Automated Behavioral Testing of Refactoring Engines

Gustavo Soares
Federal University of Campina Grande, Campina Grande, PB, Brazil
gsoares@dsc.ufcg.edu.br

Abstract

Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. Usually, compilation errors and behavioral changes are avoided by preconditions. However, defining and implementing preconditions is a complex task. As a result, even mainstream refactoring engines contain critical bugs. We propose an automated approach for testing of Java refactoring engines based on program generation. It has been useful for identifying more than 100 bugs in state-of-the-art industrial and academic refactoring engines.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Refactoring, Testing, Program Generation

1. Problem and motivation

Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. 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. 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 test suite with a low coverage level, potentially leaving many hidden bugs.

Refactoring engine developers have invested in testing. For instance, Eclipse's test suite has more than 3.000 unit tests for checking refactoring correctness. Their implementations of the Pull Up Method and Rename Method refactorings have 75% and 86% of their code covered by the tests. However, their test suites still fail to detect a number of bugs. For instance, take class `A` and its subclass `B` as illustrated in Listing 1. The `B.test()` method yields 1. If we use Eclipse 3.7 to perform the Pull Up Method refactoring on `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 3.7 or JRRTv1 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();}
}
```

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();}
}
```

Researches have tried to handle this problem by formally specifying refactorings. For instance, Schäfer and Moor [9]

specified refactorings for Java, and proposed a tool called JastAdd Refactoring Tools (JRRT) [9]. However, proving refactoring correctness for the entire language is still a challenge [10]. The same problem occurs when we apply this previous transformation by using JRRTv1[1].

As we mention, refactoring engine developers may also implement overly strong preconditions. For instance, consider the `A` class and its subclass `B` in Listing 3. `A` declares the `k(long)` method, and `B` declares methods `n` and `test`. Suppose we would like to rename `n` to `k`. If we apply this transformation by using Eclipse 3.7, it will show a warning message. However, we can apply this transformation by using JRRTv1. It performs an additional change to make the transformation behavior-preserving by adding a `super` access to the method invocation `k(2)` inside `test`.

Listing 3. Eclipse 3.7 prevents renaming B.n to B.k but JRRTv1 correctly applies the transformation.

```
public class A {
  public long k(long l) {return 1;}
}
public class B extends A {
  public long n(int i) {return 2;}
  public long test() {return k(2);}
}
```

2. Background and related work

Preconditions are a key concept of research studies on the correctness of refactorings. Opdyke [7] proposes a number of refactoring preconditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these preconditions. In fact, later, Tokuda and Batory [16] showed that Opdyke's preconditions were not sufficient to ensure preservation of behavior. Proving refactorings with respect to a formal semantics is a challenge [10]. Some approaches have been contributing in this direction. Borba et al. [1] propose a set of refactorings for a subset of Java with copy semantics (ROOL). They prove the refactoring correctness based on a formal semantics. Silva et al. [11] propose a set of behavior-preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each one of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. Yet, they have not considered all Java constructs, such as overloading and field hiding. Recently, Steimann and Thies [15] show that by changing access modifiers (`public`, `protected`, `package`, `private`) in Java one can introduce compilation errors and behavioral changes. They propose a constraint-based approach to specify Java accessibility, which favors checking refactoring preconditions and computing the changes of access modifiers needed to preserve the program behavior.

To help developers on testing refactoring engines, Daniel et al. [2] 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. Also, they implemented test oracles to evaluate engine outputs. They have identified a number of bugs that introduce compilation errors on the user's code. Later, Gligoric et al. [3] 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 (JPF) model checker as a basis for searching for all possible combinations.

In a complementary work, Murphy-Hill and Black [6] characterize problems related to the process to apply automated refactorings. They propose principles that refactoring engine developers can use to turn these tools more popular.

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

3. Approach and uniqueness

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 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. The whole process is depicted in Figure 1.

3.1 Test input generation

To perform the test input generation, we propose a Java program generator (JDOLLY [14]). It contains a subset of the Java metamodel specified in Alloy, 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.

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, the results of the closure operator in Alloy can only be achieved programmatically after considerable additional effort.

