Automated Behavioral Testing of Refactoring Engines

Gustavo Soares, *Member, IEEE*, Rohit Gheyi and Tiago Massoni

Abstract—Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. Usually, compilation errors and behavioral changes are avoided by preconditions determined for each refactoring transformation. However, to formally define these preconditions and transfer them to program checks is a rather complex task. In practice, refactoring engine developers commonly implement refactorings in an *ad hoc* manner, since no guidelines are available for evaluating the correctness of refactoring implementations. As a result, even mainstream refactoring engines contain critical bugs. We present a technique to test Java refactoring engines. It automates test input generation by using a Java program generator that exhaustively generates programs for a given scope of Java declarations. The refactoring under test is applied to each generated program. The technique uses SAFEREFACTOR, a tool for detecting behavioral changes, as oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them. We have evaluated this technique by testing 29 refactorings in Eclipse JDT, NetBeans and the JastAdd Refactoring Tools. We analyzed 153,444 transformations, and identified 57 bugs related to compilation errors, and 63 bugs related to behavioral changes.

Index Terms—Refactoring, automated testing, program generation

1 INTRODUCTION

REFACTORING is the process of changing a program to improve its internal structure without changing its external behavior [1], [2], [3]. Each refactoring may contain a number of *preconditions* to secure its behavioral preservation [1]. For instance, to pull up a method `m` to a superclass, the implementation must check whether `m` conflicts with the signature of other methods in that superclass. Widely used IDEs, such as Eclipse [4], NetBeans [5], IntelliJ [6], and JBuilder [7] contain a number of refactorings that automate precondition checking and program transformation.

Defining and implementing refactorings is a non-trivial task which the literature has treated in different ways [8], [9], [10], [11], [12], [13], [14], [15]. These include analyses of some of the various aspects of a language, such as: accessibility, types, name binding, data flow, and control flow. However, proving refactoring correctness for the entire language constitutes a challenge [16].

Consequently, refactoring engine developers use informal sets of preconditions as the basis for implementing refactorings; which may bring about differences between the implementations of the same refactoring [17]. More importantly, incorrect implementations have been reported in those engines [17], [12], [15]. Besides, while compilation errors introduced by the refactorings are easily detected in IDEs, behavioral changes may pass unnoticed (Section 2). Test suites are commonly seen as trustworthy resources for preventing such issue. However, these tests may also be affected by changes, which may render them inappropriate for testing refactoring outcomes [3].

In this work, we propose a technique to test Java refactoring engines. This technique is based on two main components: a program generator, JDOLLY (Section 4), and a tool for detecting behavioral changes, SAFEREFACTOR [18]. It uses JDOLLY to automate the test input generation. JDOLLY exhaustively generates programs for a given scope of Java declarations (packages, classes, fields, and methods). It contains a subset of the Java metamodel specified in Alloy, which is a formal specification language [19]. It also employs the Alloy Analyzer [20], a tool for the analysis of Alloy models, to generate solutions for this metamodel. The refactoring engine developer passes as input both the maximum number of the elements that generated programs may declare, and additional constraints for guiding the program generation. The refactoring under test is applied to each program generated by JDOLLY. The technique uses SAFEREFACTOR as oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them.

We have evaluated our technique[1] by testing 29 refactorings in Eclipse JDT 3.7, NetBeans 7.0.1, and two versions of the JastAdd Refactoring Tools

• *G. Soares, R. Gheyi, and T. Massoni are affiliated to the Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, 58429-900 Brazil.*
E-mail: {gsoares,rohit,massoni}@dsc.ufcg.edu.br.

1. All experimental data are available at: http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.html

(JRRT) [12], [13], [14]. JRRT was proposed to improve the correctness of refactorings by using formal techniques. We assessed 153,444 transformations, and identified 57 bugs related to compilation errors, and 63 bugs related to behavioral changes. These results confirm the bug-finding power obtained from the combination of JDOLLY and SAFEREFACTOR to detect bugs in refactorings. In short, the main contributions of this article include:

- A technique to test Java refactoring engines (Section 3);
- An evaluation of this technique on three refactoring engines, considering 29 refactorings as implemented in those tools (Section 5).

SAFEREFACTOR was described in a previous article [18] along with the evaluation of 24 specific transformations applied to small examples and real open source projects (such as JHotDraw and JUnit). SAFEREFACTOR detected a number of behavioral changes. This article describes a technique to test refactoring engines, a step of which involves SAFEREFACTOR as the oracle for detecting behavioral changes in programs generated by JDOLLY.

2 MOTIVATING EXAMPLE

In this section, we present two transformations – assumed to be refactorings – performed by Eclipse JDT 3.7 and JRRTv1[2]. They actually introduce a compilation error and a behavioral change, respectively.

Consider the class hierarchy presented in Listing 1. `A` and `B` declare fields `f` and `n`, respectively. `C` declares the `m` method, which accesses `f`. By using Eclipse JDT 3.7 to apply the Rename Field refactoring to `n`, by changing its name to `f`, the program presented in Listing 2 is generated. However, the resulting program will not compile. After the transformation, `B.f` hides `A.f`, and since the first has the private access modifier, it cannot be accessed from `C`. The following compilation error is introduced: *"The field B.f is not visible"*.

On the other hand, detecting behavioral changes is more difficult. Take class `A` and its subclass `B` as illustrated in Listing 3. `A` declares the `k` method, and `B` declares methods `k`, `m`, and `test`. The latter yields 1. Suppose we want to apply the Pull Up Method refactoring to move `m` from `B` to `A`. This method contains a reference to `A.k` using the `super` access. The use of either Eclipse JDT 3.7 or JRRTv1 to perform this refactoring will produce the program presented in Listing 4[3]. Method `m` is moved from `B` to `A`, and `super` is updated to `this`; a compilation error is avoided with this change. Nevertheless, a behavioral change was introduced: `test` yields 2 instead of 1. Since `m` is invoked on an instance of `B`, the call to `k` using `this` is dispatched on to the implementation of `k` in `B`.

Understanding inheritance, references to this/super, accessibility and other Java constructs in isolation may be simple, but nontrivial when considering them in conjunction [21]. So, it is difficult to find sufficient conditions for a refactoring to preserve behavior bearing in mind the complete Java language specifications. For example, a simple transformation changing the access modifier may have an impact on a number of Java constructs [15]. Because it is time consuming and difficult to prove all refactorings sound with respect to a formal semantics, a less costly method to evaluate the correctness of refactorings is needed. In this article, we present an approach to test refactoring engines.

3 TECHNIQUE

Our technique consists of four major steps. First, a program generator automatically yields programs as test inputs for a refactoring (Step 1); in this article, we employ JDOLLY in this step (Section 3.1). Second, the refactoring under test is automatically applied to each generated program (Step 2) (Section 3.2). The transformation is evaluated in terms of behavior preservation, using SAFEREFACTOR[4] (Section 3.3). In the end, we may have detected a number of transformations that modify behavior or introduce compilation errors. In Step 4, the detected problems are then categorized (Section 3.4). Figure 1 gives an overview of those steps.

3.1 Test Input Generation

Test input generation is performed by a Java program generator (Section 4). In this work, we present JDOLLY for this purpose, which generates Java programs from a Java metamodel specification. In JDOLLY, the tool developer can specify the maximum number (*scope*) of packages, classes, fields, and methods for the generated programs. Furthermore, JDOLLY can be parameterized with specific constraints. For example, when testing a refactoring that pulls up a method to a superclass, the input programs must contain at least a subclass declaring a method that is subject to be pulled up. We can specify these constraints in Alloy, the base specification language for JDOLLY, as detailed in Section 4.4.

3.2 Refactoring Application

The second step of our technique is to apply the refactoring under test to each generated program. This step can be performed manually (by using the IDE directly) or by the use of an API offered by the IDE infrastructure. Each refactoring checks a set of conditions, and, given the fulfillment of these conditions, the transformation is applied; otherwise,

2. The JRRT version from May 18th, 2010
3. The same problem happens when we omit the keyword **this**
4. It can be downloaded from: http://www.dsc.ufcg.edu.br/~spg/saferefactor

Listing 1: Before Refactoring

```
public class A {
  int f = 1;
}
public class B extends A {
  private int n = 2;
}
public class C extends B {
  public int m() {
    return super.f;
  }
}
```

Listing 2: After Refactoring. Applying Rename Field in Eclipse JDT 3.7 leads to a compilation error due to field hiding.

```
public class A {
  int f = 1;
}
public class B extends A {
  private int f = 2;
}
public class C extends B {
  public int m() {
    return super.f;
  }
}
```

Listing 3: Before Refactoring

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

Listing 4: After Refactoring.Applying Pull Up Method in Eclipse JDT 3.7 or JRRTv1 leads to a behavioral change due to incorrect change of **super** to **this**.

