of xx-MEPs with their corresponding definitions, revised based on earlier work [3].
Table 1. Types of xx-MEPs with definitions.

<table>
<thead>
<tr>
<th>xx-MEP</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc-MEP</td>
<td>Commonality/Commonality MEP begins with a common source node, traverses through common or variable statement fragments that do not issue messages, and ends with a common sink node.</td>
</tr>
<tr>
<td>cv-MEP</td>
<td>Commonality/Variability MEP begins with a common source node, passes common or variable statement fragments that do not issue messages, and ends with a variable sink node.</td>
</tr>
<tr>
<td>vc-MEP</td>
<td>Variability/Commonality MEP begins with a variable source node, passes common or variable statement fragments that do not issue messages, and ends with a common sink node.</td>
</tr>
<tr>
<td>vv-MEP</td>
<td>Variability/Variability MEP begins with a variable source node, passes common or variable statement fragments that do not issue messages, and ends with a variable sink node.</td>
</tr>
</tbody>
</table>

In the proposed method, MEPs differ from those in MM-path-based integration testing of single software in terms of the types of sink and source nodes, as shown in Figure 3. This distinction takes into account the structural characteristics of SPL implementation code and the nature of integration testing. In the proposed method, variable statement fragments within the same unit (e.g., nodes represented as a single dotted line in Figure 3) are assumed to be tested during unit testing, with the existence of an auxiliary test code, such as derivers or stubs, for handling variability.

In the case of the xx-MM-path, which encompass cv-, vc-, and vv-MEPs containing variable slices, determining test data to cover the path becomes impossible if the variation point remains unbound. Even with test data decided, the path remains non-executable. Whether to test these paths during platform testing or during product testing depends on the test strategy. For example, if a test engineer performs testing according to the common and reuse strategy, the test path containing variability is created during platform testing, but the test execution is delayed until the binding time of application engineering. Because lots of stubs and/or drivers are needed for testing these paths, test costs increase exponentially. Table 2 describes the possible five types of MM-paths according to the types of MEPs in Table 1. Unlike the previous version [3], vc+cv-MM-path has been newly added, and the descriptions of the remaining MM-paths have been improved for clarity of meaning.
Table 2. Types of xx-MM-paths with definitions.

<table>
<thead>
<tr>
<th>xx-MM-Path</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc-MM-path</td>
<td>Commonality/Commonality-MM-path starts with a cc-MEP, passes only cc-MEPs, and ends with a cc-MEP in which the sink node sends no messages. A set of cc-MM-path, $P_{cc}$, consists of only cc-MEPs, as described in the following: $P_{cc} = {\text{paths containing cc-MEP combination}}$, where $P_{cc} \cap {\text{vc-MEP, cv-MEP, vc-MEP}} = \emptyset$.</td>
</tr>
<tr>
<td>vc-MM-path</td>
<td>Variability/Commonality-MM-path starts with a vc-MEP, passes cc-MEPs, and ends with a cc-MEP in which the sink node sends no messages. A set of this path, $P_{vc}$, should begin with a single vc-MEP, as described in the following: $P_{vc} = {\text{vc-MEP}} \leq P_{cc}$, where $P_{vc} \cap {\text{vc-MEP}} \neq \emptyset$.</td>
</tr>
<tr>
<td>cv-MM-path</td>
<td>Commonality/Variability-MM-path starts with a cv-MEP, passes cc-MEPs, and ends with a cc-MEP in which the sink node sends no messages. A set of this path, $P_{cv}$, should include a single cv-MEP, as described in the following: $P_{cv} = P_{vc} \leq {\text{cv-MEP}}$, where $P_{cv} \cap {\text{cv-MEP}} \neq \emptyset$.</td>
</tr>
<tr>
<td>vc+cv-MM-path</td>
<td>This MM-path starts with a vc-MEP, passes cc-MEPs, and ends with a cc-MEP in which the sink node sends no messages. A set of this path, $P_{vc+cv}$, should include one vc-MEP and one cv-MEP, respectively, as described in the following: $P_{vc+cv} = {\text{vc-MEP}} \leq P_{cc} \leq {\text{cv-MEP}}$, where $P_{cc} \cap \text{cv-MEP} \cap \text{vc-MEP} \neq \emptyset$.</td>
</tr>
<tr>
<td>vv-MM-path</td>
<td>Variability/Variability-MM-path starts with a cc-MEP, vc-MEP, vv-MEP, or cv-MEP, passes xx-MEP(s), and ends with a cc-MEP, vc-MEP, or vv-MEP in which the sink node sends no messages. A set of this path, $P_{vv}$, should include at least one vv-MEP, as described in the following: $P_{vv} = {\text{paths containing cc-MEP, vc-MEP, cv-MEP, and vv-MEP combination}}$, where $P_{cv} \cap {\text{vv-MEP}} \neq \emptyset$.</td>
</tr>
</tbody>
</table>

