Making program refactoring safer

Gustavo Soares[1]

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

Abstract. Developers rely on compilation, test suite and tools to preserve observable behavior during refactorings. However, most of the refactoring tools do not implement all preconditions that guarantee the refactoring correctness, since formally identifying them is cost-prohibitive. Therefore, these tools may perform non-behavior-preserving transformations. We present a technique and tool (SafeRefactor) for making program refactoring safer. It analyzes a transformation and automatically generates a test suite that is suited for detecting behavioral changes. Moreover, we propose a program generator (JDolly) useful for generating inputs for testing refactoring tools. We have evaluated both in two experiments. First, we used SafeRefactor in seven real case study refactorings (from 3 to 100 KLOC). We reason about an original version of JHotDraw (23 KLOC) and its refactored version, and automatically detected a behavioral change. Developers have not identified this problem before. Finally, we have used SafeRefactor and JDolly to test 12 refactorings implemented by Eclipse 3.4.2. As result, we have detected a number of non-behavior-preserving transformations.

Keywords: Refactoring, Behavior-preservation, Unit-testing, Program Generation, Automated testing

1 Introduction

Refactoring is the process of change a software system in such way that improves its internal structure without changing its external behavioral [6]. Each refactoring may have preconditions that guarantee the behavioral preservation. For example, the Push Down Method refactoring moves a method from the super class to the subclasses. Before we apply this change, we need to check if others methods with same signature already exist in the subclasses. Most used IDEs such as Eclipse, NetBeans, IntelliJ, and JBuilder automate a number of refactorings. They automatically check the preconditions and perform the transformation.

However, IDEs may perform incorrect transformations that introduce compilation errors or change program behavior. Compilation errors are easier to detect; we only need to compile the refactored program. On the other hand, behavioral changes are more difficult to detect, since they are silently introduced by the tool.

Currently, each IDE implements refactorings based on an informal set of preconditions, because establishing it with respect to a formal semantics is prohibitive. An evidence of this fact is that some IDEs allow some transformations, and others do not [4]. Identifying all refactoring preconditions for complex languages as Java is not trivial and formally verifying them is indeed a challenge [17]. The current practice to avoid behavioral changes in refactorings relies on solid tests [6]. However, often test suites do not catch behavioral changes during transformations. They may also be refactored (for instance, rename method) by the tools since they may rely on the program structure that is modified by the refactoring. In this case, the tool changes the method invocations on the test suite, and the original and refactored programs are checked against different test suites. This scenario is undesirable since the refactoring tool may change the test suite meaning [16].

In this work, we propose a technique and tool (SafeRefactor) for improving confidence that a refactoring is sound. It analyzes the transformation and generates unit tests suited for detecting behavioral changes. Moreover, we propose a program generator (JDolly) useful for generating inputs for testing refactoring tools. It is based on Alloy [10], a formal specification language, and ASTGen [4], an imperative framework for generating Java programs. We show an overview of Alloy and ASTGen in Section 3.

We have evaluated SafeRefactor and JDolly in two experiments [1]. First, we evaluated SafeRefactor on seven refactorings of real Java programs (from 3 to 100 KLOC) performed by developers that used refactoring tools and unit tests to guarantee the behavior preservation. We have identified a behavioral change in a refactoring applied to JHotDraw (23 KLOC). Finally, we used SafeRefactor and JDolly to test 12 refactorings implemented by Eclipse 3.4.2. As result, we have detected that many transformations performed by Eclipse change program behavior. In summary, the main contributions of this paper are the following:

– A technique and tool for improving the confidence that a refactoring is sound (Section 4) [23,22,21];
– A Java program generator useful for automated testing refactoring implementations (Section 5) [20];
– An evaluation of 7 refactorings applied to real Java programs (Section 6);
– An evaluation of our approach on automated testing 12 refactoring implemented by Eclipse 3.4.2 (Section 7).

2 Motivating Example

In this section, we show one transformation performed by Eclipse IDE that changes program behavior. For instance, consider the program illustrated in Listing 1.1. It contains the class `A` with the methods `k(int)` and `k(long)`, and the class `B` with the method `test()`. The method `test` returns 10. If we apply the

[1] All experiment data are available at: http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.htm

Eclipse Change Method Signature refactoring to `k(int)` reducing its visibility to `private`, the refactored code will be similar to the one in Listing 1.2. Now, the method `test` returns 20 instead of 10. Therefore, this transformation changes behavior. After the refactoring, `k(int)` is not visible in `B`. However, the method `k(long)` is visible, and matches with the expression k(2) inside `test()`, leading to a change of binding.

Fig. 1. Decreasing visibility leads to behavioral change

Listing 1.1. Original version

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

Listing 1.2. Refactored version

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

The above example is small, and a small test suite can detect the behavioral change. However, it shows the complexity of a language as Java and of identifying refactoring preconditions. Although the program contains only 2 classes and few lines of code, Eclipse incorrect refactor it. Moreover, in large systems, detection of behavioral changes is more difficult.

3 Background

In this section, we show an overview of the technologies used in our program generator JDolly. First, we describe the Alloy language (Section 3.1) and next we show the ASTGen framework (Section 3.2).

3.1 Alloy

Alloy is a declarative language for formal specification [10] based on first order logic and can be used for modeling object-oriented systems. It has syntax, type system, and semantics formally defined, which allows a full automatically analysis of the model. An Alloy model is a sequence of *paragraphs* of three kinds: *signatures* are used to define new types, *facts* are used to record constraints, and analysis paragraphs are used to perform analysis of the model.

