Generating Unit Tests for Checking Refactoring Safety

Gustavo Soares[1], Rohit Gheyi[1], Tiago Massoni[1], Márcio Cornélio[2], Diego Cavalcanti[1]

[1]Department of Computing and Systems – Federal University of Campina Grande

[2]Department of Computing and Systems – University of Pernambuco

{gsoares,rohit,massoni,diegot}@dsc.ufcg.edu.br, marcio@dsc.upe.br

Abstract. *Program refactorings in IDEs are commonly implemented in an ad hoc way, since checking correctness with respect to a formal semantics is prohibitive. These tools may perform erroneous transformations, not preserving behavior. In order to detect these errors, developers rely on compilation and tests to attest that behavior is preserved. Compilation errors, for instance, are simple to identify by tools. However, changes in behavior very often go undetected. We propose a technique for generating a test suite that is specific to pinpoint incorrect refactorings. In each refactoring, we identify program parts that are common to the program before and after refactoring, and automatically generate a comprehensive set of unit tests for the initial program aided by a test generator. As such, the create test suite is run on the program before and after automatic refactoring, which does not require also to refactor the test suite itself in our technique, as may occur sometimes in traditional refactoring scenarios. Our technique is evaluated against a benchmark of* 16 *refactoring cases which present errors when performed by mainstream refactoring tools. This evaluation has been successful in detecting more than* 93% *of those errors.*

1. Introduction

The term refactoring was coined by Opdyke [Opdyke and Johnson 1990, Opdyke 1992] as behavior-preserving program transformations that improve some quality (reusability, maintainability) of the resulting code, later popularized by Fowler [Fowler 1999]. The cornerstone of his definition is that refactorings must maintain correct compilation and the observable behavior. In practice, refactoring is either performed by manual steps, which are error-prone and time consuming, or by *refactoring tools*.

Program refactorings in IDEs are commonly implemented in an *ad hoc* way, since checking correctness with respect to a formal semantics is prohibitive. Therefore, often refactoring tools allow transformations that do not compile or preserve behavior. Daniel et al. [Daniel et al. 2007] present several errors in refactoring tools that cause compilation problems. On the other hand, Schäfer et al. [Schäfer et al. 2008] and Ekman et al. [Ekman et al. 2008] detect errors in automated refactorings regarding behavior preservation. Some authors also show some possible errors, by formalization of refactorings [Borba et al. 2004, Massoni et al. 2008].

Mostly, these defects result from omissions in tool development processes and detecting the errors is not a simple task which can be easily done by the developers. The tools usually lack appropriate theories specifying all conditions that are required to refactor with behavior preservation – the transformations are not proven sound with respect to a formal semantics. For instance, Cornélio [Cornélio 2004] formalizes a comprehensive catalog of refactorings – according to a formal semantics given to a Java-like language – but the results from this and other similar approaches are hardly applied in developing refactoring tools. Formally verifying refactorings is indeed a challenge [Schäfer et al. 2009], and this effort will surely take a long time before it becomes industry standard. Currently, we need a better way to deal with *semantic errors* (non behavior-preserving transformations). This is the goal of this paper.

The current practice to avoid refactoring errors relies on compilation and tests to assure semantics preservation, which may not be satisfactory to critical software development. Compilations errors are simple to identify by IDEs; they only check whether the target version is amenable to correct compilation. However, often test suites are not good at catching semantic errors during transformations. Moreover, it is common in refactoring tools that the test suite also gets refactored, which is unsuitable, as the original intention for the test suite may be lost.

In this paper, we propose a technique for generating unit tests for sequential object-oriented programs (we focus on Java) that are useful for identifying semantic errors, specially for within refactoring tools. Our approach generates tests for the common parts of the *source* and *target* versions of a program – they represent the program before and after applying the refactoring, respectively. In order to avoid refactoring the test suite, our approach only generates tests that can be run both in the source and target programs. This increases the probability in finding errors. In our approach we generate unit tests, run on the source version, and only if no test fails, the same test suite is also run on the target version. If some defect is found on target, we conclude the behavior has changed and hence the transformation is not safe to be applied. It is important to mention that our approach can be also combined with traditional testing approaches [Fowler 1999] as well.