MM-path-based SPL integration testing, like single software’s path-based integration testing, is also source code-based. The difference between them lies in the foundation of the proposed method on the product line code base, which includes planned variation points and variants. Product line code base is the entire set of sources that is used to produce the products in a product line. MM-paths for an SPL are derived by analyzing call returns between modules based on the product line code base. This is possible because the product line code base includes all the code necessary to configure the planned products.

In addition, the proposed method is currently agnostic to variability mechanisms. While the variability mechanisms used to implement variabilities are essential aspects for executing integration testing, the proposed approach uses a product line code base to generate MM-paths, giving prominence to the method–message execution paths between different modules. If different variabilities involve interactions across modules, they create method–message execution paths, which become components of MM-paths.

MM-path-based SPL integration testing emphasizes the commonalities and variabilities in source nodes and sink nodes, which interact with different modules, as it focuses on the interfacing between various modules. Figure 4 represents an MM-path composed of cc-MEP, vc-MEP, and vv-MEP. The cases described by the paths <b1, b2, c1>, <c4, b2> in Figure 4a are examples including different variabilities.
Figure 4. Hypothetical MM-path graph for an SPL. (a) MM-path across three units. (b) MM-path expressed with MEPs.

The number of MM-paths does not depend on the number of valid products of an SPL. The proposed method employs a product line code base, so the complexity of the product line code base determines the number of MM-paths.

3.2. Test Coverage Metrics for xx-MM-Paths

Existing methods overlook platform testing and reuse problems when dealing with test coverage problems in SPLT. The proposed method, utilizing xx-MEP, effectively distinguishes between test coverage in platform testing and individual product testing of a product line, thereby offering a more comprehensive solution to avoid redundant testing. Possible SPL test coverage metrics based on the proposed method are as follows:

- \( C_{cc} \): This coverage has the advantage of being able to pre-verify commonality through a single product of a product line because it designs and executes test cases that involve commonalities only.

- \( C_{vc} \): This coverage metric includes vc-MM-paths that consist of cc-MEPs and vc-MEPs exclusively in the test run. Because only the source node of the given vc-MM-path has variability, in order to achieve this coverage during the platform testing, for each variable source node, drivers for receiving controls from another unit and initiating/resuming a program are required.

- \( C_{cv} \): This coverage metric encompasses cv-MM-paths, which are composed exclusively of cc-MEPs and cv-MEPs. Because only the sink node of the cv-MM-path is variable, achieving this coverage during platform testing necessitates the development of drivers for sending controls to another unit and stubs for returning controls for each variable sink node. Drivers and stubs should be developed to accommodate all possible values that a variation point can send or receive as controls (or messages).

- \( C_{vc+cv} \): This coverage metric encompasses MM-paths composed exclusively of cc-MEPs, vc-MEPs, and cv-MEPs. To achieve this coverage during platform testing, it is...
necessary to develop auxiliary test codes for variable source nodes and drivers and stubs for variable sink nodes.

- $C_{vp}$: vv-MM-paths are tested. Because the vv-MM-path can encompass various MEPs and multiple variants, achieving this coverage in platform testing requires the development of a significant number of stubs, drivers, and auxiliary test codes to account for the variability. Furthermore, the number of v and v combinations for each vv-MEP can become uncontrollable.

In SPLT, there may not exist an executable software product during platform development. Therefore, the cc-MM-path can perform cc-MM-path testing after deriving a representative product. Numerous studies have proposed methods for deriving such representative sample products [25]. Furthermore, vc-MM-path and cv-MM-path contain variability at the beginning and end, indicating that they can be tested during platform testing. Test coverage during product testing can be achieved by testing the vv-MM-path and application-specific parts that are not covered by platform testing.

