Poster: Grafter: Transplantation and Differential Testing for Clones

Tianyi Zhang, Miryung Kim
University of California, Los Angeles
{tianyi.zhang,miryung}@cs.ucla.edu

ABSTRACT

Code clones are common in software. When applying similar edits to clones, developers often find it difficult to examine the runtime behavior of clones. The problem is exacerbated when some clones are tested, while their counterparts are not. To reuse tests for similar but not identical clones, Grafter transplants one clone to its counterpart by (1) identifying variations in identifier names, types, and method call targets, (2) resolving compilation errors caused by such variations through code transformation, and (3) inserting stub code to transfer input data and intermediate output values for examination. To help developers examine behavioral differences between clones, Grafter supports fine-grained differential testing at both the test outcome level and the internal program state level. Our evaluation shows that Grafter can successfully reuse tests and detect behavioral differences. The tool is available for download at http://web.cs.ucla.edu/~tianyi.zhang/grafter.html and the demo video is available at https://youtu.be/1iqAeuM8s3U.

KEYWORDS

Code clones, software transplantation, differential testing

1 INTRODUCTION

Code reuse via copying and pasting is a common practice in software development. Prior studies show that up to 25% of code in modern software contains *code clones*—code similar to other code fragments elsewhere. Manually adapting clones is error-prone. Therefore, developers often rely on regression testing to check for inconsistent or missing edits on clones [7]. However, a lack of tests exacerbates such situation, where some clones are tested while their counterparts are not. In fact, our study shows that, in 46% of studied clone pairs, only one clone is tested by existing tests, but not its counterpart [6]. No existing techniques can help programmers reason about runtime behavior differences of clones, especially when clones are not identical and when clones are not tested. In the absence of test cases, developers can only resort to static analysis techniques to examine clones [3, 7], but these techniques are limited to finding only pre-defined types of cloning bugs such as renaming mistakes or control-flow and data-flow inconsistencies.

This paper describes a clone transplantation and differential testing technique called Grafter, based on our ICSE 2017 paper [6]. Given a clone pair and an existing test suite, Grafter contrasts the runtime behavior of clones using the same test. Test reuse for clones is challenging because clones may appear in the middle of a method without a well-defined interface (i.e., explicit input arguments and return type), which also makes it hard to directly adapt test for reuse. Grafter identifies input and output parameters of a clone to expose its de-facto interface and then grafts one clone in place of its counterpart to exercise the grafted clone using the same test. Grafter tracks the test status (e.g., pass or fail) as well as program states during the test so that a user can compare these values to examine the behavioral difference between clones.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195038

2 APPROACH AND TOOL SUPPORT

Similar to how organ transplantation may bring incompatibility issues between a donor and its recipient, a grafted clone may not fit the context of the target program due to variations in clone content. Grafter takes four steps to ensure the type safety of clone transplantation.

Variation identification. Grafter first performs inter-procedural analysis to identify the input and output parameters as well as method calls referenced by each clone and its subroutines. Grafter then matches the input and output parameters and method calls between clones and detects the parameters and method calls that are defined in one clone but not in the counterpart clone.

Code transplantation. To resolve compilation errors caused by clone variations, Grafter applies five transplantation rules to handle variable name variation, method call variation, variable type variation, expression type variation, and recursion.

Data propagation. Similar to how surgeons reattach blood vessels to ensure the blood in the recipient flows correctly to the vessels of the transplanted organ, Grafter also needs to make sure the input data flows correctly into the grafted clone and the output data flows back to the same assertion check of the original test. Therefore, Grafter inserts stub code to ensure that (1) newly declared variables consume the same input data as their counterparts in the recipient and (2) the updated values flow back to the same test oracle.

Figure 1 shows a grafted clone with stub code generated by Grafter for data propagation. Lines 307-311 are stub code to propagate the input data from `excludes` and `excludeList` to the corresponding variables, `includes` and `includeList`, in the grafted clone. Lines 313-314 and 323-324 are stub code to propagate the updated value from the affected variable `includeList` back to `excludeList` at the two exits of the grafted clone.