We evaluate our technique in 16 non-trivial transformations that were manually identified by Ekman et al. [Ekman et al. 2008] and by us. These transformations are contained in the catalog of mainstream refactoring tools (such as Eclipse [Eclipse.org 2009], Netbeans [Sun Microsystems 2009], JBuilder [Embarcadero Technologies 2009] and IntelliJ [Jet Brains 2009]), but they introduce subtle semantic errors in at least one of the tools. As result, our approach identified the semantic errors in 93% of the subjects. The main contributions of this paper are the following:

- An approach for generating a test suite for sequential object-oriented programs that is useful for checking refactoring safety (Section 3);
- An evaluation of 16 transformations introducing behavioral changes that are not detected by the best refactoring tools. (Section 4).

2. Motivating Example

In this section, we demonstrate the problem with refactoring tools by means of a motivating example. It shows a transformation applied by a refactoring tool that presents subtle errors in maintaining program behavior. Opdyke [Opdyke and Johnson 1990, Opdyke 1992] proposed refactorings as behavior-preserving program transformations in order to support the iterative design of object-oriented application frameworks.

Each refactoring contains a number of conditions by which it is supposed to preserve behavior. For instance, in order to introduce a class, naming conflicts must be

Figure 1. Rename Variable - Incorrect handling of super accesses

Listing 1. Source Program

```
class Circle {
  static double roundPI = 3.2;
  double radius = 1;
  double area(){
    return PI*radius*radius;
  }
}
```

Listing 2. Target Program

```
class Circle {
  static double PI = 3.2;
  double radius = 1;
  double area(){
    return PI*radius*radius;
  }
}
```

absent. Besides conditions that are useful to preserve program well-formedness, refactorings may also include conditions that preserve its semantics. The perfect scenario is a catalog of refactorings whose conditions are proved to be enough for preserving semantics, probably with respect to a formal semantics (in a theorem prover, for instance) stating the minimum set of conditions. However, this activity is very time consuming and requires considerable expertise.

Don Roberts [Roberts 1999] proposes the refactoring browser tool. Later on, several tools were built, such as: Eclipse, JBuilder, IntelliJ, JDeveloper and NetBeans. Theories that formally state some refactoring conditions are hardly applied in tool development. In order to check refactoring correctness, developers must assure behavior preservation by successive compilation and tests. This is the cornerstone of Opdyke's definition, by which refactorings must maintain correct compilation and observable behavior.

In order to understand our example, consider a graphical design system containing circles, rectangles and other geometric shapes (**Subject 2** – we are going to evaluate it using our technique in Section 4). Listing 1 shows a program with the $Circle$ class. It contains a method called $area$, which uses the static variable PI from standard package $Math$, but not using a qualified name.

In this example, we can apply the Rename Variable refactoring, renaming the static variable $roundPI$ to PI. The resulting code is presented in Listing 2. If this refactoring is applied in NetBeans and JDeveloper, the behavior is not preserved [Ekman et al. 2008]. In the case a circle with radius 1 has in the source program an area equals to 3.14 (with precision of 2 digits), which is different from the area of the target program (which is 3.2). Therefore, the transformation does not preserve behavior, then a *semantic error* is introduced. Eclipse detects this semantic error and prevents the developer from applying this transformation.

The above problem is small, likely to be detected promptly by a developer or a test suite. However, more subtle semantic errors may be very difficult to pinpoint specially in larger programs. Moreover, different refactoring tools may check a different set of enabling conditions, as noticed in the previous example. This scenario is not desirable in practice, which may reduce overall software quality.

Tests play an important role in refactoring. However, it is common during refactorings that test suites also get refactored. Consequently, they may fail to detect errors introduced by IDEs. A simple example is when we apply Rename Method refactoring. In this case, the source and target program are checked against different tests suites. Additionally, the Rename Method refactoring may not preserve the behavior of a program in some situations, as shown by Ekman et al. [Ekman et al. 2008]. Therefore, we should be careful in these situations.

In this paper, we propose a technique which generates a test suite for refactorings in Section 3 that is useful for finding non-trivial semantic errors as shown in Section 4. The generated test suite, which can be run on source and target versions, exercise the changes introduced by the transformation.