The SPLT has the following problem: numerous stub and driver implementations may be required due to absent variants during testing, and, in the worst case, running the test is not possible until the system is fully integrated. xx-MM-path-based integration testing not only distinguishes between common paths and variable paths but also enables variable paths to distinguish paths such as cv- or vc-MM-paths that can be tested before product testing. It is also possible to measure integration testing coverage both during platform testing and product testing, which was previously unavailable.

### 4. Cost and Benefit Functions for Estimation

This study introduces cost functions for the cost/benefit analysis of product line testing based on the proposed method and test coverage metrics. To begin, the following variables were defined:

- $N_{xx}$ represents the number of xx-MM-paths of a product line. For example, $N_{cc}$ indicates the number of cc-MM-paths in the product line;
- $N_{MM}$-path $(p_i)$ refers to the total number of MM-paths for product $i$; and
- $N_{xx}(p_i)$ indicates the number of xx-MM-paths for product $i$. For example, $N_{vc}(p_i)$ represents the number of vc-MM-paths for product $i$.

Next, the following cost functions were used to evaluate the proposed method by utilizing the proposed test coverage metric:

- $C_{driving\_path}$ is a function that, given the relevant parameters, calculates the cost for a test organization to derive MM-paths for a product or a product line;
- $C_{exec}$ is a function that, given the relevant parameters, calculates the average cost for a test organization to determine test inputs and execute tests for all defined MM-paths;
- $C_{aux}$ is a function that, given the relevant parameters, calculates the cost for a test organization to develop auxiliary test code (aux), including $C_{drive}$ and $C_{stub}$, representing the average cost of developing drivers and stubs, respectively (this cost function performs estimation based on the number of required drivers and stubs); and
- $C_{ccx}$ is a function that, given the relevant parameters, calculates the cost for a test organization to design and run tests based on a specific test coverage criterion. For example, $C_{cc}$ represents the total cost of the product line testing when using the cc-MM-path test coverage.

When each product of a product line using the MM-path-based integration testing method is tested in a stand-alone manner, the total cost of testing is estimated by the following Equation (1):

Cost of testing $n$ products in a stand-alone manner = \[ \sum_{i=1}^{n}(N_{MM\_path}(p_i) \times C_{exec} + C_{driving\_path}) \] (1)
Since MM-path-based integration testing uses actual codes for stubs or drivers, Equation (1) does not need to consider such costs.

The cost of testing the entire product line can vary depending on the test coverage employed in platform testing. In this cost function, the costs required for testing the product-specific parts are not taken into consideration. Equation (2) is a generic equation that does not specify the test coverage used in platform testing.

\[
C_{xx} = C_{driving\_path} + N_{xx} \times C_{aux} + \sum_{i=1}^{n} (N_{MM\_path}(p_i) - N_{xx}(p_i)) \times C_{exec}
\]

For the \( C_{xx} \) coverage metric, i.e., \( xx \) is \( cc \), there is no need for the cost of developing auxiliary test code, i.e., \( C_{aux} \) is 0. Therefore, the resulting cost function is derived as following:

\[
C_{cc} = C_{driving\_path} + \sum_{i=1}^{n} (N_{MM\_path}(p_i) - N_{cc}(p_i)) \times C_{exec}
\]

If the \( C_{cc} \) coverage is selected, both \( cc-MM \)-paths and \( vc-MM \)-paths containing variability are tested in platform testing. Since each member product of a product line does not need to retest \( cc-MM \)-paths and \( vc-MM \)-paths, the cost can be estimated with the following function derived from Equation (2):

\[
C_{cc} = C_{driving\_path} + N_{cc} \times C_{aux} + \sum_{i=1}^{n} (N_{MM\_path}(p_i) - N_{cc}(p_i)) \times C_{exec}
\]

The benefit, representing the cost savings achieved by testing through the method and metrics compared to the cost of testing in a stand-alone manner, can be calculated using the following Equation (3):

\[
Benefit = \frac{Equation \ (1) - C_{xx}}{Equation \ (1)}
\]

5. Experimental Evaluation of the Method

In this section, I present the results of the experimental evaluation of the \( xx-MM \)-path-based SPL integration testing concerning the benefits of the proposed coverage metric in reducing redundancy and measuring test coverage.

