Identifying overly strong conditions in refactoring implementations

Gustavo Soares, Melina Mongiovi, and Rohit Gheyi
Department of Computing Systems
Federal University of Campina Grande
melina@copin.ufcg.edu.br, {gsoares,rohit}@dsc.ufcg.edu.br

Abstract—**Each refactoring implementation must check a number of conditions to guarantee behavior preservation. However, specifying and checking them are difficult. Sometimes refactoring tool developers may define overly strong conditions that prevent useful behavior-preserving transformations to be performed. We propose an approach for identifying overly strong conditions in refactoring implementations. We automatically generate a number of programs as test inputs for refactoring implementations. Then, we apply the same refactoring to each test input using two different implementations, and compare both results. We use Safe Refactor to evaluate whether a transformation preserves behavior. We evaluated our approach in 10 kinds of refactorings for Java implemented by three tools: Eclipse and Netbeans, and the JastAdd Refactoring Tool (JRRT). In a sample of 42,774 transformations, we identified 17 and 7 kinds of overly strong conditions in Eclipse and JRRT, respectively.**

I. Introduction

Refactoring is the process of changing a software system to improve its internal quality yet preserving its observable behavior [1], [2]. Manually applying refactorings is an error-prone and time-consuming activity. Currently IDEs, such as Eclipse and NetBeans, automated a number of refactorings for Java. First, the user specifies the transformation parameters. Then the refactoring implementation checks a number of conditions needed for guaranteeing behavioral preservation. For instance, it is desired for the Rename Method refactoring implementation to avoid name conflicts. If all conditions are satisfied, the tool performs the transformation.

Usually refactoring tool developers implement each set of refactoring conditions based on their experience. However, they may forget to specify some conditions allowing non-behavior-preserving transformations to be performed. A number of bugs have been catalogued in the literature [3], [4], [5]. In order to test refactoring implementations, developers write unit tests to check whether an implementation prevents non-behavior-preserving transformations that they are aware of. For example, the JastAdd Refactoring Tool (JRRT) test suite is publicly available[1].

In order to avoid this scenario, another approach would be to formally specify each refactoring (conditions and transformation). Proving the refactoring correctness with respect to a formal semantics is useful for avoiding too weak conditions. However, some conditions may be overly strong, avoiding some useful behavior-preserving transformations to be applied compromising the tool's applicability (Section II). So, it is also important to prove that the set of established conditions is minimal to avoid overly strong conditions. However, both tasks are nontrivial, specially in complex languages, such as Java [6]. Moreover, even if we prove the correctness and minimality properties, we need to check that the implementation is in conformance with its specification. Some conditions may require nontrivial static analysis that is not simple to be implemented.

Some approaches have proved the correctness of some refactorings for a subset of Java [7], [8], [9]. However, none of them proved the minimality property of refactoring conditions. Schäfer and de Moor [10] propose a number of refactoring implementations for Java in JRRT. For each transformation, they specify a number of checks to guarantee behavior preservation. Their tool contains less bugs than the refactorings implemented by Eclipse. Additionally, they show some evidence that their implementation is less restrictive than Eclipse. However, there is no practical approach for identifying overly strong conditions in refactoring implementations.

In this paper, we propose a practical approach to help refactoring tool developers on checking whether refactoring conditions are overly strong. First, we use a program generator to automatically generate Java programs as test inputs for the refactoring implementations. For each generated program, we apply a refactoring using at least two refactoring implementations. Then, we compare their outputs. We use Safe Refactor [5] (Section III-B) to evaluate whether a transformation preserves behavior. This tool has been useful in identifying a number of non-behavior-preserving transformations [5]. If an implementation rejects a transformation, and the other one applies it and preserves behavior according to Safe Refactor, our approach establishes that the former implementation contains an overly strong condition. The refactoring tool developers can use this information to make their refactoring implementations more applicable.

We evaluated our approach on testing 10 kinds of refactorings implemented by three Java refactoring tools (Eclipse, Netbeans, and JRRT) in a sample of 42,757 transformations. We found that 16% and 7% of transformations rejected by Eclipse and JRRT, respectively, are behavior-preserving. The implementations have overly strong conditions avoiding correct transformations to be applied. Our approach automatically categorized them in 17 and 7 kinds of overly strong conditions

[1]http://code.google.com/p/jrrt/source/checkout

of Eclipse and JRRT, respectively.

In summary, the main contributions of this paper are the following:

- A practical approach to help refactoring tool developers on checking whether the implemented conditions are overly strong (Section IV);
- An evaluation of the approach on testing 10 kinds of refactorings implemented by Eclipse, Netbeans and JRRT (Section V).

II. Motivating Example

In this section, we show behavior-preserving transformations that are not possible to be applied due to overly strong conditions. First, consider part of a payroll system of a company, as declared next.

```
public class Employee { ... }
public class Salesman extends Employee {
  private double salary;
  public double getSalary() {
    return salary;
  }
  ...
}
```

The `Salesman` class is a subclass of the `Employee` class. A salesman declares the `salary` field and the `getSalary` method. Suppose that we would like to apply the Pull Up Field refactoring to move `salary` to `Employee`. If we use JRRT to apply this change, the tool will show the following warning message: *cannot access variable Salesman.salary*. Since this field is `private`, if we moved it to `Employee`, it would not be visible to `Salesman.getSalary()`. Therefore, JRRT does not apply this change.

However, Eclipse can apply this transformation. It increases the accessibility of `salary` from `private` to `protected`, as shown next. The transformation preserves behavior.

```
public class Employee {
  protected double salary; ...
}
public class Salesman extends Employee {
  public double getSalary() {
    return salary;
  }
  ...
}
```

After that, the developer can apply the Pull Up Method refactoring to move `getSalary` to `Employee`. Steimann and Thies [4] discuss some visibility adjustments to increase the applicability of a number of refactorings.