3. Technique

In this section, we propose a technique for generating a set of unit tests that may be useful for finding semantic errors after sequential object-oriented program refactorings. Our goal is to generate a single suite of unit tests to be run on both source (original) and target (refactored) versions of a program. This property can improve confidence that the refactoring is correct, since it avoids running different test suites to each program, a possible source of bugs in automatic refactoring. Next we present our technique, which currently is specific for Java, although it can be similarly used for other object-oriented languages.

The technique is broken in five sequential steps for each refactoring application, as depicted in Figure 2. Firstly (Step 1), a static analysis identifies methods in common in both source and target versions; these methods will be the aim of the generated tests. Step 2 aims at generating unit tests for methods identified in Step 1. Notice that the first two steps ensure that the same tests can be run on the source and target versions. In Step 3, the generated test suite is run on the source. If no test fails, the same test suite is also run on the target version (Step 4). If the tests run successfully on the Step 4, the programmer can have more confidence that the transformation does not introduce behavioral changes. Therefore, the refactoring can be safely applied (Step 5).

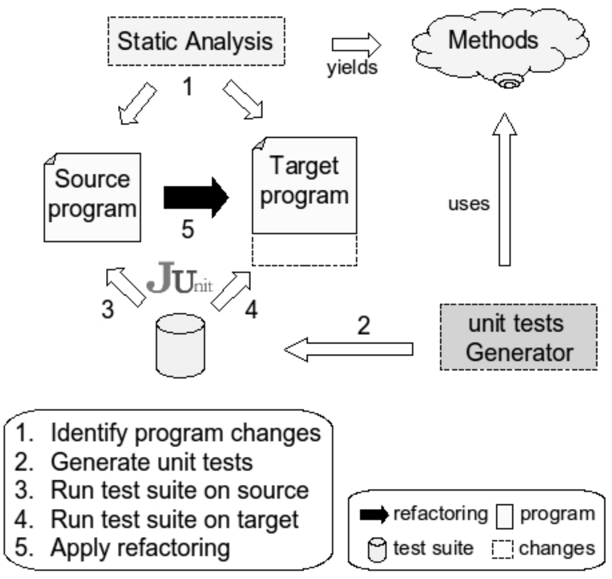

Figure 2. Refactoring Technique

In order to achieve this goal, our technique performs a *static analysis* on the source and target versions (Step 1). The idea is to identify a set of methods that are appropriate for generating tests, aiming at detecting behavioral changes. Since we want to run the same test suite in both versions, we find a set of methods that are in common in both versions. Several options may be implemented for mechanizing this technique. For instance, one can use Java reflection in order to gather common methods from the source and target versions. We consider as common methods only methods that have exactly the same modifier, return type, name, parameters types and exceptions thrown. For example, in Listing 1, both source and target programs include the $area$ method, with identical signatures.

Additionally, there are several alternatives to obtain the target version of the program for static analysis, even before the final application of the automatic refactoring. Refactoring IDEs, such as Eclipse, present features that allow users to visualize a *preview* version of the refactoring – then the static analysis can be applied to this previewed version. Also, the refactoring can be applied, then the program gets analyzed, and if the whole process detects semantic errors, the refactoring can be undone (IDEs also present the *undo* feature for refactorings).

After identifying a set of useful methods, Step 2 is carried out, in which *unit tests* are generated. For this task, RANDOOP can be used; it is fully automatic for randomly generating tests cases, requiring no inputs from the user. RANDOOP has found serious errors in widely-deployed commercial and open-source software, such as Sun's JDK 1.5 [Pacheco et al. 2007]. A test suite that is generated by RANDOOP typically consists of a sequence of method calls that create and mutate objects with random values, plus an assertion about the result of a final method call. Listing 3 presents a test case generated by RANDOOP that has found errors in the JDK 1.5 Collection API. This test consists in exercising a $LinkedList$ being wrapped by a $TreeSet$ object. Next, the $Collections$ utility class is exercised through the $unmodifiableSet$ method, whose return value's equality is unsuccessfully tested.

