http://www.diva-portal.org

Postprint

This is the accepted version of a paper presented at *Int. Conf. on Generative Programming*.

Citation for the original published paper:

Ma, L., Artho, C., Zhang, C., Sato, H. (2014)
Efficient Testing of Software Product Lines via Centralization (Short Paper).
In: *Proc. 2014 Int. Conf. on Generative Programming: Concepts and Experiences* (pp. 49-52).
GPCE 2014

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-199115

Efficient Testing of Software Product Lines via Centralization (Short Paper)

Lei Ma
The Univ. of Tokyo, Japan
malei@satolab.itc.u-tokyo.ac.jp

Cyrille Artho
AIST, Japan
c.artho@aist.go.jp

Cheng Zhang
The Univ. of Waterloo, Canada
c16zhang@uwaterloo.ca

Hiroyuki Sato
The Univ. of Tokyo, Japan
schuko@satolab.itc.u-tokyo.ac.jp

Abstract

Software product line (SPL) engineering manages families of software products that share common features. However, cost-effective test case generation for an SPL is challenging. Applying existing test case generation techniques to each product variant separately may test common code in a redundant way. Moreover, it is difficult to share the test results among multiple product variants.

In this paper, we propose the use of centralization, which combines multiple product variants from the same SPL and generates test cases for the entire system. By taking into account all variants, our technique generally avoids generating redundant test cases for common software components. Our case study on three SPLs shows that compared with testing each variant independently, our technique is more efficient and achieves higher test coverage.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools, Diagnostics, Tracing

General Terms Software reliability and quality assurance, empirical studies

Keywords Software Product Lines, automatic test generation, random testing

1. Introduction

Software product line (SPL) engineering manages a set of reusable program assets. It systematically generates families of products [14]. To assure quality, a variety of software testing techniques have been proposed to test SPLs [2, 7, 12, 15, 17].

Random testing is easy to use, effective, scalable and can be fully automated [13]. To test object-oriented programs, random testing randomly constructs object instances as the receiver and input arguments of the method under test (MUT) to exercise different paths in the MUT [4]. However, in the context of SPLs, separately performing random testing on each software product of the same SPL causes redundancies: Features shared by different products are tested repeatedly, without increasing test coverage. Furthermore, it is difficult to reuse test results among different product variants.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '14, September 15-16, 2014, Västerås, Sweden.
Copyright © 2014 ACM 978-1-4503-3161-6/14/09...$15.00.
http://dx.doi.org/10.1145/

Project centralization [10, 11] merges multiple variants of a single software product by sharing common code, preserving the behavior of each variant through code transformation. We propose to use project centralization to test SPLs, because it eliminates code redundancies by integrating multiple variants. However, existing project centralization techniques [10, 11] work on a class level. On that level of granularity, multiple variants are only shared if their classes are identical. To test SPLs, more fine-grained project centralization that can share common methods is required.

In this paper, we introduce *method-level project centralization*, which unifies and shares methods. As methods are defined in their respective classes, the new technique merges the classes and handles issues related to fields, methods, and inheritance. In general, the technique shares common code whenever possible, while preserving the behavior of each given product variants for unit testing. We implement random test case generation using Randoop [13]. In our framework, Randoop takes the centralized SPL as input and tests multiple product variants in one run. To evaluate our technique, we have conducted cases studies on 33 product variants generated from three SPLs. The results are quite promising: Compared with testing each product separately, random testing on the centralized product achieves a higher test coverage. In most cases a high coverage is achieved more quickly as well.

Although this paper focuses on testing multiple product variants of SPLs, the concept and techniques presented in this paper do not depend on domain knowledge of SPLs (such as a feature model). They can generalize to other testing scenarios for multiple similar product variants such as historical program versions from software evolution and co-evolution, and similar code branches produced by the *clone-and-own* approach.

2. Related Work

While much work on SPL testing and automatic test case generation has been done, work on fully automatic test case generation for multiple product variants from SPLs is limited.

2.1 Software Product Line Testing

When testing SPLs, test cases can be developed separately for each feature. Although it is desirable to run the prepared test cases on each generated product, this is usually not feasible due to resource limitations. Several approaches try to reduce the combinatorial product test space by product sampling and in other ways, e. g., by reducing the test executions per product [3].

