Live Feedback on Behavioral Changes

Gustavo Soares*, Emerson Murphy-Hill†, Rohit Gheyi*
*Department of Computing Systems, Federal University of Campina Grande, Brazil
{gsoares, rohit}@dsc.ufcg.edu.br
†Department of Computer Science, North Carolina State University, USA
emerson@csc.ncsu.edu

***Abstract*—The costs to find and fix bugs grows over time, to the point where fixing a bug after release may cost as much as 100 times more than before release. To help programmers find bugs as soon as they are introduced, we sketch a plugin for an integrated development environment that provides live feedback about behavioral changes to Java programs by continuously generating tests, running the tests on the current and previous versions of the program, and comparing the results. Such a tool would allow programmers to better understand how their changes affect the behavior of their programs. As a proof of concept, we developed a prototype that found a bug that remained undetected by pair programmers working on JHotDraw in a previous study. Had the programmers performed this change with our plugin, they would have been notified about the bug as soon as they introduced it.**

***Index Terms*—Live programming, refactoring, testing**

I. Motivation

Change is inevitable in software because programmers must implement new features and fix bugs. At the same time, they must also perform refactorings to make these changes easier. However, such changes may introduce bugs that persist uncaught for a long time. As Boehm and Basili [1] observed, the costs to find and fix bugs grows over the time, often becoming 100 times more expensive after delivery. Testing plays an important role in finding bugs, and to make testing more frequent, Saff and Ernst [2] proposed an approach that continuously runs tests in the background as the programmer changes the code. They have shown that this approach helps to reduce the number of bugs that programmers introduce while editing. In the tool's evaluation, though, several participants mentioned the fact that, in order for this tool to be effective, the programmer must have already written good tests.

In practice, programmers often do not have sufficient test cases to catch every bug. For example, programmers accidentally introduced a bug while refactoring exception-handling code in the JHotDraw application; this bug was not caught by the programmers nor JHotDraw's test suite [3]. In that study, two programmers working as a pair sought to extract exception-handling code from several classes into a new class. To each of these classes, they added a new field responsible for handling exceptions and replaced the original exception-handling code with an invocation of a method on the new field. Although the programmers believed that this change preserved the behavior of their program (and their test suite confirmed that belief), they forgot an important step: because several of the original classes were `Serializable`, the new field's class must also be `Serializable`. The reason that this is a bug is because when an object gets serialized and one of its fields is not `Serializable`, a `NotSerializableException` is thrown. However, the bug persisted unnoticed by the pair programmers, the tests, and the researchers until several months later when we retrospectively analyzed their changes using our SafeRefactor tool [3].

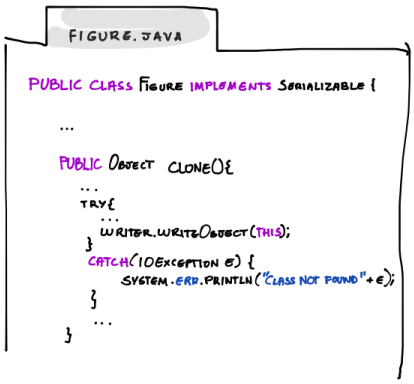
This example got us thinking – could the original programmers have used SafeRefactor to notice this bug and fix it immediately? The answer is "yes" in theory, but practically it would be cumbersome. To understand why, one has to understand how SafeRefactor works: it works by generating tests for the code before and after a change, runs the tests on both versions, and then compares the results. If the results are the same, SafeRefactor improves programmer's confidence that this change is a refactoring, but if the results are different, SafeRefactor reports it as a behavioral change. The reason why this process is cumbersome is manifold: a programmer must remember to run it; must record and explicitly choose the program versions to compare; and then must wait a significant amount of time for SafeRefactor to generate tests, compile them against both versions of the code, and then run the tests and return the results. These activities would likely distract programmers from the task at hand.

II. Sketch

In this section, we sketch the idea of a plugin for the programmer's integrated development environment (IDE) that would help programmers distinguish between refactorings and behavioral changes in a live, non-distracting way. The idea is simple: to use SafeRefactor to continuously compare the current version of a program against prior versions, then inform the programmer which changes were refactorings and which were not, and thus help the programmer confirm or disconfirm her beliefs about her changes.