Listing 3. A test case generated by RANDOOP that reveals an error in JDK 1.5 Collection API.

```
public static void test1() {
  LinkedList l1 = new LinkedList();
  Object o1 = new Object();
  l1.addFirst(o1);
  TreeSet t1 = new TreeSet(l1);
  Set s1 = Collections.unmodifiableSet(t1);
  Assert.assertTrue(s1.equals(s1)); // This assertion fails
}
```

The original RANDOOP algorithm takes four parameters: a list of target classes for the tests, a list of contracts, a list of filters, and a time limit after which the generation process halts. The default filters, contracts and time limit (10 seconds) can be used. For the technique, we use as input the list of classes (the location where the methods are defined) resulting from the static analysis performed in Step 1. More specifically, we modified RANDOOP to consider additional arguments. For instance, it is configured to generate tests that only call methods that are common from classes present in the source and target versions of the program. RANDOOP contains a method $canUse(Method\ m)$

in the $randoop.util.DefaultReflectionFilter$ class. It defines whether a method can be used in a test sequence of a test case. Besides the default checks performed by RANDOOP, we also include an additional verification on whether the method m belongs to the set of methods identified in Step 1. These changes aim at running that the same test suite on both versions, without modifications.

RANDOOP contains a set of default contracts. For example, $s1.equals(s1)$ must be true, as presented in Listing 3. Users can extend these with additional contracts, including domain-specific ones. A contract is created programmatically by implementing a public interface. Algorithm 1 summarizes Steps 1 and 2. **Require** clause defines the input parameters. In this algorithm, the $static - analysis$ method (line 1) from Step 1 defines the methods to be tested, from an intersection of classes from both versions of the program. The $modified - randoop$ (line 4) method generates a test suite (Step 2).

Algorithm 1 Test Suite Generation Algorithm

```
Require: source program $p1$.
Require: source program $p2$.
1: $methods \leftarrow$ static-analysis($p1$,$p2$)
2: $classes \leftarrow$ classes($p1$) $\cap$ classes($p2$)
3: $timelimit \leftarrow 10$
4: $suite \leftarrow$ modified-randoop ($methods$,$classes$,$timelimit$)
5: return $suite$
```

Then we can *run the generated test suite* using the JUnit framework, first on the source version (Step 3). If it runs successfully, then we run on the target version (Step 4). Developers can also run their custom made test suite in order to increase confidence that the transformation does not introduce behavioral changes. Finally, if the tests run successfully on both versions, the refactoring can be accomplished (Step 5). It is important to mention that our technique does not prove that the refactoring is correct, but it improves the developer's confidence that a given transformation preserves behavior.

4. Evaluation

In the following subsections, we show how our technique can detect a number of non-behavior preserving transformations that are considered to be refactorings by extensively tested industrial-strength tools [Schäfer et al. 2009], such as Eclipse, JBuilder, NetBeans, and IntelliJ. As mentioned before, we focus on semantic errors, since compilation errors are cheap to be detected by any compiler (we just have to compile the resulting code).

Next we characterize the subjects (Section 4.1) and give more details about some of them (Sections 4.2, 4.3and 4.4). Finally, we present a discussion (Section 4.5) about our results.

4.1. Characterization

Firstly, we describe our subjects (transformations that introduce behavioral changes that are not detected by the best refactoring tools). We considered 16 subjects in our experiment, including most transformations identified by Ekman et al. [Ekman et al. 2008] and

Figure 3. Push Down Method Refactoring - Incorrect handling of super accesses

Listing 4. Source Version

```
public class A {
  public int k(){return 23;}
}
public class B extends A {
  public int k(){return 42;}
  public int m(){
    return super.k();
  }
}
public class C extends B {
  public int test(){
    return new C().m();
  }
}
```