5.1. Research Questions

Through these case studies, I aim to obtain answers to the following research questions:

- **RQ1: Does path-based SPL integration testing allow us to avoid redundant testing?** The answer to this RQ is “yes” if the path that has already been tested in the platform testing is not tested again in the product testing, when the testing is performed in accordance with the given \( xx-MM \)-path coverage. Otherwise, the answer is “no”. This research question also confirms whether the proposed method can avoid redundant test coverage because avoiding redundant testing is closely related to avoiding redundant test coverage.

- **RQ2: Does path-based SPL integration testing reduce costs by avoiding redundant testing?** To obtain the answer to this RQ I compare the cost of product line testing performed in a stand-alone fashion with the cost of testing a product line in accordance with the selected \( xx-MM \)-path test coverage. The cost of the path-based integration testing is calculated by summing the platform testing cost performed in accordance with the selected \( xx-MM \)-path test coverage and the total product testing cost of all member products that tests only those parts not covered by the platform testing. The answer to this RQ is “yes” if the cost reduction ratio is positive and “no” if the result is negative. The detailed variables and equations needed for cost/benefit analysis are given in Section 3.

- **RQ3: Do test cases derived from path-based SPL integration testing cover all method calls?** The answer to this RQ is “yes” if test cases derived using MM-paths cover all possible
method calls of the integrated system. Otherwise, the answer is “no”. This result represents the possibility of detecting interfacing errors between modules.

5.2. Experiment Using Random Combinations

xx-MEPs constitute the MM-path. Thus, the MM-path in SPL integration testing is a combination of these xx-MEPs. In this experiment, random combinations of possible MM-paths are generated by using four types of MEPs defined in Section 2, and experimental evaluation of the proposed method and metrics in terms of redundant testing and test coverage measurement is performed using the generated MM-paths. The results give answers to RQ1 through RQ3. Since the MM-paths of the actual SPL system are eventually included in the possible MEPs, the answers obtained through this experiment are also applied to the actual SPL system. The experiment proceeded as follows:

**Step 1: Deriving MM-paths for a product line.**

To evaluate the cost savings of the proposed method and metrics, MEPs consisting of MM-paths for a product line are generated by combining four types of MEPs in 2-way, 3-way, 4-way, and 5-way configuration at random. These combinations of MEPs allow the duplication of MEPs.

**Step 2: Extracting the MM-paths of each product at random from the MM-paths for a product line generated in Step 1.**

This step is repeated “n” times for “n” products, and the number of MM-paths is randomly determined as “x”. Extraction allows for duplication. The reason for permitting duplication is that there can be multiple paths of the same type of MEPs covering different code segments.

**Step 3: Estimating costs for both platform testing and product testing based on test coverage decisions.**

Test coverage is one of the metrics for xx-MM-path defined in Section 3.2. Cost estimates are made using the following assumptions:

**Assumption 1 (C_{drive_{path}}).** The value of C_{drive_{path}} for one product is assumed to be “1”, while the value of C_{drive_{path}} for the entire product line is seven times as much as an individual product. The cost of making an asset reusable is widely estimated to be 50% of the cost of building the asset for a single use [26,27]. Given the high complexity of MM-path derivation in a product line, in this experiment, the effort required to define the path is assumed to be seven times greater. Thus, the value of C_{drive_{path}} for the product line is ‘7’.

**Assumption 2 (C_{exec}).** Most of this cost is attributed to determining test inputs. The cost is assumed to be ‘0.2’ for platform testing and ‘0.1’ for product testing, which is 50% of the platform testing cost, based on the same reason used to create Assumption 1. The effort required to determine test inputs in platform testing and product testing may not differ significantly. This is because a variant must be determined in order to run a test, even if it includes variability. This is a very conservative estimate, suggesting that it takes twice as much effort as the test execution effort in product testing.

