Scaling Testing of Refactoring Engines

Melina Mongiovi, Rohit Gheyi (advisor)

Abstract—Defining and implementing refactorings is a nontrivial task since it is difficult to define preconditions to guarantee that the transformation preserves the program behavior. Therefore, refactoring engines may apply incorrect transformations in which the resulting program does not compile, preserve behavior, or follow the refactoring definitions. These engines may also prevent correct transformations due to overly strong preconditions. We find that 84% of the test suites of Eclipse and JRRT are concerned to detect those kinds of bugs. However, the engines still have them. Researchers have proposed a number of techniques for testing refactoring engines. Nevertheless, they may have limitations related to the bug type, program generation, time consumption, and number of refactoring engines necessary to evaluate the implementations. We propose and implement a technique to scale testing of refactoring engines. We improve expressiveness of a program generator and use a technique to skip some test inputs to improve performance. Moreover, we propose new oracles to detect behavioral changes using change impact analysis, overly strong preconditions by disabling preconditions, and transformation issues. We evaluate our technique in 28 refactoring implementations of Java (Eclipse and JRRT) and C (Eclipse) and find 119 bugs. The technique reduces the time in 96% using skips while missing only 6% of the bugs. Additionally, it finds the first failure in general in a few seconds using skips. Finally, we evaluate our proposed technique by using other test inputs, such as the input programs of Eclipse and JRRT refactoring test suites. We find 31 bugs not detected by the developers.

Index Terms—Refactoring Engines, Software Testing, Program Generation, Program Analysis

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]. However, the implemented preconditions may be overly weak, allowing non-behavior preserving transformations or overly strong, preventing developers from applying useful transformations. Also, the implementation may not follow the refactoring definition [1], [2], [6], [12]. Therefore, refactoring engines may have bugs [13], [14].

In general, developers of refactoring engines manually write test cases to detect overly weak preconditions, overly strong preconditions, and transformations that do not follow the refactoring definition (transformation issues), which may be time consuming and error prone. We investigate the test suites of 20 refactoring implementations of Eclipse JDT 4.5 and JRRT (02/03/13) and find that 84% of the test assertions are concerned with identifying those kinds of bugs. Nevertheless, the bugs are still present. Testing refactoring engines is not trivial since it requires complex inputs, such as programs, and an oracle to define the correct resulting program or whether the transformation must be rejected. Manually writing test cases may be costly, and thus it may be difficult to create a good test suite considering all language constructs. Researchers have proposed a number of automated techniques for testing refactoring engines [14], [15], [16], [17]. They may automate four major steps of the testing process: (i) generating test inputs; (ii) applying the refactoring; (iii) checking the output correctness; (iv) and classifying the detected failures into distinct bugs. However, the previous approaches have limitations related to the kinds of bugs that can be detected, program generator (exhaustiveness, setup, expressiveness), time consumption, or number of refactoring engines necessary to evaluate a refactoring implementation.

In this work, we propose a technique to scale testing of refactoring engines. It automatically generates programs as test inputs using DOLLY, an automated and exhaustive Java and C program generator. Our technique can find bugs related to overly weak preconditions (compilation errors and behavioral changes), overly strong preconditions, and transformation issues. We improve a previous technique [14] with respect to DOLLY's expressiveness, reduction of the time to test the refactoring implementations, and new oracles to detect behavioral changes, overly strong preconditions, and transformation issues. We add more Java constructs in DOLLY to improve its expressiveness, extend it to generate C programs, propose a technique to skip some consecutive test inputs to reduce the costs and improve performance [18], present an oracle to identify overly strong preconditions without needing reference implementations [19], propose an oracle to identify behavioral changes [20] using change impact analysis, and introduce two oracles to identify a new kind of bug related to transformation issues.

Our technique may reduce the time to test the refactoring implementations by skipping some consecutive test inputs. Con-

• *M. Mongiovi and R. Gheyi are affiliated to the Department of Computing and Systems, Federal University of Campina Grande, PB, 58429-900 Brazil.*
E-mail: melina@computacao.ufcg.edu.br, rohit@dsc.ufcg.edu.br

secutive programs generated by DOLLY tend to be very similar, potentially detecting the same kind of bug. Thus, developers can set a parameter to skip some programs to reduce the time to test the refactoring implementations. By skipping these programs, we can reduce the Time to First Failure (TTFF), reducing the developer idle time [21]. We improve the expressiveness of DOLLY by adding abstract classes, abstract methods, and interfaces. By improving the expressiveness of the program generator, the technique may find more bugs.

The previous techniques [13], [14] use a set of oracles to evaluate the correctness of the transformations related to overly strong preconditions, compilation errors, and behavioral changes. We propose SAFEREFACTORIMPACT as the oracle to detect behavioral change transformations. SAFEREFACTORIMPACT automatically checks whether a transformation preserves the program behavior by generating test cases only for the methods impacted by a transformation.