To illustrate an Alloy specification, we use an object model example of a Java subset (Figure 2). Each box represents a set of objects, and the arrows are relations between objects. For instance, the arrow labeled `extends` from `Class` to `Class` models the class inheritance of Java. Relations may have multiplicities on both ends. If a multiplicity is omitted, it is unconstrained.

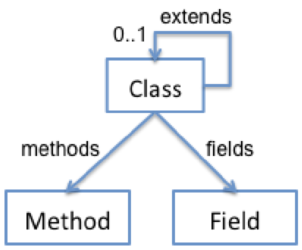


Fig. 2. Object model of a Java subset

Next we describe an Alloy model similar to the previous object model. In Listing 1.3, we specify three signatures that represents the sets of objects of our model. The signature body may contain relations of these objects. Fo instance, in the signature `Class`, we specified the relations `extend`, `fields`, and `method` similar to ones modeled on our object model. Relations may have multiplicity constraints as `set` (unconstrained), `one` (exactly one), `lone` (zero or one), `some` (one or more).

Listing 1.3. Signatures that model classes, fields and methods

```
sig Class {
  extend: lone Class,
  fields: set Field,
  methods: set Method,
  }
sig Field {}
sig Method {}
```

Besides signatures and relations, we can specify *facts* containing invariants of the system. A fact contains a set of constraints that hold for all executions of the system. We can use it for describing well-formed rules of Java. For instance, a class cannot extend itself in Java. Listing 1.4 shows a fact that has a constraint related to this rule. The `in` operator represents a subset relation. The constraint has a variable `c` of `Class` and a quantifier `no` (for none) on the left hand side. On the right hand side, there is a constraint that holds for no `c`. The keyword `^` represents the transitive closure of `c` with respect to `extend`. If we consider the `extend` relation as a graph, the transitive closure represents the set of nodes that can be achieved starting at `c.extend`.

Listing 1.4. Fact that specifies a class cannot extends itself

```
fact JavaConstraints {
  no c:Class | c in c·^extend
}
```

Next we show how to use the Alloy Analyzer to simulate instances of our model. It find all solutions that satisfy the model for a given bound (Bounded-exhaustive generation [12]). In this example, the scope is the number of classes, methods, and fields. For instance, suppose we have specified to Alloy Analyzer find solutions for the scope of three classes, two methods, and one field. Figure 3 shows one of the 858 solutions found. It has the class `Class2` and its subclasses `Class0` and `Class1`.

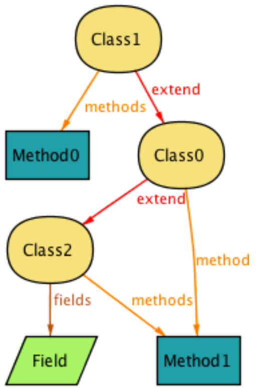


Fig. 3. Solution found by the Alloy Analyzer

3.2 ASTGen

The ASTGen is a Java framework for generating Java programs [4]. It has a number of generators that produce elements of Java abstract syntax trees (ASTs). ASTGen architecture allows the combination of the generators to produce more complex Java code. For instance, Figure 4 shows the structure of a generator that creates fields declarations. It is composed of three sub-generators that create the following elements: modifiers, types, and identifiers. Suppose these sub-generators create the elements of Java abstract syntax: public and private (modifiers), int and long (types), f1 (identifier), respectively. The FieldDeclarationGenerator will produce field declarations combining all values produced by its sub-generators. Figure 4 shows one of the 4 field declarations created.

ASTGen has more than 40 generators that are able to produce elements of Java such as control flow statements (if, while, for), and methods and fields calls expressions.

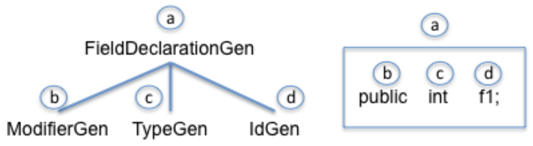


Fig. 4. Example of ASTGen FieldDeclarationGenerator

4 SafeRefactor

In this section, we present SafeRefactor, a tool for improving confidence that a refactoring is sound. SafeRefactor is an Eclipse plugin [2] that receives a source code and a refactoring to be applied (input). It reports whether it is safe to apply the transformation (output).

