Detecting Overly Strong Preconditions in Refactoring Engines

Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira

Abstract—Refactoring engines may have overly strong preconditions preventing developers from applying useful transformations. We find that 32 percent of the Eclipse and JRRT test suites are concerned with detecting overly strong preconditions. In general, developers manually write test cases, which is costly and error prone. Our previous technique detects overly strong preconditions using differential testing. However, it needs at least two refactoring engines. In this work, we propose a technique to detect overly strong preconditions in refactoring engines without needing reference implementations. We automatically generate programs and attempt to refactor them. For each rejected transformation, we attempt to apply it again after disabling the preconditions that lead the refactoring engine to reject the transformation. If it applies a behavior preserving transformation, we consider the disabled preconditions overly strong. We evaluate 10 refactorings of Eclipse and JRRT by generating 154,040 programs. We find 15 overly strong preconditions in Eclipse and 15 in JRRT. Our technique detects 11 bugs that our previous technique cannot detect while missing 5 bugs. We evaluate the technique by replacing the programs generated by JDolly with the input programs of Eclipse and JRRT test suites. Our technique detects 14 overly strong preconditions in Eclipse and 4 in JRRT.

Index Terms—Refactoring, overly strong preconditions, automated testing, program generation

1 Introduction

Refactoring is the process of changing a program to improve its internal structure while preserving its observable behavior [1], [2], [3]. Refactorings can be applied manually, which may be time consuming and error prone, or automatically by using a refactoring engine, such as Eclipse [4], NetBeans [5], and JastAdd Refactoring Tools (JRRT) [6], [7], [8], [9]. These engines contain a number of refactoring implementations, such as Rename Class, Pull Up Method, and Encapsulate Field. For correctly applying a refactoring, and thus, ensuring behavior preservation, the refactoring implementations might need to consider a number of preconditions, such as checking whether a method or field with the same name already exists in a type. However, defining and implementing preconditions is a nontrivial task. Proving the correctness of the preconditions with respect to a formal semantics of complex languages such as Java, constitutes a challenge [10].

In practice, refactoring engine developers may implement the refactoring preconditions based on their experience, some previous work [11], or formal specifications [6]. Nevertheless, refactoring engines may have overly weak and overly strong preconditions [12], [13], [14]. Overly weak preconditions allow incorrect transformations to be applied, while overly strong preconditions prevent developers from applying behavior preserving transformations. Neither of these problems are desirable. Vakilian and Johnson [15] have shown that programmers prefer that the refactoring engine does not reject a refactoring application, even if they need to manually fix some problems afterwards.

Some approaches have proved the correctness of some refactorings for a subset of Java [16], [17], [18]. They ensure that the resulting programs compile and that the transformations preserve the program behavior. However, none of them proved that such preconditions are minimal. As a result, those approaches may reject useful transformations. In practice, developers are concerned about testing whether their refactoring implementations have overly strong preconditions. For example, we find that 32 percent of the assertions used in the test suites of 10 refactoring implementations of Eclipse (30 percent) and 10 of JRRT (36 percent) are concerned with detecting overly strong preconditions. Moreover, we find more than 40 bugs related to overly strong preconditions in the Eclipse's bug tracker. More than 50 percent of them were already fixed. Testing refactoring engines is not trivial since it requires complex inputs, such as programs and an oracle to define the correct resulting program. In general, developers manually select the input programs according to their experience, which is costly, and thus it may be difficult to establish a good test suite considering all language constructs. Moreover, even if they establish useful inputs, they may not have a systematic technique to detect the overly strong preconditions.

Vakilian and Johnson [15] use refactoring alternate paths to identify usability problems related to overly strong preconditions in refactoring engines. They discover usability problems

- M. Mongiovi, R. Gheyi, and G. Soares are with the Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB 58429-900, Brazil. E-mail: melina@computacao.ufcg.edu.br, {rohit, gsoares}@dsc.ufcg.edu.br.
- M. Ribeiro is with the Computing Institute, Federal University of Alagoas, Maceió, AL 57072-900, Brazil. E-mail: marcio@ic.ufal.br.
- P. Borba and L. Teixeira are with the Informatics Center, Federal University of Pernambuco, Recife, PE 50732-970, Brazil. E-mail: {phmb, lmt}@cin.ufpe.br.

Manuscript received 15 Apr. 2016; revised 19 Mar. 2017; accepted 5 Apr. 2017. Date of publication 11 Apr. 2017; date of current version 22 May 2018. Recommended for acceptance by M. Kim.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2693982

0098-5589 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

by analyzing the interactions of users that had problems with tools in general. They manually inferred usability problems from the detected critical incidents.

Our previous work uses Differential Testing [19] to automatically identify transformations rejected by refactoring engines due to overly strong preconditions (DT technique) [13]. It automatically generates a number of programs as test inputs using JDOLLY, a Java program generator [12]. Next, it applies the same refactoring to each test input using two different implementations, and compares both results. The technique uses SAFEREFACTOR [20], [21] to automatically evaluate whether a transformation preserves the program behavior. SAFEREFACTOR automatically evaluates whether two versions of a program have the same behavior by automatically generating test cases only for the common methods impacted by the change. To use this technique, developers need access to at least two refactoring engines. However, it can only be used if both refactoring engines implement the same refactoring.

In this work, we propose Disabling Preconditions (DP), a new technique to detect overly strong preconditions in refactoring implementations by disabling preconditions. Hereafter we refer to disabling preconditions as the process of preventing to report messages to the user, raised by the preconditions. A message is reported when a precondition is unsatisfied. We automatically generate a number of programs as test inputs, using JDOLLY. For each generated program, we attempt to apply the transformation using the refactoring implementation that is being tested. When the refactoring implementation rejects a transformation, it reports a message to the user describing the problem. For each kind of message, we identify code fragments related to the precondition that yields the message. There may be a number of preconditions related to each message, but for simplicity we consider, for each refactoring implementation, one precondition per message in our technique. Next, we modify the refactoring implementation to disable the code fragments that prevented the refactoring application. We propose the DP changes to facilitate and systematize the process of modifying the code to allow disabling preconditions. They include an *If* statement around the code fragment that we want to disable.

The final step of our technique consists of evaluating whether the identified preconditions are overly strong. For each rejected transformation, we attempt to apply the refactoring that is being tested using the refactoring implementation with the preconditions that raise the reported messages disabled. If it still rejects the transformation, thus reporting another message, we repeat this process until the refactoring implementation applies a transformation. If a transformation applied by the refactoring implementation with some disabled preconditions preserves the program behavior according to SAFEREFACTOR, then we classify the set of disabled preconditions as overly strong. Otherwise, we evaluate the next rejected transformation.

We evaluate our technique using 10 refactoring implementations of Eclipse JDT 4.5 and 10 refactoring implementations of the latest JRRT version [6]. JDOLLY generates 154,040 programs as test inputs and the DP technique detects 30 overly strong preconditions in the refactoring implementations (15 in JRRT and 15 in Eclipse). So far, Eclipse developers confirmed 47 percent of them. The technique takes 0.89 and 35.72 h to detect all overly strong preconditions of JRRT and Eclipse, respectively. Moreover, the technique takes on average a few seconds to find the first overly strong precondition in JRRT and 17.41 min in Eclipse.

DP and DT techniques are complementary in terms of bug detection. The DP technique finds 11 bugs not detected by DT, and the DT technique finds 5 bugs not detected by DP (false-negatives). In addition, the DP technique does not require using another refactoring engine with the same refactorings to compare the results. So whenever possible, developers can run the DP technique and after fixing the detected bugs, they run the DT technique to search for more bugs.

We also perform another study, where we use programs from Eclipse and JRRT refactoring test suites as input for DP and DT techniques instead of the JDOLLY generated programs. These programs are used in test cases that expect the refactoring engine to prevent refactoring application. We assess the same refactoring implementations evaluated before using both techniques. We detect 23 overly strong preconditions (17 of them are new) previously undetected by the developers. In addition to these bugs, we find some false-positives due to the equivalence notion of SAFEREFACTOR. The developers did not find these overly strong preconditions because they may not have a systematic strategy to detect them, even with useful input programs in their test suite. Additionally, they may not have an automated oracle to check behavior preservation, such as SAFEREFACTOR.

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

- a new technique to detect overly strong preconditions in refactoring implementations (Section 4);
- an evaluation of the DP technique with respect to detection of overly strong preconditions and the time to find them (Section 5);
- a comparative study of the DP and DT techniques with respect to detection of overly strong preconditions (Section 5).

We organize this article as follows. We present a motivating example in Section 2, and provide some background on JDOLLY and SAFEREFACTOR in Section 3. Section 4 describes our technique to detect overly strong preconditions in refactoring implementations. Next, Section 5 presents the evaluation of our technique and its comparison with the DT technique. Finally, we relate DP technique to others (Section 6), and present concluding remarks (Section 7).

2 MOTIVATING EXAMPLE

In this section, we present a transformation rejected by Eclipse due to an overly strong precondition. Consider Listing 1, illustrating part of a program that handles queries to a database. It provides support for two database versions. Each version is implemented in a class: *QueryV1* (database version 1) and *QueryV2* (database version 2). They enable client code to swap in support for one version, or another. Those classes extend a common abstract class *Query*, which declares an abstract method *createQuery*. This method is implemented in each subclass in a different way. A query created by the *createQuery* method is executed by the *doQuery* method. Notice that this method is duplicated in both subclasses: *QueryV1* and *QueryV2*.

Listing 1. It is not possible to pull up doQuery method from QueryV1 and QueryV2 classes to Query class using Eclipse JDT 2.1 due to overly strong preconditions.

```
public abstract class Query {
  protected abstract SDQuery createQuery();
}
public class QueryV1 extends Query {
  public void doQuery() {
    SDQuery sd = createQuery();
    //execute query
  }
  protected SDQuery createQuery() {
   //create query for the database version 1
  }
}
public class QueryV2 extends Query {
  public void doQuery() {
    SDQuery sd = createQuery();
    //execute query
  }
  protected SDQuery createQuery() {
   //create query for the database version 2
  }
}
```

Listing 2. Correct resulting program.

```
public abstract class Query {
  protected abstract SDQuery createQuery();
  public void doQuery() {
     SDQuery sd = createQuery();
     //execute query
  }
}
public class QueryV1 extends Query {
  protected SDQuery createQuery() {
     //create query for the database version 1
  }
}
public class QueryV2 extends Query {
  protected SDQuery createQuery() {
     //create query for the database version 2
  }
}
```

We can pull up the *doQuery* method to remove duplication. Using Eclipse JDT 2.1 to apply this refactoring, it warns that the *doQuery* method does not have access to *createQuery*. This precondition checks whether after the transformation, the pulled up method still has access to all of its called methods. However, *createQuery* already exists as an abstract method in the *Query* class, which indicates that this precondition is overly strong. This bug was reported in Eclipse's bug tracker.[1] Kerievsky reported it when he was working out mechanics for a refactoring to introduce the Factory Method pattern [22]. He argued that "*there should be no warnings as the transformation is harmless and correct*." The Eclipse developers fixed this bug. Listing 2 illustrates a correct resulting program applied by Eclipse JDT 4.5. We found more than 40 bugs related to overly strong preconditions in the bug tracker of Eclipse. As of this writing, the Eclipse developers have already fixed more than 50 percent of them.

1. https://bugs.eclipse.org/bugs/show_bug.cgi?id=39896

We also investigated the test suite of 10 refactorings from Eclipse JDT 4.5 and JRRT: Rename Method, Rename Field, Rename Type, Add Parameter, Encapsulate Field, Move Method, Pull Up Method, Pull Up Field, Push Down Method, and Push Down Field. We classified a total of 2,559 assertions and find that 32 percent of them are concerned to overly strong preconditions. We consider the following kind of assertion as concerned to overly strong preconditions in the Eclipse test suite. It checks if Eclipse applies the transformation.

```
assertTrue("precondition was supposed to pass",
           !result.hasError())
```

In the JRRT test suite we identified one kind of assertion related to overly strong preconditions as presented next. A test failure indicates that the refactoring implementation may have overly strong preconditions.

```
fail("Refactoring was supposed to succeed;
   failed with " + rfe)
```

This way, we observe that Eclipse and JRRT developers are indeed concerned with identifying overly strong preconditions in their refactoring implementations. Moreover, they may not seem to have a systematic way to create test cases to assess the refactoring implementations with respect to overly strong preconditions.

3 Background

In this section, we present an overview of JDolly [12] (Section 3.1) and SafeRefactor [20], [21] (Section 3.2).

3.1 JDolly

JDolly is an automated and bounded-exhaustive Java program generator [12], [13] based on Alloy, a formal specification language [23]. JDolly receives as input an Alloy specification with the scope, which is the maximum number of elements (classes, methods, fields, and packages) that the generated programs may declare, and additional constraints for guiding the program generation. It uses the Alloy Analyzer tool [24], which takes an Alloy specification and finds a finite set of all possible instances that satisfy the constraints within a given scope. JDolly translates each instance found by the Alloy Analyzer to a Java program. It reuses the syntax tree available in Eclipse JDT for generating programs from those instances. Listing 3 illustrates an example of a program generated by JDolly.

An Alloy specification is a sequence of signatures and constraints paragraphs declarations. A signature introduces a type and can declare a set of relations. Alloy relations specify multiplicity using qualifiers, such as *one* (exactly one), *lone* (zero or one), and *set* (zero or more). In Alloy, a signature can extend another, establishing that the extended signature is a subset of the parent signature. For example, the following Alloy fragment specifies part of the Java metamodel of JDolly encoded in Alloy. A Java class is a type, and may extend another class. Additionally, it may declare fields and methods.