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

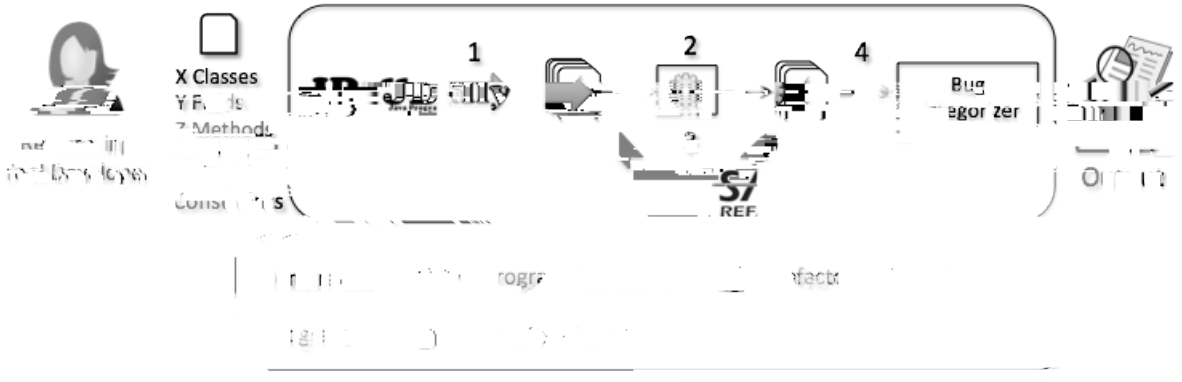


Fig. 1: A technique for testing refactoring engines.

the refactoring is rejected, and a warning message is shown.

3.3 SafeRefactor as a Test Oracle

In this step, our technique evaluates the correctness of each applied transformation. For this purpose, it uses SafeRefactor [18]. First, SafeRefactor checks for compilation errors in the resulting program, and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes.

SafeRefactor identifies the methods with matching signature (methods with exactly the same modifier, return type, qualified name, parameter types and exceptions thrown) before and after the transformation. Next, it applies *Randoop* [22], a Java unit test generator, to produce a test suite for those methods. Randoop randomly generates tests for a set of methods given a time limit. The default time limit is two seconds. Finally, SafeRefactor runs the tests before and after the transformation, and evaluates the results. If results are divergent, the tool reports a behavioral change, and displays the set of unsuccessful tests. Otherwise, developers have their confidence on behavior preservation improved.

Assuming the programs in Listings 3 and 4 as input, SafeRefactor first identifies the methods with matching signatures on both versions: `A.k`, `B.k`, and `B.test`. Next, it generates 78 unit tests for these methods within a time limit of two seconds. Finally, it runs the test suite on both versions and evaluates the results. A number of tests (64) passed in the source program, but did not pass in the refactored program; so SafeRefactor reports a behavioral change. Next, we show one of the generated tests that reveal behavioral changes. The test passes in the source program since the value returned by `B.test` is 1; however, it fails in the target program since the value returned by `B.test` is 2.

```
public void test() {
  B b = new B();
  int x = b.test();
  assertTrue(x == 1);
}
```

3.4 Bug Categorizer

The previous step may detect a number of transformations that change behavior or introduce compilation errors. Several of those failures may be caused by a single bug in the refactoring. To analyze manually all failed refactorings in order to identify whether these errors have been caused by a single bug is both time consuming and error-prone. Next, we describe a more efficient way of classifying the failing transformations.

3.4.1 Compilation Errors

We use an automatic approach proposed by Jagannath et al. [23] to classify compilation errors. It consists in splitting the failing tests based on messages from the test oracle. The goal is to group together the failing tests related to the same bug.

For example, when we apply Eclipse's Rename Method refactoring to the program shown in Listing 1, the resulting program yields the following compilation error: *The field "B.f" is not visible*. Our approach ignores (package, class, method or field) names within quotes. If the same refactoring is applied to two different programs, and they result in compilation error messages following the same template, a single bug is assigned to these two failures. We developed a tool to automate this grouping.

3.4.2 Behavioral Changes

Additionally, we propose an approach to classify behavioral changes by analyzing each detected change based on the characteristics of each pair source program-target program. Our approach is based on a set of *filters*; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program. All filters are presented in Table 1. We defined these filters by analyzing bugs found through the use of our approach, in addition to other bug reports from refactoring engines.

The filters may be applied in any order. The bug category of a behavior-changing transformation is then designated by the filters matched by its source and target programs. When a transformation does not fit any of these filters, conventional debugging is demanded from refactoring engine developers. For instance, the failure in the Pull Up Method on either Eclipse JDT 3.7 or JRRTv1 showed in Listing 4 matches the filter named "Changes **super**(**this**) to **this**(**super**)" from Table 1, in which a problem with replacing a reference to **`super`** with **`this`** is detected.

The set of filters is not complete. Currently, they focus on the Java constructs supported by JDolly. New filters can be proposed based on additional bugs found by refactoring engine developers. Currently, the classification of behavioral changing transformations is carried out manually. The process consists in analyzing each pair of programs, and testing every filter for matches.

4 JDolly

JDolly[5] is a Java program generator that exhaustively generates programs, up to a given scope. The

5. It can be downloaded from: http://www.dsc.ufcg.edu.br/~spg/jdolly

TABLE 1: Filters for classifying behavioral changes.

Filter	Description
Enables/disables overriding	After a refactoring, a method comes to be (or no longer is) overridden
Enables/disables overloading	After a refactoring, a method comes to be (or no longer is) overloaded
Enables/disables field hiding	After a refactoring, a field comes to be (or no longer is) hidden by another field declaration
Shadows class declaration	After a refactoring, a class declaration comes to be shadowed by another declaration
Changes **super**(**this or implicit this**) to **this or implicit this** (**super**)	If a method call or field access has **this or implicit this** (**super**) as target, and after a refactoring this reference is replaced by **super** (**this or implicit this**), in order to keep the link to the same previous object
Maintains **super** while changing hierarchy	A reference to **super** is moved up or down the hierarchy during refactoring
Changes accessibility	The refactoring changes the access modifier of a given field or method
The refactored program crashes	The original program is normally executed by the test suite but the refactored one throws some exceptions
Enables/disables implicit cast	After a refactoring, an implicit cast between primitive types is (or no longer is) applied where it did not take (or took) place originally

Alloy specification language [19] is employed as the formal infrastructure for generating programs; a metamodel for Java is encoded in Alloy, and the Alloy Analyzer finds solutions, which are translated into programs by JDOLLY, for user-specified constraints. For instance, Listing 3 shows an example of a program generated by JDOLLY.

Next we provide an overview of Alloy (Section 4.1). In Section 4.2, the encoding of a subset of the Java meta- model in Alloy is presented. We then describe how to translate each Alloy solution to Java (Section 4.3), and explain how to use JDOLLY for generating more specific Java programs in Section 4.4.

4.1 Alloy Overview

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures and constraints. Each *signature* denotes a set of objects associated to other objects by relations declared in the signatures. Each signature paragraph represents a type, and may declare a set of *relations* along with their types and other constraints on their included values.

We use as example part of the Java metamodel encoded in Alloy. A Java class is a type, and may extend another class. Additionally, it may declare fields and methods, as specified in the UML class diagram, as shown in Figure 2(a). Figure 2(b) presents its specification in Alloy. All classes and associations in the UML class diagram are analogous to the Alloy signatures and their relations, respectively. In `Class`, the **`set`** in relation `fields` and relation `methods` imposes no constraint on multiplicity. There are other multiplicity qualifiers, such as **`lone`**, denoting partial functions. If we omit the qualifier, the relation becomes a total function. In Alloy, one signature can extend another, establishing that the extended signature (subsignature) is a subset of the parent signature. For example, a `Class` is a subsignature of `Type`.

A number of well-formedness constraints can be specified for Java. For instance, a class cannot extend itself. In Alloy, we can declare *facts* which package formulas that always hold. The `ClassCannotExtendItself` fact specifies this constraint.

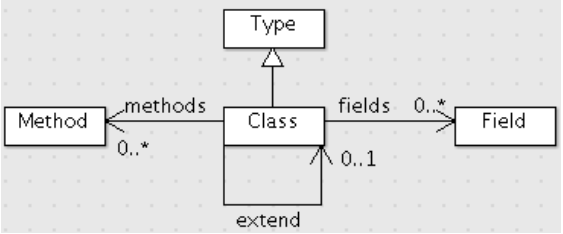
```
fact ClassCannotExtendItself {
  all c: Class | c ! in c.^extend
}
```

The **`all`** keyword represents the universal quantifier, and the **`in`** keyword denotes the set membership operator in the previous fragment. 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`.