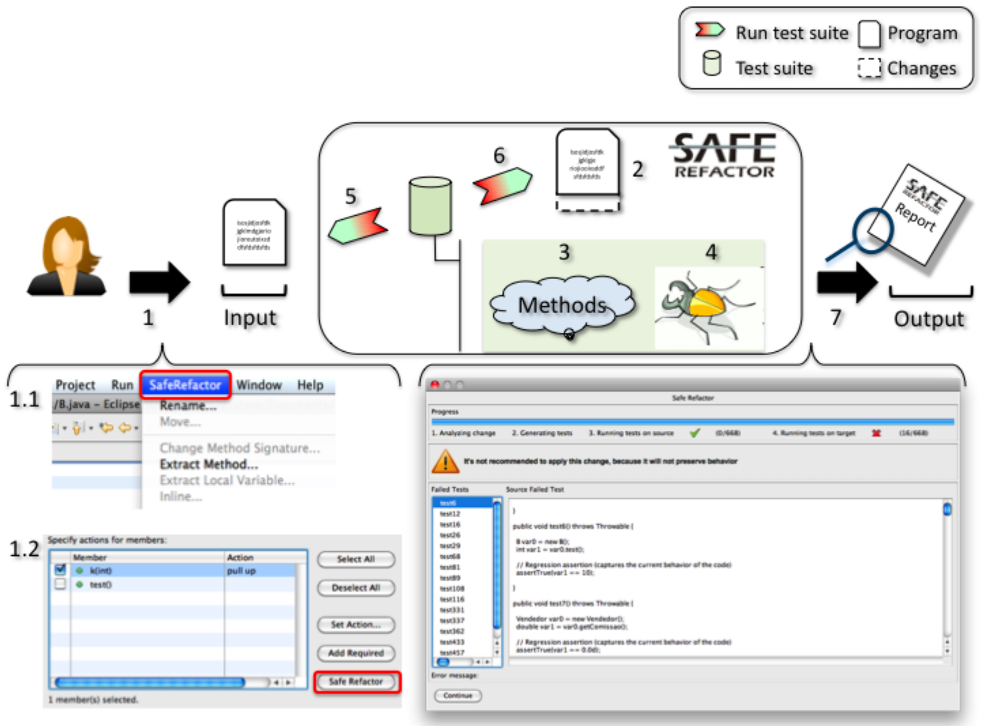
Suppose that we use SafeRefactor in Listing 1.1 program. Next we explain the whole process, which has seven sequential steps for each refactoring application (Figure 5). First the developer selects the refactoring to be applied on the source program (Step 1.1) and uses SafeRefactor (Step 1.2). The plugin starts checking the refactoring safety (Steps 2-7).

It generates a target program based on the desired transformation using Eclipse refactoring API (Step 2). In Step 3, a static analysis automatically identifies methods in common in both source and target programs. Step 4 aims at generating unit tests for methods identified in Step 3. In Step 5, the plugin runs the generated test suite on the source program. Next, it runs the same test suite on the target program (Step 6). If a test passes in one of the programs and fails in the other one, the plugin detects a behavioral change and reports to the user (Step 7). Otherwise, the programmer can have more confidence that the transformation does not introduce behavioral changes.

The goal of the *static analysis* (Step 3) is to identify *methods in common*: they have exactly the same modifier, return type, qualified name, parameters types and exceptions thrown in source and target programs. For example, Listings 1.1 and 1.2 contain `A.k(long)` and `B.test()` in common.

After identifying a set of useful methods, the plugin uses Randoop [15] to generate unit tests (Step 4). Randoop generates tests for classes within a time limit. A unit test typically consists of a sequence of method and constructor invocations that creates and mutates objects with random values, plus a JUnit assertion. Randoop executes the program to receive a feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs [15]. It creates 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. For instance, an object must be equal to itself. Our tool uses the Randoop default

[2] Available at http://www.dsc.ufcg.edu.br/~spg/saferefactor/



It's not recommended to apply this change, because it will not preserve behavior

test6 test12 test16 test26 test29 test68 test81 test89 test108 test116 test331 test337 test362 test433 test457

```
}

public void test6() throws Throwable {

  B var0 = new B();
  int var1 = var0.test();

  // Regression assertion (captures the current behavior of the code)
  assertTrue(var1 == 10);

}

public void test7() throws Throwable {

  Vendedor var0 = new Vendedor();
  double var1 = var0.getComissao();

  // Regression assertion (captures the current behavior of the code)
  assertTrue(var1 == 0.0d);
```

Fig. 5. SafeRefactor. 1. The user selects a refactoring in the menu to apply (1.1) and click on the SafeRefactor button (1.2), 2. The tool generates the target program using the Eclipse refactoring API, 3. It identifies common methods in the source and target programs, 4. The tool generates unit tests using our modified Randoop, 5. It runs the test suite on the source program, 6. The tool runs the test suite on the target program, 7. The tool shows the report to developer. If it finds a behavior change, the user can see some unit tests that fail.

contracts. We modified Randoop to remove some defects, and to pass a set of methods as parameter. All tests generated only contain calls to the methods identified in Step 3. The default time limit is 2s. Steps 3 and 4 ensure that the same tests can be run on the source and target programs.

The whole process to finish Figure 1 example takes less than 8 seconds on a dual-processor 2.2 GHz Dell Vostro 1400 laptop with 2 GB RAM, and generates 154 unit tests (151 of them failed in the target program). SafeRefactor reports the user that the refactoring should not be applied. Users can see some tests that expose the behavior change (Step 7). In other situations, SafeRefactor can report compilation errors that may be introduced by refactoring tools. If SafeRefactor does not find a behavior change or compilation error, it reports that users can improve confidence that the transformation is sound.

Opdyke compares the observable behavior of two programs with respect to the `main` method (a method in common). If it is called twice (source and target programs) with the same set of inputs, the resulting set of output values must be the same [14]. SafeRefactor checks the observable behavior with respect to randomly generated sequences of methods and constructor invocations. They only contain calls to methods in common. If the source and target programs have different results for the same input, they do not have the same behavior.

5 JDolly

In this section, we present JDolly, a program generator useful for generating inputs to test refactoring tools. First, we show its overview (Section 5.1). Next we describe its expressivity (Section 5.2), and the program generation process in detail (Section 5.3 and 5.4).

5.1 Overview

JDolly receives as input the number of packages, classes, fields, and methods. Additionally, testers can specify some structural characteristics programs must have. The generator outputs programs with this scope.