On the other hand, JRRT can correctly apply some transformations rejected by Eclipse. Consider the `A` class and its subclass `B` in Listing 1. `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 using Eclipse, it shows the warning message: *Method "A.k(long)" will be shadowed by the renamed declaration "B.k(int)"*.

Eclipse has a functionality that allows us to preview the transformation. In the previous example, Listing 2 presents the preview of the resulting program. Notice that after the transformation, the `test` method yields `20`, but in the original version it yields `10`. This transformation does not preserve behavior. This is the reason why Eclipse showed a warning message.

However, we can apply this transformation using JRRT. The resulting program is presented in Listing 3. Notice that this transformation is different from Eclipse. JRRT performs an additional change to make the transformation behavior-preserving. JRRT identifies that the call to `k` inside `test` must refer to `A.k` instead of `B.k` after the transformation. So, it adds a `super` access to the method invocation `k(2)` inside `test`. Therefore, the resulting program in Listing 3 correctly refactors the original program in Listing 1.

Although the examples presented in this section are simple, they can happen in practice in bigger examples. Refactoring tool developers have not only to specify a number of conditions, but also they must check whether some additional changes can be done to make the refactoring more applicable. Steimann and Stolz [11] present that refactoring to Role Object is rarely applicable in practice due to strong conditions. However, it is not easy to detect whether an implementation has overly strong conditions. We need a more practical approach to identify them to increase the applicability of refactoring tools.

III. Overview

In this section, we present an overview of JDolly [12], a Java program generator, and Safe Refactor [5].

A. JDolly

JDolly exhaustively generates a number of Java programs. We specify a small subset of the Java metamodel in Alloy, a formal specification language [13]. We use the Alloy Analyzer [13], a tool for performing analysis on Alloy models, for generating solutions for this metamodel. Each Alloy solution is converted into a Java program. Listing II presents an example of a program generated by JDolly.

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures and constraints. Each *signature* denotes a set of objects (similar to a UML class), which are associated to other objects by relations declared in the signatures. A signature paragraph introduces a type and a collection of *relations*, along with their types and other constraints on their included values.

We specified part of the Java language in Alloy. For example, Listing 4 shows part of the Alloy model used by JDolly to specify the concepts of Java classes, fields and methods.

Listing 4. A subset of the Java metamodel specified in Alloy.

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

Listing 1. Original version
```
public class A {
  public long k(long a) {
    return 10;
  }
}
public class B extends A {
  public long n(int a) {
    return 20;
  }
  public long test() {
    return k(2);
  }
}
```

Listing 2. Eclipse's target version after ignoring the warning message
```
public class A {
  public long k(long a) {
    return 10;
  }
}
public class B extends A {
  public long k(int a) {
    return 20;
  }
  public long test() {
    return k(2);
  }
}
```

Listing 3. JRRT target's version
```
public class A {
  public long k(long a) {
    return 10;
  }
}
public class B extends A {
  public long n(int a) {
    return 20;
  }
  public long test() {
    return super.k(2);
  }
}
```

Fig. 1. Renaming the n method to k using Eclipse and JRRT.

A Java class may declare a set of fields and methods, and may extend another class. The `set` qualifier in relations `fields` and `methods` imposes no constraint on multiplicity. The `lone` qualifier denotes a partial function. In Alloy, one signature can extend another, establishing that the extended signature (subsignature) is a subset of the parent signature. For example, `Class` is a subsignature of `Type`. Similarly we specified the other Java constructs.

Additionally, a Java program must satisfy some well-formedness constraints. We specified a number of them in Alloy. For instance, a class cannot extend itself. The fact `ClassCannotExtendItself` specifies this constraint (Listing 5). An Alloy fact packages formulas that always hold, such as invariants.

Listing 5. A well-formedness rule of Java specified in Alloy.
```
fact ClassCannotExtendItself {
  all c: Class | c ! in c.^extend
}
```

The `all` keyword represents the universal quantifier. The `in` keyword denotes the set membership operator. The operators `^` and `!` represent the transitive closure and negation operators, respectively. The dot operator (`.`) is a generalized definition of the relational join operator. For example, the expression `c.extend` yields the superclass of `c`.

JDolly performs analysis on this specification of the Java metamodel using Alloy Analyzer to find all solutions for a given scope. A scope defines the maximum number of objects allowed for each signature during analysis, assigning a bound to the number of objects of each type. For instance, Figure 2 shows part of one solution generated by the Alloy Analyzer for the previous model. It represents a program containing three classes and two of them do not declare fields. We used a scope of at most three objects for `Class`, `Method`, and `Field` signatures. JDolly converts each Alloy solution into a Java program.

B. Safe Refactor

Safe Refactor is a tool for detecting behavioral changes in transformations applied to sequential Java programs.

The Safe Refactor command line version receives as input two programs and analyzes whether they have the same behavior. The tool analyzes a transformation and generates a number of tests suited for detecting behavioral changes. The analysis consists of identifying the common methods, that is, methods with same signature before and after the transformation. Next, Safe Refactor generates a test suite for the methods previously identified. Since the tool focuses on identifying common methods, it executes the same test suite before and after the transformation. Safe Refactor uses Randoop [14], a Java unit test generator, to perform the test case generation. Randoop randomly generates tests for a set of classes and methods given a time limit or a maximum number of tests. Finally, the tool executes the tests before and after the transformation, and evaluates the results: if they are different, the tool yields a set of unit tests exposing the behavioral change. Otherwise, we improve confidence that the transformation is behavior-preserving. Figure 3 illustrates this process.