In Alloy, predicates are used to package reusable formulas and specify operations. The following Alloy fragment declares the predicate `someClassHasNoField`, stating that there is a class without fields. The **`some`** keyword represents the existential quantifier. The **`no`** keyword, when applied to an expression, denotes that the expression is empty.

```
pred someClassHasNoField [] {
  some c: Class | no c.field
}
```

The Alloy Analyzer tool [20] allows us to perform analysis on an Alloy specification; for example, in order to find a solution for a model in a pre-defined 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. The simulations performed by the Alloy Analyzer tool are sound and complete, up to a given scope.



(a)

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

(b)

Fig. 2: A UML class diagram and its representation in Alloy.

Alloy commands are used for analysis purposes. Next, we declare a **run** command that is applied to a predicate, specifying a scope for all declared signatures. For desired solutions containing as many as three of each type, class, field and method, and at least one of the classes with no fields, the Alloy Analyzer searches for all combinations that satisfy the signature and fact constraints, in addition to the `someClassHasNoField` predicate.

```
run someClassHasNoField for 3
```

4.2 Java Metamodel

We specified a subset of Java's abstract syntax and well-formedness rules in Alloy.

4.2.1 Abstract Syntax

From Java, we have only considered the *int* and *long* primitive types. A UML class diagram representing the subset of the Java metamodel encoded in Alloy is shown in Figure 3. A class is the only non-primitive type – currently, we do not consider interfaces. A Java class has an identifier, field and method declarations, and extends another class. Moreover, each class is located in a package. If a class is not explicitly related to a package, the default package is assumed.

Each field is associated with one identifier, one type, and at most one modifier, such as `public`, `protected`, or `private`. When it does not have a modifier, its accessibility is `package`. Similarly, a method declaration contains a return type, an identifier, a number of parameters, and a body. Moreover, it may contain an access modifier. In order to avoid state space explosion, we have considered methods with at most one parameter.

In Java, a method body contains a sequence of statements, whose last statement must be a *return* for every non-void method. Currently, a method body contains just a single return statement. So, the simplest return statement returns a literal value based on the return type. Return statements can also contain field accesses or method invocations. Field accesses include: `f`, `A.f`, `this.f`, `super.f` and `new A().f` – the latter is a `ConstructorFieldAccess`). `LiteralValue` represents the simplest kind of statement, extending the signature `Body`. `FieldAccess` and `MethodInvocation` contain the identifier of the accessed field and method with a single qualifier at most, respectively. For simplicity, all methods contain at most one parameter. If a method with a single parameter is called JDOLLY always passes value 2 as argument to the call.

4.2.2 Well-Formedness Rules

Well-formedness rules are specified within Alloy facts. For example a Java class cannot have two fields with the same identifier, as declared in the fact `noClassTwoFieldsSameId`.

```
fact noClassTwoFieldsSameId {
  all c: Class | all f1,f2: c.fields |
    f1 ! = f2 ⇒ f1.id ! = f2.id
}
```

Similarly, a Java class cannot contain two methods with the same signature, as presented in the fact `noClassTwoMethodsSameSignature`.

```
fact noClassTwoMethodsSameSignature {
  all c: Class | all m1,m2: c.methods |
    m1 ! = m2 ⇒
      (m1.id ! = m2.id or m1.param ! = m2.param)
}
```

We specified other elements of Java's abstract syntax and other well-formedness rules, including field access and method invocation rules.

4.3 Program Generation

The previous Alloy model is then used to generate Java programs using Alloy's **run** commands; specifically with the `generate` predicate. By default, the scope of at most three objects is used for each signature.

```
pred generate[] {}
run generate for 3
```

The Alloy Analyzer searches for solutions such as the instance depicted in Figure 4(a). The graph contains the `Class` object, which is associated

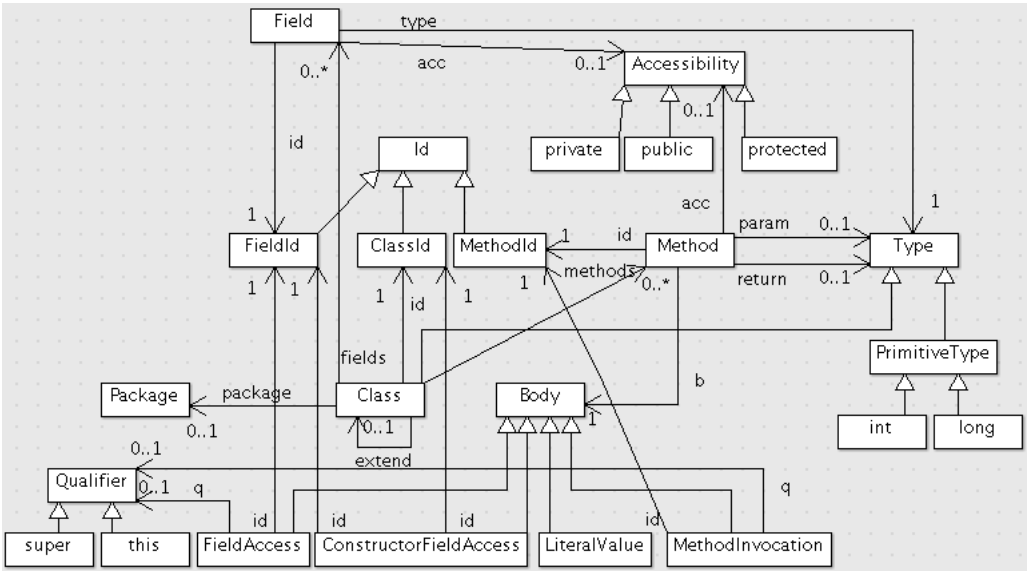


Fig. 3: The Java metamodel specified in JDOLLY.

with objects `Package`, `ClassId`, `Method`, and `Field`. Moreover, object `Field` is associated with `FieldId` and `Int_`, and `Method` is associated with `LiteralValue`, `MethodId`, `Protected`, and `Int_`. For simplicity, we distinguish class from field identifiers. For example, Figure 4(b) shows the counterpart in Java of the Alloy solution.

The Alloy Analyzer does not automatically convert an Alloy instance into a Java program. In fact, we use its API to generate *every* possible solution[6]. To complete the generation step, we reused the syntax tree available in Eclipse JDT [24] for generating programs from those solutions. For example, the Alloy objects `Class` and `Package` are mapped to a `TypeDeclaration` and a `PackageDeclaration`, respectively. The imports are automatically calculated from each Alloy instance generated; they are included in each program.

4.4 Generating More Specific Programs

With JDOLLY, we can specify different scopes to limit program generation. For instance, if we are not interested in fields, we can specify the scope of zero. Besides, the generation can be further constrained. In a context in which programs are needed with at least one class (`C2`) extending another one (`C1`), and `C2` declares at least a method (`M1`), the following Alloy fragment specifies `generate`. This particular specification is useful for testing the Pull Up Method refactoring, considering `M1`. For each instance, we pass the value given to `M1` to the refactoring.

```
one sig C1, C2 extends Class {}
one sig M1 extends Method {}
pred generate[] {
  C1 in C2·extend
  M1 in C2·methods
}
```

5 EVALUATION

In this section, we evaluate our technique on three refactoring engines, considering 29 refactorings as implemented in those tools. We intend to evaluate our technique with respect to:

- fault finding capabilities;
- performance;
- configuration effort.

In addition, we established a comparison between JDOLLY and the UDITA/ASTGen approach.

First, we describe the subjects evaluated (Section 5.1) and the experimental setup (Section 5.2). Section 5.3 contains the results of our evaluation, and Section 5.4 discusses topics related to compilation errors, behavioral changes, and JDOLLY. Additionally, we compare JDOLLY to ASTGen and UDITA. Finally, in Section 5.5 we discuss some of the threats to validity.

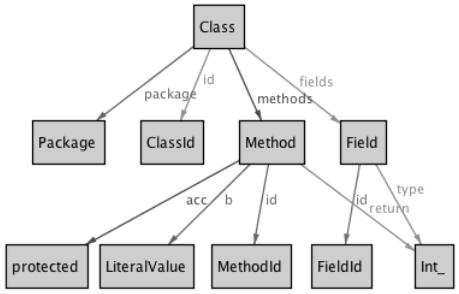
5.1 Subject Characterization

We evaluated Java refactorings implemented by Eclipse JDT 3.7 (10 refactorings), JRRTv1 and JRRTv2[7] (10 refactorings), and NetBeans 7.0.1 (9 refactorings). Table 2 summarizes all evaluated refactorings.

Eclipse JDT 3.7 contains more than 25 refactorings. The evaluated refactorings focus on a representative set of program structures. Moreover, a survey carried

6. Accessing Alloy 4 using Java API: http://alloy.mit.edu/-alloy4/api.html

7. The JRRT version from July 9th, 2011



(a)

```
package Package;
public ClassId {
  int fieldId = 1;
  protected int methodId() {
    return 2;
  }
}
```

(b)

Fig. 4: Translation of an Alloy solution to a Java program.