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

A number of well-formed constraints can be specified for Java. For instance, a class cannot extend itself. In Alloy, we can declare facts, which encapsulate formulas that always hold. The *ClassCannotExtendItself* fact specifies this constraint. The *all*, *some*, and *no* keywords denote the universal, existential, and non-existential quantifiers respectively. The ^ and ! operators represent the transitive closure and negation operators respectively. The dot operator (.) is a generalized definition of the relational join operator.

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

The Alloy model is used to generate Java programs using the *run* command, which is applied to a predicate, specifying a scope for all declared signatures in the context of a specific Alloy model. Predicates (*pred*) are used to encapsulate reusable formulas and specify operations. For example, the following Alloy fragment specifies that we should run the *generate* predicate using a scope of 3. The user can also specify different scopes for each signature.

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

The user can guide JDOLLY to generate more specific programs. For example, to generate programs to test the Pull Up Method refactoring, JDOLLY uses the following additional constraints. It specifies that a program must have at least one class (*C2*) extending another class (*C1*), and that *C2* declares at least a method (*M1*). The program in Listing 3 satisfies all constraints of this specification.

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

Furthermore, developers can specify a skip number to jump some of the Alloy instances. For a skip of size n such that $n > 1$, JDOLLY generates one program from an Alloy instance, and jumps the following $n-1$ Alloy instances. Consecutive programs generated by JDOLLY tend to be very similar, potentially detecting the same kind of bug [14], [23]. Thus, developers can set a parameter to skip some of the generated programs to reduce the time needed to test the refactoring implementations. It avoids generating an impracticable number of Alloy instances by the Alloy Analyzer.

3.2 SafeRefactor

SAFEREFACTOR [20], [21] evaluates whether a transformation preserves program behavior by automatically generating test cases for the methods impacted by the change with matching signature (methods with exactly the same modifier, return type, qualified name, parameter types and exceptions thrown) before and after the transformation. First, it decomposes a coarse-grained transformation into smaller transformations. For each small-grained transformation, it identifies the set of impacted methods. We formalized the impact of each small-grained transformation [21]. Moreover, it also identifies the methods that call an impacted method directly or indirectly. SAFEREFACTOR generates a test suite for the public common impacted methods using Randoop [25], an automated test suite generator. Randoop generates tests within a time limit specified by the user. Finally, it executes the same test suite before and after the transformation. If the results are different, the tool reports a behavioral change, and yields the test cases that reveal it. Otherwise, we improve confidence that the transformation preserves the program behavior.

For example, we can use SAFEREFACTOR to analyze whether the Pull Up Method transformation applied to Listing 3 preserves the program behavior. The refactored program is illustrated in Listing 4. The original program contains class *A* and its subclass *B*. *A* declares method *k* and *B* declares methods *k*, *m*, and *test*. Method *B.test* calls method *B.m*, which calls method *B.k* yielding 2. The transformation pulls up method *B.m* to *A*. To analyze this transformation, SAFEREFACTOR receives both programs as input. First, it identifies the public and common impacted methods. In this example, we have two small-grained transformations: remove the *B.m* method and add the *A.m* method. Since there is no other *m* method in the hierarchy, the small-grained transformations only impact these methods. Next, it identifies the methods that directly or indirectly call the impacted methods. In this example, *B.test* calls *B.m* (original program) and *A.m* (refactored program). Therefore, *B.m*, *A.m*, and *B.test* are impacted by the transformation. Only these methods may have changed their behavior after the transformation. SAFEREFACTOR only generates tests for the impacted methods, which are common to both programs (*B.m* and *B.test*) using a time limit of 0.5 s. Finally, it runs the test suite on the original and refactored programs. Since all methods yield the same value before and after the transformation, the test cases pass in both programs. Therefore, SAFEREFACTOR reports that the transformation preserves the program behavior.

Listing 3. Original program.

```java
public class A {
    protected long k(long a){
        return 1;
    }
}
public class B extends A {
    public long m(){
        return new B().k(2);
    }
    public long k(int a){
        return 2;
    }
    public long test(){
        return m();
    }
}
```

Listing 4. Refactored program after pulling up method B.m to class A.

```
public class A {
  protected long k(long a){
    return 1;
  }
  public long m() {
    return new B().k(2);
  }
}
public class B extends A {
  public long k(int a){
    return 2;
  }
  public long test(){
    return m();
  }
}
```

SafeRefactor states that two program versions have the same behavior when the public methods with unchanged signatures, which are impacted by the change also have the same behavior. It only generates tests for these methods. Methods impacted by the change with changed signatures may be called by the unchanged methods, which exercise a potential behavioral change. SafeRefactor does not consider methods with changed signatures that are not called by any method in the system.

In previous studies [20], [21], [26], SafeRefactor detected behavioral changes in transformations applied to programs with up to 100 KLOC. Moreover, SafeRefactor detected behavioral changes in transformations applied to real systems that even a manual process [27] conducted by experts did not detect [21]. Additionally, our previous techniques used SafeRefactor as the oracle to detect overly weak and overly strong preconditions in refactoring engines [12], [13], [14]. By using SafeRefactor, we detected more than 200 bugs in refactoring implementations of Eclipse, JRRT, and NetBeans. Some of them were already fixed. In those studies, we used the JDolly generated programs as the input for our technique. The programs have at least one public method, which SafeRefactor can generate tests to evaluate the transformation using a time limit of 0.5 s.

4 Detecting Overly Strong Preconditions

In this section, we explain our proposed technique to detect overly strong preconditions in refactoring implementations. Section 4.1 presents an overview of our technique. Section 4.2 describes it by using an example. Next, we explain in more details some of the steps involved: identification of different kinds of messages reported by the refactoring implementation (Section 4.3), and the process of applying transformations to allow disabling the execution of the identified preconditions (Section 4.4).

4.1 Overview

Our technique receives as input a refactoring implementation, the DP changes used to allow disabling the preconditions, and some parameters to configure JDolly, such as skip, scope, and additional constraints. Each precondition checks whether the transformation may introduce a specific problem in the program, which can result in compilation errors or behavioral changes. The technique returns the modified refactoring implementation, and all transformations that yield a set of overly strong preconditions in the original refactoring implementation. Fig. 1 illustrates the main steps of our technique.

First, JDolly automatically generates programs as test inputs (Step 1). Next, the refactoring implementation under test attempts to apply the transformations to each generated program. If the refactoring implementation rejects a transformation, we collect the messages reported to the user (Step 2). For each kind of message, we inspect the refactoring implementation code and manually identify the code fragments related to the precondition that raises it. We assume, for each refactoring implementation, that there is one precondition related to each kind of message. Then, we modify the refactoring implementation code by adding *If* statements to allow disabling the execution of the identified precondition using the DP changes (Step 3). The goal is to apply the transformation instead of reporting the message again.

Once the technique changes the refactoring implementation code to allow automatically disabling the preconditions, we evaluate them. For each transformation rejected by the refactoring implementation, it automatically tries to apply the same transformation again with a disabled precondition (Step 4). If the refactoring implementation rejects the transformation and reports another message, it repeats the process by disabling more preconditions until the refactoring implementation applies a transformation. If the modified refactoring implementation applies the transformation and the resulting program preserves the program behavior according to SafeRefactor [21], then the technique classifies the set of disabled preconditions as overly strong (Step 5). Otherwise, it analyzes the next rejected transformation. Once we classify a precondition as overly strong, we do not evaluate it again with other inputs generated by JDolly that yield the same message. Algorithm 1 summarizes the main steps.

4.2 Example

Suppose we would like to test the Pull Up Method refactoring implementation from Eclipse JDT 4.5. First we have to specify JDolly parameters (skip, scope, additional constraints). Then JDolly generates a number of programs (Step 1), such as the one presented in Listing 3. In Step 2, we apply the refactoring implementation to all generated programs. The refactoring implementation applies a transformation to some of them, and it may reject applying a refactoring to some generated programs. For instance, if we try to apply the Pull Up Method refactoring to the program in Listing 3 to move *B.m* to class *A* using Eclipse JDT 4.5, it reports the following message: *Method "B.k(...)" referenced in one of the moved elements is not accessible from type "A."*.

For each rejected transformation, we collect the reported messages and inspect the refactoring implementation code to identify the refactoring preconditions that raise them (Step 3). In each refactoring engine, we have to investigate how a message is represented (Step 3.2.1). In Eclipse, the *refactoring.properties* file defines variables representing reported messages. In this example, we find the following declaration:

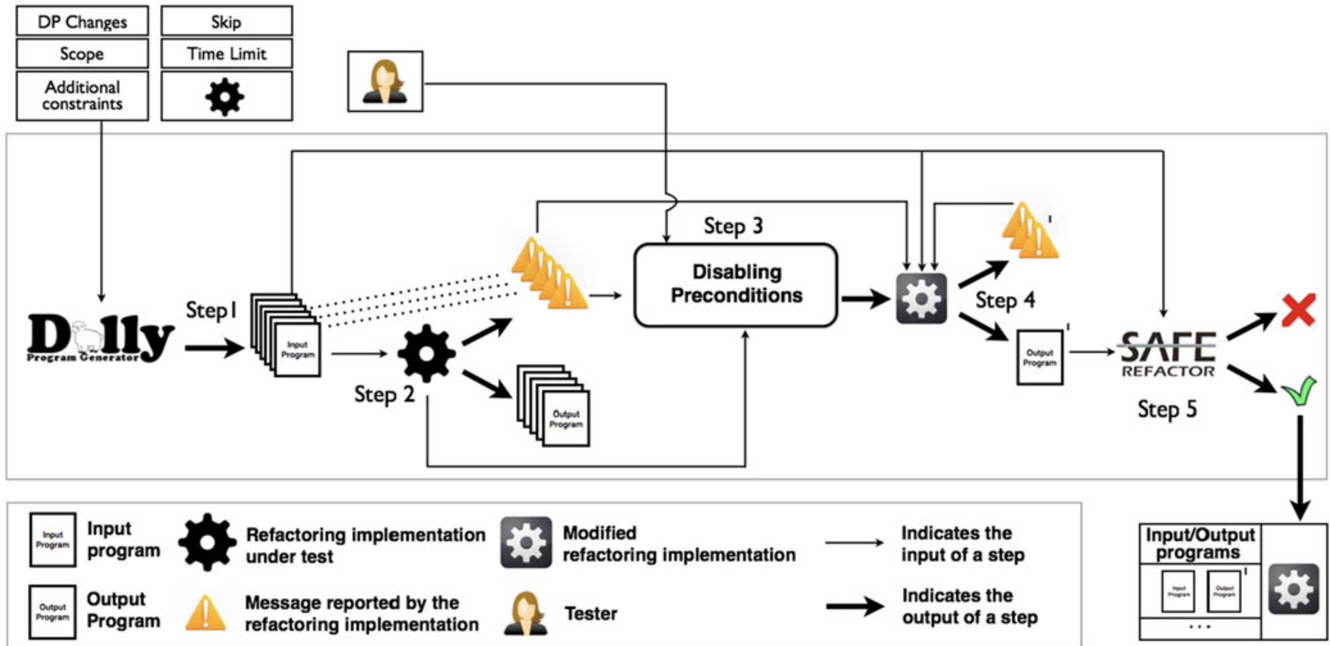

Fig. 1. A technique to detect overly strong preconditions. First, we generate the programs using JDolly (Step 1). For each generated program, we try to apply the transformation (Step 2). Next, for each kind of message reported by the refactoring implementation, we manually perform transformations in the refactoring implementation code to allow disabling the execution of a precondition by preventing to report the message to the user. (Step 3). Then, for each rejected transformation we try to apply the transformation again using the refactoring implementation with the preconditions that raise the reported messages disabled (Step 4). If the transformation preserves the program behavior according to SafeRefactor, we classify the disabled preconditions as overly strong (Step 5).

```
PullUpRefactoring_method_not_accessible =
    Method {0} referenced in one of the moved
    elements is not accessible from type {1}
```

The *PullUpRefactoring_method_not_accessible* field from the *RefactoringCoreMessages* class represents the message yielded by Eclipse when we try to apply the Pull Up Method refactoring in Listing 3.

Listing 5. Original code fragment of Eclipse JDT.

```
1  ...
2  public class PullUpRefactoringImplementation {
3    ...
4    private RefactoringStatus checkAccessedMethods(...)
     throws JavaModelException {
5      final RefactoringStatus st = new
       RefactoringStatus();
6      ...
7        if (!isAccessible) {
8          final String m = Msgs.format(RefactoringCore
           Messages.PullUpRefactoring_method_not_
           accessible,...);
9          st.addError(m, JavaStatusContext.create(method));
10       }
11     ...
12   }
13 }
```

In each refactoring engine, we investigate how to avoid reporting a message (Step 3.3). In Eclipse, we find code fragments (inside a method) that add a warning or error message in a *RefactoringStatus* object (*status*). Before applying the transformation, Eclipse checks if the *status* variable contains some warning or error message and if positive, reports the message to users. If *status* does not have errors or warning messages, Eclipse applies the transformation.

Listing 6. Code fragment after disabling a Pull Up Method refactoring precondition using DP Change 2.

```
1  ...
2  public class PullUpRefactoringImplementation{
3    ...
4    private RefactoringStatus checkAccessed
     Methods(...) throws JavaModelException {
5      final RefactoringStatus st = new
       RefactoringStatus();
6      ...
7        if (!isAccessible) {
8          final String m = Msgs.format(Refactoring
           CoreMessages.PullUpRefactoring_method_
           not_accessible,...);
9          if (ConditionsPullUpMethod.cond1) {
10           st.addError(m, JavaStatusContext.create
             (method));
11         }
12       }
13     ...
14   }
15 }
```

In our example, *checkAccessedMethods* method in Eclipse is the only method that adds *PullUpRefactoring_method_not_accessible* message to a *RefactoringStatus* object. Listing 5 illustrates part of this method. It contains code fragments of a precondition, which checks if each method called from the moved method is accessible from the destination class.

