When Project Centralization and Random Testing Meet

–Efficient Automatic Testing of Multiple Software Product Variants–

Lei Ma Cyrille Artho Cheng Zhang Hiroyuki Sato

Recently, multiple versions of a software artifact have become common occurrences in software development and maintenance. Many similar software product variants with different versions are created in software evolution and Software Product Lines (SPL). This brings challenges in testing these multiple products in a cost-effective way. Testing each product variant separately causes redundancies in testing the common code. Moreover, the test results are not easily sharable among multiple versions.

In this paper, we present a framework to test multiple product variants simultaneously. We propose the use of project centralization to represent multiple versions as a meta-product that preserves the behavior of each version. We also discuss the adaptation issues of existing random testing tool Randoop to support multiple versions and coverage analysis. Furthermore, we point out the potential issues of project centralization for more accurate and sound multi-version automatic testing.

1 Introduction

Unit testing is a widely accepted and important software quality assurance technique. A unit test of an object-oriented program consists of a sequence of method invocations as inputs to test a method. Manually crafting test sequences, however, is labor intensive. Automatic testing techniques reduce such human efforts by generating test cases automatically. Random testing [1] [8] is such an automatic testing technique. It is easy to use and scalable by generating test sequences randomly. It also has been proved to be useful in detecting unknown bugs [8]. However, existing random testing techniques do not support to test multiple similar products with different versions cost-effectively.

Multiple versions of a software artifact are common occurrences in software development and maintenance. According to Lehmann's software evolution law [4], a software system requires continuously changes to increase its functionalities, to fix bugs, and to adapt to new requirements over its life cycle. These version changes are released as a sequence of updated versions. With the widely adoption of revision control systems like *git* or *mercurial*, more and more software product variants are created in the software evolution cycle.

The creation of multiple similar products with different versions is also facilitated by the modern software development paradigm Software Product Lines (SPL) engineering [9] and the popular industrial *clone-and-own* approach. Software product line engineering (SPLE) allows to systematically generate families of similar products to address a particular market segment or to fulfill a specific mission [9]. The *clone-and-own* approach creates a new software product by copying and modifying an existing one.

Although many similar product variants are produced, there currently lacks a framework for the au-

Lei Ma, The University of Tokyo, Japan.
Cyrille Artho, RISEC, AIST, Japan.
Cheng Zhang, The University of Waterloo, Canada.
Hiroyuki Sato, The University of Tokyo, Japan.

tomatic testing of these products efficiently. Testing each product separately would cause redundancies in testing common code. Moreover, the individual testing results are also difficult to share among multiple versions to improve the overall performance. For example, if multiple software products have an identical function, independently executing the same test case on such a function for each product causes redundant executions, because the same implementation of a function shows the same run-time behavior in each of its occurrences.

Project centralization [6][7] transforms multiple software products to a single *meta-product* that shares common code while preserving the behavior of each product. It has been used to verify the distributed applications with multiple versions of multiple peers [6].

In this paper, we first summarize our preliminary results [5] by leveraging project centralization and random testing on multiple products from SPLs. Then, we discuss the limitation of existing techniques, and further elaborate our work in progress improvements of project centralization and testing strategies. Our overall goal is to build a general framework to automatically test multiple similar software products in a cost-effective way.

2 Related Work

Although there exists a large body of work on automatic test case generation, limited work has been done for fully automatic testing of multiple version simultaneously. This section summarizes the most closely related work, which is covered in more depth in our previous publication [5].

Several previous work has shown that the reuse of testing results could improve the testing coverage. OCAT [3] reused the objects captured from sample test cases as inputs for further test case generation. Palus [12] trained a method sequence model from the provided test cases and combined run-time analysis to further guide test case generation. However, they have not further investigated whether the test reusing among multiple products with different versions could improve coverage.

Xu *et al.* [11] attempted to test multiple products of an SPL by using test suite augmentation. They have studied the influence of the order of testing multiple products on the testing performance.

Compared to these works, our project centralization technique [5] performs program transformation and tests multiple products simultaneously, which does not depend on the order in which products are tested. Combining program transformation with random testing automatically generates test cases for multiple products without the necessity of having to provide test cases.

3 Project Centralization

3.1 Overview of Project Centralization

Project centralization [10][6][7] is a general technique to merge multiple software products to share common code where possible, while resolving the version conflicts of multiple versions [2]. It uses program transformation to represent multiple software products as a single meta-product, which preserves the behavior of each product after centralization. In this way, the simultaneous program analysis, verification, and testing on multiple products can be simulated by the execution of the centralized meta-product. Our previous work [6][5] has demonstrated the effectiveness of project centralization in the verification and testing of software products with multiple versions.