To illustrate the program generation, consider the program shown in Listing 1.5. First, JDolly creates the *structural parts* of the program, depicted in black in the listing. As structural parts we mean `package, class, method, field` declaration statements. This step is based on the declarative programming paradigm, and as enable technology we used the formal specification language Alloy and the Alloy Analyzer [9]. In the second step, JDolly generates the *behavioral parts* (inside the boxes in Listing 1.5), that is, the statements that initialize fields and inside of the methods bodies. For this step, we used the ASTGen [4], a Java framework for imperative generating of Java programs.

Listing 1.5. Program generated by JDolly

```
package p1;
public class A {
  private int f = 10;
  public int k(int i){ return f; }
}

package p2;
import p1.*;
public class B extends A {
  public int m(){ return super.k(2); }
}
```

5.2 Language

JDolly generates programs that contain a subset of the Java elements. In the current implementation, each program may have packages, classes, fields, and methods declarations with full access control (public, protected, private, package). Types can be primitive, String or classes declared in the program. Methods have zero or one parameter. Moreover, these elements can have structural relations such as inheritance, overloading, overriding, and field hiding.

Regarding the behavior of the classes. Fields can have assignment statements for initialization. Methods have only one statement (return) that can have a method or field call expression, or a simple value. Call expressions can use simple name (e.g. m()), qualified name (e.g. A.m()), accesses keyword (e.g. super.m(), this.m()), and new instance (e.g. new A().m()).

Next we describe the program generation process of JDolly. First we focus on the structural generation for latter explain the behavior generation.

5.3 Structural generation

The generation of the structural parts of the programs is based on Alloy [10] and Alloy Analyzer [9]. We specify a meta-model of a Java subset, and use the Alloy Analyzer for finding solutions to this model. Then we transform the solutions in Java code.

Meta-model We model in Alloy a subset of Java abstract syntax using signatures and relations. Listing 1.6 shows our current model. Packages contain imports and classes declarations. Each class has an id, a modifier, fields and methods, and can extend one class. Fields have an id and a modifier, and declare a type that can be primitive, String or a class. Methods have a return type, zero or one argument, besides modifier and id.

Listing 1.6. Meta-model of a Java subset

```
sig Id { }
abstract sig Type {}
one sig Int_,Long_, Char, Boolean, String_ extends Type {}
abstract sig Modifier {}
one sig public, private_, protected, package extends Modifier {}
sig Package {
  classes: set Class,
  imports : set Package
}
sig Class extends Type {
  id: one Id,
  modifier : one Modifier,
  extend: lone Class,
  fields: set Field,
  methods: set Method
}
sig Field {
  id : one Id,
  type: one Type,
  modifier : one Modifier
}
sig Method {
  id : one Id,
  arguments: lone Type,
  return: one Type,
  modifier: one Modifier
}
```

Well-formed rules Java programs must satisfy a number of well-formed rules. For instance, a class cannot extend itself. We have specified some of them in Alloy facts (Listing 1.7). The first constraint specifies this exemplified rule. The next three constraints specify a package cannot have two classes with same id, a class cannot have two fields with same id, and a class cannot have two methods with same signature, respectively. Finally, the last constraint specifies that a package must have an import declaration if one of its classes extends a class of other package.

Listing 1.7. Facts describing Java well-formed rules

```
fact WellFormedRules {
  no c:Class | c in c·^extend

  no c1,c2: Class | some p:Package
      c1 ≠ c2 &&
      c1 + c2 in p·classes &&
      c1·id == c2·id

  no f1,f2: Field | some c: Class |
    f1 ≠ f2 &&
    f1 + f2 in c·fields &&
    f1·id == f2·id

  no m1,m2: Method | some c: Class |
     m1 ≠ m2 &&
     m1 + m2 in c·methods &&
     m1·id == m2·id &&
     m1·arguments == m2·arguments

  no p1,p2: Package | some c1,c2: Class |
    p1 ≠ p2 &&
    c1 ≠ c2 &&
    c1 in p1·classes &&
    c2 in p2·classes &&
    c1 in c2·extend &&
    p1 !in p2·imports
}
```

Generating programs with specific structural characteristics Sometimes testers want to produce only programs with a specific structural characteristic. For example, to test the pull up method refactoring implementation, the programs must have at least a class and its subclass with one method. Listing 1.8 shows the specification used to generate programs with these characteristics. The first line models classes C1 and C2, and the fact `PullUpMethod` specifies that C2 extends C1, and C2 contains one method.

Listing 1.8. Pull Up Method Specification

```
 one sig C1,C2 extends Class {}
 fact PullUpMethod {
   C1 in C2·extend
   one m:Method | m in C2·methods
}
```

Transformation The Alloy distribution has a Java API [3] to manipulate the Alloy Analyzer commands in Java. We use it to handle the Alloy solutions, and then transform these solutions in Java code elements. To do so, we use the Java Model Eclipse API [3] , which contains several classes to model elements of Java syntax.

5.4 Behavioral generation

The next step of the generation is the creation of the behavioral parts of the programs. We have implemented in JDolly an imperative generator based on ASTGen to this function. For each structural part generated on step one, this generator creates a number of programs with distinct behavior.

The behavioral generator is composed by n generators of field initialization and m generators of method body where n and m is the number of fields and methods of the program, respectively. The current version of these generators uses the following ASTGen generators:

- ReturnStatementGenerator creates return statements that contain expressions generated by sub-generators. In our case, expressions can be method and field calls or literal values;

[3] Eclipse Java Model API tutorial: `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulationAST/index.html`

– MethodCallGenerator creates four kinds of calls to a specific method;
– FieldCallGenerator is similar to above one but for fields;
– AssignmentGenerator creates assignment statements with expressions of sub-generators. In our case, these expressions can be literal values;
– StringLiteralGenerator, NumberLiteralGenerator, and BooleanLiteralGenerator create strings, numbers, and boolean values, respectively.

