Making program refactoring safer

Gustavo Soares

Federal University of Campina Grande, Brazil
gsoares@dsc.ufcg.edu.br
home page: //http://www.dsc.ufcg.edu.br/~spg/

Abstract. Each refactoring implementation must check a number of conditions to guarantee behavior preservation. However, specifying and checking them are difficult. Therefore, refactoring tool developers may define too weak conditions, which leads to non-behavior-preserving transformations, or too strong conditions preventing behavior-preserving transformations. We propose an approach for improving the developer's confidence that the refactoring was correctly applied, and a technique to test refactoring implementations with respect to weak and strong conditions.

Keywords: Refactoring, Program Generation, Automated testing

1 Introduction

Refactoring is the process of changing a software system to improve its internal quality yet preserving its observable behavior [4]. Manually applying refactorings is an error-prone and time-consuming activity. Currently IDEs, such as Eclipse and NetBeans, automated a number of refactorings. The refactoring implementation checks a number of conditions needed for guaranteeing behavioral preservation. If the conditions are satisfied, the tool performs the transformation.

Listing 1.1. Pulling up B.k(int) using Eclipse changes program behavior

```
public class A {
  public int k(long i) { return 10; }
}
public class B extends A {
  public int k(int i) { return 20; }
  public int test() { return new A().k(2); }
}
```

However, mostly refactoring tools may implement too weak conditions, because formally establishing all of them is not trivial. Therefore, often refactoring tools allow wrong transformations to be applied with no warnings whatsoever. For instance, Listing 1.1 shows a program containing the `A` class and its subclass `B`. The `test` method yields 10. When we apply the pull up refactoring to the `k(int)` method using Eclipse 3.6, the IDE moves it to class `A`. After the transformation, the `test` method yields `20`, instead of `10`. Therefore, the transformation does not preserve behavior using the Eclipse 3.4.2 IDE. Notice that, originally, the expression inside `test` calls `A.k(long)`. However, after pulling up `k(int)`, this method is called instead.

On the other hand, tool developers may also implement too strong conditions, which avoids behavior-preserving transformations, compromising the tool's applicability.

2 Approaches

We propose an approach (SafeRefactor) [10] for checking refactoring safety in sequential Java programs. It analyzes a transformation, and generates a test suite useful for detecting behavioral changes. First, we identify methods in common (same signature) in both source and target programs. Next we use a test generator (Randoop [5]) to generate unit tests for the identified methods. Finally, we run the generated test suite on the source program and on the target program. If a test passes in one of the programs and fails in the other one, we detect a behavioral change. Otherwise, the programmer can have more confidence that the transformation does not introduce behavioral changes.

We also propose an approach to help refactoring tool developers to test their implementations with respect to too weak and too strong conditions. First, it automatically generates a number of small programs as test inputs. This step aims at generating inputs that tool developers may be unaware of. Second, it applies the refactoring implementation to each one of them. It then uses SafeRefactor to evaluate the correctness of the transformations. We report the non-behavior-preserving transformations applied due to weak conditions. Additionally, to identify too strong conditions, we apply the same transformation using at least two equivalent implementations, such as the Rename refactorings implemented by Eclipse and Netbeans. If an implementation rejects a transformation, and the other one applies it and preserves behavior according to SafeRefactor, we establish that the former implementation contains strong conditions.

To perform the program generation, we propose JDolly, a Java program generator. It contains a subset of the Java metamodel specified in Alloy, a formal specification language [3], and uses the Alloy Analyzer, a tool for analysis of Alloy models, for generating solutions for this metamodel. Each solution is translated to a Java program. JDolly receives as input the scope of the generation, that is, the maximum number of elements (packages, classes, fields, and methods) that generated programs must have, and additional constraints for generating test inputs specific for each implementation.

3 Evaluation

We used SafeRefactor to evaluate 7 refactorings performed by developers applied to real Java applications (3-100 KLOC) using tools or manual steps, and test suites [10]. It detected a behavioral change in a transformation applied to JHotDraw (23KLOC) to modularize exception handling code and two compilation errors.

