Automated Behavioral Testing of Refactoring Engines

Gustavo Soares[1], **Rohit Gheyi**[1]

[1]Department of Computing and Systems– Federal University of Campina Grande (UFCG)
Campina Grande – PB – Brazil

{gsoares,rohit}@dsc.ufcg.edu.br

Abstract. *Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. To help developers in this task, current IDEs, such as Eclipse and NetBeans, automate a number of refactorings. However, implementing refactorings is a complex task, and so, even mainstream IDEs contain critical bugs. We propose an automated approach for testing of Java refactoring engines. Its key components are:* JDOLLY*, a Java program generator, and* SAFEREFACTOR*, a program for checking behavioral changes. The technique uses* JDOLLY *to generate programs as test inputs. For each generated program, it applies the refactoring by using the engine under test, and uses oracles based on* SAFEREFACTOR *to evaluate the correctness of the transformation. In the end, it classifies the failures into distinct bugs. We have evaluated this technique by testing up to 10 refactorings from Eclipse, NetBeans and the JastAdd Refactoring Tools. Our technique tested 153,444 transformations, and identified more than 100 bugs, which were reported to engines' developers. They accepted most of them, and already fixed 35 bugs.*

1. Information about the thesis

This paper resumes the Ph.D. thesis Uma Abordagem Automatizada para Testar Ferramentas de Refatoramento of Gustavo Soares from the department of Computing and Systems of the Federal University of Campina Grande. Gustavo Soares had the supervision of Prof. Rohit Gheyi. The work was approved with distinction by the bank examiner composed by his advisor and Prof. Tiago Massoni and Prof. Patricia Machado from the department of Computing and Systems of the Federal University of Campina Grande, Prof. Paulo Borba from the Informatics Center of the Federal University of Pernambuco, and Prof. Alessandro Garcia from the Informatics Department of PUC-Rio.

2. Problem and motivation

Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality [Opdyke 1992]. Each refactoring may contain a number of preconditions needed to guarantee behavioral preservation. For instance, to pull up a method `m` to a superclass, we must check whether `m` conflicts with the signature of other methods in that superclass. In practice, testing refactoring preconditions involves manually creating an input program to be refactored and specifying a refactoring precondition failure as expected output.

However, developers choose input programs for checking just the preconditions they are aware of. Since specifying preconditions is a non-trivial task, developers may be

unaware of preconditions needed to guarantee behavioral preservation. When the implemented preconditions are insufficient to guarantee behavioral preservation, we call it as *overly weak preconditions*. Additionally, some implemented preconditions may be *overly strong*, that is, it leads the engine to refuse to apply a behavior preserving transformation.

Refactoring engine developers have invested in testing. For instance, Eclipse's test suite has more than 3.000 unit tests for checking refactoring correctness. However, their test suites still fail to detect a number of bugs [Soares et al. 2013a]. For instance, take class `A` and its subclass `B` as illustrated in Listing 1. The `B.test()` method yields 1. If we use Eclipse 4.3 to perform the Pull Up Method refactoring on method `m()`, the tool will move method `m` from `B` to `A`, and update `super` to `this` (see Listing 2). A behavioral change was introduced: `test` yields 2 instead of 1. Since `m` is invoked on an instance of `B`, the call to `k` using `this` is dispatched on to the implementation of `k` in `B`.

Listing 1. Pulling up B.k() by using Eclipse 4.3 changes program behavior.

```
public class A {
  int k() {return 1;}
}
public class B extends A {
  int k() {return 2;}
  int m() {return super.k();}
  public int test() {return m();}
}
```

Producing tests for checking refactoring preconditions by hand is not simple due to the complexity of the test inputs and the analysis of the refactoring output, which may result on a poor test suite, potentially leaving many hidden bugs, such as the previous one. Another approach to handle this problem is to formally specify refactorings. For instance, Schäfer and Moor [Schäfer and de Moor 2010] specified refactorings for Java, and proposed a tool called JastAdd Refactoring Tools (JRRT) [Schäfer and de Moor 2010]. However, proving refactoring correctness for the entire language is still a challenge [Schäfer et al.]. The same problem shown in Listings 1 and 2 occurs when we apply this previous transformation by using JRRTv1[1].

Listing 2. After pulling up method m, the test method yields 2 instead if 1.

```
public class A {
  int k() {return 1;}
  int m() {return this.k();}
}
public class B extends A {
  int k() {return 2;}
  public int test() {return m();}
}
```

3. Approach

We propose an approach for testing of Java refactoring engines. Its main novelties are its technique for generating input programs and its test oracles for checking behavioral

[1]The JRRT version from May 18th, 2010

preservation based on dynamic analysis. It performs four major steps. First, a program generator automatically yields programs as test inputs for a refactoring. Second, the refactoring under test is automatically applied to each generated program. The transformation is evaluated by test oracles in terms of overly weak and overly strong preconditions. In the end, we may have detected a number of failures, which are categorized in Step 4.