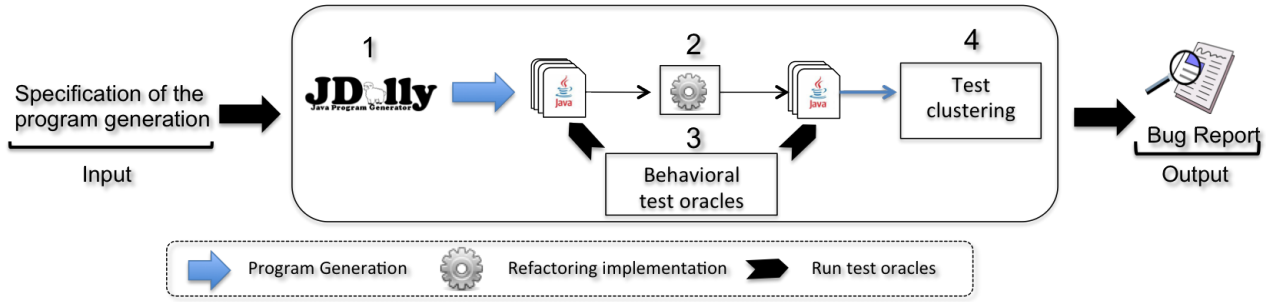

Figure 1. Automated behavioral testing of refactoring engines.

3.2 Test oracle

Overly weak preconditions. We propose SAFEREFACTOR [12], 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 [8], 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.

Assuming the programs in Listings 1 and 2 as input, SAFEREFACTOR first identifies the methods with matching signatures on both versions: `A.k`, `B.k`, and `B.test`. Next, it generates 78 unit tests for these methods within a time limit of two seconds. Finally, it runs the test suite on both versions and evaluates the results. A number of tests (64) passed in the source program, but did not pass in the refactored program; so SAFEREFACTOR reports a behavioral change.

The oracles proposed by Daniel et al. [2] try to detect behavioral changes by applying static analysis. For instance, they apply the inverse refactoring to the output program and expect that the result be equal to the input program. We believe that by using a dynamic analysis (SAFEREFACTOR) we will be able to detect behavioral changes not detect by previous oracles.

Overly strong preconditions. We propose an oracle to detect overly strong preconditions based on differential testing [13]. 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.

For example, consider the results of the Rename Method implementations for a given program illustrated in Listing 3. We compare the results of Eclipse and JRRT. While the former rejected the transformation, JRRT applied it. SAFEREFACTOR evaluates the transformations applied by JRRT, and does not find behavioral changes in the transformation applied by JRRT. We conclude that Eclipse rejected a behavior-preserving transformation due to an overly strong condition since JRRT was able to correctly apply it.

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. [4] 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 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 pattern 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 and contributions

We performed an experiment to evaluate our approach with respect to effectiveness in identifying overly weak and overly strong preconditions in refactoring engines.

4.1 Selection of subjects

We selected up to 10 refactoring implementations from Eclipse JDT 3.7, NetBeans 7.0.1, and JRRT [14]. 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 shows all evaluated refactorings. The evaluated refactorings focus on a representative set of program struc-

Table 1. Summary of scope and constraints for each refactoring; Scope = Package (P) - Class (C) - Field (F) - Method (M).

Refactoring	Scope (P - C - F - M)	Main constraint
Rename Class	2-3-0-3	**some** class
Rename Method	2-3-0-3	**some** Method
Rename Field	2-3-2-1	**some** Field
Push Down Method	2-3-0-4	**some** c:Class ‖ someSubClass[c] **and** someMethod[c]
Push Down Field	2-3-2-1	**some** c:Class ‖ someSubClass[c] **and** someField[c]
Pull Up Method	2-3-0-4	**some** c:Class ‖ someParent[c] **and** someMethod[c]
Pull Up Field	2-3-2-1	**some** c:Class ‖ someParent[c] **and** someField[c]
Encapsulate Field	2-3-1-3	**some** Field
Move Method	2-3-1-3	**some** c:Class ‖ someTargetClassField[c] and someMethodToMove[c]
Add Parameter	2-3-0-3	**some** Method

tures. Moreover, a survey carried out by Murphy et al. [5] shows the Eclipse JDT refactorings that Java developers use most: Rename, Move Method, Extract Method, Pull Up Method, and Add Parameter. Four of these are evaluated in this article. The Move Method refactoring was not supported by NetBeans by the time that this article was written.