The behavioral generator explores all combinations of values produced by the sub-generators. For example, suppose it receives as input a structural part of a program with one class, two methods, and one field. For each method, it produces 9 different return statements: 4 kinds of method calls to the other method, 4 kinds of field calls, and one literal value. The field is initialized with one literal value. The combination of values results in 81 different behavior to this program structure. Additionally, we implement filters related to well-formed rules for removing of some invalid behaviors as a method call using super to a method in same class.

6 Evaluating refactorings in real Java programs

In this section, we describe our experiment to analyze SafeRefactor with respect to the efficiency of finding behavioral changes in likely refactorings applied to real Java programs. First, we describe the subjects (Section 6.1) and the experiment setup (Section 6.2). Finally, we show the experiment results (Section 6.3).

6.1 Subject Characterization

Table 1 shows the subjects (pairs of source and target programs) used in the experiment. Each of them is uniquely identified (Subject column). They consist of refactorings performed by developers applied to real Java applications ranging from 3-100 KLOC (non-blank, non-comment lines of code) using tools or manual steps. All of them are considered behavior preserving by developers. We use SafeRefactor to evaluate whether the transformation preserves the observable behavior. Columns Program, KLOC, and Refactoring show the program name, its size in KLOC, and the refactoring applied, respectively. Next we describe these transformations.

Third-party developers performed a refactoring on JHotDraw and CheckStylePlugin (Subjects 1-2) to modularize exception handling code [25]. Fuhrer et al. [7] proposed and implemented an Eclipse refactoring to apply the infer generic type argument refactoring, enabling applications to use Java generics. They evaluated their refactoring in four real Java applications: JUnit, Vpoker, ANTLR, and Xtc (Subjects 3-6). Murphy-Hill et al. [13] performed some experiments to analyze how developers refactor. They used a set of twenty Eclipse components versions from Eclipse CVS, and manually detected the refactorings applied. We evaluate one transformation in Eclipse TextEditor module (Subject 7).

Subject	Program	KLOC	Refactoring	Tests	Error	Total Time (s)	Result
1	JHotDraw	23	Extract Exception Handler	2245	273	148	Behavior Change
2	CheckStylePlugin	20	Extract Exception Handler	5864	0	235	-
3	Junit	3	Infer Generic Type	1127	0	99	-
4	Vpoker	4	Infer Generic Type	466	0	109	-
5	ANTLR	32	Infer Generic Type	-	-	2	Compilation Error
6	Xtc	100	Infer Generic Type	-	-	4	Compilation Error
7	TextEditor	15	Replace Deprecated Code	16009	0	107	-

Table 1. Summary of SafeRefactor Evaluation in Refactoring Real Applications

6.2 Experimental Setup

We run the experiment on a dual-processor 2.2 GHz Dell Vostro 1400 laptop with 2 GB RAM and running Ubuntu 9.04. In both categories, we used a command line interface provided by our SafeRefactor. It receives three parameters: source and target program paths, and timeout to generate tests. We used the default timeout of 90s and 2s to generate the tests in the first and second categories, respectively. We did not use the SafeRefactor Eclipse graphical interface in order to automate the experiment.

6.3 Experimental Results

SafeRefactor detected a behavioral change in one refactoring and two compilation errors in less than 4 minutes in the first category. Table 1 shows total time in seconds required by SafeRefactor to yield a result, the number of generated tests, and the number of tests detecting the behavioral change (Error column) for each subject in the Total Time, Tests, Error columns, respectively. The Result column indicates whether SafeRefactor identified a behavior change or a compilation error. The symbol `-` indicates that no behavior change is detected.

Developers refactored JHotDraw in order to avoid code duplication with identical exception handlers in different parts of a system [2]. Eight programmers working in pairs performed the change: they extracted the code inside the `try`, `catch`, and `finally` blocks to methods in specific classes that handle exceptions. They relied on refactoring tools, pair review, and unit tests to assure that the behavior was preserved. Some classes that implement `Serializable` were refactored. Developers changed the `clone` method and introduced the `handler` attribute to handle exceptions. However, they forgot to serialize this new attribute. Thus, when the method `clone` try to serialize the object, an exception is thrown. Therefore, the refactored method `clone` has a different behavior.

Moreover, Eclipse wrongly applied a refactoring to ANTLR and Xtc, introducing a compilation error that was not reported [7]. Finally, SafeRefactor did not detect behavior change in Subjects 2-4 and 7. Table 1 summarize the results.

7 Testing refactoring tools

In this section, we describe our experiment to analyze SafeRefactor and JDolly with respect to the efficiency in automated testing of refactorings implementations. The test consists of three steps. First, we generate a number of programs using JDolly. Next, these programs are refactored using the refactoring API under test. Finally, we use SafeRefactor to evaluate whether the refactorings were correct applied.

First, we describe the refactorings implementations under test (Section 7.1) and the experiment setup (Section 7.2). Next, we show the experiment results and a discussion (Section 7.3 and 7.4).