TABLE 2: Summary of evaluated refactorings; Scope = Package (P) - Class (C) - Field (F) - Method (M).

Refactoring	Scope P–C–F–M	Eclipse	JRRT	NetBeans
Rename class	2–3–0–3	X	X	X
Rename method	2–3–0–3	X	X	X
Rename field	2–3–2–1	X	X	X
Push down method	2–3–0–4	X	X	X
Push down field	2–3–2–1	X	X	X
Pull up method	2–3–0–4	X	X	X
Pull up field	2–3–2–1	X	X	X
Encapsulate field	2–3–1–3	X	X	X
Move method	2–3–1–3	X	X	-
Add parameter	2–3–0–3	X	X	X

out by Murphy et al. [25] shows the Eclipse JDT refactorings that Java developers use most: Rename, Move Method, Extract Method, Pull Up Method, and Add Parameter. Four of these are evaluated in this article. The Move Method refactoring was not supported by NetBeans by the time that this article was written.

We evaluated two versions of JRRT [12], [13], [14]. First, we evaluated with our technique the refactorings implemented by JRRTv1. Later, a new version with improvements and bug fixes was released (which we call JRRTv2); this new version was also subject to our analysis. The same refactorings from Eclipse JDT were tested in both versions of JRRT.

5.2 Experimental Setup

We performed the evaluation on a 2.5 GHz dual-core PC with 1 GB of RAM. We used the SAFEREFACTOR command-line version with a time limit of one second, which is enough for testing the small generated programs. Cobertura[8] was used to collect the statement coverage of the test suite as generated by SAFEREFACTOR in the resulting program.

The scope column in Table 2 indicates the maximum number of packages, classes, fields, and methods passed as parameter to JDOLLY. For each refactoring, we specified *main constraints* for guiding JDOLLY to generate programs with certain characteristics needed to apply the refactoring. Table 3 shows these constraints; they prevent the generation of programs to which the refactoring under test is not applicable.

Exhaustively generating programs often causes state space explosion. In order to minimize the number of generated programs to a small, focused set, we have also defined *additional constraints*. These constraints were built on data about refactoring bugs gathered in the literature, enforcing properties such as overriding, overloading, inheritance, field hiding, and accessibility. For each refactoring (column Additional Constraints in Table 3), we declare Alloy facts with additional constraints. These are fully described in Table 4. If a developer has the available resources to analyze the entire scope, then it will not be required to specify additional constraints.

Each refactoring may possibly include parameters. For instance, a method can be renamed, or a field may be encapsulated. In those cases, we declare a singleton subsignature for each parameter, similar to what we have done with `C1, C2` in Section 4.4, and use it in both the main and the additional constraints.

5.3 Experimental Results

For each refactoring, we used the same set of programs to evaluate Eclipse JDT, JRRTv1, JRRTv2, and NetBeans. JDOLLY generated 153,444 programs to evaluate all refactorings. Even though Eclipse JDT, JRRT and NetBeans have their own test suites, our technique identified 120 (likely) *unique* bugs. Table 5 summarizes the bugs reported to Eclipse JDT, NetBeans and JRRT. Our approach detects bugs related to transformation failures or weak preconditions. Although it is also relevant to find overly strong preconditions in refactoring engines – which may produce potentially correct but undesirably restrictive refactorings – these bugs are not the focus of our technique (nevertheless, JDolly and SAFEREFACTOR have already been evaluated as a detection strategy

8. http://cobertura.sourceforge.net

TABLE 3: Summary of the main constraints.

Refactoring Implementation	Main Constraint	Additional Constraints
Rename Class	**some** Class	{1, 2, 3}
Rename Method	**some** Method	{2, 3, 8}
Rename Field	**some** Field	{2, 3, 9}
Push Down Method	**some** c:Class | someSubclass[c] **and** someMethod[c]	{1, 6}
Push Down Field	**some** c:Class | someSubclass[c] **and** someField[c]	{2, 4}
Pull Up Method	**some** c:Class | someParent[c] **and** someMethod[c]	{1, 6}
Pull Up Field	**some** c:Class | someParent[c] **and** someField[c]	{2, 4}
Encapsulate Field	**some** Field	{5, 6, 7}
Move Method	**some** c:Class | someTargetClassField[c] **and** someMethodToMove[c]	{1, 2}
Add Parameter	**some** Method	{1, 2, 3}

TABLE 4: Summary of the additional constraints.

Id	Additional Constraint	Description
1	someOverriding[] **or** someOverloading[Int,Int]	overriding or overloading (number of parameters passed as argument)
2	someCaller[]	At least one method body calling a method or accessing a field
3	someInheritance[]	At least one case of inheritance
4	someFieldhiding[]	At least one case of field hiding
5	someGetter[]	At least one getter method
6	someTester[MethodId]	At least one method body with a simple call to a specific method
7	somePublicField[]	At least one public field
8	someMethodsWithSameNumParameter[]	At least two methods with the same number of parameters
9	somePrimitiveFields[]	At least two primitive fields

for overly strong preconditions [26], as detailed in Section 6.2).

From our catalog, most bugs were accepted (86). Some bugs have not been dealt with by Eclipse JDT and NetBeans developers prior to this submission (23). All bugs reported to JRRTv1 (20) were fixed in JRRTv2. We have also evaluated their new version (JRRTv2) after fixing the bugs from JRRTv1, and reported 11 bugs. They did not consider 4 bugs due to the closed world assumption (CWA) adopted by them, as we discuss in Section 5.5.1. More importantly, they incorporated our test cases into their test suite[9]. Eclipse JDT and NetBeans teams have fixed 1 and 6 bugs[10], respectively, which should be included in the next version of the IDEs. Developers have already confirmed 33 and 27 bugs in Eclipse JDT and NetBeans, respectively. However, 16 bugs were considered duplicated in Eclipse JDT.

It took from 1h36m to 50h24m to evaluate each refactoring. This includes the time required to generate and compile the input programs, apply the transformations, compile the resulting programs, run SAFEREFACTOR, and collect the statement coverage. The required amount of time depends not only on the number of programs to be refactored, but also on the number of transformations to be carried out. For example, it took 6h54m to test the Rename Method refactoring on Eclipse JDT, whereas it took 13h36m to test the same refactoring in JRRTv2, with the same inputs. Time also depends on the static analysis performed by each refactoring to check preconditions. Table 6 summarizes the experimental results.

TABLE 5: Summary of bugs reported.

	Submitted	Accepted	Duplicated	Not a Bug	Not Answered	Fixed
Eclipse	34	33	16	0	1	1
JRRTv1	24	20	0	4	0	20
JRRTv2	11	6	0	5	0	6
NetBeans	51	27	0	2	22	6
Total	**120**	**86**	**16**	**11**	**23**	**33**

The results include the number of programs generated by JDOLLY, the percentage of compilable programs, the time for testing, and the number of detected failures (encompassing compilation errors and behavioral changes). It also shows the number of bugs identified by our approach in each refactoring. Table 6 indicates, for each refactoring, the mean value of the statement coverage from the refactored program.

5.3.1 Compilation Errors

Our technique detected 16 bugs in Eclipse JDT, 11 bugs in JRRTv1, 1 bug in JRRTv2, and 29 bugs in NetBeans; all related to compilation errors. Our bug categorizer (Section 3.4.1) takes a few seconds to

9. http://code.google.com/p/jrrt/source/checkout
10. The id of all bugs are available at: http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.html

TABLE 6: Overall experimental results; GP = number of generated programs; CP = number of compilable programs (%); Time = total time to test the refactoring in hours; Fail. = number of detected failures; Bug = number of identified bugs.