We evaluated our technique for testing refactoring tools by testing 22 refactoring implementations in two Java refactoring engines: the Eclipse Refactoring Module and the JastAdd Refactoring Tool (JRRT) [6]. Table 1 summarizes the results. It shows the number of programs generated by JDolly, the testing time, and the number of failures (compilation errors and behavioral changes) of each implementation. Many of these failures are related to the same bug. We analyzed them and identified 29 bugs related to compilation errors, and 38 bugs related to behavioral changes (Column Bugs). JDolly and SafeRefactor were useful for detecting bugs that have not been revealed so far. Additionally, we identified 17 too strong conditions in Eclipse and 7 ones in JRRT.

Refactoring	Programs	Time		Failures				Bugs			
				CE		BC		CE		BC	
		Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT
Rename Class	7200	03:12	03:48	194	0	145	0	1	0	1	0
Rename Method	13464	07:54	11:30	549	0	0	0	1	0	0	0
Rename Field	6080	05:18	05:58	6	76	0	16	1	3	0	1
Push Down Method	5880	04:48	04:36	595	618	186	105	2	2	6	2
Push Down Field	7488	04:42	04:26	340	0	92	0	1	0	1	0
Pull Up Method	8760	06:24	06:06	132	296	203	74	2	2	7	2
Pull Up Field	6624	05:36	06:30	222	72	546	0	3	2	3	0
Encapsulate Field	8832	05:26	05:34	2000	0	0	108	1	0	0	1
Move Method	8938	8:31	5:39	889	922	3586	1422	2	3	4	2
Add Parameter	15808	13:41	14:18	706	0	1116	137	1	0	3	2
Remove Parameter	15808	14:18	-	706	-	190	-	1	-	2	-
Change Modifier	7224	11:42	-	602	-	703	-	1	-	3	-
Total - Bugs								**17**	**12**	**28**	**10**

Table 1. Test of refactoring implementations of Eclipse and JRRT; CE = compilation error; BC = behavioral change.

4 Related Work

Schäfer et al. [7] propose a Rename refactoring implementation. It is based on the name binding invariant: each name should refer to the same entity before and after the transformation. Furthermore, Schäfer et al. [9,6] propose a number of Java refactoring implementations using an enriched language. As correctness criterion, they proposed other invariants such as control flow and data flow preservation. We evaluated ten of these implementations using our technique. In the sample used in our evaluation, the JRRT outperformed Eclipse with respect to refactoring correctness: we found 22 bugs in JRRT and 45 in Eclipse. Other approaches have proposed refactoring specifications [1,12,11]. They analyze various aspects of Java, as accessibility, types, name binding, data flow, and control flow. However, proving refactoring correctness for the whole Java language is considered a grand challenge [8].

Daniel et al. [2] proposed an approach for automated testing refactoring tools. They propose a program generator called ASTGen. It allows developers to guide the program generation by extending Java classes. Our generator allows this by specifying Alloy constraints. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. While their oracles could only syntactically compare the programs to detect behavioral changes, SafeRefactor generates tests that do compare program behavior. They found one bug related to behavioral change (we found 28 bugs). Moreover, both techniques found the same number of bugs related to compilation errors.

5 Conclusions

In this work, we aim at making program refactoring safer. Safe Refactor can be used by developers to improve their confidence that transformations preserve code's behavior. Moreover, refactoring tool developers can use JDolly and Safe Refactor to improve their implementations.

References

1. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic Reasoning for Object-Oriented Programming. Science of Computer Programming 52, 53–100 (October 2004)
2. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: FSE '07. pp. 185–194 (2007)
3. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT press (2006)
4. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
5. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. pp. 75–84 (2007)
6. Schäfer, M., , de Moor, O.: Specifying and implementing refactorings. In: OOPSLA '10. pp. 286–301 (2010)
7. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for java. In: OOPSLA '08. pp. 277–294 (2008)
8. Schäfer, M., Ekman, T., de Moor, O.: Challenge proposal: Verification of refactorings. In: PLPV '09. pp. 67–72 (2009)
9. Schäfer, M., Verbaere, M., Ekman, T., Moor, O.: Stepping stones over the refactoring rubicon. In: ECOOP '09. pp. 369–393 (2009)
10. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Software 27, 52–57 (2010)
11. Steimann, F., Thies, A.: From public to private to absent: Refactoring java programs under constrained accessibility. In: ECOOP '09. pp. 419–443 (2009)
12. Tip, F., Kiezun, A., Baumer, D.: Refactoring for Generalization Using Type Constraints. In: OOPSLA '03. pp. 13–26 (2003)