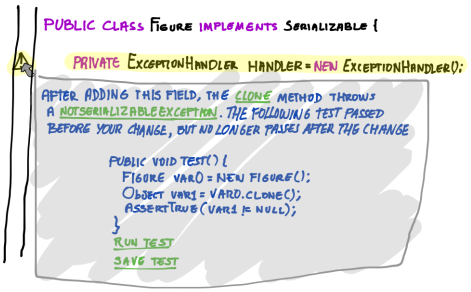
Our proposed approach is much like continuous testing in that both approaches run tests automatically in the background of a programmer's IDE. However, there are two main differences between continuous testing and our proposed approach. First, instead of using the program's existing test suite, we aim to automatically generate tests. Second, the goals of the two approaches are different. With continuous testing, the goal is to compare the program's behavior against the behavior specified in the test suite. In contrast, with our approach, the goal is to

compare the program's current behavior against the program's behavior in the past.



(a) The Figure class before being refactored.

(b) The plugin automatically detects a behavioral change, highlighting the code that was changed, and shows a warning icon.



(c) When passing the cursor over the warning icon, the plugin shows the behavioral change found and the test case that reveals the bug.

Fig. 1: Mockup of the user interface of our plugin.

To illustrate how our approach would work, let us return to (a simplified version of) the JHotDraw example. Suppose that the pair of programmers has our plugin installed in their IDE, and are working on the code in Figure 1(a). They begin by taking the `System.err.println(''Class Not Found''+ e)`, put it into a new class `ExceptionHandler`, then add a field that contains an `ExceptionHandler` to the `Figure` class (see Figure 1(b)). After adding the new field, our plugin identifies that the behavior of the code was changed, highlights the code changes between the versions, and shows a warning (Figure 1(c)). When the programmer moves the cursor over the warning icon, the plugin describes the behavioral change, and shows the test case that revealed it. The pair programmers then discusses the problem, makes `ExceptionHandler` implement `Serializable`, and the bug is removed a few seconds after it was introduced.

III. Challenges

This simple idea is more complicated than it appears. In this section we describe challenges that we identified while developing a prototype.

Defining a version. Our approach requires two discrete versions of a program to compare, yet programming is fairly continuous. At the finest level of granularity, we could consider a new version being created each time the programmer types a character, similar to how IDEs automatically give feedback about compilation errors. However, it does not seem worthwhile to run tests on an uncompilable program. Another way to define versions is that a new version is created on every changes if and only if the program compiles after the change. However, even this may be too aggressive – if a programmer types inside of a string literal, every key press would generate a new version. While our goal is to provide rapid feedback, it may be that feedback that is too frequent may be annoying to the programmer.

Which versions to compare. Sometimes programmers may want to compare the behavior of the program not against the immediately previous version, but against an earlier one. For instance, in a previous study [4], we observed how programmers performed the Extract Method refactoring. One of the participants first cut the statements to be extracted (step 1), pasted them after their original method (step 2), surrounded these statements with a new method name and brackets (step 3), added parameters to the new method declaration (step 4), and finally inserted the new method invocation at the extracted statements' original place (step 5). In this scenario, comparing the behavior between each consecutive version would mark every change as a non-refactoring, yet all 5 changes, taken as a whole, constitute a complete refactoring. Thus, it would be helpful for the plugin to check for behavioral changes not only against the immediately previous version, but also against other previous ones. The question then becomes, how far back in the program's history should the plugin look to make comparisons? If the plugin looks too far back in the history, it may report behavioral changes that are inconsequential to the programmer. If the plugin does not look back far enough, it may incorrectly identify a refactoring as a behavioral change, as in the example of the 5-step change above.

One approach is to demand that the programmer explicitly states the earliest version to compare with. This could be done by pressing a "start" button before the programmer make a coherent set of changes. However, our previous research [4] suggests that sometimes programmers do not realize that they are going to apply a refactoring, missing the opportunity to use our plugin. An alternative approach is to compare the behavior of the current version against all the previous ones, up until an implicit "start" point, such as when a file was opened or saved.

How to present behavioral changes over multiple versions. Explaining to the programmer that her previous change was a refactoring is easy; explaining that some combination of

previous changes contains *some* refactorings is not. Suppose a programmer is performing the Extract Method refactoring as described in the previous subsection. In the final step, when she adds the method invocation to complete the refactoring, we would like the plugin to be able to say that although this last change was not a refactoring, the transformation as a whole was. We aim at designing a user interface that allows the user to easily visualize the refactoring and non-refactoring changes that were applied during a programming session, yet how to design such a user interface is not obvious.