Algorithm 1. A Technique to Detect Overly Strong Preconditions
Input: refactoring implementation *R*, *skip*, *scope*, *constraints*, *timeLimit*, *DPChanges*

Step 1. *progs* = JDOLLY.generate(*skip*, *scope*, *constraints*);
$progs' = \emptyset$; ▷ A set of pairs of programs and messages
$msgs = \emptyset$; ▷ A set of all messages reported by *R*
Step 2. foreach $prog \in progs$ **do**
    msg = *R*.canApplyRefactoring(*prog*); ▷ canApplyRefactoring yields one message, for simplicity, if *R* cannot apply it
    if $msg \neq \emptyset$ **then**
        progs′.add($\langle prog, msg \rangle$);
        msgs.add(*msg*); ▷ For simplicity, it does not show that it removes some names and keywords from *msg*
$map = \emptyset$; ▷ A set of all mappings of messages to preconditions
Step 3.1. Create a class: *public class ConditionsR { public static void enableConditions() {} };*
Step 3.2. foreach $msg \in msgs$ **do**
    Step 3.2.1. Identify how *msg* is represented in *R*; ▷ Specific for each refactoring engine
    Step 3.2.2. Create a fresh public static boolean field (*cond*) in *ConditionsR*. Add *cond* = true in enableConditions;
    Step 3.2.3. *map*.add($\langle msg, cond \rangle$); ▷ It relates each message to a condition
Step 3.3. Identify how to prevent reporting messages to user in *R*; ▷ Specific for each refactoring engine
$R' = R$; ▷ R' will contain the modified refactoring implementation
Step 3.4. foreach $msg \in msgs$ **do**
    Step 3.4.1. *places* = Identify all places in *R* that can prevent reporting *msg* to user;
    Step 3.4.2. foreach $place \in places$ **do**
        R' = applyDPChange(*DPChanges*, R', *place*, *msg*, *map*); ▷ Add *if (ConditionsR.cond) {place}*. Specific for each ref. engine
$transformations = \emptyset$; ▷ A set containing all transformations applied by R'
Step 4. foreach $\langle prog, msg \rangle \in progs'$ **do**
    Step 4.1. ConditionsR.enableConditions(); ▷ It enables all preconditions
    Step 4.2. ConditionsR.(*map*.getCondition(*msg*)) = false; ▷ It disables a condition related to *msg*
    Step 4.3. *msg* = R'.canApplyRefactoring(*prog*);
    if $msg \in msgs$ **then**
        go to **Step 4.2**;
    else if $msg = \emptyset$ **then**
        transformations.add($\langle prog, R'$.applyRefactoring$(prog) \rangle$); ▷ It saves a transformation that does not yield a message
    else
        continue; ▷ For simplicity, it does not focus on disabling preconditions related to messages not reported in Step 2
$result = \emptyset$;
Step 5. foreach $t \in transformations$ **do**
    if SAFEREFACTOR(*t.input*, *t.output*, *timeLimit*).hasSameBehavior() **then**
        result.add(*t*); ▷ It saves a behavior preserving transformation applied by R'
Output: $\langle R', result \rangle$; ▷ It returns R', and all transformations that yield a set of overly strong preconditions in *R*

If this precondition is not satisfied, Eclipse creates the appropriate message (Line 8) and adds an error message (Line 9).

To disable a precondition, we have to add an *If* statement before each place that enables reporting a message related to the precondition (Step 3.4.2). Each *If* statement contains a boolean field associated to a message. Our goal is to bypass checking the precondition, and we do so by avoiding reporting messages to the user. In Eclipse, we must prevent adding a warning or error message in a *RefactoringStatus* object. Listing 6 illustrates the code to allow disabling the precondition of our example. It includes an *If* statement (Line 9). We create a class *ConditionsPullUpMethod* (Step 3.1) that only declares boolean variables (Step 3.2.2). In our example, *ConditionsPullUpMethod.cond1* is related to *PullUpRefactoring_method_not_accessible* message.

In Step 4, we enable all conditions (Step 4.1), set *ConditionsPullUpMethod.cond1* to *false* (Step 4.2), and try to apply the transformation again in the program presented in Listing 3 by using the modified refactoring implementation (Step 4.3). By disabling Line 9, the modified refactoring implementation does not report the message and continues its execution. In our example, it yields a program presented in Listing 4. Step 5 analyzes whether the output program (Listing 4) compiles and preserves behavior according to SAFEREFACTOR (Section 3.2). Since SAFEREFACTOR classifies this transformation as a refactoring, our technique classifies this precondition as overly strong. Eclipse developers confirmed this bug.[2] The technique returns the modified refactoring implementation, and all transformations that yield a set of overly strong preconditions in the original refactoring implementation. Developers need to reason about adjusting the precondition to avoid preventing correct transformations, such as this transformation.

4.3 Identifying Messages

This section explains how to identify the different kinds of messages reported by the refactoring implementation when it rejects transformations. For each JDOLLY generated program (Step 1), we attempt to apply the transformation using the refactoring implementation under test (Step 2). It may apply some transformations and reject others. We collect all

2. https://bugs.eclipse.org/bugs/show_bug.cgi?id=399788

DP Change 1. ⟨Avoid throwing an exception in JRRT⟩

```
ds
class C extends D {
  cs
  T m(...) {
    Stmts
    throw new RefExc(msg);
    Stmts'
  }
}
```

→

```
ds
class C extends D {
  cs
  T m(...) {
    Stmts
    if (Conditions.condN) {
      throw new RefExc(msg);
    }
    Stmts'
  }
}
```

messages reported by the refactoring implementation when it rejects transformations.

Next, we classify the different kinds of messages, among the set of reported messages. We ignore the parts of the message that contain keywords, and names of packages, classes, methods, and fields. For example, the considered message in Step 2 reported by Eclipse in Listing 3 is: *Method referenced in one of the moved elements is not accessible from type*. To automate this process, we implement a message classifier using an approach similar to the one proposed by Jagannath et al. [28]. In each refactoring engine, we have to investigate how a message is represented (Step 3.2.1). In Eclipse, the *refactoring.properties* file defines variables representing reported messages. JRRT creates messages in *RefactoringException* objects.

4.4 Disabling Refactoring Preconditions

In this step, we change the refactoring implementation code to allow disabling the execution of refactoring preconditions that prevent the engine from applying the refactorings. We use the identified messages in Step 3.2.1. For each refactoring engine, we identify how to avoid reporting messages to the user (Step 3.3), and all places (Step 3.4.1) that can prevent reporting a message (*msg*). In Eclipse, we have to avoid adding errors or warnings in *RefactoringStatus* objects containing *msg*. In JRRT, we have to avoid throwing a *RefactoringException* containing *msg*. The goal is to change the refactoring implementation to avoid reporting messages by including *If* statements (Step 3.4.2). We formalize these transformations with DP Changes.

4.4.1 DP Changes

A DP change specifies a Java program template before and after the code modification. The left-hand side template specifies the method body in a Java program. When the code fragment that we want to disable the precondition matches the left-hand side template, we change the refactoring implementation code by following the right-hand side template. Each DP change adds an *If* statement in the refactoring implementation code and is applied within a method body.

DP changes contain Java constructs and meta-variables. The DP changes of JRRT and Eclipse have the following common meta-variables: *C* specifies a class (it extends a *D* class); *ds* specifies a set of class and interface declarations of the refactoring implementation code; *m* specifies a method name; *T* specifies a type name; *Stmts* specifies a sequence of statements; *msg* specifies a message reported to the user by the refactoring implementation when it rejects a transformation; and *cs* specifies a set of class structures, such as methods, attributes, inner classes, and static blocks. *C* contains *cs* and declares *m*, which contains *Stmts* and calls a method by passing *msg* as a parameter. Meta-variables equal on both sides of a DP change means that the transformation does not modify them.

We specified one DP change for JRRT and two for Eclipse in our evaluation presented in Section 5. The left-hand side template of a DP change specifies a *C* class in the refactoring implementation code, which can extend a *D* class, and other classes and interfaces declarations of the refactoring implementation code (*ds*). *C* may contain a set of class structures, such as methods, attributes, inner classes, and static blocks (*cs*). It also declares the *m* method, which has a return type *T* and a sequence of statements (*Stmts*).

For each refactoring implementation, we create a class (*Conditions*) that declares public static boolean fields (*cond1*, *cond2*, ..., *condN*). For each message (msg_i) that a refactoring implementation yields in Step 2, we create a boolean variable $cond_i$ associated to it (Step 3.2.2). $cond_i$ will be used in all *If* statements added for a specific msg_i. *Conditions* declares a public static void method *enableConditions* that sets all boolean variables declared in the class to true.

DP changes help developers to systematically modify the refactoring implementation to disable refactoring preconditions. If there is no DP change to match, developers analyze the minimum changes necessary to allow disabling the code fragments that prevent the refactoring precondition to propose a new kind of DP change. If this new kind of DP change cannot be reused to allow disabling other preconditions, we leave it as a specific case.

DP Changes in JRRT. JRRT always throws a *RefactoringException* (*RefExc*) that contains a *msg* to terminate the execution and report the error message to the user. To avoid reporting *msg*, we propose DP Change 1. We include an *If* statement immediately before throwing a *RefactoringException* that receives as a parameter the message related to the precondition that we wish to disable.

DP Changes in Eclipse. Eclipse implements a class (*RefactoringStatus*) that stores the outcome of the preconditions checking operation. It contains methods, such as *addError*, *addEntry*, *addWarning*, *createStatus*, *createFatalErrorStatus*, *createErrorStatus*, and *createWarningStatus*. Those methods receive a message and other arguments, describing a specific problem detected during the precondition checking. The methods started with *create* return a *RefactoringStatus* object. The messages are stored in the *refactoring.properties* file.

DP Change 2. ⟨Avoid adding a refactoring status in Eclipse⟩

```
ds
class C extends D {
   cs
   T m(...) {
      Stmts
      status.s(..., msg, ...);
      Stmts'
   }
}
```

→

```
ds
class C extends D {
   cs
   T m(...) {
      Stmts
      if (Conditions.condN) {
         status.s(..., msg, ...);
      }
      Stmts'
   }
}
```

A field from the *RefactoringCoreMessages* class represents them. They can be directly accessed by a field call or through a variable, parameter of the method, or the return of a method call. The refactoring implementations of Eclipse check the status of a transformation, in a *RefactoringStatus* object, after evaluating the preconditions. If it contains some warning or error messages, Eclipse rejects the transformation and reports the messages to the user. We propose the Eclipse DP changes by analyzing the smallest code fragment, which we need to disable for avoiding the engine to add a new error or warning status in a *RefactoringStatus* object.

DP Change 2 prevents Eclipse from reporting error messages. It has the following specific meta-variables: *status* specifies an object of *RefactoringStatus* type, and *s* is one of the methods of *RefactoringStatus* described in the beginning of this section. We include an *If* statement immediately before a call to a method from the *RefactoringStatus* class that receives as a parameter the message related to the precondition that we want to disable.

4.4.2 Applying DP Changes

The proposed DP changes are general specifications that can be used as guidelines to modify the refactoring implementation to allow disabling a precondition. To apply a DP change, the refactoring implementation must match the left-hand side template of it. We find all places in the refactoring implementation code that matches the left-hand side template of a DP change, by searching for the message (msg_i) related to the refactoring precondition that we want to disable (Step 3.4.1). The code structure must match all Java constructs and meta-variables specified in the left-hand side template.

For example, we use DP Change 2 to disable the precondition illustrated in Listing 5. *T* matches *RefactoringStatus*, *m* matches *checkAccessedMethods*, *Stmts* matches the sequence of statements from the beginning of the method until Line 6; *status* matches the *st* variable of type *RefactoringStatus*; *s* matches the *addError* method; *msg* matches *m*; and *Stmts'* matches the sequence of statements from Line 8 until the end of the method. We use the right-hand side of this same DP change to modify the code to allow disabling the precondition (Step 3.4.2). Listing 6 illustrates the modified program. In our example, *checkAccessedMethods* method in Eclipse is the only method that adds *PullUpRefactoring_method_not_accessible* message to a *RefactoringStatus* object. So, we do not need to apply more DP changes. We automate the DP changes proposed for Eclipse and JRRT using aspect-oriented programming [29] (see Appendix).

5 Evaluation

We evaluate our technique using 10 refactoring implementations of Eclipse and 10 refactoring implementations of JRRT.[3] First, we present the research questions (Section 5.1) and planning (Section 5.2). Next, Sections 5.3 and 5.4 present and discuss the results, respectively. Finally, Section 5.5 describes some threats to validity, and Section 5.6 summarizes the main findings.

5.1 Research Questions

Our experiment has two goals. The first goal is to evaluate the DP technique to detect overly strong preconditions with respect to its ability to detect overly strong preconditions and its performance from the point of view of refactoring engine developers in the context of refactoring implementations from Eclipse and JRRT. For this goal, we address the following research questions:

- *Q1* Can the DP technique detect bugs related to overly strong preconditions in the refactoring implementations?
  We measure the number of bugs related to overly strong preconditions for each refactoring implementation.
- *Q2* What is the average time to find the first failure using the DP technique?
  We measure the time to find the first failure in all refactoring implementations.
- *Q3* What is the rate of overly strong preconditions detected by the DP technique among the set of assessed preconditions?
  We measure the rate of preconditions that are overly strong in each refactoring implementation.