Listing 5. Target Version

```
public class A {
  public int k(){return 23;}
}
public class B extends A {
  public int k(){return 42;}
}
public class C extends B {
  public int m() {
    return super.k();
  }
  public int test(){
    return new C().m();
  }
}
```

others identified by us. We also included some subjects that are very similar to the subjects from Ekman et al., although with variations that improve evaluation. Each subject is composed by a small set of classes which can report non-trivial errors in refactorings tools. For instance, our subjects contain methods with primitive and non-primitive parameters and return types; we also consider void methods. Due to RANDOOP, the current implementation of our technique only generates tests for public methods of sequential programs. Moreover, our technique does not work when threads are considered. For evaluating each subject, we used an automated tool we have developed. For this purpose, we kept separate source and target versions implemented in Java – the refactoring is applied before the evaluation. This separation does not invalidate the results achieved, since the focus is on the effectiveness of the technique in finding semantic errors during refactorings activities. We considered in our evaluation the following refactorings: Extract Method, Encapsulate Field, Pull Up Method, Push Down Method and Rename (Class, Field, Method and Local Variable). Some of them are described in detail in this section; the entire set of tested subjects are showed in Table 1. All subjects present a behavioral change that goes undetected by at least one refactoring tool: Eclipse 3.4.2, JBuilder 2007, NetBeans 6.0, JDeveloper 11g and IntelliJ 8.0.

4.2. Push Down Method

Subject 7. Listing 4 shows a program with three classes amenable to refactoring.

Suppose that we would like to apply the Push Down Refactoring [Fowler 1999], pushing down the method m from class B to C. The resulting code is presented in Listing 5. We can apply this refactoring in Eclipse, IntelliJ, JBuilder and NetBeans. However the behavior is not preserved. If we run the $test$ method in the source and target versions, it yields 23 and 42, respectively.

Using our technique, when we analyze the versions of Figure 3, we first identify the methods in common in both versions: $A.k()$, $B.k()$ and $C.test()$; the method m does not is considered because it is in different classes in source and target versions. Then, we use our modified RANDOOP to automatically generate a number of unit tests specifically

Figure 4. Rename Method Refactoring - Renaming Method Can Lead to Shadowing of Statically Imported Method

Listing 7. Source Version

```java
import static java.lang.String.*;
public class A {
  static String m(int i) {
    return "42";
  }
  public int test() {
    return Integer.parseInt(
           valueOf(23));
  }
}
```

Listing 8. Target Version

```java
import static java.lang.String.*;
public class A {
  static String valueOf(int i){
    return "42";
  }
  public int test() {
    return Integer.parseInt(
           valueOf(23));
  }
}
```

applied to these methods. In the previous example, it generates 1943 tests. Finally, we run them in JUnit and find that 1807 tests fails in the target version. Listing 12 presents a test case generated by RANDOOP. This test case passes in the source version, but not in the target one.

Listing 6. Test Case that reveals an error

```java
public void testclasses7() throws Throwable {
  C var0 = new C();
  int var1 = var0.test();
  assertEquals((int)23, (int)(java.lang.Integer)var1);
}
```

4.3. Rename Method

Subject 4. Listing 7 shows a program to which the Rename refactoring is applied, specifically to method m that becomes $valueOf$ in the class A. In the source version, $valueOf$ presented in $test$ is imported from the class $String$. The resulting code is presented in Listing 8. The method $valueOf$ presented in $test$ refers to the local $valueOf$ method defined.

We can apply this refactoring in Eclipse, JBuilder, NetBeans, and JDeveloper. If we run the $test$ method in the source and target versions, it yields 23 and 42, respectively. Therefore, it does not preserve behavior. Our technique generates 100 unit tests. JUnit reports 99 failures in the target version.

4.4. Encapsulate Field

Subject 5. Listing 9 shows a program with two classes and a public attribute i. We would like to apply Encapsulate Field refactoring in the local public variable i. The resulting code is presented in Listing 10.

We can apply this refactoring in Eclipse, JBuilder, NetBeans, IntelliJ, and JDeveloper. If we run the $getValue$ method in the source and target versions, it yields 23 and 42, respectively, showing that behavior is not preserved. Using our technique, we generate 2098 unit tests. JUnit reports 2095 failures in the target.

Figure 5. Encapsulate Field Refactoring - No Check for Overloading Problems

Listing 9. Source Version

```
public class A {
  public int i;
}
public class B extends A {
  public int getI() {
    return 42;
  }
  public int getValue() {
    A a = new B();
    a.i = 23;
    return a.i;
  }
}
```

Listing 10. Target Version

```
public class A {
  private int i;
  public void setI(int i) {
    this.i = i;
  }
  public int getI() {
    return i;
  }
}
public class B extends A {
  public int getI() {
    return 42;
  }
  public int getValue() {
    A a = new B();
    a.setI(23);
    return a.getI();
  }
}
```

4.5. Discussion

The current implementation of our technique detects semantic errors of public methods that return values (*int*, *double* or objects). Suppose the example presented in Listing 11.