7.1 Subject characterization

We tested 12 refactoring implemented in Eclipse 3.4.2. Table 2 shows the refactorings evaluated and the number of test cases in the Eclipse test suite for each one. Eclipse team does not separate Pull up and Push down test cases between methods and fields. Similarly, the Change method signature test suit has no distinction between add parameter, remove parameter, and change access modifier test cases.

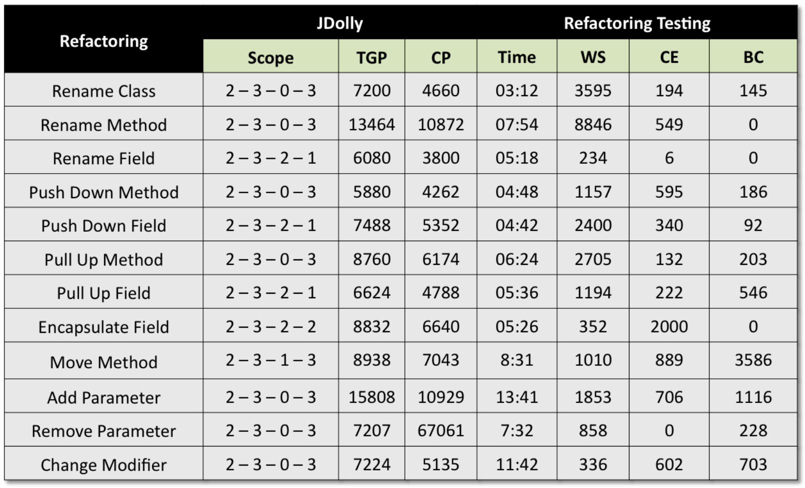
Evaluated Refactorings		Eclipse test cases
Rename	Class	229
	Method	355
	Field	73
Push Down	Method	96
	Field	
Pull Up	Method	134
	Field	
Change Method Signature	Change Access Modifier	133
	Add Parameter	
	Remove Parameter	
Move	Method	147
Encapsulate Field		36

Table 2. Summary of evaluated refactoring implementations

7.2 Experiment Configuration

For each refactoring implementation under test, we modeled an Alloy specification describing some characteristics generated programs must have and defined the scope of the generation. In summary, we specified 12 executions of our technique illustrated in Table 3. Each line represents one execution. Column Scope shows the number of packages, classes, fields, and methods of the programs.

We performed the experiment on a 2.2 GHz dual-core Dell Vostro laptop running Ubuntu 9.04. As we mentioned before, the Eclipse version under test was the 3.4.2.



Refactoring	JDolly			Refactoring Testing			
	Scope	TGP	CP	Time	WS	CE	BC
Rename Class	2 – 3 – 0 – 3	7200	4660	03:12	3595	194	145
Rename Method	2 – 3 – 0 – 3	13464	10872	07:54	8846	549	0
Rename Field	2 – 3 – 2 – 1	6080	3800	05:18	234	6	0
Push Down Method	2 – 3 – 0 – 3	5880	4262	04:48	1157	595	186
Push Down Field	2 – 3 – 2 – 1	7488	5352	04:42	2400	340	92
Pull Up Method	2 – 3 – 0 – 3	8760	6174	06:24	2705	132	203
Pull Up Field	2 – 3 – 2 – 1	6624	4788	05:36	1194	222	546
Encapsulate Field	2 – 3 – 2 – 2	8832	6640	05:26	352	2000	0
Move Method	2 – 3 – 1 – 3	8938	7043	8:31	1010	889	3586
Add Parameter	2 – 3 – 0 – 3	15808	10929	13:41	1853	706	1116
Remove Parameter	2 – 3 – 0 – 3	7207	67061	7:32	858	0	228
Change Modifier	2 – 3 – 0 – 3	7224	5135	11:42	336	602	703

Table 3. Summary of the experiment; Scope = packages, classes, fields, and methods; TGP = generated programs; CP = compilable programs; Time = total time in hh:mm; WS = warning status; CE = compilation errors; BC = behavioral changes

7.3 Results

Our approach detected faults in all refactorings implemented by Eclipse. Table 3 shows the experiment results. We can see JDolly results in Column JDolly, which is divided in three sub-columns. Our generator may create invalid programs. While Column TGP shows the number of generated programs, Column CP shows only the programs that compile. The total time of the test, number of transformations that leads to warning status, compilation errors, and behavioral changes appear in Time, WS, CE, BC Columns, respectively.

Bounded-exhaustve testing can generate different test cases that reveal a common fault. Jagannath et al. [11] proposes an approach called Oracle-based Test Clustering (OTC) for automatically separating the failing tests by faults.

Their main idea is to split the tests based on the template of the error message. We plan to use this approach to split the failing tests related to compilation errors. Regarding the non-behavior-preserving transformations, we plan to manually analyze them reporting identified bugs to Eclipse bug report. All non-behavior-preserving transformations shown in this paper are examples of bugs automatically detected by our approach.

7.4 Discussion

Our approach was useful for testing the refactorings implemented by Eclipse. It revealed a number of faults that have not been detected so far using manually written test cases or previous approach for automated testing.

While previous approach of automated refactoring tools testing [4] focused on detecting compilation error faults, we focused one behavioral changes. We believe that in most cases a behavioral change is more critical than a compilation error in the refactored program, since the first one is silently introduced by the tool, taking more time to be discovered. SafeRefactor [23] allowed us to detect non-behavior-preserving transformations. We configured it with a timeout of one second for test generation and it created around 100 unit tests for each transformation.