The second goal is to evaluate two techniques (DP and DT [13]) to detect overly strong preconditions in refactoring implementations for the purpose of comparing them with respect to detecting overly strong preconditions from the point of view of refactoring engine developers in the context of refactoring implementations of Eclipse and JRRT. We address the following research question for this goal:

- *Q4* Do DP and DT techniques detect the same bugs?
  We measure the bugs detected by both techniques: DP and DT techniques.

3. All tools and experimental data are available online at: http://www.dsc.ufcg.edu.br/~spg/DPtechnique.html

TABLE 1
Summary of the DP Technique Evaluation in the JRRT and Eclipse Refactoring Implementations; Refactoring = Kind of Refactoring; Skip = Skip value Used by JDOLLY to Reduce the Number of Generated Programs; GP = Number of Generated Programs by JDOLLY; CP = Rate of Compilable Programs (%); LOC cov. = Rate of Lines of Code Coverage for the Set of Generated Programs (%); N. ass. prec. = Number of Assessed Refactoring Preconditions in Our Study; OSC = Number of Detected Overly Strong Preconditions in the Refactoring Implementations; Time (h) = Total Time to Evaluate the Refactoring Implementations in Hours; TTFF (min) = Time to Find the First Failure in Minutes; "*na*" = not assessed

Refactoring	Skip	GP	CP (%)	LOC Cov. (%)		N. ass. prec.		OSC		Time (h)		TTFF (min)	
				JRRT	Eclip.	JRRT	Eclip.	JRRT	Eclip.	JRRT	Eclip.	JRRT	Eclip.
Move Method	no	22,905	69	17.7	9.4	6	3	6	3	0.01	4.50	0.30	10.2
	10	2,290				6	3	6	3	0.01	0.40	0.08	0.60
	25	916				4	3	4	3	0.02	0.19	0.06	1.21
Pull Up Method	no	8,937	72	12.9	7.1	4	2	2	2	0.12	0.75	0.16	4.92
	10	893				4	2	2	2	0.04	0.10	0.46	5.61
	25	357				4	2	2	2	0.01	0.03	0.18	0.90
Push Down Field	no	11,936	79.1	11.9	8.4	3	2	1	0	0.10	3.11	0.18	*na*
	10	1,193				3	2	1	0	0.01	0.30	0.11	*na*
	25	477				3	2	0	0	0.01	0.12	*na*	*na*
Rename Method	no	11,264	79.5	11.9	8.0	3	3	1	3	0.12	0.06	0.06	0.05
	10	1,126				3	3	1	3	0.01	0.01	0.08	0.06
	25	450				3	3	1	3	0.01	0.06	0.08	0.20
Push Down Method	no	20,544	78.5	14.4	8.7	3	3	3	1	0.15	7.15	2.53	49.43
	10	2,054				3	3	3	1	0.16	1.09	0.46	4.75
	25	821				3	3	3	1	0.01	0.39	0.20	1.91
Pull Up Field	no	10,928	79.7	12.3	8.1	1	1	0	1	0.08	0.01	*na*	0.11
	10	1,092				1	1	0	1	0.01	0.01	*na*	0.11
	25	437				1	1	0	1	0.01	0.01	*na*	0.08
Add Parameter	no	30,186	63	12.4	15.5	4	3	2	2	0.31	11.48	0.36	91.26
	10	3,018				4	3	2	1	0.04	1.61	0.08	9.48
	25	1,207				4	3	2	1	0.02	0.65	0.08	3.66
Encapsulate Field	no	2,000	92.8	11.8	3.4	0	1	*na*	1	*na*	0.01	*na*	0.33
	10	200				0	1	*na*	1	*na*	0.01	*na*	0.28
	25	80				0	1	*na*	1	*na*	0.01	*na*	0.53
Rename Field	no	19,424	79.2	12.0	3.8	0	3	*na*	0	*na*	5.98	*na*	*na*
	10	1,942				0	3	*na*	0	*na*	0.44	*na*	*na*
	25	776				0	3	*na*	0	*na*	0.18	*na*	*na*
Rename Type	no	15,916	65.5	11.3	4.5	0	4	*na*	2	*na*	2.67	*na*	0.11
	10	1,591				0	4	*na*	2	*na*	0.26	*na*	0.05
	25	636				0	4	*na*	2	*na*	0.11	*na*	0.08
Total/Average	no	154,040	72.8	12.8	7.6	24	25	15	15	0.89	35.72	0.58	17.41
	10	15,399				24	25	15	14	0.28	4.22	0.21	2.35
	25	6,157				22	25	12	14	0.09	1.75	0.12	1.01

5.2 Planning

In this section, we describe the subjects used in our study and the experiment setup.

5.2.1 Subject Selection

We tested 10 refactoring implementations of Eclipse JDT 4.5 and 10 of JRRT (02/03/2013) [6]. Eclipse is a widely used engine and JRRT was proposed to improve the correctness and applicability of refactorings by using formal techniques [6]. Among the evaluated refactorings (Column Refactoring of Table 1), we evaluated popular refactorings, such as the Rename Method [27], [30] and refactorings that are predominantly performed with automated tool support, such as Encapsulate Field and Rename Class [31].

5.2.2 Setup

We ran the experiment on two computers with 3.0 GHz Core i5 with 8 GB RAM running Ubuntu 12.04. We used SAFEREFACTOR [21] 2.0 with a time limit of 0.5 second to generate tests. This time limit is enough to test transformations applied to small programs [21]. We used EclEmma 2.3.3 to measure the code coverage. We executed the experiment using JDOLLY 1.0 with Alloy Analyzer 4 and SAT4J solver 2.0.5 to generate the programs with no skip and skips of 10 and 25. To allow disabling the preconditions in our experiment, we manually modified the Eclipse and JRRT code. We used the proposed DP changes to systematically modify most of refactoring preconditions (58 places).

We used the same Alloy specifications defined before [12], [14] as input parameters of JDOLLY to generate the programs. JDOLLY generates programs with at most two packages, three classes, two fields and three methods to test the refactoring implementations of Eclipse and JRRT. The specification defines some main constraints for guiding JDOLLY to generate programs with certain characteristics needed to apply the refactoring. To test the Rename Class, Method, and Field refactorings, we specified some additional constraints: the programs must declare at least one Class, Method, and Field, respectively. To test the Push Down

Method/Field refactorings, the programs must declare a method/field in a superclass. To test the Pull Up Method/Field refactorings, the programs must declare a method/field in a subclass. To test Encapsulate Field and Add Parameter refactorings, the programs must declare at least one public field and method, respectively. Finally, to test the Move Method refactoring, the programs must declare at least two classes. One of the classes must declare a method and a field of the same type of the other class.

The Alloy specification used by JDOLLY also defines some constraints to reduce the number of generated programs, such as some overloading or overriding methods and some primitive fields. Furthermore, we specified that the programs must have at least one public method for enabling SAFEREFACTOR to generate tests for evaluating the transformations. We automated the refactoring applications of Eclipse and JRRT by investigating their refactoring test suites to learn how to apply the refactorings using their source code. We implemented in the same way, replacing the input programs with the JDOLLY generated programs. We used the same setup for both evaluated techniques (DP and DT).

5.3 Summary of the Results

Concerning the JRRT evaluation, we identified 24 refactoring preconditions and found 15 (62 percent) overly strong preconditions in its refactoring implementations. The DP technique did not detect 3 bugs using a skip of 25 in the Move Method and Push Down Field refactorings of JRRT. It took 0.89 h to evaluate all JRRT refactoring implementations without skip to generate programs. Using skips of 10 and 25, the technique took 0.28 and 0.09 h, respectively. In average, the technique needed a minute to find the first failure.

Concerning the Eclipse evaluation, we identified 25 refactoring preconditions and found 15 (60 percent) different kinds of bugs in its refactoring implementations. The DP technique did not detect 1 bug using skips of 10 and 25 in the Add Parameter refactoring of Eclipse. It took 35.72 h to evaluate all Eclipse refactoring implementations without skip to generate programs. Using skips of 10 and 25, the technique took 4.22 and 1.75 h, respectively. It took on average 17.41 min to find the first failure using no skip. Using skips of 10 and 25, the technique took on average 2.35 and 1.01 min to find the first failure, respectively. SAFEREFACTOR generated an average of 45 test cases (ranging from 1 to 179) to evaluate transformations applied by JRRT and 59 (ranging from 1 to 268) to evaluate transformations applied by Eclipse.

JDOLLY generated 154,040 programs to evaluate all refactorings without skip. Considering all generated programs, the rate of compilable programs was 72.8 percent. For future work, we intend to specify more well-formedness rules to reduce the rate of uncompilable programs and the cost of analysis. Still, the uncompilable programs do not affect our results concerning bug detection.

Given the set of generated input programs for each refactoring implementation, we measured LOC coverage for both JRRT (*AST* package) and Eclipse (*org.eclipse.jdt.internal.corext.refactoring*) implementations. LOC coverage for Eclipse is 7.6 percent, while for JRRT is 12.8 percent. The coverage rates are low because the Eclipse *org.eclipse.jdt.internal.corext.refactoring* and JRRT *AST* packages contain all refactorings implemented in these engines. Table 1 summarizes the evaluation results of JRRT and Eclipse refactoring implementations.

We also compared the proposed technique with the DT technique. The DP technique found nine new bugs that the DT technique could find in the refactoring implementations of JRRT, and two new bugs in the refactoring implementations of Eclipse. It did not detect five bugs that the DT technique detected in the refactoring implementations of Eclipse. Concerning the use of skips, the DP technique did not detect four bugs using a skip of 25 and one bug using a skip of 10. The DT technique missed no bug using skips of 10 and 25. We calculated the number of missed bugs using skips by comparing with the number of detected bugs using no skip. We need to execute the same technique without skip to find the missed bugs. Table 2 summarizes the evaluation results of the comparison between DP and DT techniques.

TABLE 2
Summary of the Comparison Between DP and DT Techniques Using Input Programs Generated by JDOLLY; Refactoring = Kind of Refactoring; Skip = Skip Value Used by JDOLLY to Reduce the Number of Generated Programs; DP = DP Technique; DT = DT Technique; Overly Strong Preconditions = Number of Detected Overly Strong Preconditions in the Refactoring Implementations; "*na*" = not Assessed

Refactoring	Overly Strong Preconditions				
	Skip	Eclipse		JRRT	
		DP	DT	DP	DT
Move Method	no	3	3	6	1
	10	3	3	6	1
	25	3	3	4	1
Pull Up Method	no	2	2	2	2
	10	2	2	2	2
	25	2	2	2	2
Push Down Field	no	0	0	1	0
	10	0	0	1	0
	25	0	0	0	0
Rename Method	no	3	3	1	0
	10	3	3	1	0
	25	3	3	1	0
Push Down Method	no	1	3	3	1
	10	1	3	3	1
	25	1	3	3	1
Pull Up Field	no	1	0	0	0
	10	1	0	0	0
	25	1	0	0	0
Add Parameter	no	2	1	2	2
	10	1	1	2	2
	25	1	1	2	2
Encapsulate Field	no	1	1	*na*	*na*
	10	1	1	*na*	*na*
	25	1	1	*na*	*na*
Rename Field	no	0	3	*na*	*na*
	10	0	3	*na*	*na*
	25	0	3	*na*	*na*
Rename Type	no	2	2	*na*	*na*
	10	2	2	*na*	*na*
	25	2	2	*na*	*na*
Total	no	15	18	15	6
	10	14	18	15	6
	25	14	18	12	6

5.4 Discussion

In this section, we discuss the results of our evaluation.

5.4.1 Assessed Preconditions

We identified 24 preconditions from JRRT and 25 preconditions from Eclipse, based on reported messages when they reject transformations. We relate each reported message to one precondition for each refactoring implementation. Table 3 illustrates some of the Eclipse and JRRT assessed preconditions considered in our evaluation. For each one, we explain what the precondition checks (fourth column), the message reported by the refactoring engine when the precondition is unsatisfied (fifth column), and if our technique classified it as overly strong in this study (sixth column).

For example, Precondition 1 prevents JRRT from moving a method when it overrides (or is overwritten by) different methods before and after the transformation. Without this precondition, the transformation may change the program behavior. However, our technique classified this precondition as overly strong because it also prevents moving an overwritten method when there is no other method in the program calling it. Precondition 4 avoids the same problem in the Add Parameter refactoring of JRRT, since changing a method signature may change method overriding. Our technique also classified it as overly strong for this refactoring. Preconditions 2 and 3 prevent JRRT to push down or pull up a field to a class that already contains a field with the same name, respectively. Both preconditions avoid introducing compilation errors in the resulting program, since a class cannot declare two fields with the same name. According to this evaluation, they are not overly strong.

Precondition 7 prevents Eclipse from moving a method to a class that already declares a method with the same name. It avoids introducing compilation errors and behavioral changes in the resulting program. However, our technique found that this precondition is overly strong because the methods can have different types of parameters. Preconditions 8 and 9 prevent Eclipse from renaming a method when there is another method in the same package or type in the renamed method hierarchy, with the same name but different parameter types and with the same signature, respectively. They also avoid introducing compilation errors and behavioral changes in the resulting program. For example, it can introduce compilation errors related to the reduction of inherited method visibility or can introduce behavioral changes when the renamed method changes the binding of a method call. Our technique classified both preconditions as overly strong because in some cases the renamed method is not public and there is no other method in the program calling it.

