Safira: A Tool for Evaluating Behavior Preservation

Melina Mongiovi
Department of Computing Systems, Federal University of Campina Grande
{melina@copin.ufcg.edu.br}

Abstract

We propose a tool (Safira) capable of determining if a transformation is behavior preserving through test generation for entities impacted by transformation. We use Safira to evaluate mutation testing and refactoring tools. We have detected 17 bugs in MuJava, and 27 bugs in refactorings implemented by Eclipse and JRRT.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Tools and Techniques

General Terms Reliability, Design

Keywords Refactoring, Testing

1. Introduction

Evaluating whether a program transformation is behavior preserving is required in several tasks, such as refactoring and mutation testing. In refactoring activities, tools are required to ensure that a transformation preserves behavior. On the other hand, mutation testing tools must introduce behavior changes on to programs. Every transformation must contain a set of conditions stating when behavior must be preserved. However, it is not an easy task to formally establish conditions for Java on account of its non-trivial semantics. For this reason, a number of refactoring and mutation testing tools are likely to present bugs.

Some approaches [3, 9] have formally established a number of refactoring conditions for a subset of Java. Steimann and Thies [9] have formally specified a set of refactoring conditions related to Java visibility. However, these conditions are not proven sound with respect to a formal semantics. In fact, it represents a challenge to formally propose sound refactoring [4]. A similar problem happens in the mutation testing area. Some approaches have suggested ways to avoid certain kinds of *equivalent mutants* — a mutant that is functionally equivalent to the original program. Based on previous work [9], Steimann and Thies have proposed an approach to generate mutants based on negating the conditions required for a refactoring [10]. Schuler and Zeller [5] have proposed an approach to detect equivalent mutants based on changes in test coverage. However, a simpler, safer and more practical approach to evaluate whether a transformation preserves behavior is required.

We propose a tool, called Safira, to evaluate whether a transformation is behavior preserving. This tool generates a test suite focusing on exercising only the entities impacted by a transformation. Safira yields a test case whenever it detects a behavior change. We propose an approach to evaluate mutation testing tools based on Safira and on a Java program generator (JDolly [8]). We use it to evaluate 11 mutations of MuJava [6] and found 17 bugs in all of them. We have also used Safira and JDolly to test refactoring implementations following a previous approach [8]. We found 27 bugs in refactorings implemented by Eclipse and JRRT [3].

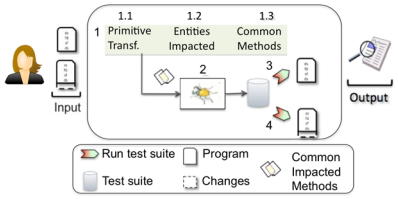

Figure 1. Safira architecture

2. Safira

The major steps performed by Safira are explained as follows: Firstly, the original (source) and the modified (target) programs are passed as parameters by developers. The change impact analyzer checks both the original and modified programs (Step 1), beginning by decomposing the transformation into primitive ones (Step 1.1). Secondly, for each primitive transformation, Safira identifies the entities impacted by it (Step 1.2). Finally, Safira identifies a set of methods in common that exercises, directly or indirectly, impacted entities (Step 1.3). A method in common must have the same signature in the source and target programs. After this a test suite is generated only for the identified methods (Step 2). The tests are then executed on the source (Step 3) and on the target programs (Step 4). Safira reveals a set of tests that passes on source program but fails on target program. If this set is empty, the developer increases confidence that the transformation has not introduced behavior changes. Otherwise, test cases show behavior changes. Our approach is illustrated in Figure 1.

In order to identify the entities impacted by a transformation, we decompose a transformation into primitive ones by following a similar approach as the one used by Chianti [2]. The set of entities impacted by the transformation encompasses the union of the entities impacted by each primitive transformation. We have considered nine primitive transformations: add and remove an empty class, add and remove an empty method, add and remove a field, add and remove extends and modify a method body. We have formalized the set of impacted methods and constructors for each primitive transformation. Safira uses Randoop [1] to automatically generate unit tests. Randoop, randomly, generates a test suite for the classes and methods received as parameters within a time limit specified by the developer. Randoop executes the program to receive a feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs. It creates

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

Table 1. Results of the MuJava evaluation using Safira and JDolly

MuJava Evaluation					
Mut. Typ	Mut.	Equiv. Mut.	Bugs	1st Bug (s)	Time (s)
AOIU	227	1 (0.44%)	1	1040	1338
ISD	101	2 (1%)	2	337	1969
AOIS	454	57 (12%)	1	1070	9727
OMR	185	31 (16%)	2	34	2000
IHD	58	25 (43%)	2	16	1227
OMD	100	56 (56%)	1	27	1580
IOR	69	66 (95%)	2	40	2857
IHI	100	38 (38%)	2	23	2652
IOD	21	12 (61%)	1	6	196
JID	195	63 (32%)	2	35	5870
JSI	100	100 (100%)	1	26	109

method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. Each sequence is executed and checked against a set of contracts.