**Assumption 3 (C_{aux}).** The development efforts for stubs or drivers are estimated at 0.08 per unit. This value is estimated to be 40% of the test execution effort (0.2). cv-MM-path and vv-MM-path require both stubs and drivers, while vc-MM-path only requires drivers. In the case of cv-MM-path, the development efforts for stubs or drivers are estimated to be 0.2 per unit due to the combinatorial complexity among variable values. In addition, vc+cv-MM-path includes vc-MEP, which requires drivers only and is calculated as ‘1.9 × 0.08’ instead of ‘2 × 0.08’.
Step 4: Calculating costs and benefits based on test coverage metrics defined in Section 3.2.

As a result of Step 1, a product line consisting of 155 MM-paths was created. This product line comprises 20 individual products as members. Table 3 shows the details on the number of products within the product line, the number of each type of xx-MM-path, and the estimated cost for each product based on test coverage. For P1, there are three vc+cv-MM-paths, two cv-MM-paths, one vc- & cv-MEP, and eight vv-MM-paths. In total, P1 has 24 MM-paths, including 11 cc-MM-paths. The cost for testing xx-MM-path for each product is determined using the equations outlined in Section 3, taking into account Assumption 1 through Assumption 3.

Table 3 presents the results of the cost estimation for each product’s testing based on the test coverage criteria. If the cc-coverage criterion is selected, only common MM-paths are covered during platform testing. Consequently, each product’s testing must perform testing for the remaining MM-paths. For example, if the cc-coverage criterion is selected, testing for P1 covers only 13 MM-paths, the cost of which is calculated by multiplying 0.1, representing a cost for defining test data, resulting in 1.3. However, vv-MM-path coverage leads to more platform testing but reduces the cost of testing each product individually. With the vv-MM-path coverage criterion, all MM-paths are tested during platform testing, eliminating the need for additional testing costs for individual products, resulting in ‘0’ additional cost in the product testing. Conversely, the cost for stubs/drivers increases significantly. As another example, if cv-coverage criterion is selected, testing for P1 covers MM-paths, except for cc- and cv-MM-paths, because these two types of paths are covered in platform testing. The last row represents the total cost of testing all the products for each coverage criterion. The results indicate that when platform testing covers a greater number of paths, the costs for testing in product testing are lower.

Table 3. Number of xx-MM-paths and cost estimates for testing each product.

<table>
<thead>
<tr>
<th>Products</th>
<th>Number of xx-MM-Paths</th>
<th>Cost for Each Product Testing by Coverage Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>cc-</td>
</tr>
<tr>
<td>P1</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>P2</td>
<td>39</td>
<td>11</td>
</tr>
<tr>
<td>P3</td>
<td>28</td>
<td>11</td>
</tr>
<tr>
<td>P4</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>P5</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>P6</td>
<td>30</td>
<td>11</td>
</tr>
<tr>
<td>P7</td>
<td>25</td>
<td>11</td>
</tr>
<tr>
<td>P8</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>P9</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>P10</td>
<td>36</td>
<td>11</td>
</tr>
<tr>
<td>P11</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>P12</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>P13</td>
<td>34</td>
<td>11</td>
</tr>
<tr>
<td>P14</td>
<td>41</td>
<td>11</td>
</tr>
<tr>
<td>P15</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>P16</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>P17</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>P18</td>
<td>30</td>
<td>11</td>
</tr>
<tr>
<td>P19</td>
<td>36</td>
<td>11</td>
</tr>
<tr>
<td>P20</td>
<td>40</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>155</td>
<td>11</td>
</tr>
</tbody>
</table>
Table 4 shows the following information at the product line level: the number of MM-paths; the estimated costs, which include the number of MM-paths to be tested in platform testing based on the selected test coverage criterion; the total costs for product testing in accordance with the coverage criterion; and the total cost estimates required for product line testing. Each cost is estimated using the appropriate equations derived from Equation (2) in accordance with the selected coverages.

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>Stand-Alone Manner</th>
<th>cc-MM-Path</th>
<th>vc-MM-Path</th>
<th>cv-MM-Path</th>
<th>vc+cv-MM-Path</th>
<th>vv-MM-Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of xx-MM-path ((N_{xx}))</td>
<td>155</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>10</td>
<td>92</td>
</tr>
<tr>
<td>Number of MM-paths tested in platform testing</td>
<td>0</td>
<td>11</td>
<td>28</td>
<td>53</td>
<td>63</td>
<td>155</td>
</tr>
<tr>
<td>Cost for testing xx-MM-paths in platform testing</td>
<td>0</td>
<td>1.1</td>
<td>5.86</td>
<td>10.1</td>
<td>18.38</td>
<td>72.48</td>
</tr>
<tr>
<td>Total cost for testing all products within a product line</td>
<td>78.4</td>
<td>36.4</td>
<td>34.9</td>
<td>68.9</td>
<td>27.1</td>
<td>0</td>
</tr>
<tr>
<td>Total costs for product line testing ((C_{xx}))</td>
<td>78.4</td>
<td>44.5</td>
<td>47.76</td>
<td>86</td>
<td>55.38</td>
<td>89.88</td>
</tr>
</tbody>
</table>