How to generate tests. Since the effectiveness of our approach on detecting behavioral changes depends on the quality of the generated tests, it is important to have a good test generator. Researchers have been proposing a number of approaches for test generation. We used the random test generator Randoop [5] to generate tests for SafeRefactor, which has helped us to detect a number of behavioral changes in Java programs [3], [6]. Other options include Seeker, a test generator that combines static and dynamic analysis for generating test suites [7]. While Seeker tends to generate tests with higher code coverage than Randoop, Randoop tends to have better performance, which is important for our live programming application. We plan to evaluate several test generators to find an acceptable balance between performance and test quality.

There are at least two approaches that we can use to optimize test generation for our plugin. First, we do not have to test the entire program for every change. If a method is changed, at least the plugin should test that method. In other cases, the plugin may need to tests unchanged parts of the program; for instance, a change to a field may have an impact on the methods that use this field. A way to deal with this problem is to use change impact analysis to calculate which methods are impacted by a code change. For instance, we could use the approach used by Wloka et al. [8] to assess the impact of a change so that we could focus on testing only the methods impacted by the change. Second, we do not have to generate completely new tests for each version: any previously generated test that can compile against both versions under test can be reused to expose behavioral changes. However, the challenge is that we need to determine which tests are compatible with both versions.

Definition of behavior preservation. It is not straightforward to generate tests for all types of changes. Consider a change to a method in an abstract class; how should the plugin generate tests for it? We cannot generate tests based on only this class because the abstract class cannot be instantiated. If the class has subclasses, we can instantiate a subclass and test the method as it was inherited. If the class has no subclasses, then we can generate a mocked subclass and test that in the same way. However, this latter case introduces a wider problem – if an abstract class has no concrete subclasses, what does it mean for a change to be behavior preserving? Since the class is abstract, it cannot be instantiated, so any change is technically behavior preserving with respect to the program under analysis. However, the class may be instantiated in some third-party code. This is essentially an issue of assuming a *closed world* versus assuming an *open world*. Our intuition is that assuming an open world is more beneficial for the programmer, and that our plugin should be implemented using this assumption. This is because, in the abstract class example, the programmer likely wants to know whether a change to an abstract class is behavior preserving with respect to all possible subclasses. In essence, the plugin should present information the same way for abstract classes as it does with concrete ones.

To take another example, suppose in the middle of an Extract Method refactoring, a programmer adds code to write to a log file. What should the plugin tell the programmer now? Since writing to a log is unlikely to be a bug, marking the entire set of changes as non-behavior preserving seems unhelpful. We think it would be more helpful to say that the change as a whole is a refactoring with respect to what the method returns, but is not a refactoring with respect to what is written to the log. We refer to these as two different "domains" of behavior preservation, and believe that our plugin should specify what domain a behavior change belongs to because this will help the programmer make a more informed judgement about whether the change modified behavior in any way meaningful to her. How the plugin can communicate this to the programmer is an open question.

IV. Implementation

In this section, we describe an in-progress implementation of our plugin for the Eclipse IDE. Our goal was to evaluate the feasibility of our idea with respect to performance. Can our approach work fast enough to provide feedback during coding activities? Our proof of concept compares only two versions: a *base* version, meaning the version of the program the last time the file was saved, and the *current* version, the most recent compilable version of the program. Next, we explain the process our plugin uses for detecting behavioral changes.

First, we identify the method that is being edited by the programmer, its parameters and the constructors of its declaring class. We also include its callers and subclasses that inherit from it. We use these elements to generate tests using Randoop. Similar to SafeRefactor, we only include the methods with the same signature between versions to guarantee that the tests compile before and after the change.

Randoop randomly generates tests for a specified period of time – the longer the time, the more tests it generates. Since we focus on only a small part of the total program, and because the plugin is intended to be used during live coding, we ask Randoop to generate as many tests as it can in one second. To reduce overhead, we invoke Randoop on a running, separate Java Virtual Machine (JVM) with the binary classes of the base version pre-loaded. We use Java Remote Method Invocation[1] (RMI) to communicate between the plugin and the separate JVM. To reduce the time to test each version, we do not generate the JUnit test files. Instead, Randoop generates the test cases in memory and simultaneously executes those

[1] http://docs.oracle.com/javase/tutorial/rmi/index.html

tests against the base version of the program. Then, we send those test cases over RMI to the JVM running the current version of the program. That JVM then invokes the tests cases using reflection.[2] Finally, we compare the results of the two test executions; if they are different, we report a behavioral change, otherwise, a refactoring.