Product sampling selects a representative subset of products from the valid product space of an SPL and only considers these sampled products for testing. Appropriate sampling strategies aim to fulfill given coverage criteria [2, 12, 15], with N-way combinatorial sampling [12] being the most widely used approach.

Other work uses program analysis to reduce the test executions, by running a test case only on a configuration that influences it [8, 15]. Kästner et al. [7] explore the execution strategies of a unit test for all products of an SPL without generating each product in a brute-force way. They encode the variability of an SPL either in the testing tool (a white-box interpreter) or in a meta-product that represents all products (combined with black-box testing using a model checker) to simulate test execution on all products.

Compared with these techniques, we use code transformation to combine multiple products from an SPL, improving testing efficiency by reducing redundancy in test executions. We need only a set of products as input for testing without requiring provided test cases or domain knowledge on an SPL, such as a feature model.

Other work on sharing the results of test executions exists. Xu et al. [17] use a test suite augmentation technique to test multiple products and investigate the influence of the order in which products are tested. In our approach, no specific product testing order is required, and we use code transformation on the products (rather than the test cases) to share tests among all products.

2.2 Randomized Test Case Generation

The critical step in random test case generation for object-oriented programs is to prepare the input objects with desirable object states. Most recent random techniques create the required input objects by *method sequence construction* [5, 13, 18].

Randoop [13] randomly creates a new method sequence by concatenating previously generated sequences as inputs. OCAT [5] adopts object-capture and replay techniques, where object states are captured from sample test executions, and then used as input for further testing. Palus [18] leverages a trained sequence model from combined static analysis (for method relevance) and dynamic analysis (for method invocation order) to guide test case generation.

However, these techniques are designed for a single software product; they do not share the test results among multiple products.

3. Our Approach

Program centralization merges multiple instances of the same application so that they can be executed as a single entity [16]. Project centralization extends this technique and transforms multiple products into a single project, preserving the original behavior of each product. Using project centralization, we can generate test cases for all product variants simultaneously. However, our previous centralization [10, 11] only shares a class among multiple products if it has the same implementation throughout. For SPL unit testing, a more fine-grained centralization is required to increase code sharing.

3.1 Method-Level Project Centralization

A *project* represents the code of a product variant, which has a unique project identifier and a set of classes. Each class has a unique name, a set of fields and methods, along with a set of annotations, and attributes (super classes, interfaces, etc.) [9].

Method-level project centralization transforms a set of projects, $P = \{p_1, \ldots, p_n\}$, into a single project p_{centr} such that each method and its implementation from every $p_i \in P$ is preserved in p_{centr}. Methods from different projects are generally different from each other, since they are defined in their respective declaring classes. To share as many methods as possible, we adapt our class-level project centralization [10, 11] to individual methods.

When merging classes from different projects, techniques of class-level project centralization are first applied, representing all classes with the same name as a separate set. For each such a set of classes $C = \{cl_1, \ldots, cl_k\}$, where each $cl_i \in C$ occurs in P and all of them have the same name, the method-level project centralization merges its classes that satisfy the following conditions:

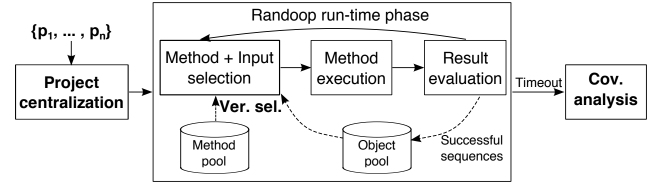

Figure 1. Centralization-based test case generation.

1. The classes are consistent in their class attributes (class versions, super classes, interfaces, etc.).
2. All fields with the same name are consistent. A field $cl.f$ of class cl is consistent with another field $cl'.f'$ of class cl', if $cl.f$ and $cl'.f'$ have the same type, annotations, and attributes.
3. All methods with the same name and descriptor are consistent.[1] Method consistency of normal method is similar to that of fields, except that the method bodies may differ.
   However, the static initializer `<clinit>` of each class in C needs special treatment, as it is executed only once, at class load time, even if the initialization of multiple variants of a class is to be simulated. Static initializers can be considered consistent if their code is identical, or if their method body instructions are totally ordered with respect to the subset relation.