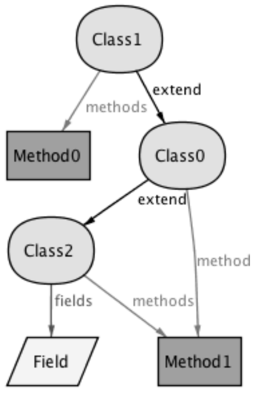


Fig. 2. Part of a solution found by the Alloy Analyzer.

Consider the transformation presented in Figure 1. Safe Refactor receives as input the programs shown in Listings 1 and 2. First, it identifies the methods with the same signature on both versions: `A.k(long)` and `B.test()`. Next, it generates 12 unit tests for these methods in 1 second (time limit). Finally, it runs the test suite on both versions and evaluates the results. A number of tests (11) passed in the source program

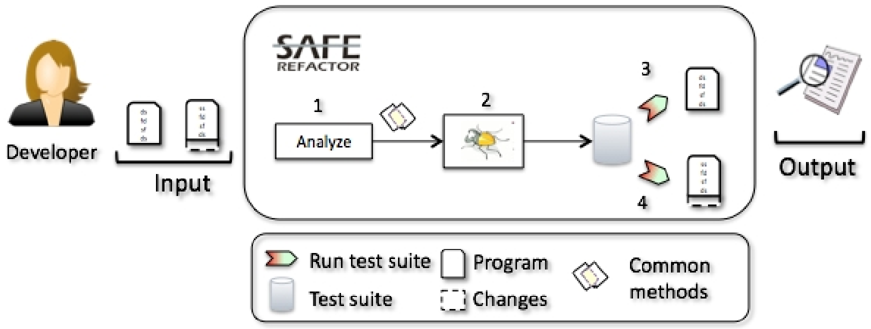


Fig. 3. Safe Refactor's technique.

but they did not pass in the refactored program. Listing 6 shows one of the generated tests that reveals the behavioral change. The test passes in the source program since the value returned by `B.test()` is `10`, but it fails in the target program since the value returned by `B.test()` in this version is `20`. Therefore, Safe Refactor reports a behavioral change.

Listing 6. A unit test exposing a behavioral change in the transformation performed by NetBeans from Listing 1 to Listing 2.

```
public void test() {
  B b = new B();
  long x = b.test();
  assertTrue(x == 10);
}
```

Previously, we have used Safe Refactor to evaluate transformations applied to programs with up to 100KLOC. It has been useful for identifying behavioral changes that were not detected by IDEs [5].

IV. Technique

Our goal is to identify behavior-preserving transformations that are not possible to be applied by refactoring tools due to overly strong conditions in their implementations. In this section, we describe a technique for detecting them.

Our technique consists of four major steps. First, we automatically generate programs as test inputs for the refactoring implementations (Step 1) using a program generator (JDolly) (Section IV-A). Next we apply the same refactoring to each test input using two different implementations (Step 2) (Section IV-B). We compare the outputs of the implementations. If one of them cannot apply the transformation, and the other one performs it maintaining the program's behavior according to Safe Refactor, we identify a transformation that has an overly strong condition (Section IV-C). The first implementation rejects a valid behavior-preserving transformation (Step 3). At the end of this step, we may have detected a number of rejected behavior-preserving transformations. In Step 4, we classify them (Section IV-D). The whole process is depicted by Figure 4.

A. Test input generation

The first step is to generate test inputs using our Java program generator (Section III-A). The tool developer can specify the maximum number (scope) of classes, fields, and methods that generated programs must have. Furthermore, JDolly can be parameterized with specific additional constraints for the generated programs. For example, considering the Pull Up Method refactoring, the generated programs must contain at least a subclass declaring a method that we would like to pull up to its superclass. We can specify this constraint and guide JDolly to generate useful test inputs. The `PullUpMethod` fact shows some constraints we used to test this refactoring.

```
fact PullUpMethod {
  some c:Class |
    someSuperClass[c] and (some m:Method | m in c·methods)
}
```

The `some` keyword represents the existential quantifier. The `someSuperClass` predicate states that the `c` class has a superclass. We apply the Pull Up Method refactoring to the value given to `m` by the Alloy Analyzer in each solution. We can also add other constraints to reduce the number of instances generated.

B. Performing the refactoring

The second step of our technique is to apply a refactoring using two different implementations to each program generated by JDolly. Each refactoring implementation checks a set of conditions, and, if all of them are satisfied, it applies the transformation. Otherwise, the refactoring is rejected, and a warning message is shown.

For instance, JDolly generated 6,830 programs to test the Rename Method refactoring. We save them in a folder. Listing 1 presents one of the programs generated by JDolly. Then, we applied this refactoring to each generated program using the Rename Method implementations of Eclipse, JRRT, and NetBeans. We used their refactoring API to apply the refactoring. This transformation applied to Listing 1 cannot be performed by Eclipse. It gives the following warning message: *Method A.k(long) will be shadowed by the renamed declaration B.k(int)*. However, JRRT can perform the refactoring, and yields the program in Listing 3. NetBeans can also perform the transformation. It yields a target program presented in Listing 2. However, the transformation performed by Netbeans does not preserve behavior.