3. Evaluation

We used Safira to assess mutation testing tools (Section 3.1) and refactoring engines (Section 3.2).

3.1 Mutation Testing

Mutation testing can help developers to evaluate their test suite. It consists of introducing defects on to the code in order to modify its behavior. If a test suite fails to detect behavior change, it needs to be improved. There are a number of mutation testing tools, such as MuJava [6]. However, these tools may generate equivalent mutants. Then when a test suite does not kill a mutant, developers do not know whether the problem resides in their test suite or it is an equivalent mutant. We can use Safira in the program and on its mutant. If Safira does not detect any behavior changes, developers improve confidence that it is an equivalent mutant. However, if it finds a behavior change, Safira yields a test case by means of which developers can improve their test suite.

Testing mutation testing tools is nontrivial, since one needs structurally complex inputs such as programs. Moreover, an oracle is required to determine whether a transformation is behavior preserving. We propose an approach to test mutation testing tools based on Safira. To begin with, a number of programs are generated by JDolly, a Java program generator which, exhaustively, generates programs up to a given scope. It specifies a subset of Java metamodel in Alloy, a formal specification language. JDolly receives, as input, a scope (the maximum number of packages, classes, fields, and methods that a program must have), and the additional constraints. For every program generated, we use MuJava to generate mutants. Finally, Safira evaluates whether each mutant is functionally different from the original program.

For each mutation, we generated 100 programs using JDolly. We evaluated 11 mutations of MuJava and tested all of them by using a scope up to 4. For instance, our approach found 25 equivalent mutants for the IHD mutation. Some of them are related to the same bug. We manually analyzed them and classified in 2 distinct bugs. The first equivalent mutant was detected in 16 seconds. We found 17 bugs in 11 mutations.

3.2 Refactoring

Safira was also used in the refactoring context to assist developers in refactoring activities. We implemented an Eclipse plug-in — the user selects a refactoring to apply. Then the plug-in reports whether the transformation preserves behavior or not. Soares et al. [8] proposed an approach based on JDolly and SafeRefactor [7] to test refactoring tools. They found a number of bugs in refactoring implementations of Eclipse and JRRT. We used their approach, but we replaced SafeRefactor for Safira. The impact analysis performed by Safira allows to generate tests guided by the change impact. We evaluate the results comparing our approach with theirs in terms of time consuming, test suite and correctness. We evaluated 10 kinds of refactorings. We detected 27 bugs related to behavioral changes.

4. Conclusion

Safira found a number of bugs in refactoring and mutation testing tools. To understand inheritance, this/super, package, accessibility modifiers and other Java constructs in isolation may be simple. However, when they are taken together, the task becomes nontrivial. So, it is difficult to propose all the conditions required by a transformation to preserve behavior, considering the whole scope of Java language. For example, a simple transformation, changing the access modifier, may have an impact on a number of Java constructs [9].

Steimann and Thies [9] formalized some refactoring conditions to change visibility. By negating some conditions, they proposed an approach to generate mutants [10]. However, they did not formally prove the conditions were correct. It is not simple to formally specify such conditions. Javalanche has identified a number of program invariants. If a mutation does not violate these invariants, then they are more likely to be equivalent mutants [5]. The tool implements four mutations. When Safira detects some behavior change, it yields a test case different from all previous approaches. Our technique is simple, and can be used to evaluate any kind of mutations. It can also be useful for testing tools.

Schäfer et al. [3] formally specified a number of refactorings and implemented them in JRRT, which outperformed Eclipse in terms of correctness. However, we have identified a number of bugs in JRRT. They already fixed some of the bugs found by us. SafeRefactor [7] has also identified the same bugs which were detected in Eclipse and JRRT. However, we have reduced the test suite and the total analysis time in 74% and 30%, respectively, on account of our change impact analyzer. Moreover, Safira is less dependent on the time limit used to generate tests. As future work, we intend to evaluate other refactoring and mutations testing tools.

References

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[2] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *ICSE*, pages 664–665, 2005.

[3] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *OOPSLA*, pages 286–301, 2010.

[4] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: Verification of refactorings. In *PLPV*, pages 67–72, 2009.

[5] D. Schuler and A. Zeller. (Un-)Covering equivalent mutants. In *ICST*, pages 45–54, 2010.

[6] Y. seung Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15: 97–133, 2005.

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

[8] G. Soares, M. Mongiovi, and R. Gheyi. Identifying too strong conditions in refactoring implementations. In *ICSM*, 2011. To appear.

[9] F. Steimann and A. Thies. From public to private to absent: Refactoring java programs under constrained accessibility. In *ECOOP*, pages 419–443, 2009.

[10] F. Steimann and A. Thies. From behaviour preservation to behaviour modification: constraint-based mutant generation. In *ICSE*, pages 425–434, 2010.