Listing 11. A subject containing a method returning an Object.

```
public class A {
  public int k() { return 23; }
}
public class B extends A {
  public int k() { return 42; }
  public int m() { return super.k(); }
}
public class C extends B {
  public D test(){
    D d = new D();
    d.setF(m());
    return d;
  }
}
public class D {
  int f = 23;
  public void setF(int x) {
    this.f = x;
  }
  public int getF() {
    return this.f;
  }
}
```

Applying the Push Down Method refactoring by moving m from class B to class C, the method $test$ yields a D object with a different behavior. With our technique, this kind of error can be detected, since RANDOOP generates the following test case presented in Listing 12:

Listing 12. RANDOOP Test Case

```
public void testclasses22() throws Throwable {
  C var0 = new C();
  D var1 = var0.test();
  int var2 = var1.getF();
  assertEquals((int)23, (int)(java.lang.Integer)var2);
}
```

Notice that the test sequence contains a call to $test$, and it applies the method $getF$. The getter methods must be present, so RANDOOP can generate a sequence of calls that exposes the semantic error. However, if the method is $void$, we may not detect a semantic error if the class does not declare the appropriate getter methods for the fields. Accordingly, this need is certainly a guideline that developers should follow when using our approach.

Our benchmark is composed by subjects that cover different errors [Ekman et al. 2008]. For instance, a Push Down Method refactoring may lead to incorrect handling of super, field, type and private method accesses, whereas the Extract Method may lead to incorrect handling of dataflow analysis. The Encapsulate Field refactoring may introduce a problem due to overloading problems. Moreover, some errors are caused by Rename refactorings: undiagnosed shadowing, shadowing of inherited member types, shadowing of static import, shadowing by fields and shadowing of statically imported method.

In summary, our technique detected problems in all 16 subjects but 1. Table 1 shows the complete results of our experiment. It shows the number of classes and methods that are in common in both versions, the number of tests generated by RANDOOP, and the number of tests that identify semantic errors in the target version. We only include subjects small enough to expose the error. If we include larger examples, RANDOOP tends to generate more tests. It also may be interesting to increase the time limit, so more tests can be generated in order to exercise the changes introduced by the transformation.

Subject 14 contains a difference in the standard output ($System.out.println$) messages, which the current implementation of our technique cannot identify. As a future work, we aim at changing RANDOOP to generate tests based on testing patterns that are useful for identifying those kinds of errors.

Finally, Subject 15 extracts a class with the same name ($Point$) of a different class in another package ($java.awt.Point$). In this case, behavior is changed. Since the class is renamed, we do not generate any test for the new and old classes. We generate a test suite that can be run in both versions. If the common base code does not exercise the modified parts, it is difficult to detect a semantic error introduced by renaming refactorings. As the technique focus on common methods and classes, names are important. However, errors introduced by the Rename Field refactoring are easier to be detected. Subject 15 may be very rare in practice, since it is easy to introduce a compilation error. It is common to

Table 1. Summary of the Refactoring Experiment

Subject	Classes	Methods	Refactoring	Tests	Errors
1	4	2	rename class	101	99
2	1	1	rename field	6	4
3	1	2	rename local variable	100	99
4	1	1	rename method	100	99
5	2	2	encapsulate field	2098	2095
6	1	2	extract method	2740	2737
7	3	3	push down method	1943	1807
8	2	1	push down method	101	99
9	3	3	push down method	101	99
10	2	2	push down method	101	99
11	3	3	pull up method	4	1
12	3	3	push down method	2190	2145
13	3	5	push down method	2682	548
14	3	3	push down method	300	0
15	2	0	extract class	100	99
16	3	4	push down method	3263	1404

some class to have a $java.awt.Point$ variable calling a method that is not presented in the new extracted class. Therefore, we may have a compilation error. In our case, this does not happen because the experiment only contains a few classes and methods.