At the end of this step, we have the results of each refactoring implementation for each generated program: the

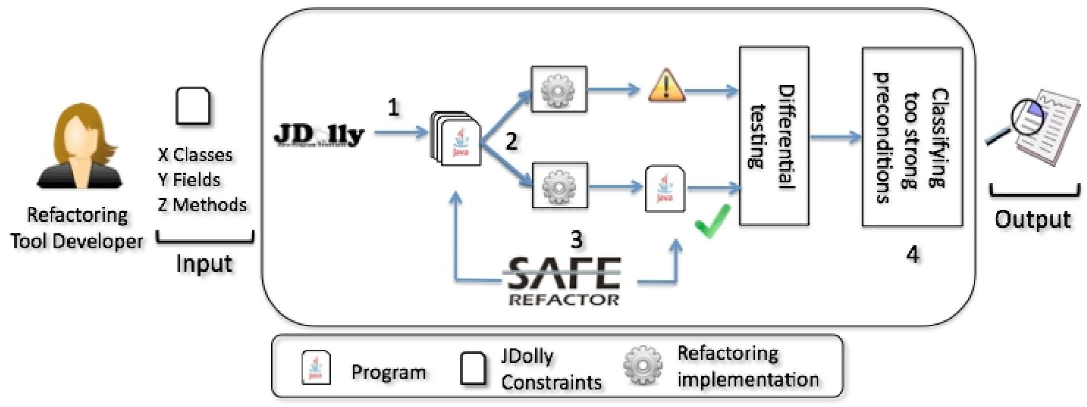


Fig. 4. A technique for testing refactoring implementations with respect to overly strong conditions.

target program when the transformation is applied or a warning message when it is rejected.

C. Test oracle

We propose an oracle to detect behavior-preserving transformations rejected by refactoring implementations. For the same input and refactoring, we compare the results of two refactoring implementations: if one implementation rejects the transformation, and the other one applies it, and Safe Refactor does not find behavioral changes, we establish that the former implementation rejected a behavior-preserving transformation due to an overly strong condition. This technique is also known as differential testing [15]. If both implementations cannot apply a transformation, we cannot conclude anything.

For example, consider the results of the Rename Method implementations for a given program illustrated in the previous section (Section IV-B). We compare the results of Eclipse, NetBeans, and JRRT. While the former rejected the transformation, NetBeans and JRRT applied it. Safe Refactor evaluates the transformations applied by JRRT and NetBeans. It 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. Moreover, it detects a bug (too weak condition) in the transformation applied by NetBeans. Listing 6 shows a test given by Safe Refactor exposing behavioral change.

D. Classifying overly strong conditions

The previous step detects a set of behavior-preserving transformations rejected by refactoring implementations. However, since JDolly generates a number of programs, we can have some rejected transformations referring to the same overly strong condition.

Manually analyzing each rejected behavior-preserving transformation to identify whether they show the same kind of overly strong condition is time consuming and error-prone. To automate this process, we use an approach proposed by Jagannath et al. [16] that focuses on splitting the failing tests based on the oracle messages. The goal is to group the failing tests related to the same fault together. We categorized the warning messages thrown by a refactoring implementation when a condition is not satisfied based on templates.

For example, when we apply the Rename Method refactoring of Eclipse to the program shown in Listing 1, the tool yields the following warning messages, respectively: *Method "A.k(long)" will be shadowed by the renamed declaration "B.k(int)"*. Our approach ignores the parts inside quotes, which contain names of packages, classes, methods, and fields. If there is another message that has the same template, the rejected transformations are automatically classified in the same category of overly strong condition.

At the end of this step, our approach reports the behavior-preserving transformations rejected by the implementation categorized by overly strong conditions. Since Safe Refactor does not prove that a transformation is behavior-preserving, it is important to manually check them to avoid false positives. After fixing the overly strong conditions, refactoring tool developers can run our approach again.

V. Evaluation

In this section, we evaluate our technique in 27 refactoring implementations. First, we describe these implementations (Section V-A) and the experimental setup (Section V-B). Finally, Section V-C presents the results of our evaluation.

A. Refactoring implementations

We tested 27 refactorings implementations for Java of three tools: Eclipse (10 refactorings), JRRT (10 refactorings), and NetBeans (7 refactorings). Table I summarizes all evaluated refactorings.

Eclipse is the most used Java IDE [17], and contains a number of automated refactorings (currently, more than 25). NetBeans is also a popular Java IDE. A number of related approaches [18], [4], [3] have studied the correctness of their transformations.

JRRT implements a number of refactorings [3], [19], [10]. They aim at outperforming the refactoring implementations of Eclipse in terms of overly strong and too weak conditions. Some refactorings may have invariants to be preserved. For

TABLE I
SUMMARY OF EVALUATED REFACTORING IMPLEMENTATIONS.

Refactoring	Eclipse	JRRT	Netbeans
Rename class	X	X	X
Rename method	X	X	X
Rename field	X	X	X
Push down method	X	X	X
Push down field	X	X	X
Pull up method	X	X	X
Pull up field	X	X	X
Encapsulate field	X	X	
Move method	X	X	
Add parameter	X	X	

instance, their Rename Method refactoring implementation is based on the name binding invariant: each name should refer to the same entity before and after the transformation. They proposed other invariants such as control flow and data flow preservation. To alleviate the problem of overly strong conditions, their implementations may also perform additional changes, such as the one presented in the transformation from Listing 1 to Listing 3.

In our experiment, we evaluate 10 kinds of refactorings (Table I). Murphy et al. [17] conducted a survey with Eclipse developers and found that the most used refactorings are: Rename, Move, Extract Method, Pull Up Method, and Add Parameter. We evaluated four of them and focused on structural refactorings, transformations that operate at or above the level of methods. We did not evaluate the Extract Method refactoring due to current limitation of JDolly to generate more elaborated method bodies. We tested only 7 refactorings in NetBeans. The Move Method refactoring is not supported. Moreover, the Encapsulate Field and Add Parameter refactorings were not evaluated due to lack of documentation support.

B. Experimental setup

We performed the experiment on a 2,7 GHz dual-core PC with 4 GB of RAM running Ubuntu 10.04. We evaluated Eclipse 3.4.2, NetBeans 6.9.1 and JRRT 1.0. For each generated input, we compare the outputs of these three tools.