Figure 5 compares the costs of different test coverages with the cost of testing in a stand-alone manner. The cost of testing in the stove-piped manner is estimated using Equation (1). The benefits of these test coverages are estimated using Equation (3). The solid line indicates the cost of testing in a stand-alone manner. It is evident that the highest benefit can be obtained when using cc-MM-path coverage, vc-MM-path coverage, and vc+cv-MM-path coverage, whereas cost loss occurs when using the cv-MM-path coverage and vv-MM-path coverage. Although cv-MM-path coverage is costly, a positive benefit/cost ratio can be achieved by adopting either vc- or vc+cv-MM-path coverage.

![Figure 5. Total cost and benefit of each xx-MM-path’s coverage.](image)

5.3. Case Study Using the Elevator System SPL

This section evaluates the effectiveness of path-based integration testing for SPL using the elevator system product line as a case study. The elevator system product line
consists of six features, one mandatory and five optional, resulting in a total of 20 products. The elevator system product line includes a java file containing five java sources and one main method. Figure 6 is the feature model of the elevator system product line.

![Feature model of the elevator system product line](image)

**Figure 6.** Feature model of the elevator system product line.

The elevator system product line was implemented using FeatureIDE ([https://www.featureide.de](https://www.featureide.de) (accessed on 24 Nov 2023)) [28], resulting in a 450LOC-scale system comprising five classes. A call graph enables a graphical analysis of program call relations and flow sequencing. For analyzing the method call relationship of the elevator system product line, the CallGraph tool provided by Certiv Analytics is utilized. Due to the complexity of the graph, it is challenging to represent and analyze all caller-callee relationships as a single CallGraph. Therefore, in this study, all callees were found starting from the main caller, after which the path graph was created. Figure 7 shows a part of the path graph, illustrating the scenario where the ‘ExecutiveFloor’ and ‘Empty’ features are selected.

![Part of the XX-MEP-paths for the elevator product line](image)

**Figure 7.** Part of the XX-MEP-paths for the elevator product line.
In Figure 7, 11 cc-MEPs, one cv-MEP and two vv-MEPs are analyzed. Based on these results, three cc-MM-paths and two cv-MM-paths are derived. The resulting paths are given below:

**CC-MM-path:**
- A.randomSequenceOfActions/X.XXCall/C.Person/D.addWaitingPerson
- A.randomSequenceOfActions/B.isIdle/B.anyStopRequested/E.getFloor/D.hasCall
- B.timeShift/C.getDestination

**CV-MM-path:**
- B.timeShift/B.leaveElevator/C.leaveElevator/C.getName/||vp_B.isEmpty

**VV-MM-path:**
- B.timeShift/B.stopRequestedAtCurrentFloor/||vp_B.isExecutiveFloorCalling/E.getFloorID/D.hasCall/| |vp_B.isExecutiveFloor

The experiment was conducted during product testing following the sample application test strategy. This study addressed whether the xx-MM-path was covered by the integration testing test case using EclEmma version 3.1.0, a code coverage measurement tool. The next test case involves passengers moving from the executive floor to the ground floor. These test cases cover cc-, cv-, and vv-MM-path as defined (with method call coverage at 100% and all relevant branches covered). cc-MM-paths are covered in one sample product testing, and they are not retested in other product testing. Testing cc-MM-paths through a single sample product offers the advantage of achieving the required coverage while addressing the challenge of platform testing with high-complexity platform features that lack executable implementation.

```java
@Test
public void Specification1() {
    Environment env = new Environment(5);
    Elevator e = new Elevator(env, false);
    Actions a = new Actions(env, e);
    a.bobCall();//going from Executive floor to Ground floor
    for (int i = 0; i < cleanupTimeShifts && ! e.isBlocked(); i++) {
        if (e.isIdle())
            return;
        e.timeShift();
    }
}
```