Refactoring	GP	CP (%)	Total Time (h)				Compilation Error								Behavioral Change								Statement Coverage (%)			
							Eclipse		JRRTv1		JRRTv2		NetBeans		Eclipse		JRRTv1		JRRTv2		NetBeans					
			Ecl.	JRRTv1	JRRTv2	NetB	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Ecl.	JRRTv1	JRRTv2	NetB
Rename class	15,322	74.3	6.7	23	18.3	22.84	1,016	3	0	0	0	0	3,352	4	145	1	0	0	0	0	15	1	54	63	67	56
Rename method	11,263	79.5	6.9	8.7	13.6	23.2	559	1	0	0	0	0	1,731	2	0	0	0	0	482	2	1,231	2	83	84	90	86
Rename field	19,424	79.2	29.3	22.4	30.4	50.41	48	1	520	2	0	0	326	3	0	0	167	1	0	0	1,667	1	100	100	100	100
Push down method	20,544	78.5	11.9	11.6	16.9	31.9	1,267	2	1,989	2	0	0	10,323	4	853	5	258	4	716	3	1,485	6	90	90	93	90
Push down field	11,936	79.1	6	3.7	4.6	13.7	342	1	0	0	0	0	6889	4	92	1	0	0	0	0	270	2	100	100	100	100
Pull up method	8,937	72	7.3	6.3	6.9	13.5	269	2	549	2	0	0	3,049	3	202	3	78	2	10	1	1,073	5	90	90	92	89
Pull up field	10,927	79.7	8.6	5.3	7.7	12.1	518	1	80	2	0	0	1,128	4	546	4	0	0	0	0	239	2	100	99	100	100
Encapsulate field	2,000	92.8	2.5	1.6	2.3	5.7	238	1	0	0	0	0	234	1	0	0	344	1	437	1	439	1	66	81	86	76
Move method	22,905	69	10.3	4.5	5.9	-	214	2	1,098	3	9	1	-	-	3,586	3	1,759	3	6,944	3	-	-	82	82	86	-
Add parameter	30,186	63	34.69	24.61	25.05	50.36	1,663	2	0	0	0	0	5,824	4	2,238	2	378	2	0	0	2,186	2	87	87	90	87
Total	**153,444**	**68.8**	**124.2**	**112.15**	**131.65**	**223.7**	**6,134**	**16**	**4,236**	**11**	**9**	**1**	**32,856**	**29**	**7,662**	**18**	**2,984**	**13**	**8,152**	**10**	**8,605**	**22**				

automatically classify all failures of a refactoring. For instance, our technique detected 1,267 compilation failures in the Push Down Method refactoring implementation of Eclipse JDT. The described approach classified them into two groups: some transformations produced the message *"The method [M] from the type [T] is not visible"*, while others produced the message *"No enclosing instance of the type [T] is accessible in scope"*. Consequently, two bugs were catalogued.

Even though all evaluated refactorings implemented by Eclipse JDT and NetBeans contain at least one bug related to compilation errors, our approach did not find bugs related to compilation errors in 50% and 90% of the refactorings of JRRTv1 and JRRTv2, respectively. In Eclipse JDT, the Rename Class refactoring contains three bugs; from JRRTv1 and JRRTv2, the Move Method refactoring showed more bugs than the other refactorings. In NetBeans, three refactorings contain four bugs each. Notice that the Rename Field, Pull Up Field and Move Method implemented by JRRTv1 have more bugs than the similar implementation of Eclipse JDT. After fixing them, JRRTv2 presented fewer bugs than Eclipse JDT.

5.3.2 Behavioral Changes

We identified 18, 13, 10 and 22 bugs in Eclipse JDT, JRRTv1, JRRTv2 and NetBeans, respectively, all related to behavioral changes. We devised an additional, manual bug categorizer (Section 3.4.2) to classify these bugs. For each refactoring, it took approximately two hours to manually classify behavioral changes. As future work, we intend to implement tools to automate this process. For instance, Listings 3 and 4 show a bug of the Pull Up Method refactoring implemented in the Eclipse JDT, categorized as *"Change super to this"*.

5.4 Discussion

Next we discuss some issues related to compilation error, behavior preservation and JDOLLY. In addition, we provide a separate study comparing JDolly with ASTGen/UDITA.

5.4.1 Compilation Errors

Changing the name, location, or accessibility of a declaration can lead to compilation errors. All engines but JRRTv2 produced transformations that reduced the accessibility of an inherited method, which is not allowed in Java. Most compilation errors were due to dereferences of inaccessible or nonexistent declarations. For example, in Listing 5, `m` accesses the `f` field of its super class. If we apply the Pull Up Field refactoring of Eclipse JDT 3.7 to `B.f`, it yields the uncompilable program presented in Listing 6. After the transformation, `B.f` hides `A.f`, and since it is private, it cannot be accessed from `C`. To prevent such errors, JRRT statically checks whether every identifier refers to the same declaration as before. In that case, however, JRRTv1 introduced another compilation error by re-qualifying field access `super.f` to `((A)super).f`, which has a syntax error. We reported this bug to JRRT developers, and they fixed it. JRRTv2 correctly applies the transformation by re-qualifying the `super.f` field access to `((A)this).f`.

Moreover, JRRT refactorings translate the programs into a richer language, which provides a more straightforward specification. After this, the programs are translated back into Java. Although the implementation of the refactoring itself becomes simpler, it does require some effort to translate the program back from the enriched language into the base language. Our technique detected some failures in JRRTv1 that may be related to this step. For instance, some of the refactored programs presented compilation errors due to method invocations for non existing declarations, such as `unknown()`.

Although we only evaluated 9 refactorings from NetBeans, those refactorings contained more bugs related to compilation errors than Eclipse JDT and

Listing 5: Before Refactoring

```
public class A {
  long f = 1;
}
public class B extends A {
}
public class C extends B {
  private long f = 2;
  public long m(){
    return super.f;
  }
}
```

Listing 6: After Refactoring. Pull Up Field implemented by Eclipse JDT 3.7 introduces a compilation error due to an invisible field.

```
public class A {
  long f = 1;
}
public class B extends A {
  private long f = 2;
}
public class C extends B {
  public long m(){
    return super.f;
  }
}
```

JRRT. It seems that NetBeans does not implement a number of expected preconditions. Since its refactorings present a lower rate of rejections, it takes, in general, more time to evaluate NetBeans than the other tools.

5.4.2 Behavioral Changes

Some bugs related to overloading and overriding have been known by Eclipse JDT developers for years. For instance, a bug related to the Add parameter refactoring has demanded the inclusion of additional preconditions since 2004[11]. Nevertheless, it is difficult to establish and check preconditions to avoid these bugs. While the Add Parameter bug is still open, Eclipse JDT developers implemented simpler preconditions for Rename Method, checking whether there are other methods in the hierarchy with the same signature as that of the refactored method. If so, the engine warns the user that the transformation may introduce behavioral changes. In this case, it is up to the user to analyze whether the transformation is safe.

For each refactoring, we analyzed the statement coverage of the random test suite used by SAFEREFACTOR over the program after refactoring; from these, we calculated the mean value of the statement coverage (see Table 6). The minimum mean value of the statement coverage of Eclipse JDT, JRRTv1, JRRTv2, and NetBeans in our evaluation was 54%, 63%, 67%, and 56%, respectively, for the Rename Class refactoring. These numbers can be partially explained by the tests generated only for methods in common. Additionally, most of the programs generated by JDOLLY contain at most four methods, and fewer than 15 LOC. If a class or a method is renamed, and they are not referred to by methods with unchanged signatures, the statement coverage decreases significantly. Since refactorings engines may allow different transformations, and the test suite is randomly generated in SAFEREFACTOR, the mean value of the statement coverage may be different between engines.

The detected bugs can be fixed either by modifying preconditions or changing the transformation itself. For instance, one bug reported to JRRT generates a program with the following code fragment: `((A)super)`. This is an invalid Java expression. We can fix this bug by modifying the transformation applied by JRRT, which rewrites a command with the incorrect fragment. However, fixing bugs may not be as straightforward as it appears to be. For example, consider the transformation showed in Listings 5 and 6. We can fix this bug by adding a precondition avoiding this kind of transformation. However, adding preconditions may avoid useful behavior-preserving transformations. JRRTv1 can apply this transformation, and yet preserve program behavior by replacing the super field access to a qualified this field access, `((A)this).f`.

In some situations, differential testing [27] may help identifying whether a bug can be attributed to some incorrect transformation performed by the engine or due to missing preconditions. Suppose that we use an engine called *X* to apply a refactoring and the SAFEREFACTOR detects a behavioral change. If *Y* applies the same refactoring, and the SAFEREFACTOR cannot find any behavioral changes, then it is likely that the transformation is incorrect in *X*. This solution may work within a limited range of refactoring failures. Although our technique may help debugging refactoring engines in some cases, its main contribution is the detection of bug presence.

5.4.3 JDOLLY

During evaluation, we specified the scope of the program generation in JDOLLY based on previous examples of bugs in refactorings. For instance, we used the scope of two packages since Steimann and Thies [15] show accessibility problems when moving elements between packages. Schäfer et al. [28] show

11. See Eclipse JDT Bug 58616

non-behavior-preserving transformations in programs with up to three classes and four methods/fields. Since JDOLLY exhaustively generates programs for a given scope, this approach has been useful for detecting bugs that have not been detected so far.

JDOLLY generated uncompilable programs. The lowest percentage of compilable programs was in the Add Parameter (63%), and the highest was in the Encapsulate Field (92.8%). Considering all generated programs, the percentage of compilable programs was 68.8%. For future work, we intend to specify more well-formedness constraints so as to minimize uncompilable programs.

Our Java metamodel does not include constructs such as the **static** modifier, inner classes, interfaces, and richer method bodies. Therefore, the currently implementation of JDOLLY cannot reveal some previously identified bugs in manual experiments [28]. We aim at improving the expressiveness of the programs generated by JDOLLY by adding more constructs to our model. This will increase the state space for the Alloy Analyzer to find solutions and, consequently, the number of programs generated by JDOLLY, which will take longer to evaluate all transformations. We plan to investigate the possibility of generating a greater range of programs, specifying as well a time limit, or limiting the number of generated programs. As a result, we will be able to evaluate refactorings by means of more sophisticated programs, though without considering the entire solution space.