We used the Safe Refactor command line version using the time limit of 1 second to generate tests, which is enough for testing the small programs generated by JDolly. We also used the JDolly command line version.

C. Experimental results

Our technique evaluated 27 refactoring implementations of Eclipse, NetBeans, and JRRT. Based on the scope and the additional constraints used for each refactoring, JDolly generated 42,774 programs[2]. Eclipse and JRRT did not apply a number of transformations, from which 32% and 16% were behavior-preserving, respectively. They reject them due to overly strong conditions. We automatically classified these transformations in categories. As a result, we identified 17 and 7 kinds of overly strong conditions in Eclipse and JRRT, respectively. We did not find overly strong conditions in the refactorings implemented by NetBeans.

Table 3 summarizes the experiment results. For each refactoring, we show the results of each implementation (Eclipse, NetBeans, and JRRT). The number of programs generated by JDolly is shown in Column *Program*. Column *Rejected Transformation* shows the number of transformations that were rejected by each implementation for not satisfying refactoring conditions. The number of behavior-preserving transformations that were rejected due to an overly strong condition of the implementation is shown in Column *Rejected B. Pres. Transformation*. Finally, Column *Overly Strong Condition* shows the number of overly strong conditions that were categorized by our technique.

Most transformations can be applied in NetBeans. It did not reject transformations except for the Rename Class refactoring. All transformations rejected by it were also rejected by Eclipse and JRRT. Therefore, we did not find problems related to overly strong conditions in NetBeans. However, it performed a number of non-behavior-preserving transformations that were rejected by Eclipse and JRRT. NetBeans contains a number of bugs (too weak conditions), as presented elsewhere [12]. The focus of this work is not on identifying too weak conditions but in detecting overly strong conditions. Since Netbeans specifies too weak conditions, it allows not only non-behavior-transformations, but also a number of behavior-preserving transformations that cannot be applied by other tools. Since the oracle of our technique is based on differential testing (Section IV-C), performing almost all transformations using NetBeans was useful for identifying whether transformations rejected by Eclipse and JRRT could, in fact, be applied.

Eclipse was the tool that rejected more transformations. It rejected 21,759 transformations, from which 32% are behavior-preserving. We found overly strong conditions in all Eclipse's implementation but the Push Down Field refactoring. For instance, its Rename Method refactoring implementation rejected 5,995 out of 6,830 transformations but 4,802 of them could be applied without changing programs' behavior.

Renaming a method in the presence of features such as overloading and overriding may lead to behavioral changes in some situations due to changes in name bindings [3]. Eclipse developers may have implemented overly strong conditions for simplicity in order to avoid non-behavior-preserving transformations. However, this overly strong condition also rejected a number of useful behavior-preserving transformations since overloading and overriding are commonly used by Java developers.

[2]All experiment data are available at: http://dsc.ufcg.edu.br/~spg/papers.html

TABLE II

SUMMARY OF THE EXPERIMENT; PROGRAM = NUMBER OF PROGRAMS GENERATED BY JDOLLY; REJECTED TRANSFORMATION = NUMBER OF TRANSFORMATIONS REJECTED BY THE IMPLEMENTATION; REJECTED B. PRES. TRANSFORMATION = NUMBER OF BEHAVIOR-PRESERVING TRANSFORMATIONS THAT WERE REJECTED; OVERLY STRONG CONDITION = NUMBER OF OVERLY STRONG CONDITIONS CLASSIFIED BY OUR TECHNIQUE.

Refactoring	Program	Rejected Transformation			Rejected B. Pres. Transformation			Overly Strong Condition		
		Eclipse	Netbeans	JRRT	Eclipse	Netbeans	JRRT	Eclipse	Netbeans	JRRT
Rename class	2037	1658	1212	1212	446	0	0	2	0	0
Rename method	6830	5995	0	1666	4802	0	419	4	0	1
Rename field	2647	324	0	0	200	0	0	2	0	0
Push down method	3822	2056	0	2065	59	0	40	1	0	1
Push down field	3043	1551	0	1551	0	0	0	0	0	0
Pull up method	5201	2907	0	3065	251	0	398	2	0	2
Pull up field	4151	976	0	912	744	0	584	1	0	1
Encapsulate field	3754	472	-	2736	176	-	1536	1	-	1
Move method	6316	5083	-	4757	367	-	135	2	-	1
Add parameter	4973	737	-	1189	79	-	0	2	-	0
Total	42774	21759	1212	19153	7124	0	3112	17	0	7

JRRT rejected 19,153 transformations. In 16% of them, the program's behavior could be preserved. We found overly strong conditions in 6 out of 10 refactorings evaluated: Rename Method, Push Down Method, Pull Up Method, Pull Up Field, Encapsulate Field, and Move Method.

Manually analyzing and classifying overly strong conditions in thousands of rejected transformations is time-consuming and error-prone. To avoid that, our technique automatically classifies them according to the template of the message shown by the implementation when a transformation is rejected. We analyze all warning messages in transformations that are behavior-preserving in at least another refactoring implementation. Our technique categorized 17 kinds of overly strong conditions in Eclipse, and 7 ones in JRRT. Table III shows the overly strong conditions identified in Eclipse and JRRT, respectively. Each line in the table contains a warning message template. The brackets abstract the names of packages, classes, methods, and fields, as described in Section IV-D.

We manually checked the overly strong conditions we found by randomly selecting a sample of 10 transformations for each kind of overly strong conditions, and we did not find false positives (a transformation that does not represent an overly strong condtion) or false negatives (the same template of warning message representing different overly strong conditions).