To test the performance of our prototype, we used a MacBook Pro Core i5 with 4 GB of ram to perform the JHotDraw refactoring described in Section I. Our plugin generated 26 test cases, 14 of which exposed the behavioral change that programmers accidentally introduced. It detected this bug about 2 seconds after performing the change. We feel this is fast enough to provide live feedback on behavioral changes. Even so, further optimizations are possible to reduce this time.

V. Related Work

Rachatasumrit and Kim [9] investigated the adequacy of regression tests for validating refactorings in real-world programs. They found that refactorings' correctness was not well-tested: only 22% of code impacted by refactorings was covered by existing test suites. Also, a survey performed at Microsoft found that a lack of tests may discourage programmers from performing refactorings [10]. Our plugin is designed to improve programmers' confidence when performing refactorings.

To help programmers perform refactorings, IDEs provide refactoring tools, which automatically check preconditions to assure behavioral preservation, and if the preconditions are satisfied, they perform the desired transformation. However, it is difficult for tool builders to identify all preconditions needed for each refactoring. Even state-of-the-art academic and industrial Java refactoring tools may introduce bugs [6]. Also, Ge et al. [4] show that programmers sometimes fail to recognize that they are going to refactor before manually starting the refactoring, leading to an underuse of refactoring tools. They propose BeneFactor, a plugin that detects when the programmer is manually refactoring the code, and reminds her that there are tools available to complete the change. Nevertheless, current refactoring tools can only check a pre-defined set of refactorings; because of the difficulty in defining such refactorings, BeneFactor has only 3 refactorings implemented. By providing live feedback on behavioral changes, we aim at automatically validating any kind of refactoring without the user needing to explicitly invoke the plugin.

Developers can also use static analysis tools, such as FindBugs [11], to help them find bugs early in the development process. In fact, we could have detected the bug introduced in the JHotDraw example by using Findbugs. The tool would show a warning in the new field of the `Figure` class. Similar to refactoring tools, though, its difficult to specify all kinds of bugs that may occur. We propose a complementary approach that uses dynamic analysis to help programmers find bugs.

Jin et al. [12] propose an approach called BERT for automated regression testing. Given two versions of a program, they generate tests for the changed parts of the program, run the tests on both versions, and compare the results. However, programmers can use BERT only when the change does not alter method signatures to guarantee the generated tests will compile on both versions of the program. In contrast, we identify the common methods between both versions to guarantee the compatibility of the tests.

Lahiri et al. [13] propose a tool for equivalence checking and displaying behavioral differences. The tool translates the program into an intermediate language, and, for each pair of methods (before and after the change), checks equivalence by using a program verifier. Symdiff took up to 180 seconds to evaluate programs with less than 570 lines of code. We seek to provide a plugin that provides feedback almost immediately.

[2] http://docs.oracle.com/javase/tutorial/reflect/index.html

VI. Conclusions

In this paper, we investigate how to design a plugin for the programmer's IDE that provides live feedback on behavioral changes by generating and running tests on multiple program versions. Although, at first, we thought that it could be easily implemented based on previous works on continuous testing [2] and generating tests for checking behavioral changes [3], during its development, new challenges emerged. We described our current implementation, which provides evidence that it is feasible to provide live feedback on behavioral changes based on test generation and execution. We think that this plugin will change the programming experience, but only long term use will help us find out how.

Acknowledgment

Thanks to Brittany Johnson, Xi Ge, Jim Shepherd, Yoonki Song, and Michael Bazik. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES).

References

[1] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[2] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *ISSTA*, 2004, pp. 76–85.

[3] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, pp. 52–57, 2010.

[4] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *ICSE*, 2012, pp. 211–221.

[5] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007, pp. 75–84.

[6] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE TSE*, vol. 39, no. 2, pp. 147–162, 2013.

[7] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *OOPSLA*, 2011, pp. 189–206.

[8] J. Wloka, E. Hoest, and B. Ryder, "Tool support for change-centric test development," *IEEE Software*, vol. 27, no. 3, pp. 66 –71, 2010.

[9] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *ICSM*, 2012, pp. 357 –366.

[10] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *FSE*, 2012, pp. 50:1–50:11.

[11] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA*, 2004, pp. 132–136.

[12] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *ICST*, 2010, pp. 137–146.

[13] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: a language-agnostic semantic diff tool for imperative programs," in *CAV'12*, 2012, pp. 712–717.