Dynamic Analysis of Shared Execution in Software Product Line Testing

Bo Wang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University,
Beijing, China
wangbo_15@pku.edu.cn

ABSTRACT

Software product line (SPL), a family-based software development process, has proven to be a more effective technology than single software systems. Testing SPL products individually is redundant for product lines testing. Meanwhile, the complexity of systematically testing SPL programs is combinatorial, which limits the scalability of testing SPL.

In this paper, I will introduce the idea of accelerating SPL testing by dynamic analysis. I intend to investigate a dynamic analysis of SPL testing, which analyzes all variants during the execution of a SPL, and forks the execution only a variant leads to a different state. In addition, to get the preliminary result, we can encode a family of variants into a single program to simulate a realization of SPL testing. The experimental result collected on the mutated C program, *printtokens* in Software-artifact Infrastructure Repository, shows our approach can accelerate the SPL testing with a speedup up to 4.83X than testing each variants at a time and pruning 87% of the variants.

Keywords

Software product line testing; Dynamic analysis; Shared execution

1. INTRODUCTION AND MOTIVATION

Software Product Line (SPL) is an important technique in software engineering, based on the idea that related software products of a family can share some core functionality, while each of them differs in some specific variants.

The main challenge of in SPL testing is to deal with the combinatorial explosion induced by the number of possible products, which will be 2^N for N variants in the worst case. For this reason current attempts to test SPL are unsound and incomplete even in simple cases. Testing each variant in isolation in a brute-force fashion dose not scale for SPL projects with more than a few variants. And the others test SPL projects against some specific variants, sacrificing variant coverage and soundness for efficiency. The techniques of static pruning redundancy, Varexj [7], can filter invalid configurations and remove valid but unnecessary configurations for further test, Major[5] removes the useless testing running, and mutation schemata[8] remove the compiling redundancy. There are several dynamic approaches base on module analysis. SPLat[6] records the configuration options that are accessed during the test execution, and ignore the configurations that only differ on the unaccessed options. Devroey, X. et al. presents an implementation performing mutation and test case execution at the model level by sharing the same models[2, 3]. There are also general approaches to reuse computed result of function or basic block[4, 1].

I intend to propose a code based dynamic analysis of SPL testing for lossless acceleration. Unlike static approaches that analyzes variant in a prepass, dynamic analysis of SPL testing starts with a process representing all variants, and forks the execution when a variant leads to a new system state. The main asset of dynamic analysis is that the major redundancy in part of the execution can be removed, because the executions before the changed state are shared. If a test turns on two variants, the executions before the first encountered variant are completely the same. However, we cannot prune this redundancy by static analysis. We can benefit from sharing the same execution against one test. This approach also can subsumes aforementioned static analysis methods for the further reduction.

To get the preliminary result, we implemented our approach on C programming language and evaluated it on *printtokens*, one subject of Software-artifact Infrastructure Repository. In mutation testing, each mutants holds a slight change and shares the remaining part of the original program. Although the granularity is at statement level, mutations can be treated as features in term of SPL. By encoding mutations into the program, we get a family of SPL products. The evaluation shows that our approach can accelerate the SPL testing with a speedup up to 2.22X than testing each variant at a time.

2. RESEARCH ISSUES AND METHODOLOGY

We first describe the redundant execution in standard SPL testing that dynamic analysis of SPL testing can avoid. 1(a), 1(b) and 1(c) describes the execution of three variants. The circles represent system states and the states with the same number are the same. The arrows represent the transitions of states after the executions of code blocks. At state 2, the three variants execute three different code blocks, and lead to either state 3 or state 5. Though variant 1 and variant 3

SPLC '16, September 16-23, 2016, Beijing, China
© 2016 ACM. ISBN 978-1-4503-4050-2/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2934466.2966351

executes different code blocks, the two code blocks lead to the same system state. As a result, state 1, state 2, and the transitions leading to them are completely the same among the three variants, forming redundant computations. Also, state 5 and the transitions leading to it are the completely same among variant 1 and variant 3, also forming a redundant computation. The redundant states are denoted as solid circles or grey circles based on their degrees of redundancy.