Firstly, we tried to use the ASTGen approach combined with SafeRefactor to improve the testing of refactoring tools. However, we realized that some structural parts of the program are difficult to specify in a imperative way. For example, ASTGen has a generator to produce inheritance between two classes. However, to program the generation of inheritance for three classes `A`, `B` and `C` in such a way that it produces all valid (and none invalid) inheritance relations is not simple in a imperative language. On the other hand, we needed just one constraint in our Java meta-model specified in Alloy (Listing 1.6) to the Alloy Analyzer find all valid inheritance relations. So, the main motivation to propose JDolly was the need for a program generator that allows to specify the structural characteristics of the programs in a simple manner and that generates programs with expressivity for testing refactoring tools. The programs generated by JDolly have had enough expressivity for detecting a number of corner cases in refactoring tools usage.And in our experiment, it has been easier to specify the structural parts of the program using the declarative approach of JDolly than using ASTGen generators. However, behavioral parts of the program as expressions and method sequences are difficult to specify in a declarative language, because of that we used the imperative approach of ASTGen in this step. Therefore, the combination of the imperative and declarative generation seems to achieve better results than generation approaches that use only one of them [4,12].

We have not found behavioral changes problems in rename method implementations. Ekman et al. [16] manually catalogued examples of non-behavior-preserving transformation applied by Eclipse rename refactoring. However, the current Java meta-model of JDolly does not have enough expressivity to generate programs useful for detected them. We aim at improving our meta-model with elements such as static members and inner classes to find more bugs. Besides that, we can see in Table 3 that 8846 out of 10872 transformations (82%) performed by rename refactoring produced warning status. We plan to investigate whether the preconditions implemented are forbidding behavior-preserving transformation. Our technique can be useful for specifying a smaller set of preconditions that guarantee the behavioral preservation.

8 Conclusions

In this paper, we propose a technique and tool (SafeRefactor) for improving confidence that a refactoring is sound. SafeRefactor analyzes a transformation and generates tests useful for detecting behavioral changes. Moreover, we develop a Java program generator (JDolly) useful for generating inputs for testing refactoring tools. We evaluate both in two experiments. In the first one, we analyzed SafeRefactor with respect to the efficiency in detecting behavioral changes in likely refactorings of Real Java programs. SafeRefactor detect one behavioral change and two compilation errors. In the next experiment, we analyze the efficiency of SafeRefactor and JDolly in automated refactoring tool testing. It has been useful for detecting a number of non-behavior-preserving transformation.

Most of IDEs do not implement all preconditions to guarantee the refactoring correctness. Therefore, they may perform transformations that introduce compilation errors or behavioral changes. Our work can make program refactoring safer. It can be used by developers to perform refactorings with more safety. Additionally, tool developers can use it to improve automated refactoring tool testing.

8.1 Related Work

Daniel et al. [4] proposed an approach for automated refactoring tools testing. They developed a framework for generating Java program called ASTGen, and used the programs generated by it as input for refactoring tools. They implemented some oracles to evaluate the refactoring results. For example, one of the oracles checks for compilation errors. Other one applies the inverse refactoring to the output and compares the result with the input; if they were different, the tester manually analyzes them. As result, they identified 21 bugs in Eclipse and 24 in NetBeans. Almost all of them were bugs that produce compilation errors. Our approach improves the approach proposed by them. We combine Alloy with ASTGen to create a program generator with more expressivity regarding the generation of the structural parts of the program, since the declarative approach makes easier to specify some of these structures as inheritance relations. Moreover, SafeRefactor allows identifying faults that leads to behavioral changes.

Ekman et al. proposed to evaluate the correctness of a refactoring based on specific invariants of the transformation. For example, the rename method refactoring must preserve the binding between names and entities. They proposed a technique for creating symbolic names that are guaranteed to bind to a desired entity in a particular contex by inverting lookup functions. To implement this, they used the JastAdd Extensible Java Compiler. They proposed solutions for the rename and extract method refactoring [16,18]. They used ASTGen, the Eclipse test suite, and their own test suite to evaluated the correctness of their implementations. We plan to use JDolly and SafeRefactor to evaluate them.

Fuhrer et al.[26] proposed inference rules that guarantee the behavior preservation with respect to types. They proposed and implemented in Eclipse the refactoring Infer Generic Type Arguments, which transforms a program to use the Generics feature of Java 5. In Section 6.1, we have shown two transformations applied by this Eclipse refactoring that lead to compilation errors on resulting program. In Java, changing modifier accessibility (`public`, `protected`, `package`, `private`) can change the static biding of classes, methods, and fields. Steimann e Thies [24] formal specified the Java accessibility and proposed constraints related to this that guarantee static binding preservation. These approaches can be benefited of ours. Since formal proves are difficult, our technique can be useful to improve the confidence that their implementations are sound.

Borba et al. [1] proposed a set of refactorings for a subset of Java with copy semantic (ROOL). They proved the refactorings correctness based on a formal semantic. This work can help to identify preconditions for Java refactorings. Silva et al. [19] proposed a set of behavior-preserving transformation laws for a sequential oriented-objected language with reference semantics (rCOS). They proved the correctness of each one of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. However, they did not consider some Java functionalities, such as overloading and field hiding. As we show in this work, our technique can detect behavioral changes introduced due to these functionalities. Duarte [5] extended ROOL refactorings for Java. They considered many Java features and parallelism. However, he did not prove these laws with respect to a formal semantics. Therefore, they may have problems. In fact, we manually analyzed some of them and verified that they allow non-behavior-preserving transformations. Our work is related with above approaches. We proposed a practical approach for detecting behavioral changes related to missing preconditions. We plan to automated testing some of these laws, helping to refine them and improving the confidence that they are sound.