Precondition 10 prevents Eclipse to push down a field when there is a method referencing it. It avoids introducing compilation errors when the field does not hide other field, and behavioral changes, otherwise. Precondition 12 prevents Eclipse from adding a parameter in a method when there is another method in the same class with the same signature. It avoids introducing compilation errors in the resulting program, since a class cannot declare two methods with the same signature. Both preconditions (10 and 12) are not overly strong in this evaluation.

This set of assessed preconditions is a subset of the existing preconditions. The evaluated refactoring implementations may have more overly strong preconditions. Developers may consider programs with different program constructs to detect them. In some cases, preconditions of different refactoring implementations, such as Preconditions 1 and 4, and Preconditions 2 and 3, are implemented by the same code fragments. The refactoring engine reports the same message when the preconditions are unsatisfied. Even so, we consider them as different preconditions because we analyze each refactoring implementation separately. To test all preconditions of a refactoring implementation, we need to select a set of input programs that leads the refactoring implementation to report all messages when it rejects transformations.

5.4.2 Disabling the Assessed Preconditions

We identified some patterns followed by the developers to reject a transformation due to an unsatisfied precondition. In these cases, we propose DP changes, and we can use them to disable preconditions by preventing to report messages to the user. However, in some specific cases, we did not find a pattern for the right-hand side template to disable a precondition. So, we could not propose DP changes to modify the refactoring implementation to disable a precondition. We need to reason about the refactoring implementation code to identify the specific changes necessary to disable the precondition. We call those kinds of changes as specific cases. We applied 58 DP changes (22 in JRRT and 36 in Eclipse) and 25 specific cases to allow disabling the execution of the Eclipse and JRRT assessed preconditions in this study. In some places of the code we may apply more than one transformation, since some preconditions of different refactoring implementations report the same message. Developers can restructure the refactoring engine code to enable using the DP changes to disable the code fragments of preconditions that we classified as specific cases.

For each precondition, we may apply more than one DP change or the same DP change more than once because each message may appear in a number of places in the code. The number of messages may impact the performance of Steps 3 and 4 of our technique because we need, for each message, to identify the precondition that raises it and change the refactoring engine code to allow disabling the precondition.

JRRT throws an exception with the message in all cases and aborts its execution. Eclipse opens a dialog to report the message describing the problem to the user. The user can cancel the refactoring application or continue in some cases. In our study, 36, 46, and 14 percent of the changes we made in the Eclipse code prevent warning, error, and fatal error problems, respectively. In only one change (6 percent) we cannot assert by static analysis whether it prevents a warning or error message.

Listings 7 and 8 illustrate the original and modified JRRT code to allow disabling the execution of a Move Method refactoring precondition (Precondition 1 of Table 3), respectively. We can use the DP Change 1 to disable this precondition. The transformation was applied to the unlockOverriding method from the AST.MethodDecl class. This precondition evaluates if a set of overridden methods in the original program (old_overridden.equals) is equal to the set of overridden methods in the program after the transformation (overriddenMethods). JRRT rejects the transformation by throwing a RefactoringException when the precondition

TABLE 3
Subset of Eclipse and JRRT Assessed Preconditions

Id	Eng.	Refactoring	Precondition	Message	OS (DP)
1	JRRT	Move Method	It checks whether the method after the transformation still overrides precisely the same methods as before the transformation	overriding has changed	yes
2	JRRT	Push Down Field	It checks whether the subclass, where the field will be pushed down, already declares a field with the same name	field of the same name exists	no
3	JRRT	Pull Up Field	It checks whether the target class already declares a field with the same name of the moved field	field of the same name exists	no
4	JRRT	Add Parameter	It checks whether the method after the transformation still overrides precisely the same methods as before the transformation	overriding has changed	yes
5	JRRT	Pull Up Method	It checks whether the transformation violates a type constraint	type constraint violated	yes
6	JRRT	Push Down Method	It checks whether the transformation can preserve method name bindings. It is unsatisfied when a method name cannot be accessed even with qualifiers	cannot access method	yes
7	Eclipse	Move Method	It checks whether the class, where the method will be moved, already declares a method with the same name	a method with name <method> already exists in the target type	yes
8	Eclipse	Rename Method	It checks whether there is other method, in the same package or type in the renamed method hierarchy, with the same name and number of parameters but different parameter type names of the new method's signature	<package> or a type in its hierarchy defines a method, <method> with the same number of parameters, but different, parameter type names	yes
9	Eclipse	Rename Method	It checks whether there is other method, in the same package or type in the renamed method hierarchy, with the same new method's signature	<package> or a type in its hierarchy defines a method, <method> with the same number of parameters and the same, parameter type names	yes
10	Eclipse	Push Down Field	It checks whether some method refers the pushed down field	pushed down member <method> is referenced by <method>	no
11	Eclipse	Pull Up Method	It checks whether the methods called by the pulled up method are accessible from the class where the method will be pulled up	method <method> referenced in one of the moved elements is not accessible from type <type>	yes
12	Eclipse	Add Parameter	It checks whether the class, which contains the method to be changed, already declares a method with the same signature of the new method signature	duplicate method in type <type>	no

Eng. = Refactoring engine that contains the precondition; Refactoring = Kind of refactoring; Precondition = precondition checking; Message = reported message when the precondition is unsatisfied; OS (DP) = yes if the DP technique found this precondition as overly strong in this experiment, otherwise no.

is not satisfied. Our technique classified this precondition as overly strong. We reported this overly strong precondition to the JRRT developers and they classified it as bug due to imprecise analysis. The bug has not been fixed yet.

Listing 7. Original code fragment of the Move Method refactoring implementation of JRRT.

```
public void unlockOverriding() {
    . . .
    if(!old_overridden.equals(overriddenMethods()))
        throw new RefactoringException
        (''overriding has changed'');
    . . .
}
```

Listing 8. Code fragment after we change the JRRT code by using the DP Change 1.

```
public void unlockOverriding() {
    . . .
    if(!old_overridden.equals(overriddenMethods()))
        if (ConditionsMoveMethod.cond1) {
            throw new RefactoringException
            (''overriding has changed'');
        }
    . . .
}
```

Among the 25 specific cases, there are six different kinds of transformations. For example, in some cases we included in the *then* clause of the *If* statement, introduced to disable

the precondition, some statements to avoid crashing the refactoring engine. We also had to add in some cases a *return null* command after the *If* statement, which disables the precondition, to avoid compilation errors in the refactoring implementation code.

In some cases, when we disabled a precondition, the refactoring implementation reported another message and we needed to disable another precondition that raises another reported message, and so on. For seven refactoring implementations, we disabled a number of refactoring preconditions at the same time: Move Method, Push Down Method, and Add Parameter of JRRT and Move Method, Pull Up Method, Rename Method, and Rename Type of Eclipse. In the Move Method refactoring of JRRT we needed to disable up to four preconditions at the same time to find a bug (among the set of six assessed preconditions). In the other refactoring implementations, we disabled up to two preconditions.

Regarding Step 3, the first author took around one day of work to understand how Eclipse and JRRT check their refactoring preconditions, raise messages, and reject transformations. After that, she took some minutes to manually change the refactoring engine code to allow the disabling of each refactoring precondition using the proposed templates. She also took some minutes where specific cases are concerned.

5.4.3 Bugs Detected by DP Technique

Among the 49 assessed preconditions, we identified 30 overly strong preconditions (61 percent) in the Eclipse and JRRT refactoring implementations using the DP technique. For example, Listing 3 illustrates a Pull Up Method refactoring rejected by Eclipse due to overly strong precondition (Precondition 11 of Table 3). We manually analyzed all bugs detected in our evaluation, and did not find the same program yielding an overly strong precondition in two different refactoring implementations. Most of the Eclipse and JRRT overly strong preconditions found by our technique are related to method accessibilities and name conflicts. Others are related to changes in overriding methods, type constraints violations, shadow declarations, unimplemented features, transformation issues, and changes in method invocations.

JRRT applied transformations to all of the programs generated by JDolly in three refactoring implementations: Encapsulate Field, Rename Field, and Rename Type. We did not detect overly strong preconditions in those refactoring implementations. Different from JRRT, Eclipse rejected some transformations in these refactoring implementations and we found some overly strong preconditions. In both refactoring engines we identified 15 overly strong preconditions using the DP technique.

We reported all detected bugs to the Eclipse developers. So far, they confirmed 47 percent of them (seven bugs), and did not answer for 27 percent (four bugs). The remaining four bugs were considered duplicates (13 percent) or invalid (13 percent). We investigated the duplicated bugs (IDs 434881 and 399183) and reopened them because we think they are not duplicated, since the reported messages are different. We also reopened a bug marked as invalid by the developers (ID 399181). Developers argued that the refactoring implementation does not show a message in this case. However, we tested the same bug in Eclipse JDT 4.6 and it still reports the message. So far, they have not responded. The other invalid bug was in the Pull Up Field refactoring (ID 462994). Developers marked it as invalid because the transformation changes the value of a field not called by any method in the original program. The equivalence notion adopted by SafeRefactor does not consider that this kind of change modifies the program behavior. SafeRefactor only evaluates the behavior of the common public impacted methods. Developers did not fix all confirmed bugs because they have very limited resources working on the refactoring module. We reported the new JRRT bugs to its developers. These bugs were not detected by our previous technique. JRRT developers believe that most of these bugs are due to imprecise analysis or unimplemented features. So far, they have not answered for two of them.

The goal of our technique is to propose a systematic way to evaluate the implemented preconditions. We do not suggest removing the overly strong preconditions found by our technique. By removing them, the refactoring implementation may apply incorrect transformations. Developers need to reason about the preconditions and choose the best strategy to slightly weaken them without making them overly weak. They can use the DP and DT techniques and our previous technique to detect overly weak preconditions [12] to reason about their preconditions.

5.4.4 Time

We computed the time for the automated steps of the DP technique. The time to evaluate the JRRT refactoring implementations was smaller than the time to evaluate Eclipse ones in all cases but two: Rename Method and Pull Up Field refactorings. In those refactoring implementations, all assessed preconditions of Eclipse are overly strong while this is not true for JRRT. The execution of the technique finishes when we find that all preconditions being tested are overly strong. The execution to evaluate Eclipse finished earlier than the JRRT ones in the Rename Method and Pull Up Field refactorings. However, the total time to evaluate all of JRRT and Eclipse refactoring implementations was 0.89 and 35.72 h, respectively.

Our previous study [14] showed that by using skips could substantially reduce the time to test refactoring implementations while missing a few bugs related to overly weak and overly strong preconditions with the DT technique. In this study, we evaluated the influence of skip in the time reduction and bug detection of the DP technique. Using skips of 10 and 25, the total time to evaluate all refactoring implementations was reduced by 87 and 94 percent, while missing 3 and 13 percent of the bugs, respectively. The total time to evaluate the Move Method refactoring of JRRT and Rename Method refactoring of Eclipse using a skip of 25 was higher than using a skip of 10. In those cases, the technique found that all preconditions under test are overly strong using a skip of 10 earlier than using a skip of 25.

Eclipse took 11.48 and 7.15 h to evaluate the Add Parameter and Push Down Method refactorings, respectively. These times were higher than the time to evaluate the other refactoring implementations. JDolly generated more programs to evaluate these refactoring implementations (30,186 for Add Parameter and 20,544 for Push Down Method) and

only some of their assessed preconditions were classified as overly strong.

The average time to find the first failure in the JRRT refactoring implementations (few seconds) was also smaller than in Eclipse (17.41 min). The average time to find the first failure in Eclipse was affected by two refactorings that took much longer than the average time to find the first failure, namely the Push Down Method and Add Parameter refactorings. Our technique found the first failure in JRRT and Eclipse after generating 255 and 2,898 programs, respectively. In the Add Parameter refactoring, our technique found the first failure in JRRT and Eclipse after generating 328 and 8,258 programs, respectively. The average time to first failure in the Eclipse refactoring implementations without considering these two higher values is 2.62 min. Using skips of 10 and 25, the average time to find the first failure in all refactoring implementations reduced by 85 and 93 percent, respectively.

Using skips, developers can run the technique and find a bug in a few seconds or minutes, fix the bug, run the technique again to find another bug, and so on. Developers can also run the technique to find a number of bugs in a few minutes or hours. Before a release, they may run the technique without skipping instances to find some missed bugs and improve confidence that the implementations are correct.

In our previous work [21], we analyzed the influence of the time limit passed to SafeRefactor to generate tests. We used different time limits, such as 0.2, 0.5, and 20 s, to evaluate transformations applied to different kinds of programs in SafeRefactor. To evaluate transformations applied to small programs, with at most four classes, methods, and fields, similar to those used in our evaluation, we used a time limit of 0.2 s, which was enough to generate tests for this context.

5.4.5 Comparison of DP and DT Techniques Using Input Programs Generated by JDolly

The techniques are complementary in terms of bug detection. The DP technique detected 11 new bugs (37 percent of the bugs) that the DT technique cannot detect in the Pull Up Field and Add Parameter refactorings of Eclipse and in the Move Method, Rename Method, Push Down Method, and Push Down Field refactorings of JRRT. The DT technique cannot detect some bugs when the other refactoring engine used in the differential testing has overly weak preconditions or also has overly strong preconditions. In the former case, the other refactoring engine applies a transformation that does not preserve the program behavior or the resulting program does not compile. In the latter case, the other refactoring engine also rejects to apply the transformation.