A total of 26 methods are part of the MM-path originating from the ‘timeshift()’ method, which has the most method call relationships within the elevator system product line. Of these, nine methods encompass a total of 44 branches. The following additional test cases are derived from the previously mentioned MM-paths:
/*@Test*/
public void Specification1() {
    Environment env = new Environment(5);
    Elevator e = new Elevator(env, false);
    Actions a = new Actions(env, e);
    a.aliceCall(); // going from 3 to 0
    a.bobCall(); // going from Executive floor to Ground floor
    for (int i = 0; i < cleanupTimeShifts && !e.isBlocked(); i++) {
        e.timeShift();
    }
    a.bigMacCall(); // going from 1 to 3
    for (int i = 0; i < cleanupTimeShifts && !e.isBlocked(); i++) {
        e.timeShift();
    }
    a.aliceCall(); // going from 3 to 0
    a.chuckCall(); // going from 1 to 3
    for (int i = 0; i < cleanupTimeShifts && !e.isBlocked(); i++) {
        if (e.isIdle())
            return;
    }
    e.timeShift();
}

As a result, all methods in the path were covered. However, two of the 46 branches were left uncovered. A non-covered branch is one that depends on the specificity of the variability implementation mechanism used. Such a branch is analyzed as one that can be executed when the variability value is ‘none’. MM-path-based integration testing informs test engineers about the variability in the execution path during integration testing, helping them determine the integration testing strategy.

5.4. Answers to Research Questions

Through two experiments I obtained the following answer to each RQ:

- **The answer to RQ1**: MM-path-based SPL integration testing eliminates redundant testing. All xx-MM-paths identified in Experiment 1 of Section 5.2 are independent and do not require retesting in each product testing if they are covered during platform testing based on test coverage decisions. MM-path-based integration testing fundamentally prevents redundant testing. Additionally, MM-path-based SPL integration testing simplifies test coverage measurement and resolves redundancy issues. As demonstrated in Experiment 1 of Section 5.2, the proposed method and metrics provide clarity regarding the scope of platform testing and product testing based on test coverage determination. In summary, MM-path-based SPL integration testing facilitates easy test coverage measurement.

- **The answer to RQ2**: As depicted in Figure 5, cc-MM-path coverage yields the highest benefit, while vv-MM-path coverage results in the lowest benefit. Specifically, the cost of cv-MM-path coverage was higher than that of testing in a stand-alone manner due to the increased expense associated with developing stubs and drivers when adopting cv-MM-path coverage. In contrast, adopting vc-MM-path coverage requires only the development of drivers. Consequently, the cost of vc+cv-MM-path coverage is lower than that of cv-MM-path coverage because vc-MEPs of vc+cv-MM-paths help offset the cost incurred by cv-MM-paths.

- **The answer to RQ3**: MM-path-based SPL integration testing efficiently supports testing method calls across different modules with a reduced test set. In Experiment 2 of Section 5.3, applying the proposed method and metrics to the elevator product line using defined xx-MM-paths demonstrates that test cases cover all method calls.
without redundancy. This is contingent upon the test input adequately covering method invocations while considering interactions that may lead to defects.

6. Discussion

This paper revised the initial version of MM-path-based SPL integration testing and validated the effectiveness of it with test coverage metrics through two experimental evaluations. The path-based integration testing method, extended for SPLT, can be executed during platform testing if the driver and stub for the call-return of variability-related methods are implemented, as variability is part of the path. However, with the proposed method and metrics, similar to single software testing, finding paths can be challenging due to the large size of the software. This challenge can be addressed by using the call graph to search for callee methods beginning with the start method. Experimental evaluation of the proposed method and metrics may encounter the following issues:

- **Worst-case xx-MM-path-based testing**: cc-MM-paths, cv-MM-paths, and vc-MM-paths can be exceedingly rare. In such cases, it is possible to separate the thread of the path into sub-paths, rather than testing end-to-end paths. The sub-path containing variants can then be tested during platform testing, facilitated by driver or stub implementation. This approach allows for an increased number of testable paths during platform testing. While this scheme may reduce the advantages of path-based integration that relies on actual code without the need for driver or stub implementation, it proves valuable when only a small number of variabilities exist in the path due to the characteristics of SPLT. Variability may be implemented during platform development or by introducing drivers and stubs for testing. If variabilities are implemented, testing must encompass every possible combination that can be bound to the variation point. This enables the subdivision of paths containing a variation point into multiple detail paths.