5. Related Work

Refactoring tools may contain bugs. Ekman et al. [Ekman et al. 2008] catalog a number of non-trivial compilation and semantic errors resulting from refactoring tools, when applying refactorings such as Rename Element and Encapsulate Field. Schäfer et al. [Schäfer et al. 2008] present an approach to solve the problem for rename refactoring. We present a simple technique for finding semantic errors, regardless the refactoring category. It is hard to examine all kinds of refactorings in order to define all enabling conditions. Their benchmark [Ekman et al. 2008] is used in this paper as a benchmark, and our technique was able to pinpoint all their semantic errors, except two of them: one with concurrency, that our technique does not support; and other with changes in the standard output ($System.out.println$). The current implementation of our technique cannot detect them.

Daniel et al. [Daniel et al. 2007] propose an approach to check whether refactoring tools do not introduce compilation errors, which are simple to identify. Their approach generates a number of inputs to check whether the tools are correct. This approach complements ours. We focus on finding behavioral changes that are more difficult to be detected.

Formally verifying program refactorings is a challenge [Schäfer et al. 2009]. Some approaches have been contributes in this direction. An approach [Gheyi 2007] proposes and formally proves in a theorem prover with respect to a formal semantics and formal equivalence notion encoded in Prototype Verification System (PVS) a number of refactorings for a formal specification language (Alloy). Borba et al. [Borba et al. 2004, Borba et al. 2003, Cornélio 2004] propose a set of program refactorings for the Refinement Object-Oriented Language (ROOL), which is a sequential Java-like language but with copy semantics. A set of primitive laws (bidirectional program refactorings) is defined, and proved that they are behavior preserving based on a weakest preconditions semantics for ROOL [Borba et al. 2004]. By composing their laws, they formalize several refactorings, such as *Pull Up Field* and *Push Down Field* refactorings [Fowler 1999]. A closely related approach is developed by Tip et al. [Tip et al. 2003]. They realized that some enabling conditions and modifications to source code for refactorings involving generalization, for automation in Eclipse, depend on relationships between types of variables. These type constraints enable the tool to selectively perform transformations on source code, avoiding type errors that would otherwise prohibit the overall application of the refactoring. They proved that these refactorings preserve typing. However the previous approaches focus on a restricted set of Java and do not consider dynamic semantics, respectively. Proving that a refactoring for Java is sound considering full Java is a bit more difficult. So in this paper, we propose a more practical approach for identifying semantic errors.

Murphy-Hill and Black [Murphy-Hill and Black 2008b, Murphy-Hill and Black 2008a] characterize a number of problems with current refactoring tools. They report on a user study that compares their new tools proposed against existing tools. The results of the study show that speed, accuracy, and user satisfaction can be increased. They extend the set of usability recommendations proposed by Roberts [Roberts 1999] and propose new ones. Our work is complementary. As mentioned before, since the conditions of each refactoring are not formally defined, it is very simple to compromise the refactoring tools correctness. In our work, we help developers catching semantic errors that are not captured by refactoring tools.

6. Conclusions

In this paper, we propose a technique for generating tests for sequential object-oriented program refactorings. Since it is well known that refactoring tools contain defects [Ekman et al. 2008], our goal is to increase the chances to identify semantic errors. We do not focus on compilation errors since they are cheap to detect. As tools present defects, we generate a test suite that can be run on the source and target versions. If the test suite runs successfully, developers improve confidence that the transformation preserves behavior. We also developed an Eclipse plug-in [1] in order to automate the steps of our technique. We applied our technique to 16 tricky transformations, divided in 5 kinds of refactoring, namely: Extract Method, Encapsulate Field, Pull Up Method, Push Down Method, Rename (Class, Field, Method and Local Variable). Three out of five kinds of refactorings are in the top four mostly used refactorings by Eclipse developers [Murphy et al. 2006, Murphy-Hill and Black 2008a]. In 93% of the cases, our approach identified the semantic errors. We consider subjects containing methods returning any primitive or $Object$ types. It is important to mention that our approach is complementary to the traditional refactoring steps. The developer can continue to use the traditional approach [Fowler 1999], using ours to increase the confidence that the transformation pre-

[1]The tool is available at: `http://www.dsc.ufcg.edu.br/~spg`

serves behavior. If a transformation introduces a behavioral change, our technique shows a test case for developers indicating a method call sequence that introduces the change.