Our technique can use any program generator. Specifically, JDOLLY systematically explores the entire combinatorial solution space for the Alloy model within a given scope of elements – a bounded-exhaustive generation method. As a consequence, it generates programs that a purely random generation might miss. There is no randomness in JDOLLY. Likewise, with our technique any test suite generator can replace Randoop. For instance, we can use TestFul [29] instead of Randoop. Randoop is the choice because it detected interesting bugs in real programs [22]. In our initial evaluation, it generated tests that detected several bugs in refactorings [18]. As future work, we intend to evaluate our technique by using different test suite generators.

Test data adequacy criteria provide measurements of test quality. Moreover, it may provide explicit rules to determine when it is appropriate to end the testing phase [30], [31]. There are a number of notions of test data adequacy. For instance, test data adequacy can be defined in terms of covering all productions in grammar-based testing. In our work, we have used a similar test data adequacy criteria. JDOLLY generates *every* possible program, for a subset of the Java metamodel, within a given scope of constructs. As such, the generator covers every terminal symbol and non-terminal production rule from the metamodel, which are represented by signatures and relations from the underlying Alloy specification. In the evaluation of the refactorings (Table 2), JDOLLY generated programs covering from 71% to 85% of the 41 signatures and relations of the metamodel. Some signatures and relations were not covered because we had specified a scope of 0 for `Field`. In other cases, some additional constraints implied that some relations could not have values.

Extending Method Bodies

In this article, we have dealt mainly with testing refactorings that operate at or above the level of methods. The method bodies contain only one statement (see relation `b` of `Method`). We can extend our Alloy specification to test other refactoring, such as Extract Method. First of all, we must change the relation of the `Method` signature. Now, `b` must contain a sequence of statements.

```
sig Method {
  b: seq[Body]
}
```

Moreover, we can extend `Body` to represent other kinds of statements. For instance, we can create the following signature representing method invocation. Notice that we need a new kind of `Id` to represent the variable name that invokes a method id.

```
sig InstanceMethodInvocation extends Body {
  id: one VarId,
  method: one MethodId
}
```

In this way, JDOLLY can generate more elaborated method bodies, such as the one presented next.

```
public void m() {
  A a = new A();
  a.z(2);
  a.y();
}
```

We can guide JDOLLY to generate method bodies with a number of statements (by specifying the main and additional constraints on `b`). For example, we can state that the second and third statements should be extracted on to another method. Additional constraints reduce the space state. In order to avoid generating a number of uncompilable programs, we can consider that the method bodies have a certain pattern, as the one followed by ASTGen.

Comparison with ASTGen and UDITA

There are two program generators that have been used in the literature for testing refactoring engines: ASTGen [17] and UDITA [32]. Although JDOLLY, UDITA, and ASTGen exhaustively generate programs for a given scope, they follow different approaches.

ASTGen follows the generative approach, which means that the tester directly implements how the Java constructs will be combined to generate programs. The more combinations the tester implements, the more programs it will generate. As basis technology, ASTGen uses the Java language and its `Iterable` interface.

However, combining some other Java elements does require some effort. JDOLLY and UDITA follow the filtering approach, that is, the generator automatically searches for all possible combinations of Java constructs to generate programs. Moreover, the tester can specify constraints to filter the program generation. The more constraints the tester specifies, the fewer programs it will generate. While, UDITA uses the Java Path Finder (JPF) model checker as a basis for searching for all possible combinations, JDOLLY grounds its search on SAT solvers used by the Alloy Analyzer. Moreover, the SAT Solvers and the JPF also automatically avoid generating several symmetric programs. On the other hand, in ASTGen, the tester is in charge of this task. UDITA also allows combining generating and filtering approach together.

Besides using different technologies for searching for solutions, JDOLLY and UDITA specify constraints in different styles. While in UDITA the constraints are specified in a Java-like language, in JDOLLY they are specified in Alloy, which is a declarative language.

In previous experiments [32], UDITA was found to be more expressive and easier to use than ASTGen, usually resulting in faster program generation as well. We carried out a similar comparison on the differences between JDOLLY and UDITA. We have considered the generation of inheritance graphs for Java programs in UDITA and JDOLLY. We have compared them with respect to time, specification (or program) size, number of generated programs, and, from these, rate of compilable, isomorphic and non-isomorphic programs. Scopes ranging from 1 to 4 were applied throughout the evaluation. Table 7 summarizes the results of the comparison.

The JDOLLY specification was smaller in size — 14 lines of specification — than the 46 LOC program in UDITA. Alloy logic presented, as expected, a higher level of abstraction than Java-like code. For example, the results of the closure operator in Alloy can only be achieved programmatically after considerable additional effort. Due to the simplicity of the programs generated in this experiment, all programs generated by JDOLLY and UDITA were compilable. However, when evaluating refactorings, both generate uncompilable programs since neither specifies the complete set of Java's well-formedness rules. Moreover, in contrast with JDOLLY, UDITA did not generate 2, 7 and 37 non-isomorphic programs in scopes 2, 3 and 4, respectively. Those programs may be useful for detecting bugs. On the other hand, JDOLLY generated more structurally-equivalent (isomorphic) programs programs than UDITA; this metric has been quantified by a simple algorithm (developed by us) that compares isomorphism between two Java programs. These isomorphic programs may increase the time for testing refactoring engines. Nonetheless, JDOLLY was a bit faster than UDITA in this experiment (Table 7).

Eight out of ten refactorings evaluated by us were also evaluated using ASTGen [17]. JDOLLY generates more programs than ASTGen even with a smaller subset of Java in all refactorings except for the Encapsulate Field refactoring of ASTGen. A similar thing happens in the UDITA evaluation. This indicates that ASTGen and UDITA used stronger additional constraints. Additionally, the number of detected bugs can be used to measure the quality of programs generated by JDOLLY. JRRT developers have used ASTGen to test their renaming refactorings, but no bugs were found [12]. By using JDOLLY, on the other hand, we were able to detect five bugs in the Rename Method and Field refactorings on JRRT.

5.5 Threats to Validity

Next we identify some threats to validity from the evaluation performed.

5.5.1 Construct Validity

Construct validity refers to whether the bugs that we have detected are indeed bugs in the refactoring engines. Tool developers follow a closed world assumption (CWA) as the correctness criterion. In refactoring, CWA means that the test suite is considered part of the program that is being refactored. For instance, the method calls started by the test suite are the only ones to be considered in the precondition checking by the refactoring engine. Our technique, on the other hand, follows an open world assumption (OWA), in which every public method can be a potential target for the test suite generated by SAFEREFACTOR.

Some bugs were not accepted by the JRRT team due to this difference in criteria. Consider the transformation illustrated by Listings 3 and 4, performed by the Pull Up Method refactoring in both JRRTv1 and Eclipse JDT. Under OWA, this transformation does not preserve behavior. The method `test` yields 1 before the transformation, and 2 afterwards. However, under CWA, this transformation preserves behavior, since there is no instance of `B` calling `test`.

Despite the different criteria, many other reported bugs were accepted by JRRT, Eclipse and NetBeans developers (see Table 5). Although our technique may produce false positives, it was considered useful by those developers in practice. In particular, the feedback given by the JRRT team shows evidence that our technique is convenient in detecting bugs under both CWA and OWA criteria.

TABLE 7: Comparison of JDOLLY and UDITA; Prog.: Number of generated programs; Comp.: number of compilable programs; Isomor: number of isomorphic programs; Unique: number of unique programs; NG: number of unique programs that were not generated.

	JDolly							UDITA						
Scope	Lines	Time (s)	Prog.	Comp.	Isomor.	Unique	NG	Lines	Time (s)	Prog.	Comp.	Isomor.	Unique	NG
1	14	1.6	2	100%	0	2	0	46	1	2	100%	0	2	0
2		1.1	6	100%	0	6	0		2	4	100%	0	4	2
3		1.5	29	100%	5	24	0		3	18	100%	1	17	7
4		2.9	230	100%	81	149	0		4	123	100%	11	112	37

5.5.2 Internal Validity

Concerning JDOLLY generation with Alloy, additional constraints may hide possibly detectable bugs. These constraints can be too restrictive with respect to the programs that can be generated by JDOLLY, which shows that one must be cautious when creating constraints for JDOLLY.

The results provided by SAFEREFACTOR deserve closer analysis. If, out of the programs generated by JDOLLY no compilation error or behavior change is detected, no definitive conclusion can be drawn from the refactoring under test. Our technique cannot, based on the absence of behavior changes, claim that a refactoring is correct. Nevertheless, developers have stronger evidence that the refactoring is correctly implemented, in practice; we use a test suite to evaluate the transformation.