Since the three eventually lead to different states, the redundancy happens only in part of the computation and cannot be removed by static approaches. Dynamic analysis of SPL testing removes this kind of redundancy by dynamically forking the execution. In dynamic analysis of SPL testing, we start with one main process that represents all variants, as shown in 1(d) . At each state, we check the code blocks that each represented variant is going to execute. If the code blocks are the same, we execute the block normally. If the statements are different, we analyze their different executions and fork new processes if necessary. At state 2, the three variants have three different code blocks to execute, so we need to analyze the executions of the code blocks.

To perform the analysis, we first conduct a trial execution of the statements and collect their changes to the system state. Then we cluster their changes to the system state into equivalence classes. Two variants are in the same equivalent class if their changes lead to the same new system state. In the example, variant 1 and variant 3 are clustered into one equivalence class and variant 2 is in another class.

If number of equivalence class is more than one, say n, we fork n - 1 new processes. Each forked process represents the variants in one equivalence class, and we apply the change from the equivalence class to the state of the forked process. Finally, the change from the remaining equivalent class is applied to the original process, and the original process now represents only the variants in the remaining class. This process continues for each process until all processes terminate. In the example, we fork a new process for the equivalent class of variants 1 and 3, and apply the change from the equivalent class to change the state to state 5. In the original process, we apply the change of the other equivalent class to change the state to state 3, and the original process now represents only variant 2. Then the two processes continue their execution independently. Again, at state 5, the newly forked process detects that the two variants need to execute different statements and the two statements lead to different system states, so this process again forked into two processes, each representing one variant.

From 1(d) we can see that, the transitions leading to states 1, 2, and 5 are only performed once, avoiding the aforementioned redundant computation in the standard execution of SPL testing.

3. PRELIMINARY KEY RESULTS

The result shows our approach had significant speedups on the subject over existing techniques. Comparing with the brute-force method , our dynamic approach decreased the number of mutation from 853 to 110, and the time usage decreased from 29s to 6s, with a 4.83X speedup.

4. WORK PLAN

So far, we have implemented a preliminary evaluation of our approach based on mutation testing. In the next year, we will verify our approach on more and larger projects, first on the mutation testing area, then verify the dynamic approach on the SPL area. After these work finish, we intent to commit our work to software engineering related conferences. In the future, we will expand the dynamic analysis approach to other fields.

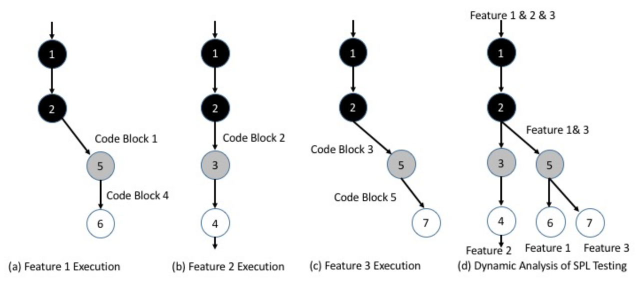

Figure 1: Dynamic Analysis of SPL Testing with Forking

5. REFERENCES

[1] D. Conners and W.-m. W. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO*, pages 158–169, 1999.

[2] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P.-Y. Schobbens. A variability perspective of mutation analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 841–844. ACM, 2014.

[3] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 655–666. ACM, 2016.

[4] M. A. Hammer, U. A. Acar, and Y. Chen. Ceal: a c-based language for self-adjusting computation. In *PLDI*, pages 25–37. ACM, 2009.

[5] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.

[6] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D&#039;Amorim. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 257–267, New York, NY, USA, 2013. ACM.

[7] J. Meinicke. Varexj: A variability-aware interpreter for java applications, 2014.

[8] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139–148. ACM, 1993.