3.1. Test input generation

To perform the test input generation, we propose a Java program generator (JDOLLY [Soares et al. 2013a]). It contains a subset of the Java metamodel specified in Alloy [Jackson 2012], a formal specification language. It employs the Alloy Analyzer, a tool for the analysis of Alloy models, to generate solutions for this metamodel. Each solution is translated into a Java program. In JDOLLY, the user can specify the maximum number (*scope*) of packages, classes, fields, and methods for the generated programs. The tool exhaustively generates programs for a given scope. In this way, it may generate input programs capable of revealing bugs that developers were unaware of. Furthermore, JDOLLY can be parameterized with specific constraints. For example, when testing a refactoring that pulls up a method to a superclass, the input programs must contain at least a subclass declaring a method that is subject to be pulled up. We can specify these constraints in Alloy.

3.2. Test oracle

We propose SAFEREFACTOR [Soares et al. 2010], a tool for checking behavioral changes, as oracle for weak preconditions. First, the tool checks for compilation errors in the resulting program, and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes. SAFEREFACTOR identifies the methods with matching signature before and after the transformation. Next, it applies Randoop [Pacheco et al. 2007], a random unit test generator for Java, to produce a test suite for those methods. Finally, it runs the tests before and after the transformation, and evaluates the results. If results are divergent, the tool reports a behavioral change.

We propose an oracle to detect overly strong preconditions based on differential testing [Soares et al. 2011c]. When the refactoring implementation under test rejects a transformation, we apply the same transformation by using one or more other refactoring implementations. If one implementation applies it, and SAFEREFACTOR does not find behavioral changes, we establish that the implementation under test contains an overly strong condition since it rejected a behavior-preserving transformation.

3.3. Test clustering

Our technique may produce a large number of failures, and some of them may be related to the same fault. Jagannath et al. [Jagannath et al. 2009] propose an approach to split failures based on oracle messages (Oracle-based Test Clustering - OTC). We adopt this approach to classify failures that introduce compilation errors in the output program. The failures are grouped by the template of the compiler error message, so that each group contains a distinct fault. We also use OTC to categorize the overly strong precondition failures based on the template of the warning message thrown by a refactoring engine. However, we cannot use this approach for classifying failures related to behavioral changes since

Table 1. Overall experimental results.

Refactoring	Program	Time(h)	Comp. error.	Behav. cha.	Overly strong
Rename Class	15322	6.7	4368	160	446
Rename Method	11263	6.9	2290	1713	5221
Rename Field	19424	29.3	894	1834	200
Push Down Method	20544	11.9	13579	3312	99
Push Down Field	11936	6	7231	119	0
Pull Up Method	8937	7.3	3867	1363	649
Pull Up Field	10927	8.6	1726	785	1328
Encapsulate Field	2000	2.5	472	1220	1712
Move Method	22905	10.3	1321	12289	502
Add Parameter	30186	34.69	7487	4802	79
Total	153444	124.19	43235	27597	10236

there is no information from our oracle (SAFEREFACTOR) that could be used to split the failures. Instead, we propose an approach to classify them based on *filters* that check for structural patterns in each pair of input and output programs. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the output program, relatively to the input program.

4. Results

We performed an experiment to evaluate our approach with respect to effectiveness in identifying overly weak and overly strong preconditions in refactoring engines. We selected up to 10 refactoring implementations from Eclipse JDT 3.7, NetBeans 7.0.1, and JRRT [Soares et al. 2013a]. We evaluated two versions of JRRT. First, we tested the refactorings implemented by JRRTv1, and reported the bugs we found. Later, a new version was released with improvements and bug fixing (which we call JRRTv2); this new version was also subject to our analysis.

Table 1 summaries the experiment results. The evaluated refactorings focus on a representative set of program structures. Columns Program and Time show the number of programs generated by JDOLLY for each refactoring, and the average time for testing the refactoring implementations from each engine, respectively. Columns Comp. error., Behav. cha., and Overly strong show the total number of transformations applied by Eclipse, NetBeans, JRRTv1, and JRRTv2 that produced compilation errors, behavioral changes, and that were not applied due to overly strong preconditions, respectively. Considering all refactorings, JDOLLY generated 153,444 programs, and our technique detected 43,235 transformations that introduce compilation errors, 27,597 ones that introduce behavioral changes, and 10,236 that were not applied due to overly strong preconditions.

Even though Eclipse, JRRT and NetBeans have their own test suites, our technique classified 144 (likely) *unique* bugs. Table 2 summarizes the bugs identified and reported to Eclipse JDT, NetBeans and JRRT. Our technique identified 34 overly weak preconditions in Eclipse. Although all of them were accepted by the Eclipse developers, 16 of them were labeled as duplicated. So far, they have fixed just two of them. In NetBeans, our technique identified 51 overly weak preconditions. NetBeans team has already accepted

Table 2. Summary of reported bugs.