In principle, the concept of project centralization can be applied to class or method granularity. Class-level project centralization performs code transformation without merging files, while method-level project centralization shares more common code by merging files while possible. As the program behavior preservation with file merg-

ing requires knowledge of underlying programming language structure, method-level project centralization has the advantage of sharing more code, at the expense of being more language-specific. On the other hand, class-level project centralization can be easily generalized to different languages.

Unit testing exercises the software under test to explore more program paths and states to uncover defects. Sharing more code by file merging is desirable because it reduces redundancies in testing the same code multiple times in each of its occurrences in different versions.[†1]

3.2 Testing Multiple Products of SPLs

In our previous work [5], we have investigated the benefit of using method-level project centralization in testing multiple software products of SPLs. We first identify all consistent class files for merging and rename those classes that cannot be merged due to semantic restrictions in the underlying program language. After renaming, we merge all consistent classes. The centralized meta-product and transformation map are then used as the input project of our adapted random testing tool Randoop [8] that supports the version switching between multiple versions. We have also implemented the code coverage analysis separation for each product by only analyzing the centralized product. In this way, we enable the testing multiple software products efficiently, which improves the code coverage by using less test generation time.

Figure 1 gives an example of method-level project centralization for three software products P_1, P_2 and P_3. In this example, both classes A and B are consistent for merging in P_1 and P_2. However, the class A is inconsistent between P_1, P_2, and P_3, due to the incompatible class attributes *field* with different types. Therefore, we rename the inconsistent classes in P_3 and merge classes in P_1 and P_2. To merge the class A, we first merge all attributes from different versions of A in P_1 and P_2. Method $m2$ only occurs in P_2. It has only one partition and could be directly merged as a method in the centralized code. The case for method $m1$ is more involved, because it occurs in P_1 and P_2 as different versions. To merge $m1$, we first create a centralized method $m1$ and rename original methods to different names. The renamed method names are *m1_P1* and *m1_P2*, respectively (see Fig. 1(b)). In the centralized method body of $m1$, we use the *switch* statement to check out and forward the corresponding method through run-time version selec-

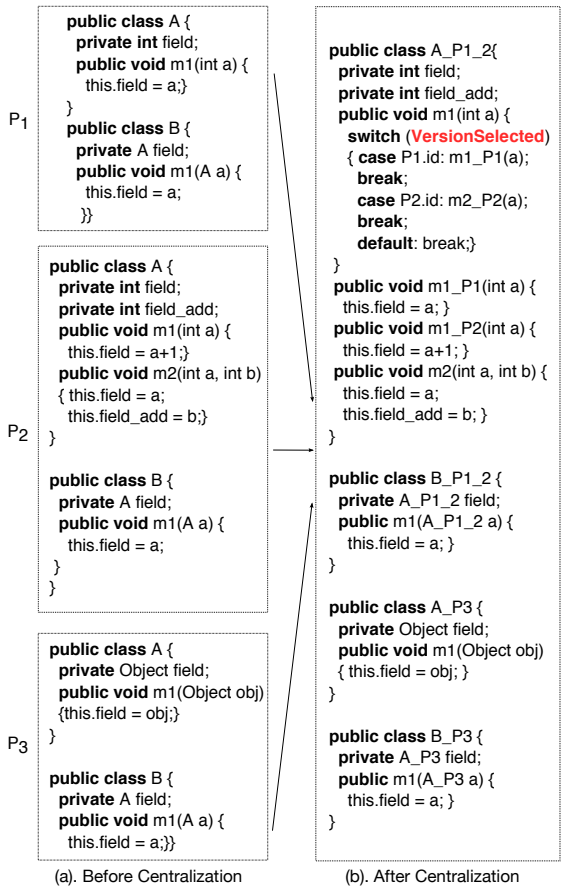

図 1 Example of Method-level Project Centralization

†1 Test sequence execution is one of most expensive activities in the automatic test case generation.

tion. Our transformation preserves the method behavior of each original product in the centralized meta-product for testing.

However, our previous work has not considered static class attributes [5]. This could cause false positives (failed tests) when testing centralized products. Such test failures do not happen when testing the original products. The reason for this is that class static attributes are unique for each class. Multiple versions of a centralized product would interfere each other on such attributes without proper transformation. For class file merging, our previous work renames all those classes cannot be merged and propagates the rename effects so that the remained consistent classes are merged. This loses chance to further share consistent classes, because classes that depend on them are renamed differently based on internal code dependencies [7]. Moreover, we test each method equally often, without favoring common classes that are shared among multiple products. In the rest of this section, we discuss our in progress solutions to these issues.