SAFEREFACTOR only generates test suites that exercise methods with unchanged signatures. Methods with changed signatures may be called by the unchanged methods, which exercise a potential change of behavior. Otherwise, methods not called by others are not considered, in our approach, part of the overall behavior of the system under test; changes in these methods will not affect the system behavior. A stronger notion of equivalence could be used: testing every changed method of the system and creating a mapping between two versions of the modified versions, for comparing their results. We believe that this approach would add considerable costs with limited benefits to testing refactoring engines.

5.5.3 External Validity

We believe that other refactoring engines can be tested as well with our technique. This exercise can be accomplished by applying a test generator for the target language (a substitute for Randoop) and adaptations to SAFEREFACTOR. Also, the target language's metamodel must be provided to JDOLLY; or else we can use a different program generator. Therefore, refactoring engines targeted at other object-oriented programming languages can benefit from our technique.

Regarding some refactoring transformations other than the ones evaluated in this article, we have showed that our technique is applicable to any transformation, because it does not rely on specific properties of the transformation. In order to generate programs that exercise a specific refactoring, we may have to change the Alloy specification in JDOLLY.

The bug categorizer described in Section 5 is limited, since the classification is not complete. We have only considered a subset of Java. Still, it is non-trivial to pinpoint a bug in a refactoring. Each refactoring engine may incorporate different design choices. Our bug categorizer is an approximation, and it may help refactoring engine developers with this task. For example, our approach may classify two distinct bugs under the same category. After fixing the identified bugs, the developer should re-run the technique to catch possibly missed bugs. Moreover, our approach may identify two distinct bugs that are, in fact, just one. Developers can easily detect whether two different test cases are related to the same bug by fixing each bug and running all bugs again after. In spite of that, the technique reduced from thousands of failing test cases to 120 unique bugs to be checked by refactoring engine developers. This classification was useful when reporting a number of bugs in refactorings in Eclipse JDT, NetBeans and JRRT. Tool developers accepted a number of those bugs.

6 RELATED WORK

6.1 Program Generation

Regarding automatic generation of programs, Grammar-Based Test Generation (GBTB) is a well-known technique for automatically generating programs based on a formal grammar definition [33]. Using this technique, a generator is capable of building valid (or intentionally invalid) sentences of the target language. GBTB has been successfully used, for instance, to generate programs for testing the correctness and error messages in compilers [33], [34]. JDOLLY, by comparison, uses Alloy to specify the Java metamodel using signatures and relations. By performing analysis using the Alloy Analyzer, each Alloy solution is translated into a Java program. Moreover, we can guide JDOLLY to generate programs with properties that are specific to a given target domain (Section 4.4). In contrast, context-free

grammars are somewhat limited for this purpose, being usually extended by operational definitions or even by code snippets for adapting generation to the desired class of test cases.

Recently, GBTB has been mixed with other advanced combinatorial techniques for generating programs of a language grammar. An approach that is very related to JDOLLY's generation technique has been described by Hoffman et al. [35]. It uses grammars instrumented by tags and code snippets written in Python that further constrain the generated test cases. In the referred tool, YouGen, tags inject parameters to the generation. For instance, parameters adjust the depth of a generation tree, limiting the derivation over recursive production rules. This feature is analogous to JDOLLY's scope. Also, while JDOLLY makes use of Alloy Analyzer's exhaustive search to generate a comprehensive set of programs, YouGen uses combinatorial techniques, such as mixed-strength covering arrays. In both cases, they evaluate all possible combinations. Their application contexts are in essence different, however: YouGen has been used for testing XML-based tools and network protocols, whereas JDOLLY is tailored for testing refactoring engines, using SAFEREFACTOR as a test oracle. Still, both tools can be adapted for diverse application cases.

6.2 Refactorings

Steimann and Thies [15] show that by changing access modifiers (`public`, `protected`, `package`, `private`) in Java one can introduce compilation errors and behavioral changes. They propose a constraint-based approach to specify Java accessibility, which favors checking refactoring preconditions and computing the changes of access modifiers needed to preserve the program behavior. Such specialized approach is extremely useful for detecting bugs regarding accessibility-related properties; our approach, on the other hand, is general enough for detecting bugs with respect to other program constructs. They identified a number of bugs that can also be detected by our approach. We have also detected new bugs related to the Java access modifiers. Both approaches can be complementary for testing refactorings that affect accessibility constraints.

Another specialized approach for testing refactorings – generalization-related refactorings such as Extract Interface and Pull Up Method – is proposed by Tip et al. [10]. Their work proposes an approach that uses type constraints to verify preconditions of those refactorings, determining which part of the code they may modify. Using type constraints, they also propose the refactoring Infer Generic Type Arguments [36], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [37]. These refactorings are implemented in the Eclipse JDT. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have preconditions related to other constructs. Our general-purpose testing approach evaluates a refactoring independently of program structures being affected by the refactoring. The bugs detected by our tool are often due to misplaced or omitted preconditions.

Preconditions are a key concept of research studies on the correctness of refactorings. Opdyke [1] proposes a number of refactoring preconditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these preconditions. In fact, later, Tokuda and Batory [38] showed that Opdyke's preconditions were not sufficient to ensure preservation of behavior. Proving refactorings with respect to a formal semantics is a challenge [16]. Some approaches have been contributing in this direction. Borba et al. [8] propose a set of refactorings for a subset of Java with copy semantics (ROOL). They prove the refactoring correctness based on a formal semantics. Silva et al. [11] propose a set of behavior-preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each one of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. Yet, they have not considered all Java constructs, such as overloading and field hiding. Our testing approach still applies formal verification techniques (first-order logic and Alloy Analyzer) that are combined for a practical and less costly solution for increasing confidence when refactoring Java programs.

Mens et al. [39] use graph rewriting for formalizing program refactorings. Two refactorings are specified for a subset of Java, and the authors propose a static semantics for Java, which is preserved by the two refactoring specifications. Graph-based verification is more ambitious than testing, aiming at full structural analysis, although presenting limited scalability. They have recognized that some refactorings, such as Move Method, which may deal with nested structures, require complex graph manipulation. Such analysis becomes considerably costly, which limits its results, in comparison with a more lightweight testing approach.

Schäfer et al. [40] propose refactorings for concurrent programs. They have proved the correctness based on the Java memory model. Currently, we do not deal with concurrency, since SAFEREFACTOR can only evaluate sequential Java programs. However, they have demonstrated that some useful refactorings are not influenced by concurrency. In those situations, we can use SAFEREFACTOR.

Overbey and Johnson [41] propose a technique to check for behavior preservation. They implement it in a library containing preconditions for the most common refactorings. Refactoring engines for different languages can use their library to check refactoring

preconditions. The preservation-checking algorithm is based on exploiting an isomorphism between graph nodes and textual intervals. They evaluate their technique for 18 refactorings in refactoring engines for Fortran 95, PHP 5 and BC. In our approach, we use SAFEREFACTOR to evaluate whether a transformation is behavior-preserving. The authors acknowledge the challenge of specifying preconditions for refactoring; they provide a simple and expressive way of specifying these preconditions. Nevertheless, producing correct preconditions in languages with a complex semantics remains a difficult problem. We chose to build a practical approach for improving quality of refactoring engines, independently of how preconditions are specified. Our technique can complement their approach to improve confidence that the refactorings are sound.

Testing refactoring correctness can be useful in other contexts as well. Dig and Johnson [42] studied the API changes of some frameworks. They have discovered that more than 80% of the changes that break API clients are refactorings. This suggests that refactoring-based migration engines should be used to update applications. API users can use SAFEREFACTOR for checking whether API changes modify their programs' behavior. Furthermore, Reichenbach et al. [43] propose the program metamorphosis approach for program refactoring. It breaks a coarse-grained transformation into small transformations. Although these small transformations may not preserve behavior individually, they guarantee that the coarse-grained transformation preserves behavior. Our approach can be used to increase confidence that the set of small transformations, applied in sequence, indeed preserve behavior.

Soares et al. [26] propose a technique to identify overly strong conditions based on differential testing [27]. If a tool correctly applies a refactoring according to SAFEREFACTOR and another tool rejects the same transformation, the latter has an overly strong condition. In a sample of 42,774 programs generated by JDolly, they evaluated 27 refactorings of Eclipse JDT, NetBeans and JRRT, and found 17 and 7 types of overly strong conditions in Eclipse JDT and JRRT, respectively. This approach is useful for detecting whether the set of refactoring preconditions is minimal. This work complements the technique presented here. Our approach focuses on weak conditions in refactorings. Therefore a combination of these approaches may be useful for detecting whether the set of refactoring preconditions is complete and minimal.