Engine	Submitted	Accepted	Duplicated	Not accepted	Not answered	Fixed
Eclipse	51	40	17	1	0	2
JRRTv1	31	27	0	4	0	23
JRRTv2	11	6	0	5	0	6
NetBeans	51	24	0	2	25	7

30 of them and fixed 7 bugs. Meanwhile, we reported 24 overly weak preconditions to JRRTv1, from which 20 were accepted and fixed (4 of the bugs were not considered bugs due to a closed-world assumption of JRRT developers). We reported more 11 bugs to JRRTv2, from which 6 were accepted and fixed. JRRT team also incorporated our test cases into their test suite. Our technique did not find overly strong preconditions in NetBeans, but identified 17 ones in Eclipse and 7 ones in JRRTv1. JRRT developers fixed 3 out of the 7 overly strong preconditions.

5. Related work

To help developers on testing refactoring engines, Daniel et al. [Daniel et al. 2007] proposed an approach to automate this process. They used a program generator (ASTGen) to generate programs as test inputs. ASTGen allows users to directly implement how the program will be generated. Later, Gligoric et al. [Gligoric et al. 2010] proposed UDITA, a Java-like language that extends ASTGen allowing users to specify what is to be generated (instead of how to generate), and uses the Java Path Finder model checker as a basis for searching for all possible combinations. Although JDOLLY and UDITA use the same ideal for generating programs, they use different technologies for searching for solutions, and specify constraints in different styles. Alloy logic presents a higher level of abstraction than Java-like code. For example, a constraint containing the transitive closure operator in Alloy can only be achieved programmatically after considerable additional effort in Java. Also, Daniel et al. [Daniel et al. 2007] implemented test oracles to evaluate engine outputs. They have identified a number of bugs that introduce compilation errors on the user's code and one bug that introduces a behavioral change. While the oracles of previous approaches can only syntactically compare the programs to detect behavioral changes, SAFEREFACTOR generates tests that do compare program behavior. We found 63 bugs related to behavioral changes.

6. Publications and Awards

The technique for testing of refactoring engines with respect to overly weak conditions was published in the IEEE Transactions of Software Engineering [Soares et al. 2013a] (Qualis A1 - JCR=2,292). The technique for testing of refactoring engines with respect to overly strong condition was published in the IEEE International Conference on Software Maintenance [Soares et al. 2011c] (Qualis A2). SAFEREFACTOR was published in the IEEE Software Magazine [Soares et al. 2010] (Qualis A1 - JCR=1,23). Later, further evaluations of SAFEREFACTOR on analyzing refactoring actives on software repositories were published in the Brazilian Symposium on Software Engineering [Soares et al. 2011a] (Qualis B3) and Journal of Systems and Software [Soares et al. 2013b] (Qualis A2 - JCR=1,245). An extension of SAFEREFACTOR

for supporting aspect-oriented programming was published in the Brazilian Symposium on Programming Languages [Soares et al. 2011b] (Qualis B3).

The quality of the work was recognized by some awards. Gustavo Soares's Ph.D. work was ranked the best graduate research work in the ACM Student Research Competition at SPLASH 2012. Gustavo Soares also won the John Vlissides Award in the Doctoral Symposium at SPLASH, 2012. The award is presented annually to a doctoral student participating in the doctoral symposium showing significant promise in applied software research. The paper "Analyzing Refactorings on Software Repositories" [Soares et al. 2011a] won the best paper award at the 25th Brazilian Symposium on Software Engineering (SBES 2011). The poster "Making program refactoring safer" was ranked 2nd Best Poster at ECOOP 2011.

References

Daniel, B., Dig, D., Garcia, K., and Marinov, D. (2007). Automated testing of refactoring engines. In *ESEC/FSE '07*. ACM.

Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., and Marinov, D. (2010). Test generation through programming in udita. In *ICSE '10*, pages 225–234.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jagannath, V., Lee, Y. Y., Daniel, B., and Marinov, D. (2009). Reducing the costs of bounded-exhaustive testing. In *FASE '09*, pages 171–185.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Pacheco, C., Lahiri, S., Ernst, M., and Ball, T. (2007). Feedback-directed random test generation. In *ICSE '07*, pages 75–84.

Schäfer, M. and de Moor, O. (2010). Specifying and implementing refactorings. In *OOPSLA '10*, pages 286–301. ACM.

Schäfer, M., Ekman, T., and de Moor, O. Challenge proposal: verification of refactorings. In *PLPV '09*, pages 67–72. ACM.

Soares, G., Catão, B., Varjão, C., Aguiar, S., Gheyi, R., and Massoni, T. (2011a). Analyzing refactorings on software repositories. In *SBES '11*, pages 164–173.

Soares, G., Cavalcanti, D., and Gheyi, R. (2011b). Making aspect-oriented refactoring safer. In *SBLP'11*.

Soares, G., Gheyi, R., and Massoni, T. (2013a). Automated behavioral testing of refactoring engines. *IEEE TSE*, 39(2):147–162.

Soares, G., Gheyi, R., Murphy-Hill, E., and Johnson, B. (2013b). Comparing approaches to analyze refactoring activity on software repositories. *JSS*, 86:1006,1022.

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

Soares, G., Mongiovi, M., and Gheyi, R. (2011c). Identifying overly strong conditions in refactoring implementations. In *ICSM '11*, pages 173–182.