3.3 Refinement of Method-level Project Centralization

Our first refinement of method-level project centralization is to support static class attribute transformation so that each product has its own version of the static attribute in the centralized project. Like in previous work [10], we also transform each static field into an array and add one dimension if the field is an array itself. All the references of the static fields are also transformed correspondingly so that we could use the project identity to run-time access these fields.

To share as much code as possible, we refine our project centralization further by adding the re-merging phase. We leverage our D-graph representation [7] for multiple products. In the D-graph, each node has a class name cl and a constraint graph cg to store the version relation of all classes with the same name as cl so that different versions are connected by edges in cg. To centralize multiple products, we first initialize the corresponding D-graph. Since we also consider class file merging, we mark those consistent classes with no version conflict and remove the corresponding edges in the constraint graph. Then, we start the iterative version constraint equation solving stage until no *conflict edges* exist. Next, we start the second renaming and re-merging transformation phase to output the meta-product. We process each node of the D-graph in the topological order. We update their renamed references and re-check class consistency for each graph node and merge all consistent class in each node. In this way, some consistent classes after reference updates whose dependent classes are renamed differently could still be shared.

To favor test common methods, we measure the *importance* of each method by calculating its weight and use the weight to probabilistically selected each method from all the methods under test. The weight formula of a method m is defined as $Weight(m) = \sqrt{\#Shared(m)}$, where $\#Shared(m)$ represents the number of products that share m after project centralization. A more advanced weight formula could also be devised when other run-time information is available, like the coverage ratio of methods. Furthermore, we could balance the testing of multiple products with different versions based on the product or method coverage, or user-defined weights.

4 Conclusion and Future Work

In this paper, we have discussed the challenges of automatically testing multiple versions of software. We have proposed project centralization as the general framework to test the multiple software products simultaneously. Essential issues of project centralization are covered by past work, but as shown

in this paper, some refinements are still needed.

Our main future work is to finish the implementation of our proposed techniques. We are also considering to include some program analysis techniques, such as the coverage analysis, to enhance automatic testing of multiple products. We also aim at evaluating our tool on a wide range of benchmarks, such as multiple versions from software evolution and SPLs.

Acknowledgement This work was supported by the SEUT program, the University of Tokyo.

参 考 文 献

[1] Hamlet, R.: Random Testing, *Encyclopedia of Software Engineering*, Marciniak, J.(ed.), Wiley, 1994, pp. 970–978.

[2] Hnetynka, P. and Tuma, P.: Fighting class name clashes in Java component systems, *Modular Programming Languages*, LNCS, Vol. 2789, Springer, 2003, pp. 106–109.

[3] Jaygarl, H., Kim, S., Xie, T., and Chang, C. K.: OCAT: Object Capture-based Automated Testing, *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, Trento, Italy, 2010, pp. 159–170.

[4] Lehman, M. and Ramil, J.: Software evolution in the age of component-based software engineering, *Software, IEEE Proceedings*, Vol. 147, No. 6(2000), pp. 249–255.

[5] Ma, L., Artho, C., Cheng, Z., and Sato, H.: Efficient Testing of Software Product Lines via Centralization, *Proceedings of 13th ACM International Conference on Generative Programming: Concepts & Experiences*, GPCE '14, 2014 (to appear).

[6] Ma, L., Artho, C., and Sato, H.: Analyzing Distributed Java Applications by Automatic Centralization, *Computer Software and Applications Conference Workshops*, COMPSACW '13, Kyoto, Japan, 2013, pp. 691–696.

[7] Ma, L., Artho, C., and Sato, H.: Project Centralization Based on Graph Coloring, *Proceedings of ACM 29th Annual Symposium on Applied Computing*, SAC'14, Gyeongju, South Korea, 2014, pp. 1086–1093.

[8] Pacheco, C. and Ernst, M.: Randoop: Feedback-directed Random Testing for Java, *Proceedings of 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, Montreal, Canada, 2007, pp. 815–816.

[9] Paul, C. and Linda, N.: *Software Product Lines: Practices and Patterns*, Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2001.

[10] Stoller, S. D. and Liu, Y. A.: Transformations for model checking distributed Java programs, *SPIN '01: Proc. 8th international SPIN workshop on Model checking of software*, NY, USA, Springer-Verlag New York, Inc., 2001, pp. 192–199.

[11] Xu, Z., Cohen, M. B., Motycka, W., and Rothermel, G.: Continuous Test Suite Augmentation in Software Product Lines, *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, Tokyo, Japan, 2013, pp. 52–61.

[12] Zhang, S., Saff, D., Bu, Y., and Ernst, M. D.: Combined Static and Dynamic Automated Test Generation, *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, Toronto, Ontario, Canada, 2011, pp. 353–363.