- **Path-explosion and path management**: Because real-world SPLs include hundreds of variabilities, it is challenging to apply path-based testing comprehensively. This challenge also applies to path-based testing in individual software products. In single software testing, MM-path-based integration testing is used for integration-level testing and ASF (atomic system function) testing, which verifies function visibility at the system level [24]. This paper focuses on method and message-level MM-path-based integration testing for SPL. One approach to reduce the number of paths is to base them on input and output port events of the atomic system function, rather than methods or messages. However, ASF graphs become available only after lower-level verification is completed. Path-based testing can adapt path threads to different levels, such as unit, integration, subsystem, or system, depending on the context. While the detailed discussion of this adaptation is beyond the paper’s scope, it offers a solution to the challenges of path explosion and path management stemming from numerous path threads.

- **Test data (test inputs) definition**: A significant challenge in path-based testing involves defining test data that covers path threads [29]. Research into automatic test data generation methods, including those for path-based software testing [30], is currently ongoing. This challenge is not unique to SPLT and requires ongoing investigation to develop automatic test generation methods that consider the specific characteristics of SPLT, similar to what is performed in single software testing.

- **Handling variability dependencies and constraints**: The proposed method considers variability dependencies (optional, mandatory, alternative) and variability constraints based on the feature model. This is intended to define the expected results of test cases that cover the paths. However, if the implementation deviates from the specification, including variability dependencies and constraints of the feature model, and message passing occurs on an invalid path, it is evident that MM-paths may include invalid paths.
7. Threats to Validity

The proposed method has the following internal and external threats to validity:

- **Instrumentation issue**: The values of cost factors, such as $C_{\text{drivingpath}}$, $C_{\text{exec}}$, and $C_{\text{aux}}$, are not calculated based on actual data; rather, they are assumed values derived from the data reported in an existing technical report. Even though the estimation has been performed conservatively, their values may vary from one SPL organization to another and from one product line to another within the same organization. Consequently, the values of these cost factors are not stable, which may result in inconsistent results when applying the proposed method.

- **Random selection bias**: To validate the cost savings of the proposed method, this paper randomly generates possible MM-paths for an SPL. While the paper generates a specific number of MM-paths for an SPL using a combinatorial method to reduce selection bias, this approach may still pose a threat to drawing generalized conclusions about the proposed method.

- **Limited generalizability**: (1) The proposed approach analyzes MM-paths based on a product line code base of generative product lines. Therefore, the proposed method is only effective for SPLs that employ variability mechanisms supporting generative product lines based on the product line code base. (2) The proposed method has been validated using a small product line, and, as a result, the cost savings attributed to removing redundant testing may be overshadowed by the cost of deriving a large number of MM-paths in a larger size of product lines.

Given these potential threats to validity, the conclusions presented in this paper may not be guaranteed to hold when applying the proposed method in real-world SPLs.

8. Conclusions and Future Work

This paper revised and conducted experimental evaluations for the MM-path-based SPL integration testing method, which analyzes call–return paths between methods, integrates based on paths, and defines test cases covering these paths. The experimental evaluation confirmed the effectiveness of the proposed method and metrics in addressing the issue of redundant test coverage in SPLT, which is closely related to avoiding redundant testing. The coverage metrics defined in Section 3.2 proved valuable for determining coverage criteria during platform testing, allowing for more efficient product testing while minimizing duplication. Additionally, by utilizing the cost functions and equations defined in Section 3, this paper assessed the potential cost reduction for each test coverage metric of the proposed method, providing valuable support for test engineers in deciding SPLT test coverage. Furthermore, this paper verified that the proposed method effectively captures interactions between different units, ensuring that faults within these interactions are not overlooked. Future research will apply the proposed method and metrics to real-world SPLs.

Funding: Not applicable.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: This paper was supported by research funds from Jeonbuk National University in 2023.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The author declares no conflict of interest.


Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.