For example, Listing 9 presents a JDolly generated program. It contains the *A* class and its subclasses *B* and *C*. Both *A* and *B* classes contain the *f* field and the *B* class declares the *test* method that calls the *B.f* field, yielding 1. If we attempt to use JRRT to apply the Push Down Field refactoring from moving *A.f* to class *C*, it rejects this transformation due to an overly strong precondition. By disabling the precondition that prevents the refactoring application, we can apply the transformation without changing the program behavior. Listing 10 illustrates the resulting program. The *B.test* method yields 1 before and after the refactoring. We only detected this overly strong precondition using the DP technique. The DT technique cannot detect it because Eclipse also rejects this transformation. We reported this bug to JRRT developers and they agreed that this transformation should be applied.

The DT technique detected five bugs that the DP technique cannot detect in the Eclipse Push Down Method and Rename Field refactorings. The DP technique cannot detect those bugs because when we disable the code fragments of a precondition, Eclipse applies a non-behavior preserving transformation. JRRT applies a transformation that includes a cast (two bugs in the Rename Field) or a *super* modifier (one bug in the Rename Field) in a field call to preserve the program behavior.

For example, Listing 11 presents another JDolly generated program. It contains the *B* class, and its subclass *C*. The *B* class contains the *f1* field. The *C* class contains the *f0* field and declares the *test* method that calls *f1* yielding 0. By using Eclipse to rename field *C.f0* to *f1*, it rejects this transformation due to an overly strong precondition. JRRT applies this transformation without changing the program behavior. Listing 12 illustrates a resulting program applied by JRRT. Method *C.test* yields 0 before and after the refactoring. We only detected this overly strong precondition using the DT technique. The DP technique cannot detect it because when we disable the precondition, Eclipse applies a non-behavior preserving transformation. It does not include a cast of the *B* class in the field call inside the *test* method. Without this cast, *test* calls *C.f1* instead of *B.f1* yielding 1.

Listing 9. Pushing down field A.f to class C is rejected by JRRT due to overly strong preconditions. Bug detected by DP technique and not detected by DT technique.

```
public class A {
    private int f = 0;
}
public class B extends A {
    protected int f = 1;
    public long test(){
        return f;
    }
}
public class C extends A {}
```

Listing 10. A possible correct resulting program applied by JRRT.

```
public class A {}
public class B extends A {
    protected int f = 1;
    public long test(){
        return f;
    }
}
public class C extends A {
    private int f = 0;
}
```

Listing 11. Renaming C.f0 to f1 is rejected by Eclipse JDT 4.5 due to overly strong preconditions. Bug detected by DT technique and not detected by DP technique.

```
public class B {
    protected int f1 = 0;
}
public class C extends B {
    private int f0 = 1;
    public long test() {
        return this.f1;
    }
}
```

Listing 12. A possible correct resulting program applied by JRRT.

```
public class B {
    protected int f1 = 0;
}
public class C extends B {
    private int f1 = 1;
    public long test() {
        return ((B) this).f1;
    }
}
```

An advantage of the DP technique is that it does not need another refactoring engine. Although in DP we need to manually identify the preconditions from the set of reported messages, we propose a systematic strategy to perform this activity (see Algorithm 1). Table 4 summarizes the number of added lines of code and number of modified methods in all refactoring implementations of JRRT and Eclipse in Step 3.4.2.

An advantage of the DT technique is that it can show useful transformations performed by another refactoring engine (see example in Listing 12), which can help developers to identify and fix the overly strong preconditions. However, it needs at least two refactoring engines. When possible, we suggest that developers can run the DP technique and after fixing the detected bugs, then they run the DT technique to find more bugs.

The DT technique took on average 17 and 66.2 h to test the refactoring implementations of JRRT and Eclipse, respectively. The DP technique took on average 0.89 and 35.7 h to test the same refactoring implementations of JRRT and Eclipse. DT technique takes longer to apply and analyze the transformations since it uses two refactoring engines.

5.4.6 *Comparison of DP and DT Techniques Using Input Programs of Eclipse and JRRT Refactoring Test Suites*

We evaluated the DP technique by replacing the JDOLLY generated programs with input programs used by developers in the Eclipse and JRRT test suites. The goal was to analyze if our technique can find overly strong preconditions using other input programs in Eclipse and JRRT refactoring implementations already evaluated in the previous study (see Section 5.2.1). We only selected input programs used in the test cases where the refactoring engine rejects the transformation.

TABLE 4
Summary of Lines of Code Added and Number of Modified Methods in Step 3.4.2 in Eclipse and JRRT Refactoring Implementations

Refactoring	Added LOC		Modified Methods	
	JRRT	Eclipse	JRRT	Eclipse
Move Method	11	12	6	10
Pull Up Method	13	11	6	9
Push Down Field	6	2	4	2
Rename Method	8	3	3	2
Push Down Method	8	12	3	10
Pull Up Field	1	1	1	1
Add Parameter	9	5	4	4
Encapsulate Field	-	2	-	2
Rename Field	-	3	-	3
Rename Type	-	5	-	4

We identified 272 input programs to evaluate the refactoring implementations. The engines reported a total of 71 messages when we attempted to apply the transformations in the evaluated Eclipse and JRRT refactoring implementations. We related the messages to 71 refactoring preconditions.

The DP technique detected 18 overly strong preconditions not detected by the Eclipse and JRRT developers. The DP technique detected at least one overly strong precondition in 70 and 20 percent of the evaluated Eclipse and JRRT refactoring implementations, respectively. We also evaluated the same refactoring implementations using the DT technique and the same input programs. The DT technique detected 15 overly strong preconditions not detected by the Eclipse and JRRT developers. The DT technique detected at least one overly strong precondition in 60 and 10 percent of the evaluated Eclipse and JRRT refactoring implementations, respectively.

The DP technique detected eight overly strong preconditions not detected by DT technique in the Pull Up Method, Pull Up Field, Add Parameter, Rename Method, and Encapsulate Field refactorings of Eclipse, and also in the Push Down Method and Pull Up Method refactorings from JRRT. The DT technique detected five overly strong preconditions not detected by DP technique in the Move Method, Pull Up Method, Push Down Method, Rename Field, and Rename Method refactorings of Eclipse. In total, we assessed 71 preconditions and detected 23 overly strong preconditions not detected by the developers.

Our technique cannot detect 17 of the bugs using the current JDOLLY version. We need to add more Java constructs in JDOLLY to detect them. Besides the 23 detected bugs, we found 12 false-positives in this study. In these bugs, the input programs do not have public methods. SAFEREFACTOR did not identify any public method to generate tests and classified the transformations as behavior preserving. We did not have this problem with the JDOLLY generated input programs, as they have at least one public method. We reported all new bugs to the Eclipse developers but they have not answered yet. Table 5 illustrates the main results of this evaluation.

The developers did not find those overly strong preconditions because they do not seem to have enough support to reason about their preconditions and a systematic strategy to evaluate whether a precondition is overly strong.

TABLE 5
Summary of the Comparison Between DP and DT Techniques Using Input Programs of Eclipse and JRRT Refactoring Test Suite; Refactoring = Kind of Refactoring; Input Programs = Number of Selected Input Programs of the JRRT and Eclipse Refactoring Test Suite; N. of Assessed Preconditions = Number of Assessed Refactoring Preconditions in Our Study; Overly Strong Preconditions = Number of Detected Overly Strong Preconditions in the Refactoring Implementations; DP = DP Technique; DT = DT Technique

Refactoring	Input Programs		N. of assessed preconditions		Overly Strong Preconditions			
	JRRT	Eclipse	JRRT	Eclipse	JRRT		Eclipse	
					DP	DT	DP	DT
Move Method	12	27	10	8	0	0	3	4
Pull Up Method	30	25	8	9	1	0	1	1
Push Down Field	0	3	0	3	0	0	0	0
Rename Method	10	92	4	6	0	0	4	3
Push Down Method	12	5	5	3	3	2	0	1
Pull Up Field	0	3	0	2	0	0	1	0
Add Parameter	0	5	0	2	0	0	1	0
Encapsulate Field	0	2	0	2	0	0	1	0
Rename Field	2	26	1	4	0	0	2	3
Rename Type	18	0	4	0	0	0	0	0
Total	84	188	32	39	4	2	14	13

Furthermore, as they expect the refactoring engine to reject those transformations, they believe that the transformations may change the program's behavior. In fact, developers may not have an automated oracle to check behavior preservation, such as SafeRefactor.

5.4.7 Testing Other Refactoring Implementations

To test different refactoring implementations, we have to adapt at most two steps of our technique (Steps 1 and 3.4.2 in Algorithm 1). We have to analyze whether JDolly generates programs that can be refactored. Moreover, we may need to propose more DP changes to disable preconditions. In Step 1, approximately 46 percent of the Eclipse and JRRT refactoring implementations not evaluated in this experiment could be evaluated by our technique using the current version of JDolly. We just have to set up the additional constraints parameter to generate programs that exercise a specific kind of refactoring. For example, we can setup our technique to test the Move Field refactoring by reusing our Java meta-model and well-formedness rules. We need to generate programs with at least two classes (*C1* and *C2*) and one field (*F1*) in one of the classes. The following Alloy fragment specifies it.

```
one sig C1, C2 extends Class {}
one sig F1 extends Field {}
pred generate[]{
    F1 in C1.fields
}
```

We did not evaluate more refactoring implementations due to time constraints.

To evaluate the remaining refactoring implementations (54 percent), we need to extend JDolly to generate programs considering richer method bodies and different kinds of Java constructs (interfaces, inner classes, and so on). Currently, JDolly generates methods with a simple body (*return* statement). So, we cannot test refactoring implementations applied within a method body, such as Extract Method. In our previous work [14], we specified method bodies in CDolly (C program generator) using a similar approach, tested some Eclipse CDT refactoring implementations (such as Extract Function) and found some bugs. Richer method bodies in Java can be specified similarly in JDolly.

We may also need to propose new DP Changes (Step 3.4.2). We analyzed the code of some Eclipse refactoring implementations that we did not evaluate in this study, such as Extract Method, Inline Method, and Move Inner to Top. We randomly selected five messages from each analyzed refactoring implementation, which are reported by Eclipse when it rejects a transformation due to an unsatisfied precondition. For all of them, we can use the current set of DP changes for Eclipse to allow disabling the refactoring precondition related to each message. We can also reuse the DP changes of JRRT to allow disabling the refactoring preconditions of other refactoring implementations not evaluated in this study.

5.4.8 Testing Other Refactoring Engines

To test different refactoring engines such as NetBeans, we may have to adapt Steps 1, 3.2.1, 3.3 and 3.4.2 of our technique. In Step 1, we follow the same guidelines presented in Section 5.4.7. In Step 3.2.1, we have to identify how a reported message is represented in the refactoring implementation. In NetBeans, the *Bundle.properties* file defines variables that represent reported messages. In Step 3.3, we have to identify how to prevent reporting messages to the user. NetBeans refactoring implementations create an object of type *Problem*, which receives as parameter a message describing the problem when a precondition is unsatisfied. We have to prevent creating this kind of object. Finally, in Step 3.4.2, we have to propose DP Changes. Before applying a transformation, the NetBeans refactoring implementations check whether there are problems with the transformation, and report the messages to the user, when applicable. We can propose DP changes to avoid creating an object of type *Problem* by adding an *If* statement before its creation. For instance, DP Change 3 prevents creating a problem in NetBeans. We do not need to change the other steps of our technique to test the NetBeans refactoring implementations.

TABLE 6
Summary of the DP Technique Evaluation in the NetBeans Refactoring Implementations; Refactoring = Kind of Refactoring; GP = Number of Generated Programs by JDOLLY; N. ass. prec. = Number of Assessed Refactoring Preconditions in Our Study; LOC = Number of Lines of Code Added in Step 3.4.2; Meth. = Number of Modified Methods in Step 3.4.2; OSC = Number of Detected Overly Strong Preconditions in the Refactoring Implementations

Refactoring	GP	LOC	Meth.	N. ass. prec.	OSC
Pull Up Field	100	6	2	1	1
Push Down Field	100	2	1	1	0
Add Parameter	100	6	1	1	0
Rename Field	100	2	1	1	0
Pull Up Method	100	6	2	1	0
Rename Method	100	4	1	1	0

As a feasibility study, we evaluate some refactoring implementations of NetBeans 8.2. Table 6 indicates the number of generated programs by JDOLLY, number of assessed refactoring preconditions, number of detected overly strong preconditions in the refactoring implementations, and number of lines of code added and modified methods in Step 3.4.2. We found a bug by using the DP technique. NetBeans 8.2 cannot pull up *C.f* to *B.f* in the program presented in Listing 13. It reports the following message: *Member "f" already exists in the target type*. By disabling this precondition, we can apply a behavior preserving transformation.

Listing 13. Pulling up field C.f to class B is rejected by NetBeans 8.2 due to an overly strong precondition.

```
package p1;
public class A {
    protected int f = 0;
}
package p1;
public class B extends A {}
package p0;
import p1.*;
public class C extends B {
   protected int f = 1;
   public long m() {
      return this.f;
   }
}
```

5.5 Threats to Validity

In this section, we discuss some threats to the validity of our evaluation.

Construct Validity. Construct validity refers to whether the overly strong preconditions that we have detected are indeed overly strong. Eclipse developers considered two bugs reported by us as invalid. Some preconditions that we found may not be overly strong with respect to the equivalence notion adopted by the developers. Our equivalence notion is related to the behavior of the public methods with unchanged signatures. These methods can exercise methods with changed signatures. Otherwise, the methods with changed signatures may not affect the overall system behavior. So far, they have confirmed 47 percent of the reported bugs.