Differential testing. Grafter supports behavior comparison at two levels. The test-level comparison runs the same test on two clones and compares the test outcomes. The state-level comparison compares the intermediate program states for affected variables at the exit(s) of the clones. Grafter instruments code clones to

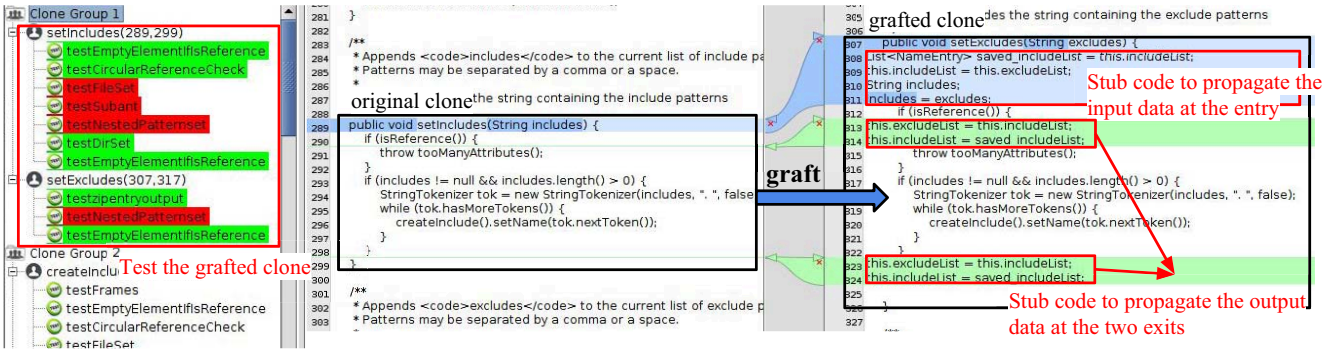

Figure 1: Experimenting the clone transplantation in the Grafter GUI. A user can view and test the grafted clone. The grafted clone passes the test cases colored in green but fails the test cases colored in red.

capture the updated program states at the exit(s) of clones. Grafter uses the XStream library[1] to serialize the program states of affected variables in an XML format. Then it checks if two clones update corresponding variables with the same values.

Grafter is implemented as a Java desktop application. Both the tool and the evaluation dataset are publicly available.[2]

3 EVALUATION

We evaluated Grafter on 52 pairs of nonidentical clones from three open-source projects: Apache Ant, Java-APNS, and Apache XML Security. Grafter successfully grafts 49 out of 52 pairs of clones without inducing compilation errors. Successfully reusing tests in 94% of the cases is significant, because currently no technique enables test reuse for nonidentical clones appearing in the middle of a method. Grafter inserts up to 33 lines of stub code (6 on average) to ensure type safety during grafting, indicating that code transplantation and data propagation in Grafter are not trivial.

To assess fault detection capability, we systematically seed 361 mutants as artificial faults using the MAJOR mutation framework.[3] We use a static cloning bug finder [3] as a baseline for comparison. Grafter is more robust at detecting injected mutants than the static approach—31% more using the test-level comparison and almost 2X more using the state-level comparison. Grafter's state-level comparison also narrows down the number of variables to inspect to three variables on average. Therefore, Grafter should complement static cloning bug finders by detecting runtime behavioral discrepancies. Our grafting technology also have the potential to assist code reuse and repair [1, 5]. The evaluation is detailed in [6].

4 RELATED WORK

The most related test reuse technique is Skipper. Skipper copies associated portions of a test suite when developers reuse code [4]. It determines relevant test cases and transforms them to fit the target system. Skipper is built on Gilligan [2] and requires users to provide a *pragmatic reuse plan* to establish the mapping of reused entities (e.g., classes, methods) from the original system to the target system. Skipper assumes that clones are full-feature clones at the level of methods and classes, making it difficult to apply to sub-method level clones. The way Grafter grafts clones for test reuse resembles existing software transplantation techniques [1].

5 SUMMARY AND FUTURE WORK

Up to a quarter of software systems consist of code clones from somewhere else. However, the cost of developing test cases is high, which makes test reuse among similar code attractive. This paper showcases Grafter, a test transplantation and runtime behavior comparison approach for clones. Our evaluation shows that Grafter's code transplantation succeeds in 94% of the cases and its fine-grained differential testing feature enables users to inspect the runtime behavior differences of clones.

To address the challenge of reusing test among sub-method clones, Grafter transplants source code of clones rather than test cases. However, transplanting tests makes it easier for developers to understand adapted tests and also reduces the risk of porting a clone to an unintended, different testing context. As future work, we will investigate how to transplant tests directly as opposed to transplanting a clone. Such a technique will enable automated test adaption for clones with minimal human intervention.

[1] http://x-stream.github.io/

[2] The tool and dataset are available at http://web.cs.ucla.edu/~tianyi.zhang/grafter.html

[3] http://mutation-testing.org/

REFERENCES

[1] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269. DOI: http://dx.doi.org/10.1145/2771783.2771796

[2] Reid Holmes and Robert J. Walker. 2012. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology* 21, 4 (Nov. 2012), 20:1–20:44. DOI: http://dx.doi.org/10.1145/2377656.2377657

[3] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 55–64.

[4] Soha Makady and Robert J. Walker. 2013. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice & Experience* 43, 9 (Sept. 2013), 1039–1070. DOI: http://dx.doi.org/10.1002/spe.2134

[5] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[6] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 665–676.

[7] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 111–122.