Before actually merging any classes, we rename the classes that do not satisfy the above conditions, marking them as distinct classes from different products. We also update any references to these classes accordingly.

Method-level centralization operates on the set of consistent classes with the same name, $C = \{cl_1, \ldots, cl_k\}$, and merges their common code. The first three steps are: (1) create a centralized class with the same name and attributes as a class $cl_i \in C$, (2) use the union of all fields of the classes in C to synthesize the set of fields of the centralized class, (3) create a *proxy method* that forwards method calls to the right implementation.

A proxy method is created as follows: For all methods $M = \{m_1, \ldots, m_l\}$ that occur in C and have the same name and descriptor, we first partition M into a set of distinct methods $MP = \{mp_1, \ldots, mp_h\}$ according the the method body (implementation) of the methods in M. All methods in M with the same method body are mapped to the same element in MP. A *transformation map* keeps track of project identities by associating each method in MP with the project identities of all matching methods in M.

Then, we create a new *proxy method*, which has the same attributes as all the methods in M except for the method body. We rename the representative method from each partition to a new unique name to distinguish the different variants. We add both proxy methods and the renamed methods into the centralized class. We also keep the project identities for each representative method to remember which projects it represents. In the method body of the proxy method, we use a switch statement that checks the project identity against the transformation map, and forwards a method invocation from the proxy method to the corresponding renamed method. This allows testing different versions of a method (with both same name and descriptor) by only providing its corresponding project identity and the proxy method. The proxy method accurately forwards the call to the method matching the given project.

3.2 Integration with Randoop

Randoop [13] is a state-of-the-art random test case generation tool. Given the program to test, Randoop first extracts all publicly vis-

[1] A method descriptor in Java bytecode includes both the return type and the input parameter types of a method.

ible methods and puts them into a *method pool*, which contains all methods under test (see Fig. 1). To test multiple product variants $P = \{p_1, \ldots, p_n\}$ simultaneously, we first perform project centralization on P, and feed both the centralized project and the transformation map into our modified version of Randoop.

Randoop starts by randomly selecting a method $m(T_1 \ldots T_k)$ to test from its method pool. All input objects with type $T_1 \ldots T_k$ must also be prepared to test m. Randoop randomly selects the corresponding inputs from its object pool and concatenates previously known input sequences to derive these objects, to test m. Upon successful input construction, m is executed. Execution results of each method call are analyzed against a few predefined contracts. If the generated sequence is new and its execution does not cause any failures, Randoop adds this *successful sequence* to the object pool (see Fig. 1). Randoop continues to test more methods until a time limit is hit.

We modify Randoop such that when a method m is selected from the method pool, we randomly set the version, which is represented by the corresponding project identity. Using the transformation map, we select a version from those projects that contain m, and execute the generated test sequence using the selected version.

When memorizing executed test sequences, we also memorize the selected version of each successful method sequence, so that we can generate these tests as JUnit test code for further analysis. Instead of re-executing the same method sequence repeatedly, we can change the version of each successful sequence in the object pool when creating new sequences.

As the version of a method sequence can change according to the random selection, it is possible to introduce extra method sequences and execution behavior that do not exist when testing each project separately. For example, a method $m(T\ t)$ from a project might use a method sequence (returns an object with type T) that can only be generated in another project. As long as it does not matter how test data is prepared, such sequences produce correct results. If methods heavily depend on product-specific preconditions and invariants, though, a method call sequence with mixed versions may sometimes produce false positives or false negatives. We will investigate this issue further in future work.

After Randoop hits the time limit, we recover the code coverage for each product by only analyzing the coverage of each method in the centralized project according to the transformation map.

4. Case Study

To evaluate our work, we have implemented a tool and applied it to 33 products from three SPLs. Our implementation of method-level project centralization is based on Java bytecode transformation using the ASM library [1], and it is integrated with Randoop and JaCoCo v0.6.4[2] which is used for code coverage analysis. In our case study, we investigate two major research questions:

RQ1: Is project centralization effective in sharing the common code among multiple products?

RQ2: Does testing using project centralization increase code coverage, compared with testing each sampled product independently?