We have no prior knowledge over the refactoring engines' code, since we are not developers of these engines. We may not identify all code fragments related to the preconditions being tested. Developers may identify a different set of preconditions and may have better results when using our technique. Additionally, changing the refactoring implementation code may introduce problems in the refactoring implementation. It may result in applying incorrect transformations that do not follow the refactoring definition. We minimize this threat by systematizing the process of disabling preconditions. We propose DP changes where each one alters one line of code. Even the specific cases change a few lines of code.

A false-positive result of SAFEREFACTOR indicates that it did not detect a behavioral change. In our technique, a false-positive may incorrectly classify a precondition as overly strong. However, in this study, we manually analyzed each overly strong precondition before reporting it. We only found some false-positives in the experiment using the input programs of the refactoring engines' test suites. The false-positives were related to changes in the standard output or changes in non-public methods that cannot be detected by SAFEREFACTOR 2.0.

Finally, we specify in Table 3 some preconditions based on the available source code and documentation of JRRT and Eclipse [6], [7], [9], [11], [32], [33], [34], [35]. Still, some definitions may be incomplete or incorrect as we are not developers of the refactoring engines.

Internal Validity. Additional constraints in JDOLLY may hide possibly detectable overly strong preconditions. 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 specifying constraints for JDOLLY. Our current setup for testing Eclipse has memory leaks. This may have an impact on the time to test its refactoring implementations. Another threat is related to the bugs detected only by the DP technique. The DT technique did not identify some bugs because the other engine (JRRT or Eclipse) used to perform differential testing also has overly strong preconditions or overly weak preconditions that allow incorrect transformations. Using another refactoring engine to perform differential testing may identify some of those bugs.

External Validity. We can use our technique to test other refactoring implementations and other refactoring engines, as explained in Sections 5.4.7 and 5.4.8, respectively.

5.6 Answers to the Research Questions

Next, we answer our research questions.

- *Q1* Can the DP technique detect bugs related to overly strong preconditions in the refactoring implementations?
  We found a total of 30 bugs (11 new bugs) related to overly strong preconditions in 14 (70 percent) refactoring implementations. We did not find bugs in the Push Down Field and Rename Field refactorings of Eclipse, and Pull Up Field, Encapsulate Field, Rename Field, and Rename Type refactorings of JRRT.
- *Q2* What is the average time to find the first failure using the DP technique?
  The technique can find the first bug in each JRRT

DP Change 3. ⟨Avoid creating a problem in NetBeans⟩

```
ds
class C extends D {
  cs
  T m(...) {
    Stmts
    p = createProblem(..., msg, ...);
    Stmts'
  }
}
```

→

```
ds
class C extends D {
  cs
  T m(...) {
    Stmts
    if (Conditions.condN) {
      p = createProblem(..., msg, ...);
    }
    Stmts'
  }
}
```

refactoring implementation in 0.59 min on average. Finding the first bug in the Eclipse evaluation took an average of 17 min. The average time to find the first failure in Eclipse was affected by some values, such as the time to first failure in the Push Down Method and Add Parameter refactorings.

- *Q3* What is the rate of overly strong preconditions detected by the DP technique among the set of assessed preconditions?
  In the Eclipse and JRRT refactoring implementations, 60 and 62 percent of the evaluated preconditions in this study are overly strong, respectively.
- *Q4* Do DP and DT techniques detect the same bugs?
  The techniques detect 19 bugs in common. The DT technique cannot detect 11 bugs that the DP technique detected in the Add Parameter and Pull Up Field refactorings of Eclipse, and in the Move Method, Push Down Field, Rename Method, and Push Down Method refactorings of JRRT. When both refactoring engines under test have overly strong preconditions, the DT technique fails to detect bugs. The DT technique detected 5 bugs in Eclipse that the DP technique cannot detect in the Push Down Method and Rename Field refactorings of Eclipse.

6 Related Work

Opdyke [1] proposed a set of refactoring preconditions to ensure that the transformations preserve the program behavior. Later, Roberts [36] automated the basic refactorings proposed by Opdyke. However, Opdyke did not prove correctness and completeness of the proposed preconditions. Indeed, Tokuda and Batory [37] demonstrated that the preconditions proposed by Opdyke are not sufficient to guarantee behavior preservation after applying transformations. Moreover, proving refactorings with respect to a formal semantics considering all language constructs constitutes a challenge [10]. In this work, we propose a technique to test refactoring implementations with respect to overly strong preconditions by disabling some preconditions.

Soares et al. [12] presented an automated approach to detect overly weak preconditions in refactoring implementations. They use JDolly to generate Java programs and SafeRefactor [20] to evaluate whether a transformation preserves the program behavior. They found 106 bugs related to compilation errors and behavioral changes in 39 refactoring implementations. We also use JDolly to generate programs, and SafeRefactor with the change impact analysis parameter activated to evaluate behavior preservation. Our work complements their work in the sense that we can detect overly strong preconditions and they can detect overly weak preconditions. Developers can combine both techniques to produce more accurate refactoring implementations regarding the defined preconditions.

Daniel et al. [38] proposed an approach for automated testing of refactoring engines with respect to overly weak preconditions. The technique is based on ASTGen, a Java program generator, and a set of programmatic oracles. The user implements in Java how the program elements will be combined together. We use JDolly to automatically generate the test inputs, which employs the Alloy specification language [23] as the formal infrastructure for program generation. To evaluate correctness, they implemented six oracles that evaluate the output of each transformation. For instance, the oracles check for compilation errors and warning messages. They found a number of bugs in the refactoring implementations of Eclipse JDT and NetBeans. However, different from our work, they do not have oracles to detect bugs related to overly strong preconditions.

Later, Gligoric et al. [39] proposed UDITA, a Java-like language that extends ASTGen, allowing users to describe properties using any desired mix of filtering and generating style, as opposed to ASTGen, which uses a purely generating style. UDITA evolved ASTGen to be more expressive and easier to use, usually resulting in faster program generation. They found four bugs related to compilation errors in Eclipse in a few minutes. Our technique found a number of bugs related to overly strong preconditions. Besides using different technologies for searching for solutions, JDolly and UDITA specify constraints in different styles. In UDITA the constraints are specified in a Java-like language while in JDolly they are specified in Alloy, a declarative language. Moreover, some kinds of constraints are simpler to be specified in Alloy instead of implemented in Java, such as the transitive closure operator of Alloy that can emulate recursive functions.

Gligoric et al. [40] used real systems to reduce the effort for writing program generators using the same oracles of their previous work [39]. They found 141 bugs related to compilation errors in refactoring implementations for Java and C. Applying refactorings in large systems and minimizing the failure into a small program to categorize the bugs may be costly. Moreover, evaluating transformations on large real programs may be error prone and time consuming. Our focus is to detect overly strong preconditions. We can

adapt our technique to use real systems as test inputs such as them. In this case, we may need to increase SafeRefactor's time limit to generate tests as we did in our previous work to analyze large real programs [21], [26].

Vakilian and Johnson [15] presented a technique to detect usability problems in refactoring engines. It is based on refactoring alternate paths. They adapted the Critical Incident Technique (CIT) [41] in the context of refactorings. CIT aims to discover usability problems by analyzing the interactions of users that had problems with tools in general. They manually inferred usability problems from the detected critical incidents. Our technique yields to the developer the set of overly strong preconditions. They used real programs while we use JDolly to automatically generate programs as test inputs. Moreover, their technique found two usability problems related to overly strong preconditions in real programs while our technique found 30 bugs related to overly strong preconditions in the refactoring implementations of Eclipse and JRRT.

Schäfer et al. [6] presented a number of Java refactoring implementations. They translated a Java program to an enriched language that's easier to specify and check preconditions, and apply the transformation. They aim to improve correctness and applicability of the Eclipse refactoring implementations. Although we found the same number of overly strong preconditions in JRRT and Eclipse using the DP technique, we found more overly strong preconditions in Eclipse than in JRRT using the DT technique (6 in JRRT and 18 in Eclipse).

Rachatasumrit and Kim [42] studied the impact of refactoring transformations on regression tests by using the version history of Java open source projects. They found that refactoring changes are not well tested: regression test cases cover only 22 percent of impacted entities. Moreover, they found that 38 percent of affected test cases are relevant for testing the refactorings. In our previous work, we extend SafeRefactor to generate tests only for the methods impacted by the change [21]. Using the change impact analysis, SafeRefactor reduced the time to test the refactoring implementations and generated more relevant tests to evaluate the refactoring changes. To evaluate transformations applied to small programs similar to those used in our evaluation, we used a time limit of 0.2 s, which was enough to generate tests for them. In this experiment, we decided to be more conservative and chose a time limit of 0.5 s.

Steimann and Thies [43] proposed 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. The proposed approach improves the applicability of the refactoring implementations of Eclipse. The DP technique can be used to evaluate whether their refactoring implementations have overly strong preconditions.

Garrido and Johnson [44], [45] proposed CRefactory, a refactoring engine for C. They specified a set of refactoring preconditions that support programs in the presence of conditional compilation directives and implemented the refactorings. We can use our technique to test their refactoring implementations with respect to overly strong preconditions.

Tip et al. [46] presented 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 proposed the refactoring Infer Generic Type Arguments [32], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [47]. Eclipse implemented these refactorings. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have preconditions related to other constructs. Moreover, they did not verify whether the preconditions are overly strong. Our technique may be helpful in these situations.

Li and Thompson [48] introduced a technique to test refactorings with respect to overly weak preconditions using a tool, called Quvid QuickCheck, for Erlang. They evaluated a number of implementations of the Wrangler refactoring engine. For each refactoring, they stated a number of properties that it must satisfy. If a refactoring applies a transformation, but does not satisfy a property, they indicate a bug in the implementation. They established a number of basic properties that check for engine crashes and compilation errors. These properties should hold for all refactorings. Additionally, they wrote specific properties for a number of refactoring types concerning structural changes of the program. 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.

Borba et al. [17] proposed a set of refactoring laws for a subset of sequential Java with copy semantics. They formally specified all preconditions and proved that each transformation is sound with respect to a formal semantics. Silva et al. [18] formally specified and proved a set of refactoring transformation laws for a sequential object-oriented language with reference semantics that guarantee behavior preservation. Some of these laws can be used in the Java context. However, they have not considered all Java constructs, such as overloading and field hiding. Moreover, they did not prove the minimality property of refactoring preconditions. Our technique can evaluate whether a refactoring implementation has overly strong preconditions in the absence of formal proofs.

Overbey and Johnson [49] proposed a technique to check for behavior preservation in transformations performed by refactoring engines. They implemented it in a library containing preconditions for the most common refactorings. Refactoring engines for different languages can use their library to check preconditions. The preservation-checking algorithm is based on exploiting an isomorphism between graph nodes and textual intervals. They evaluated their technique for 18 refactorings in engines for Fortran 95, PHP 5 and BC. We can use our technique to test the refactoring implementations.

7 Conclusion

In this work, we propose a novel technique to detect overly strong preconditions in refactoring engines. We automatically generate a number of programs using JDolly and attempt to refactor them. If the refactoring engine rejects the transformation, we disable the execution of the refactoring preconditions that prevent the transformation. If the

refactoring engine with the preconditions disabled for execution applies a behavior preserving transformation according to SafeRefactor, we consider the disabled preconditions as overly strong. Refactoring engine developers can reason about their proposed preconditions to refine and slightly weaken them. As a result, they can improve the applicability of their refactoring implementations.

We generated 154,040 test inputs to test 10 refactoring implementations of Eclipse and 10 of JRRT [6] and detected 30 overly strong preconditions. So far, the Eclipse developers confirmed 47 percent of them. The technique took on average a few seconds or minutes to find the first failure.

We also compared the proposed technique (DP technique) with our previous technique based on differential testing (DT technique) [13]. The techniques are complementary in terms of bug detection. The DP technique found 11 bugs not detected by the DT technique, and the DT technique found 5 bugs not detected by the DP technique. Additionally, the DP technique does not need another refactoring engine to evaluate the refactoring implementations.

We also used the proposed technique with input programs from the Eclipse and JRRT refactoring test suites, instead of the JDolly generated programs. The goal was to analyze if it can find overly strong preconditions using other input programs. We detected 18 overly strong preconditions not detected by the developers. We cannot detect 17 of them using the specification used in JDolly 1.0. Developers did not find these overly strong preconditions because they do not seem to have a systematic strategy to detect them. Additionally, they do not have an automated oracle to check behavior preservation, such as SafeRefactor.

Developers can improve their testing process by using our proposed techniques. Whenever possible, they can run the DP technique and after fixing the detected bugs, they run the DT technique to find more bugs. The DT technique can show some additional changes that a refactoring implementation can perform to enable applying a safe transformation, such as replacing *super* with *this* (or *this* with *super*), or adding a cast in a field or method call. They can also run our techniques using their input programs instead of JDolly generated programs, as we did in part of our evaluation.

As future work, we aim at testing different types of refactoring implementations. We can extend JDolly to add new Java constructs (interfaces, inner classes) and richer method bodies following a similar approach used for generating C programs [14]. Moreover, we intend to test refactoring implementations from other tools, such as NetBeans and Visual Studio. For refactoring implementations already tested, we can reuse JDolly's specification and just have to propose a set of DP Changes to disable preconditions in Step 3.