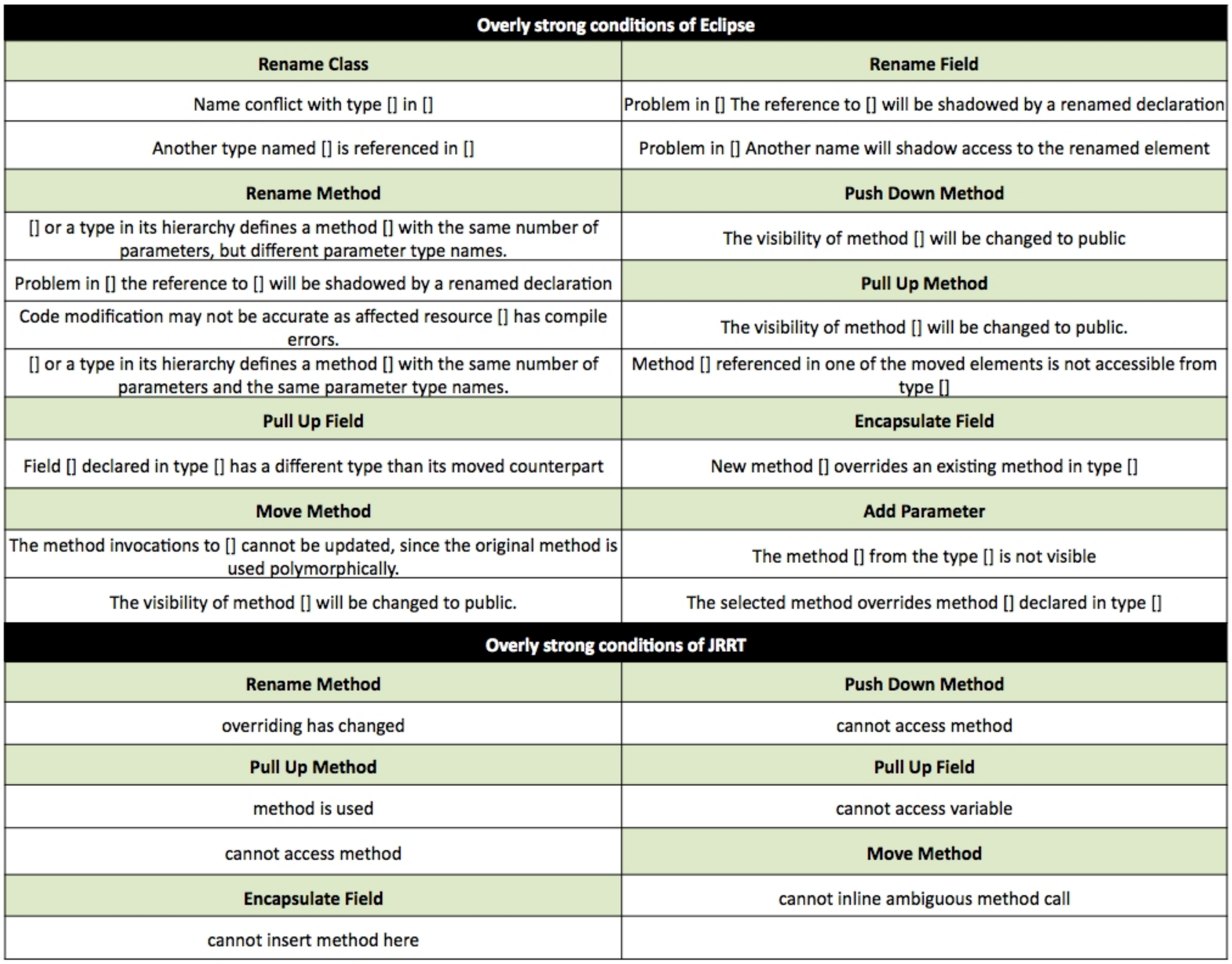
In five refactorings, we found less overly strong conditions in JRRT than Eclipse: Rename Class, Rename Method, Rename Field, Move Method, and Add parameter. Moreover, in four refactorings (Push Down Method, Pull Up Method, Pull Up Field, and Encapsulate Field), we found the same number of overly strong conditions in both tools. Finally, only in the Push Down Field refactoring, we did not find overly strong conditions.

Our technique identified 8 kinds of overly strong conditions in Rename Class, Method, and Field implementations of Eclipse, and only one in JRRT implementations. JRRT checks whether name bindings are preserved. Each name should refer to the same entity before and after the transformation [3]. Moreover, JRRT implementations may also check whether it is possible to re-qualify a name in order to preserve the name binding. This approach alleviates the problem of overly strong conditions. Figure 1 shows an example in which JRRT re-qualifies a name adding a `super` access to avoid name binding changes. Eclipse follows a different approach.

The overly strong condition found in the Rename Method refactoring of JRRT is related to overriding. This implementation has the invariant that overriding must be preserved. We also detected a condition in Eclipse related to that but NetBeans successfully applied a number of transformations that change overriding yet preserving program's behavior. In other refactorings, such as Move Method and Add Parameter, JRRT does not check conditions related to overriding.

Furthermore, we identified overly strong conditions related to accessibility. Both, Eclipse and JRRT, rejected transformations in the Push Down Method and Pull Up Method refactorings due to inaccessible methods. However, these transformations were performed by NetBeans. Changing access modifiers is not simple. It may change the name binding leading to behavioral changes [4]. Making these changes in ad-hoc way may be error-prone. Steimann and Thies [4] propose a number of conditions for applying refactorings with respect to Java accessibility. They show that these conditions are less strong than the ones implemented in Eclipse. While Eclipse

TABLE III
SUMMARY OF OVERLY STRONG CONDITIONS OF ECLIPSE AND JRRT.