4.1 Evaluation Subject and Settings

We evaluate our tool on three SPL subjects that were developed and used in previous studies based on FeatureHouse product generation (see Table. 1). All the selected SPLs are currently included in the release of FeatureIDE v.2.7.0[3].

Each of these selected subjects is accompanied by both a feature model and source code. For example, GPL has 38 features and 42 constraints in its feature model, which can generate 156 valid products in total by selecting different feature configurations. As pairwise feature coverage is widely used in SPLs as the product sampling approach, we therefore sample valid products with 100 % pairwise feature coverage by using SPLATool v0.3 [6].

Table 1. Case study subjects: Size and complexity metrics.

SPL desc.	Classes (*.java)	LOC	Features	Constraints	# Products (total)	# Products (pairwise)
Elevator	19	1,223	7	3	20	6
GameOfLife	39	1,702	23	17	65	8
GPL	57	2,957	38	42	156	19

To compare our approach with independent test case generation for each product, we perform project centralization on the sampled products to generate the centralized project for each SPL. We run our modified Randoop tool chain on the centralized project, while running the original Randoop on each of the sampled product separately.[4] We run each configuration for $1,000$ seconds, after which no noticeable coverage improvement can be observed anymore.

4.2 Results

Table 2 summarizes the results of our experiments. Column two gives the number of sampled products for each SPL. Columns three and four give the total number of classes (*.class) and methods in all sampled products, respectively. Columns five and six show the corresponding number of *public* classes and *public* methods. Column seven lists the total number of branches. Column eight is the number of classes after performing centralization on all sampled products, while columns nine and ten describe the project size before and after centralization. Finally the average method coverage and branch coverage for all sampled products by both approaches are listed in the next four columns, followed by the centralization transformation time and the total execution time of each experiment (the non-centralized cases were run for the full duration in *each* configuration).

In all three cases, centralization shares common code and needs less storage. The centralized project takes 21.6 %, 21.3 %, and 20.3 % of the space required by original sampled products of Elevator, GameOfLife, and GPL. Centralization improves both method and branch coverage, compared with independently testing each project, even though we allot k times the test case generation time to individual testing of k sampled products to have a similar number of test cases in each setting. Our centralization transformation is also efficient and finishes in a few seconds (insignificant compared with the test generation time) in our studied subjects.

4.3 Discussion

To understand the improvement of method coverage, we need to review the test case generation procedure for each method. To test a method, Randoop randomly selects input objects from the object pool. If there is no object with a compatible type, Randoop skips that method. Randoop adopts a fixed method pool. If a method m requires an input object that is not returned by any method in the method pool, m may never be tested. This often happens when testing each sampled product independently.

After centralization, public methods from multiple sampled products are put into the method pool, and all generated method sequences are shared among all products. This increases the chance to cover more methods, by providing more diverse object types generated by multiple products. Therefore, even if independently testing a product p cannot generate an object typed T to cover a method

[2] `http://www.eclemma.org/jacoco/`

[3] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

[4] Class `GolView` from *GameOfLife* is excluded for individual product testing, because it constantly creates GUI frames that crash the local OS.

Table 2. Results of our experiments. All experiments were run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running Mac OS X 10.9.3 and Oracle's Java VM (JVM), version 1.7.0_21 with a memory limit of 3 GB for the JVM.

Sampled pairwise prod.	#Sampled products	#All		# Public		#	#cl.	Size (KB)		Avg. cov. non-centr.		Avg. cov. centr.		Transf.	Exec. time (s)	
		cl.	m.	cl.	m.	branch	centr.	non-centr.	centr.	m. [%]	br. [%]	m. [%]	br. [%]	time (s)	non-centr.	centr.
Elevator	6	90	694	72	552	1,658	15	187.0	40.3	88.7	83.9	89.2	88.9	1.3	6,000	1,000
GameOfLife	8	326	1,073	122	833	1,636	57	369.3	78.5	59.6	45.4	64.9	53.9	2.3	8,000	1,000
GPL	19	285	1,543	195	1,264	1,062	21	283.8	57.5	84.7	59.1	86.2	66.8	2.2	19,000	1,000

$m(T\ t)$ in p, $m(T\ t)$ may still be covered by using an object instance typed t from another product p' in the centralization-based testing. Mixing data from different products therefore increases the availability of data types, for example, when public methods of a given product are removed in other products. Although our initial case studies have not found any adverse effects, we will investigate such data reuse in greater depth in the future, on software that depends on product-specific preconditions and invariants.