Additionally, we will adapt our technique for other domains. First we may need to change JDolly's specification to add more Java constructs (Step 1). To perform Step 3, we have to understand: (i) how messages are represented in the tool (Step 3.2.1), (ii) and how the tool prevents yielding a message to the user (Step 3.3). We have to propose DP Changes (Step 3.4.2) to disable preconditions. Finally, we have to use an automated oracle in Step 5 to define when a transformation is correct. For example, we can use our technique to detect overly strong preconditions in mutation testing tools [50]. Each mutant operator implementation may have some preconditions to avoid introducing compilation errors or generating equivalent mutants. When the mutation testing tool rejects to apply a mutant operator to generate a mutant due to an unsatisfied precondition, we collect the message. We have to search in the code for places that may yield this message to the user. Then, we modify the mutant operator implementation and add *If* statements to prevent yielding this message. It is important to formalize repeated transformations in DP changes. So, we may follow similar transformations for other messages or mutant operators. After that, we apply the mutant operator again by disabling precondition and use SafeRefactor to check whether the transformation changes the program behavior. By using this approach, we can identify mutants that are not generated by the tool due to overly strong preconditions. In general, we can apply our technique to test tools that check some preconditions and then apply transformations.

Furthermore, we will modify SafeRefactor to use other test suite generators, such as EvoSuite [51], instead of Randoop to see whether this can improve the detection of other overly strong preconditions. EvoSuite generates and optimizes whole test suites towards satisfying a coverage criterion. We intend to evaluate multiple types of refactorings simultaneously. For each refactoring implementation, we specify additional constraints in Alloy to guide JDolly on generating programs in which the refactoring engine can apply the transformation. To handle multiple types of refactoring simultaneously, we have to specify constraints to generate programs in which more than one refactoring implementation can be applied. For example, to test the Pull Up Method and Pull Up Field refactoring implementations, we need to generate programs that contain at least a method and a field in a subclass. For each program that JDolly generates, we apply two refactorings, and analyze them separately.

We also aim at conducting interviews with refactoring engine developers about how they select program inputs, reason about the proposed preconditions, and creating the test cases. We intend to evaluate the test suites of other refactoring implementations and use real programs to try to detect more overly strong preconditions using the DP and DT techniques. Finally, we will evaluate our technique by using different time limits, and allowing the user to indicate the maximum number of tests to consider.

Appendix
Aspect-Oriented Implementation

Aspect-Oriented Programming aims to increase modularity by allowing the separation of crosscutting concerns [52]. Disabling refactoring preconditions can be seen as a crosscutting concern of the refactoring engine. We implemented in AspectJ [53] all DP changes. The abstract aspect *DisablingPreconditions* (Listing 14), declares an abstract pointcut *methodMsg* to collect calls to methods with a *String* parameter (*msg*). The pointcut refers to the left-hand side of a DP change. It also declares an *around* advice to allow executing only the methods collected in *methodMsg*, which the list *Messages.reportedMsgs* does not contain *msg* (*executePrecond* method). While DP changes include an *If* statement in the right-hand side program, the aspects use the around advice to achieve the same goal, avoiding some method executions in the resulting

program. *Messages.reportedMsgs* stores the messages related to the preconditions that we want to disable and *msg* is the message related to the evaluated precondition. We implement specific aspects to disable the preconditions of Eclipse and JRRT. They extend *DisablingPreconditions*. Developers can extend the aspects if they need to create more DP changes. They need to specify the pointcut to collect specific method calls and implement the advice to allow disabling the preconditions.

Listing 14. Abstract aspect to disable preconditions.

```
public abstract aspect DisablingPreconditions {
    abstract pointcut methodMsg(String msg);
    void around(String msg): methodMsg(msg) {
        if (executePrecond(msg)) {
            proceed(msg);
        }
    }
    public boolean executePrecond(String msg) {
        return !Messages.reportedMsgs.contains(msg);
    }
}
```

Listing 15. Aspect to disable refactoring preconditions of Eclipse.

```
public aspect DisablingPreconditions
Eclipse extends DisablingPreconditions {
    pointcut methodMsg(String msg):
        call (void RefactoringStatus.addError
        (String,..)) && args(msg,..) ||
        call (void RefactoringStatus.addWarning
        (String,..)) && args(msg,..) ||
        call (void RefactoringStatus.addEntry
        (int,String,..)) && args(int,msg,..);
    pointcut methodMsgNonVoid(String msg):
        call (RefactoringStatus RefactoringStatus.
        createErrorStatus(String,..)) &&
        args(msg,..) ||
        call (RefactoringStatus RefactoringStatus.
        createWarningStatus(String,..)) &&
        args(msg,..) ||
        call (RefactoringStatus RefactoringStatus.
        createFatalErrorStatus(String,..)) &&
        args(msg,..) ||
        call (RefactoringStatus RefactoringStatus.
        createStatus(int,String,..)) &&
        args(int,msg,..);
    RefactoringStatus around(String msg):
    methodMsgNonVoid(msg) {
        if (executePrecond(msg)) {
            return proceed(msg);
        } else {
          return new RefactoringStatus();
        }
    }
}
```

The specific aspect to disable the preconditions of Eclipse avoids adding a new warning or error status in a *RefactoringStatus* object. The *RefactoringStatus* class declares some void methods that add a new status in a *RefactoringStatus* object (methods starting with *add*). It also declares methods that create a new *RefactoringStatus* object, add the status, and return this object (methods starting with *create*). We specify a pointcut and implement an advice for both kinds of methods. The *methodMsg* pointcut collects calls to the *addError*, *addWarning*, and *addEntry* methods of *RefactoringStatus* and the *methodMsgNonVoid* pointcut collects calls to the *createStatus*, *createErrorsStatus*, *createWarningStatus*, and *createFatalErrorStatus* methods. We create this pointcut because those methods return a *RefactoringStatus* object. The refactoring implementations of Eclipse do not add or create a new status when setting the *Messages.reportedMsgs* list with the messages related to the preconditions that we want to disable. Listing 15 illustrates the aspect used to disable Eclipse preconditions. Similarly, we implement the aspect to disable JRRT preconditions.

Acknowledgments

We would like to thank Miryung Kim, Jonhnanthan Oliveira, and the anonymous reviewers. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq 307190/2015-3, 465614/2014-0, 306610/2013-2 and 460883/2014-3, CAPES 175956 and 117875, FACEPE APQ-0570-1.03/14, FAPEAL PPGs 14/2016, and DEVASSES PIRSES-GA-2013-612569.

References

[1] W. Opdyke, “Refactoring object-oriented frameworks,” Ph.D. dissertation, Univ. Illinois, Urbana-Champaign, Champaign, IL, USA, 1992.
[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley/Longman, 1999.
[3] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
[4] Eclipse.org, “Eclipse project,” 2017. [Online]. Available: http://www.eclipse.org
[5] NetBeans.org, “NetBeans IDE,” 2017. [Online]. Available: http://www.netbeans.org/
[6] M. Schäfer and O. de Moor, “Specifying and implementing refactorings,” in *Proc. 25th ACM Int. Conf. Object-Oriented Program. Syst. Languages Appl.*, 2010, pp. 286–301.
[7] M. Schäfer, M. Verbaere, T. Ekman, and O. Moor, “Stepping stones over the refactoring rubicon,” in *Proc. 23rd Eur. Conf. Object-Oriented Program.*, 2009, pp. 369–393.
[8] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, “Correct refactoring of concurrent Java code,” in *Proc. 24th Eur. Conf. Object-Oriented Program.*, 2010, pp. 225–249.
[9] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, “Refactoring Java programs for flexible locking,” in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 71–80.
[10] M. Schäfer, T. Ekman, and O. Moor, “Challenge proposal: Verification of refactorings,” in *Proc. 3rd Workshop Program. Languages Meets Program Verification*, 2008, pp. 67–72.
[11] R. Fuhrer, A. Kiezun, and M. Keller, “Refactoring in the Eclipse JDT: Past, present, and future,” in *Proc. Workshop Refactoring Tools ECOOP*, 2007, pp. 30–31.
[12] G. Soares, R. Gheyi, and T. Massoni, “Automated behavioral testing of refactoring engines,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013.
[13] G. Soares, M. Mongiovi, and R. Gheyi, “Identifying overly strong conditions in refactoring implementations,” in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 173–182.
[14] M. Mongiovi, G. Mendes, R. Gheyi, G. Soares, and M. Ribeiro, “Scaling testing of refactoring engines,” in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 371–380.

[15] M. Vakilian and R. Johnson, "Alternate refactoring paths reveal usability problems," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1106–1116.
[16] F. Tip, A. Kieżun, and D. Bäumer, "Refactoring for generalization using type constraints," in *Proc. 18th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2003, pp. 13–26.
[17] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic reasoning for object-oriented programming," *Sci. Comput. Program.*, vol. 52, pp. 53–100, 2004.
[18] L. Silva, A. Sampaio, and Z. Liu, "Laws of object-orientation with reference semantics," in *Proc. 6th IEEE Int. Conf. Softw. Eng. Formal Methods*, 2008, pp. 217–226.
[19] W. Mckeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.
[20] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Softw.*, vol. 27, no. 4, pp. 52–57, Jul./Aug. 2010.
[21] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, "Making refactoring safer through impact analysis," *Sci. Comput. Program.*, vol. 93, pp. 39–64, 2014.
[22] J. Kerievsky, *Refactoring to Patterns*. London, U.K.: Pearson Higher Education, 2004.
[23] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, 2nd ed. Cambridge, MA, USA: MIT Press, 2012.
[24] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: The Alloy constraint analyzer," in *Proc. 31st Int. Conf. Softw. Eng.*, 2000, pp. 730–733.
[25] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 75–84.
[26] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *J. Syst. Softw.*, vol. 86, no. 4, pp. 1006–1022, 2013.
[27] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan./Feb. 2012.
[28] V. Jagannath, Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Proc. 12th Int. Conf. Fundam. Approaches Softw. Eng.: Held Part Joint Eur. Conf. Theory Practice Softw.*, 2009, pp. 171–185.
[29] G. Kiczales, et al., "Aspect-oriented programming," in *Proc. 11th Eur. Conf. Object-Oriented Program.*, 1997, pp. 220–242.
[30] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 287–296.
[31] S. Negara, N. Chen, M. Vakilian, R. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. 27th Eur. Conf. Object-Oriented Program.*, 2013, pp. 552–576.
[32] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *Proc. 19th Eur. Conf. Object-Oriented Program.*, 2005, pp. 71–96.
[33] M. Schäfer, A. Thies, F. Steimann, and F. Tip, "A comprehensive approach to naming and accessibility in refactoring Java programs," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1233–1257, Nov./Dec. 2012.
[34] M. Schäfer, T. Ekman, and O. Moor, "Sound and extensible renaming for Java," in *Proc. 23th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2008, pp. 277–294.
[35] M. Schäfer, "Specification, implementation and verification of refactorings," Ph.D. dissertation, Univ. Oxford, Oxford, U.K., 2010.
[36] D. Roberts, "Practical analysis for refactoring," Ph.D. dissertation, Univ. Illinois, Urbana-Champaign, Champaign, IL, USA, 1999.
[37] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Automated Softw. Eng.*, vol. 8, pp. 89–120, 2001.
[38] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 185–194.
[39] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proc. 32nd Int. Conf. Softw. Eng.*, 2010, pp. 225–234.
[40] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *Proc. 27th Eur. Conf. Object-Oriented Program.*, 2013, pp. 629–653.
[41] L. Shattuck and D. Woods, "The critical incident technique: 40 years later," in *Proc. Human Factors Ergonom. Soc. 38th Annu. Meet.*, 1994, pp. 1080–1084.
[42] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 357–366.
[43] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *Proc. 23rd Eur. Conf. Object-Oriented Program.*, 2009, pp. 419–443.
[44] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.*, 2003, pp. 323–326.
[45] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 379–388.
[46] F. Tip, A. Kieżun, and D. Bäumer, "Refactoring for generalization using type constraints," in *Proc. 18th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2003, pp. 13–26.
[47] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proc. 20th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2005, pp. 265–279.
[48] H. Li and S. Thompson, "Testing Erlang refactorings with QuickCheck," in *Proc. 19th Int. Symp. Implementation Appl. Functional Languages*, 2008, pp. 19–36.
[49] J. Overbey and R. Johnson, "Differential precondition checking: A lightweight, reusable analysis for refactoring tools," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 303–312.
[50] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.
[51] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. 19th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 416–419.
[52] G. Kiczales, et al., *Aspect-Oriented Programming*. Berlin, Germany: Springer, 1997, pp. 220–242.
[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AspectJ," *Commun. ACM*, vol. 44, no. 10, pp. 59–65, 2001.

Melina Mongiovi received the doctoral degree in computer science from the Federal University of Campina Grande. She is a professor in the Department of Computer Science at Federal University of Campina Grande. Her research interests in software engineering include refactoring, software testing, and program analysis.

Rohit Gheyi received the doctoral degree in computer science from the Federal University of Pernambuco. He is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods, and software product lines.

Gustavo Soares received the doctoral degree in computer science from the Federal University of Campina Grande. He is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include developing program synthesis techniques and tools for data science, software developers, and end-users.

Márcio Ribeiro received the doctoral degree in computer science from the Federal University of Pernambuco, 2012. He is a professor in the Computing Institute at Federal University of Alagoas. He received the ACM SIGPLAN John Vlissides Award (2010). His PhD thesis has been awarded as the best in Computer Science of Brazil in 2012. In 2014, Márcio was the General Chair of the most important scientific event in Software of Brazil, the Brazilian Conference on Software (CBSoft). His research interests include configurable systems, variability-aware analysis, refactoring, empirical software engineering, and software testing.

Paulo Borba is professor of Software Development in the Informatics Center at the Federal University of Pernambuco, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.

Leopoldo Teixeira received the doctoral degree in computer science from the Federal University of Pernambuco. He is a professor in the Informatics Center, Federal University of Pernambuco. His research interests include software product lines, refactorings, and formal methods.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.