Overly strong conditions of Eclipse	
Rename Class	**Rename Field**
Name conflict with type [] in []	Problem in [] The reference to [] will be shadowed by a renamed declaration
Another type named [] is referenced in []	Problem in [] Another name will shadow access to the renamed element
Rename Method	**Push Down Method**
[] or a type in its hierarchy defines a method [] with the same number of parameters, but different parameter type names.	The visibility of method [] will be changed to public
Problem in [] the reference to [] will be shadowed by a renamed declaration	**Pull Up Method**
Code modification may not be accurate as affected resource [] has compile errors.	The visibility of method [] will be changed to public.
[] or a type in its hierarchy defines a method [] with the same number of parameters and the same parameter type names.	Method [] referenced in one of the moved elements is not accessible from type []
Pull Up Field	**Encapsulate Field**
Field [] declared in type [] has a different type than its moved counterpart	New method [] overrides an existing method in type []
Move Method	**Add Parameter**
The method invocations to [] cannot be updated, since the original method is used polymorphically.	The method [] from the type [] is not visible
The visibility of method [] will be changed to public.	The selected method overrides method [] declared in type []
Overly strong conditions of JRRT	
Rename Method	**Push Down Method**
overriding has changed	cannot access method
Pull Up Method	**Pull Up Field**
method is used	cannot access variable
cannot access method	**Move Method**
Encapsulate Field	cannot inline ambiguous method call
cannot insert method here	

implements some heuristics for that, Schäfer and de Moor [10] intend to integrate these conditions to JRRT.

Eclipse and NetBeans contain test suites for evaluating their refactoring implementations. For instance, the Eclipse test suite contains more than 2,600 unit tests. JRRT has a different test suite. Schäfer and de Moor [10] also evaluated JRRT over more than 1,000 unit tests of Eclipse's test suite. They used them not only for evaluating correctness, but also for identifying overly strong conditions [10]. Schäfer and de Moor checked whether all rejected transformations of Eclipse could be applied by JRRT. They identified some overly strong conditions in Eclipse. However, they also identified overly strong conditions in JRRT in the Add Parameter, the Move Method, and the Push Down refactorings. The overly strong conditions were related to visibility adjustment. However, they do not propose an approach to evaluate whether refactoring implementations have overly strong conditions. We can do it by using JDolly and Safe Refactor.

In our evaluation, JDolly generated small programs (up to 15 LOC) with up to two packages, three classes, four methods, and two fields. These programs contain some common features of Java such as inheritance, overloading, and overriding. Although simple, they were useful for identifying 24 kinds of overly strong conditions in Eclipse and JRRT. The test suite of Eclipse and JRRT also contain small programs. However, the programs have some Java constructs such as interface, abstract classes and generics, that are currently not supported by JDolly. By improving the expressivity of JDolly, our technique can be useful for identifying other overly strong conditions.

VI. RELATED WORK

A. Refactoring implementation

The term refactoring was coined by Opdyke [20], [21] as a behavior-preserving program transformation that improves some quality (reusability, maintainability) of the resulting code. Opdyke [21] proposes a number of refactoring for C++ and specifies conditions to guarantee behavior preservation. Roberts [22] automated the basic refactorings proposed by Opdyke. However, there was no formal proof of the correctness and minimality of these conditions. Later on, Tokuda and Batory [23] found some overly weak conditions.

Schäfer et al. [3] propose a Rename Class, Method and Field refactoring implementations. They state that a renaming must preserve name bindings, that is, each name should refer to the same entity before and after the transformation. To alleviate the problem of overly strong conditions, their implementation re-qualifies a name to preserve the name binding where otherwise a conflict would exist. Our evaluation shows that the implementations are less restrictive than the Eclipse ones. We detected one overly strong condition related to overriding in the Rename Method implementation.

Furthermore, Schäfer et al. [19], [10] present a number of Java refactoring implementations. They translate a Java program to an enriched language that is easier to specify and check conditions, and apply the transformation. As correctness criteria, besides using name binding preservation, they used other invariants such as control flow and data flow preservation. We evaluated ten refactoring implementations. In our sample, we identified 7 kinds of overly strong conditions in JRRT.

Steimann and Thies [4] show that changing access modifiers in Java can introduce behavioral changes. They formalize a number of conditions related to accessibility to preserve behavior. Moreover, they alleviate overly strong conditions of a number of refactorings implemented by Eclipse by performing additional changes to access modifiers. They concluded that the proposed approach increases the applicability of the refactorings presented in Eclipse. As a future work, we intend to evaluate their implementation using our approach.

Steimann and Stolz [11] present a refactoring to the Role Objects pattern. They argue that the previous implementation was restricted, and it would be rarely applicable in practice due to its overly strong conditions. They propose a new implementation that increases the applicability of this refactoring. We intend to use our approach to show that their refactoring implementation is less restrictive than the initial one.

Borba et al. [8] propose a set of refactorings for a subset of Java with copy semantics. Silva et al. [9] present a set of behavior-preserving transformations for a subset of Java with reference semantics. They formally specify all conditions and prove that each transformation is sound with respect to a formal semantics. Tip et al. [7] formally specify sound refactorings with respect to type constraints. However, they do not prove that the conditions are minimal. In the absence of formal proofs, our technique can be useful for improving the confidence that the established set of conditions does not contain overly strong conditions.

Reichenbach et al. [24] propose the program metamorphosis approach for program refactoring. It breaks a coarse-grained transformation into smaller transformations. Although the smaller transformations may not preserve behavior individually, they guarantee that the coarse-grained transformation preserves behavior. We propose an approach to detect overly strong conditions in refactoring implementations.