Marinov e Khurshid [12] proposed TESTERA, a framework for automated specification-based testing of Java programs. It uses Alloy to specify the pre- and post-conditions of a method under test. Using this specification, it automatically generates the test inputs and checks the postcondition. However, they not focus on generating complex test inputs as Java programs. Gligoric et at. [8] propose a language UDITA that combines the imperative and declarative approaches for generating test inputs. They compared UTIDA with ASTGen on testing refactoring tools. Using UDITA, they could specify program generations that are difficult to specify in ASTGen, and realized that the combination of the imperative and declarative paradigms improves the program generation. They found 4 bugs (2 in Eclipse and 2 in Netbeans) related to compilation errors. Our work is related to this, since we also have the idea of combine the declarative and imperative generation. However, the main difference of our work is that we focused on improving faults detection related to behavioral changes too.

Murphy-Hill et al. [13] analyzed the usage of the refactoring tools of Eclipse by developers. The most used refactorings based on this work are: rename, extract local variable, move, extract method, and change method signature. We tested three of them (rename, move, change method visibility). To evaluate extract method and local variable refactorings we need methods with sequence of statements. The current version of JDolly only produces one statement at time, but this limitation can be resolved by exploring more combinations of ASTGen generators to produce method sequences.

8.2 Future Work

Proving refactoring correctness with respect to a formal semantic is difficult. Some approaches in the state of art of refactoring proposed solutions to specific kinds of refactorings [16,18] or to specific aspects of the language [24]. As future work, we aim at testing these approaches, helping to improve them.

We also plan to improve JDolly expressivity. We can add more Java constructions to our Java meta-model specification and explore more combinations of the ASTGenerators to create sequence of statements on methods bodies. In this way, we can test more refactorings such as extract method and extract inline variable. Finally, JDolly can be used to test other systems that receive programs as input, such as compilers and model checkers.

Acknowledgment

We gratefully thank the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) [4], funded by CNPq grants 573964/2008-4 and 477336/2009-4.

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. Cabral, B., Marques, P.: Exception handling: A field study in java and .net. In: 21st European Conference on Object Oriented Programming. pp. 151–175 (2007)
3. Community, A.: Accessing alloy4 using java api (2009), at http://alloy.mit.edu/alloy4/api.html
4. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Foundations of Software Engineering. pp. 185–194 (2007)
5. Duarte, R.M.: Parallelizing Java Programs Using Transformation Laws. Ph.D. thesis, Universidade Federal de Pernambuco (2008)

[4] www.ines.org.br

6. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
7. Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., Keller, M.: Efficiently refactoring java applications to use generic libraries. In: 19th European Conference on Object Oriented Programming. pp. 71–96 (2005)
8. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in udita. In: ICSE. ACM (2010)
9. Jackson, D., Schechter, I., Shlyahter, H.: Alcoa: the alloy constraint analyzer. In: 22nd ICSE. pp. 730–733. ACM Press (2000)
10. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT press (2006)
11. Jagannath, V., Lee, Y.Y., Daniel, B., Marinov, D.: Reducing the costs of bounded-exhaustive testing. In: FASE 2009 (March 2009)
12. Marinov, D., Khurshid, S.: Testera: A novel framework for automated testing of java programs. In: ASE '01. p. 22. IEEE Computer Society (2001)
13. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: International Conference on Software Engineering. pp. 287–296 (2009)
14. Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
15. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering. pp. 75–84 (2007)
16. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for java. In: OOPSLA. pp. 277–294 (2008)
17. Schäfer, M., Ekman, T., de Moor, O.: Challenge proposal: Verification of refactorings. In: Programming Languages meets Program Verification. pp. 67–72 (2009)
18. Schäfer, M., Verbaere, M., Ekman, T., Moor, O.: Stepping stones over the refactoring rubicon. In: ECOOP. pp. 369–393. Springer-Verlag (2009)
19. Silva, L., Sampaio, A., Liu, Z.: Laws of object-orientation with reference semantics. SEFM pp. 217–226 (2008)
20. Soares, G.: Making program refactoring safer. In: Student Research Competition at ICSE '10. ACM (2010)
21. Soares, G., Cavalcanti, D., Gheyi, R., Massoni, T., Serey, D., Cornélio, M.: Saferefactor - tool for checking refactoring safety. In: Tools Session at Brazilian Symposium on Software Engineering. pp. 49–54. Fortaleza, Brazil (2009)
22. Soares, G., Gheyi, R., Massoni, T., Cornelio, M., Cavalcanti, D.: Generating unit tests for checking refactoring safety. In: SBLP. pp. 159–172 (2009)
23. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Software 27, 52–57 (2010)
24. Steimann, F., Thies, A.: From public to private to absent: Refactoring java programs under constrained accessibility. In: ECOOP. pp. 419–443 (2009)
25. Taveira, J.C., Queiroz, C., Lima, R., Saraiva, J., Soares, S., Oliveira, H., Temudo, N., Araújo, A., Amorim, J., Castor, F., Barreiros, E.: Assessing intra-application exception handling reuse with aspects. In: SBES. pp. 22–31 (2009)
26. Tip, F., Kiezun, A., Baumer, D.: Refactoring for Generalization Using Type Constraints. In: OOPSLA. pp. 13–26 (2003)