However, there exist a few cases where centralization decreases coverage. Centralization increases the method testing space, by introducing more methods and additional product version dimensions. Our current strategy of both method and version selection adopts a uniform distribution. However, the difficulty of covering branches of different sampled products varies. A better selection strategy to select methods based on coverage and sharing ratio is likely to improve the effectiveness of our approach.

4.4 Threats to Validity

The representativeness of selected subjects is the primary external threat to validity. We carefully select three SPLs from different categories that have been widely used in previous studies. We also use their recent implementation based on FeatureHouse.

Another external threat to validity is caused by the randomness of Randoop. We fix and use the same random seed and run each configuration long enough to diminish this threat.

The main threat to internal validity is caused by potential bugs of our tool implementation. We decrease this threat by performing unit testing and using the internal verification tool of ASM to check the correctness of our code transformation.

5. Conclusion and Future Work

In this paper, we propose method-level project centralization and its integration with random testing to test multiple product variants from SPLs. Our technique shares common code whenever possible, while preserving the behavior of each method for unit testing. The evaluation on three SPLs shows the promising of our approach in sharing common code and obtaining higher code coverage, compared with testing each product independently.

Our work is a general testing technique to handle multiple products with different versions. These products could come from an SPL, software evolution, or by the clone-and-own approach. Our tool does not include a feature model. However, it would be interesting to compare our technique with some feature-ware automatic test generation techniques for SPLs and also automatic test generation techniques for multiple versions from software evolution (such as regression testing). Other work includes investigating the impact of using data from different products in a single test case. To evaluate this, we will conduct studies on the quality of generated test cases, and the bug-finding ability of our approach.

Acknowledgments

This work was partially supported by the ⌈SEUT⌋ program from the University of Tokyo. We also thank the anonymous reviewers for their insightful suggestions to improve this paper.

References

[1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[2] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *Proc. 14th Int. Conf. on Software Product Lines*, SPLC'10, pages 241–255, South Korea, 2010.

[3] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida. Strategies for testing products in software product lines. *SIGSOFT Softw. Eng. Notes*, 37(6):1–8, Nov. 2012.

[4] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[5] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: Object capture-based automated testing. In *Proc. 19th Int. Symposium on Software Testing and Analysis*, ISSTA '10, pages 159–170, Italy, 2010.

[6] M. F. Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proc. 16th Int. Software Product Line Conf.*, SPLC '12, pages 46–55, Brazil, 2012.

[7] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *Proc. 4th Int. Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8, Dresden, Germany, 2012.

[8] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Tenth Int. Conf. on Aspect-oriented Software Development*, AOSD '11, pages 57–68, Brazil, 2011.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addi. Wesl. Prof., 2013.

[10] L. Ma, C. Artho, and H. Sato. Analyzing distributed Java applications by automatic centralization. In *Computer Softw. and Applications Conf. Workshops*, COMPSACW'13, pages 691–696, Japan, 2013.

[11] L. Ma, C. Artho, and H. Sato. Project centralization based on graph coloring. In *Proc. ACM 29th Annual Symposium on Applied Computing*, SAC'14, pages 1086–1093, South Korea, 2014.

[12] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proc. 14th Int. Conf. on Software Product Lines*, SPLC'10, pages 196–210, South Korea, 2010.

[13] C. Pacheco and M. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion*, OOPSLA'07, pages 815–816, Canada, 2007.

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

[15] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proc. 15th Int. Conf. on Fundamental Approaches to Software Engineering*, FASE'12, pages 270–284, Estonia, 2012.

[16] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Int. SPIN workshop on Model checking of software*, SPIN '01, pages 192–199, Canada, 2001.

[17] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel. Continuous test suite augmentation in software product lines. In *Proc. 17th Int. Softw. Product Line Conf.*, SPLC '13, pages 52–61, Japan, 2013.

[18] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. Int. Symp. on Softw. Testing and Analysis*, ISSTA '11, pages 353–363, Canada, 2011.