B. Automated testing

Daniel et al. [18] present a technique for automated testing refactoring implementations to detect too weak conditions. They developed a Java program generator called ASTGen to generate programs as input to refactoring engines. To evaluate the refactoring correctness, they implemented oracles that evaluate the output of each transformation. For instance, one of them checks whether a transformation introduces a compilation error. They evaluated the technique by testing 21 refactoring implementations, and identified 21 bugs in Eclipse and 24 in NetBeans. Most of them are transformations that introduce compilation errors. Moreover, since their oracle for detecting behavioral changes is syntactic, it may yield some false positives and negatives. On the other hand, we use Safe Refactor as an oracle for checking whether a transformation preserves behavior.

Jagannath et al. [16] propose an approach called Oracle-based Test Clustering to reduce the manual inspection to identify all faults in bounded-exhaustive testing. The idea is to split the failing test cases based on the template of the error message. We used their approach for classifying overly strong conditions based on the warning messages.

McKeeman [15] presents differential testing. It is a technique where two or more comparable systems are evaluated against an exhaustive generated test cases. The different results between the systems are candidates for tests that expose faults. They used this technique for testing compilers. We propose a technique that uses differential testing to detect behavior-preserving transformations that were rejected by refactoring implementations due to conservative analysis.

C. Program generation

The program generator (ASTGen) proposed by Daniel et al. [18] exhaustively generates programs for a given scope. The user implements in Java how the elements of the programs will be combined together (*generating* approach). However, for some Java elements, implementing how they will be combined requires some effort. In this way, later, Gligoric et at. [25] propose a technique for generating programs that allows not only a *generating* approach but also a *filtering* approach. In the latter, the user specifies what should be generated instead of how as in the former. They present a non-deterministic Java based language called UDITA. In this way, the user can create generators that exhaustively generate solutions in a non-deterministic way. To filter the generation, the user specifies in Java the characteristics of a valid solution. By using the same oracles used by ASTGen, they found 4 new bugs related to compilation errors in 6 refactoring implementations of Eclipse and NetBeans.

Both program generators specify a subset of Java. While, in JDolly, the Alloy Analyzer yields all Java programs of a given scope, UDITA uses the Java Path Finder model checker as a basis for non-deterministic choices. In ASTGen, we have to deal with how the programs must be generated. JDolly uses a filtering approach similar to UDITA but the user specifies the constraints of the generation in Alloy instead of Java. Some kinds of constraints are simpler to be specified in Alloy instead of implemented in Java.

VII. Conclusions

In this work, we propose a practical approach for detecting overly strong conditions in refactoring implementations. A number of test inputs are generated by JDolly. Each program is applied by at least two refactoring implementations. Then, we compare the results. If there is a transformation that can be correctly applied by a tool, but it is rejected by the other one, we detect a strong condition. We evaluated Eclipse, JRRT and NetBeans with respect to overly strong conditions, and identified 24 kinds of overly strong conditions that reduce the applicability of Eclipse and JRRT refactoring implementations.

It took from 3 to 8 hours to test each kind of refactoring implemented by Eclipse, Netbeans and JRRT. We can optimize this process. For example, compiling thousands of programs takes time. We can optimize this step by performing incremental compilation since some programs generated are similar.

Acknowledgment

We gratefully thank Paulo Borba, Augusto Sampaio and the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 477336/2009-4, and 304470/2010-4.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[3] M. Schäfer, T. Ekman, and O. de Moor, "Sound and extensible renaming for java," in *Object-Oriented Programming, Systems, Languages and Applications*, 2008, pp. 277–294.

[4] F. Steimann and A. Thies, "From public to private to absent: Refactoring java programs under constrained accessibility," in *European Conference on Object-Oriented Programming*, 2009, pp. 419–443.

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

[6] M. Schäfer, T. Ekman, and O. de Moor, "Challenge proposal: Verification of refactorings," in *Programming Languages meets Program Verification*, 2009, pp. 67–72.

[7] F. Tip, A. Kiezun, and D. Baumer, "Refactoring for Generalization Using Type Constraints," in *Object-Oriented Programming, Systems, Languages and Applications*, 2003, pp. 13–26.

[8] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic Reasoning for Object-Oriented Programming," *Science of Computer Programming*, vol. 52, pp. 53–100, 2004.

[9] L. Silva, A. Sampaio, and Z. Liu, "Laws of object-orientation with reference semantics," in *Software Engineering and Formal Methods*, 2008, pp. 217–226.

[10] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *Object-Oriented Programming, Systems, Languages and Applications*, 2010, pp. 286–301.

[11] F. Steimann and F. U. Stolz, "Refactoring to role objects," in *International Conference on Software Engineering*, 2011, pp. 441–450.

[12] G. Soares, R. Gheyi, and T. Massoni, "A technique to test refactoring engines," Technical Report TR-UFCG-DSC-201103109. Federal University of Campina Grande, Brazil, 2011, at http://www.dsc.ufcg.edu.br/˜spg/papers.html.

[13] D. Jackson, *Software Abstractions: Logic, Language and Analysis*. MIT press, 2006.

[14] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[15] W. Mckeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[16] V. Jagannath, Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Fundamental Approaches to Software Engineering*, 2009, pp. 171–185.

[17] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[18] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Foundations on Software Engineering*, 2007, pp. 185–194.

[19] M. Schäfer, M. Verbaere, T. Ekman, and O. Moor, "Stepping stones over the refactoring rubicon," in *European Conference on Object-Oriented Programming*, 2009, pp. 369–393.

[20] W. Opdyke and R. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990, pp. 145–160.

[21] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[22] D. Roberts, "Practical Analysis for Refactoring," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1999.

[23] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Automated Software Engineering*, vol. 8, no. 1, pp. 89–120, 2001.

[24] C. Reichenbach, D. Coughlin, and A. Diwan, "Program metamorphosis," in *European Conference on Object-Oriented Programming*, 2009, pp. 394–418.

[25] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *International Conference on Software Engineering*, 2010, pp. 225–234.