6.3 Automated Testing

Daniel et al. [17] propose an approach for automated testing refactoring engines. They used ASTGen to generate programs as input to refactoring engines. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. For instance, one of them checks for compilation errors, while another applies the inverse refactoring to the target program, and compares the result with the source program. If they were syntactically different, the refactoring engine developer would manually check whether they have the same behavior. They evaluated the technique by testing 21 refactorings, and identified 21 bugs in Eclipse JDT and 24 in NetBeans. In Eclipse JDT, 17 bugs were related to compilation errors, 3 were related to incomplete transformations (e.g. the Encapsulate field did not encapsulate all field accesses), and one was related to behavioral change. Later, Gligoric et at. [32] used the same approach to evaluate UDITA. They found 4 new compilation error bugs in 6 refactorings (2 in Eclipse JDT and 2 in NetBeans). While the oracles of previous approaches can only syntactically compare the programs to detect behavioral changes, SAFEREFACTOR generates tests that do compare program behavior. We found 63 bugs related to behavioral changes. Moreover, both techniques found a similar number of bugs related to compilation errors.

Li and Thompson [44] propose an approach to test refactorings for Erlang using a tool called Quvid QuickCheck. They evaluate a number of implementations of the Wrangler refactoring engine. For each refactoring, they state a number of properties that it must satisfy, which is still a challenge. If a refactoring applies a transformation but does not satisfy a property, they indicate a bug in the implementation. We evaluate behavior preservation by using SAFEREFACTOR. We propose a similar approach for testing refactorings for Java. Their approach applies refactorings to a number of real case studies and toy examples. In contrast, we apply refactorings to a number of programs generated by JDOLLY.

Korel and Yami [45] propose an approach to automated regression test generation [46]. They use TESTGEN, a test data generation system for Pascal programs. Similarly, a component of our approach, SAFEREFACTOR, tests evaluate whether a transformation preserves behavior. It uses the Randoop test generator. They test the parts of the programs whose functionality is unchanged after modifications. SAFEREFACTOR automatically detects the methods with unchanged signatures and generates tests for them. We are concerned with testing refactorings in this article.

A more recent contribution concerning regression test generation is provided by BERT [47], a tool that focuses on detection of state changes from one version of a given class to its next version, considering transformations of any category (not only refactoring). The main distinction between the two approaches is their test oracle. SAFEREFACTOR uses a simple oracle that compares outputs of methods with unchanged

signatures for the same input. If any changed behavior is, directly or indirectly, exercised by one of these methods, there is a high probability that the test goes wrong, and a behavior change is detected. BERT, on the other hand, focuses on identifying differences in several structural aspects of the target program: return values of all methods, field values, and even output (textual) results. If a change is detected, there is an indication of a regression bug, although this may be not the case (false positives). Since they evaluate any kind of transformation, developers have to analyze whether the behavioral changes have been intentionally introduced.

Marinov and Khurshid [48], [49] propose 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 post-conditions. This approach is similar to JDOLLY for generating test inputs, but we generate programs as test inputs.

7 Conclusion

In this article, we propose a technique to test Java refactoring engines. This technique is made up of a Java program generator (in our proposal, JDOLLY) and a test system for refactorings (SAFEREFACTOR). For each refactoring, the technique generates a number of Java programs, followed by the application of the refactoring, with these programs as target. It uses SAFEREFACTOR to evaluate the correctness of the transformations. Finally, the technique classifies the failing transformations by kind of behavioral change or compilation error introduced by them. We propose a Java program generator (JDOLLY) to run the program generation step of our technique. It uses Alloy [19] and the Alloy Analyzer [20] to create programs for a given scope of elements (packages, classes, fields, and methods). We have evaluated our technique by testing 29 refactorings, and found 57 and 63 bugs related to compilation errors and behavioral changes, respectively.

Implementing refactorings is not simple. Even refactoring engines written with correctness in mind, such as JRRT, still have bugs. We have shown some corner cases automatically detected by our technique. With these results, we have demonstrated how the combination of JDOLLY and SAFEREFACTOR is powerful to detect bugs in refactorings. In the absence of formal proofs, our technique can be useful for the improvement of previous solutions. We have reported all bugs to Eclipse JDT, NetBeans and JRRT, and a number of them have already been accepted prior to this submission. Moreover, Eclipse JDT and NetBeans developers have fixed some of them, and JRRT developers have already fixed all accepted bugs. They have also included our test cases in their test suite.

Acknowledgments

We gratefully thank Paul Strooper, Paulo Borba, Augusto Sampaio, Max Schäfer and the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 477336/2009-4, 304470/2010-4, and 480160/2011-2.

References

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

[2] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, pp. 126–139, February 2004.

[4] Eclipse.org, "Eclipse project," 2011, at http://www.eclipse.org.

[5] Sun Microsystems, "NetBeans IDE," 2011, at http://www.netbeans.org/.

[6] Jet Brains, "IntelliJ Idea," 2011, at http://www.intellij.com/idea/.

[7] Embarcadero Technologies, "JBuilder," 2011, at http://www.codegear.com/br/products/jbuilder.

[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, August 2004.

[9] M. Cornélio, "Refactorings as Formal Refinements," Ph.D. dissertation, Federal University of Pernambuco, 2004.

[10] F. Tip, A. Kieżun, and D. Bäumer, "Refactoring for generalization using type constraints," in *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 13–26.

[11] L. Silva, A. Sampaio, and Z. Liu, "Laws of object-orientation with reference semantics," in *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 217–226.

[12] M. Schäfer, T. Ekman, and O. de Moor, "Sound and extensible renaming for Java," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 277–294.

[13] M. Schäfer, M. Verbaere, T. Ekman, and O. Moor, "Stepping stones over the refactoring rubicon," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ser. ECOOP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 369–393.

[14] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *Proceedings of the 25th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 286–301.

[15] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ser. ECOOP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 419–443.

[16] M. Schäfer, T. Ekman, and O. de Moor, "Challenge proposal: verification of refactorings," in *Proceedings of the 3rd workshop on Programming Languages Meets Program Verification*, ser. PLPV '09. New York, NY, USA: ACM, 2008, pp. 67–72.

[17] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194.

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

[19] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[20] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the Alloy constraint analyzer," in *Proceedings of the 22nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 730–733.

[21] J. Gosling, B. Joy, G. L. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[23] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FASE '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 171–185.

[24] Eclipse.org, "JDT core component," 2011, at http://www.eclipse.org/jdt/core/.

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

[26] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, September 2011, pp. 173–182.

[27] W. Mckeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[28] M. Schäfer, T. Ekman, R. Ettinger, and M. Verbaere, "Refactoring bugs," 2011, at http://code.google.com/p/jrrt/wiki/RefactoringBugs.

[29] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for Java classes," in *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 281–284.

[30] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *SIGPLAN Notes*, vol. 10, pp. 493–510, April 1975.

[31] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Survey*, vol. 29, pp. 366–427, December 1997.

[32] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 225–234.

[33] A. Celentano, S. C. Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti, "Compiler testing using a sentence generator," *Software: Practice and Experience*, vol. 10, no. 11, pp. 897–918, November 1980.

[34] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Transactions on Software Engineering*, vol. 8, pp. 343–353, July 1982.

[35] D. M. Hoffman, D. Ly-Gagnon, P. Strooper, and H.-Y. Wang, "Grammar-based test generation with YouGen," *Software: Practice and Experience*, vol. 41, pp. 427–447, April 2011.

[36] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP '05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 71–96.

[37] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 265–279.

[38] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Automated Software Engineering*, vol. 8, pp. 89–120, January 2001.

[39] T. Mens, S. Demeyer, and D. Janssens, "Formalising behaviour preserving program transformations," in *Proceedings of the 1st International Conference on Graph Transformation*, ser. ICGT '02. London, UK: Springer-Verlag, 2002, pp. 286–301.

[40] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent Java code," in *Proceedings of the 24th European Conference on Object-Oriented Programming*, ser. ECOOP '10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 225–249.

[41] J. L. Overbey and R. E. Johnson, "Differential precondition checking: A lightweight, reusable analysis for refactoring tools," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. New York, NY, USA: ACM, 2011, pp. 303–312.

[42] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 389–398.

[43] C. Reichenbach, D. Coughlin, and A. Diwan, "Program metamorphosis," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ser. ECOOP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 394–418.

[44] H. Li and S. Thompson, "Testing Erlang Refactorings with QuickCheck," in *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, ser. Lecture Notes in Computer Science, vol. 5083. Springer, 2008, pp. 19–36.

[45] B. Korel and A. M. Al-Yami, "Automated regression test generation," in *Proceedings of the 4th International Symposium on Software Testing and Analysis*, ser. ISSTA '98. New York, NY, USA: ACM, 1998, pp. 143–152.

[46] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, pp. 242–257, December 1970.

[47] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Proceedings of the 23rd International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 137–146.

[48] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 22–34.

[49] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, "Software assurance by bounded exhaustive testing," *IEEE Transactions on Software Engineering*, vol. 31, pp. 328–339, April 2005.