As a future work, we aim at using our technique in a real case study. Additionally, we aim at proposing some specific RANDOOP contracts for each refactoring. This may be useful to increase the chances to find more semantic errors. Modifying RANDOOP to consider non public methods may be another direction.

In the static analysis step, we only consider the classes and methods that are in common in the source and target versions. Sometimes our approach does not perform well if we consider the Rename method and class refactorings. The technique does not generate tests for them. We need a common base code that exercises the changes in order to detect a problem. Notice that we considered subjects that apply Rename Method, Field and Local Variable refactorings and our approach detected the behavioral changes. As a future work, we intend to evaluate our approach in more refactorings and propose some specific strategies. Additionally, we intend to generate some test patterns that are useful for detecting differences in the messages presented in the standard output. Our current implementation does not detect differences in those messages.

We can improve our current static analysis step by taking into account the hierarchy or other object-oriented property. For example, if the method m is push down from class A to B, m is not included in the resulting set using the static analysis since m has a different signature in the source an target versions. It would be useful if we include the method m called from class B, hence satisfying both versions. Another approach can refine the set of methods by taking into account the refactoring characteristic. For example, it would be interesting only to consider methods that exercise the changes. In this case we can have a small test suite, hence taking less time to run all tests. Even with our current *static analysis*, we found good results, as presented in Section 4.

Acknowledgments

We would like to thank all members of the Software Productivity Group (SPG) at UFCG. This work was supported by CAPES and CNPq, grant 484813/2007-2, and by the National Institute of Science and Technology for Software Engineering (INES [2]), also funded by CNPq, grant 573964/2008-4.

References

Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100.

Borba, P., Sampaio, A., and Cornélio, M. (2003). A refinement algebra for object-oriented programming. In *European Conference on Object-Oriented Programming*, pages 457–482.

Cornélio, M. (2004). *Refactorings as Formal Refinements*. PhD thesis, Federal University of Pernambuco.

Daniel, B., Dig, D., Garcia, K., and Marinov, D. (2007). Automated testing of refactoring engines. In *Foundations of Software Engineering*, pages 185–194.

[2]http://www.ines.org.br

Eclipse.org (2009). Eclipse project. At http://www.eclipse.org.

Ekman, T., Ettinger, R., Schafer, M., and Verbaere, M. (2008). Refactoring bugs in eclipse, idea and visual studio. At http://progtools.comlab.ox.ac.uk/refactoring/bugreports.

Embarcadero Technologies, I. (2009). Jbuilder. At http://www.codegear.com/br/products/jbuilder.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gheyi, R. (2007). *A Refinement Theory for Alloy*. PhD thesis, Federal University of Pernambuco.

Jet Brains, I. (2009). Intellij idea. At http://www.intellij.com/idea/.

Massoni, T., Gheyi, R., and Borba, P. (2008). Formal model-driven refactoring. In *Fundamental Approaches to Software Engineering*, pages 362–376.

Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83.

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

Murphy-Hill, E. R. and Black, A. P. (2008b). Breaking the barriers to successful refactoring: observations and tools for extract method. In *International Conference on Software Engineering*, pages 421–430.

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

Opdyke, W. and Johnson, R. (1990). Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Object-Oriented Programming emphasizing Practical Applications*, pages 145–160.

Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84.

Roberts, D. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign.

Schäfer, M., Ekman, T., and de Moor, O. (2008). Sound and extensible renaming for java. In *Object-oriented programing, systems, languages, and applications*, pages 277–294.

Schäfer, M., Ekman, T., and de Moor, O. (2009). Challenge proposal: Verification of refactorings. In *Programming Languages meets Program Verification*, pages 67–72.

Sun Microsystems, I. (2009). Netbeans ide. At http://www.netbeans.org/.

Tip, F., Kiezun, A., and Baumer, D. (2003). Refactoring for Generalization Using Type Constraints. In *Object-oriented programing, systems, languages, and applications*, pages 13–26.

Wloka, J., Ryder, B. G., and Tip, F. (2009). Junitmx - a change-aware unit testing tool. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 567–570, Washington, DC, USA. IEEE Computer Society.