4.2 Experiment design

Table 1 indicates the maximum number of packages, classes, fields, and methods passed as parameter to JDOLLY. For each refactoring, we specified *main constraints* for guiding JDOLLY to generate programs with certain characteristics needed to apply the refactoring. Column Main Constraint shows these constraints; they prevent the generation of programs to which the refactoring under test is not applicable.

In order to minimize the number of generated programs to a small, focused set, we have also defined *additional constraints*. These constraints were built on data about refactoring bugs gathered in the literature, enforcing properties such as overriding, overloading, inheritance, field hiding, and accessibility. For each refactoring, we declared Alloy facts with additional constraints. If a developer has the available resources to analyze the entire scope, then it will not be required to specify additional constraints. For each refactoring, we used the same set of programs as test inputs to evaluate Eclipse JDT, JRRTv1, JRRTv2, and NetBeans.

4.3 Results

Table 2 summaries the experiment results. 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. 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 conditions, respectively.

Considering all refactorings, JDOLLY generated 153,444 programs, and our technique detected 43,235 transformations with compilation errors, 27,597 ones with behavioral changes, and 70,832 that were not applied due to overly strong conditions.

Even though Eclipse, JRRT and NetBeans have their own test suites, our technique identified 120 (likely) *unique* bugs. Table 3 summarizes the bugs 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 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. Moreover, It identified 7 overly strong preconditions in JRRTv1, from which 3 were fixed in JRRTv2.

4.4 Threats do validity

We check overly strong conditions by comparing two or more refactoring engines. These engines may not share the same concept with respect to the evaluated refactoring. We plan to confirm with the engine developers whether the bugs identified by our technique are indeed related to overly strong conditions. With respect to internal validity, constraints specified for JDOLLY may be too restrictive with respect to the program generation, which may hide possibly detectable bugs. We must be cautious when creating these constraints. Concerning the external validity, we are going to select a representative set of refactorings, which should include refactorings that are applied to different structures of the program.

4.5 Future work

So far, we have specified a subset of the Java metamodel in JDolly, which allows us to deal mainly with testing refac-

Table 2. Overall experimental results.

Refactoring	Program	Time(h)	Comp. error.	Behav. cha.	Overly strong
Rename Class	15322	6.7	4368	160	4528
Rename Method	11263	6.9	2290	1713	4003
Rename Field	19424	29.3	894	1834	2728
Push Down Method	20544	11.9	13579	3312	16891
Push Down Field	11936	6	7231	119	7350
Pull Up Method	8937	7.3	3867	1363	5230
Pull Up Field	10927	8.6	1726	785	2511
Encapsulate Field	2000	2.5	472	1220	1692
Move Method	22905	10.3	1321	12289	13610
Add Parameter	30186	34.69	7487	4802	12289
Total	153444	124.19	43235	27597	70832

Table 3. Summary of reported bugs.

Engine	Submitted	Accepted	Duplicated	Not accepted	Not answered	fixed
Eclipse	34	34	16	0	0	2
JRRTv1	24	20	0	4	0	20
JRRTv2	11	6	0	5	0	6
NetBeans	51	24	0	2	25	7

torings that operate at or above the level of methods. The method bodies contain just one return statement, such as the example in Listing 1. Additionally, our current Java metamodel does not include some structural Java elements such as interface and inner classes. We are going to extend our Alloy specification in order to test the evaluated refactorings with more elaborated input programs, and also test other refactorings, such as Extract Method.

Acknowledgments

I gratefully thank Rohit Gheyi, Max Schäfer and the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES).

References

[1] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *SCP*, 52: 53–100, 2004.

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

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

[4] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE '09*, pages 171–185, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23:76–83, July 2006.

[6] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[7] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[11] L. Silva, A. Sampaio, and Z. Liu. Laws of object-orientation with reference semantics. In *SEFM '08*, pages 217–226, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[14] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE TSE*, 99(PrePrints), 2012.

[15] F. Steimann and A. Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *ECOOP '09*, pages 419–443, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8:89–120, January 2001. ISSN 0928-8910.