In our previous work [13], we use Differential Testing (DT oracle) to identify overly strong preconditions in refactoring engines. It applies the same refactoring to each test input using two different implementations, and compares the results. The DT oracle needs at least two refactoring engines. This approach can only be used if the engines implement the same refactoring. In this work, we propose an oracle to identify overly strong preconditions by Disabling Preconditions (DP oracle). For each program generated by DOLLY, we apply the transformation using the refactoring engine under test. Next, we collect the different kinds of messages reported by the refactoring engine when it rejects transformations. For each kind of message, we inspect the refactoring engine and manually identify the refactoring preconditions that can raise it. We change the refactoring engine code to allow disabling the preconditions that prevent the refactoring. If the engine, with some preconditions disabled applies the transformation, and it preserves the program behavior according to SAFEREFACTORIMPACT [20], then we classify the set of disabled preconditions as overly strong.

We propose two oracles to identify transformation issues in refactoring implementations: Differential Testing (DT) and Structural Change Analysis (SCA) oracles. DT oracle compares the outputs of two refactoring implementations. For this, we implement a program that compares two Java programs concerning their Abstract Syntax Tree (AST). When the outputs compile and preserve the program behavior, we use our comparator to check if they are different. If the comparator identifies some difference, we manually inspect the transformations to analyze if one of them (or both) has issues. SCA oracle automatically analyzes whether the input and output programs have some expected properties necessary to satisfy the refactoring definition. We implement a program to check the refactoring definitions. For each output that compiles and preserves the program behavior, the technique checks whether the transformation follows the refactoring definition.

After identifying the failures, the proposed technique uses a set of automated bug categorizers to classify all failing transformations into distinct bugs. In our previous work [13], we used an approach similar to the approach proposed by Jagannath et al. [21] (Oracle-based Test Clustering) to automate the classification of failures related to overly strong preconditions. We implement an automated issue categorizer to classify the outputs of DT and SCA oracles into different kinds of issues. It is based on the kinds of differences between the outputs (for DT oracle) and the kinds of refactoring definitions that the transformations do not follow (for SCA oracle). Soares et al. [14] specified a systematic, but manual approach to categorize failures related to behavioral changes. We automate it in this work. For simplicity we use the term transformation to refer to a refactoring or a failing transformation.

We evaluate our proposed technique to scale testing of refactoring engines in 28 refactoring implementations of JRRT [6], Eclipse JDT (Java), and Eclipse CDT (C). We generate 294,648 programs using DOLLY and find 119 bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, 35 bugs related to overly strong preconditions (30 bugs using DP oracle and 24 using DT oracle), and 18 transformation issues related to the refactoring definition. We also compare the impact of the skip on the time consumption and bug detection in our technique. The technique reduces the time in 90% and 96% using skips of 10 and 25 in DOLLY while missing only 3% and 6% of the bugs, respectively. By using skips, we find the first failure related to compilation error, behavioral change, or overly strong preconditions in general in a few seconds. So, the refactoring engine developer can quickly find a bug in the refactoring implementation, fix it, run our technique again to find another bug, and so on. Before a release, tool developers can run the technique without skip to find the missed bugs.

We detect 30 overly strong preconditions in 20 refactoring implementations from Eclipse JDT and JRRT using the DP oracle. So far, Eclipse developers confirmed 47% of them. It takes around 36min to detect all overly strong preconditions of JRRT and Eclipse. Our current setup to the test the refactoring implementations of Eclipse is costlier than the JRRT ones. The DP oracle takes on average a few seconds to find the first overly strong precondition in JRRT and on average 17.41min in Eclipse. We compare the DP oracle with our previous one (DT oracle) by using the same input programs. The DP oracle detects 11 bugs (37% of new bugs) not detected by the DT oracle, while missing 5 bugs (21% of the bugs detected by the DT oracle). In addition, the DP oracle does not require using another engine with the same refactorings to compare the results. So, whenever possible, developers can run the DP oracle and after fixing the detected bugs, they run the DT oracle to find more bugs.

We also perform another study in which we use programs from the Eclipse and JRRT refactoring test suite as inputs for our technique instead of the automatically generated ones from DOLLY. Our goal is to analyze if our technique can find bugs using other input programs. We evaluate the same refactoring implementations evaluated before. We detect 23 overly strong preconditions (17 of them were not detected using the programs generated by DOLLY), 6 bugs related to compilation errors, and 2 bugs related to behavioral changes previously undetected by the developers. We reported the bugs to the Eclipse developers and so far, they did not answer. The developers do not find these bugs because they may not have a systematic strategy to detect overly strong preconditions, even with useful input programs in their test suite. Additionally, they may not have an automated oracle to check behavior preservation. We use SAFEREFACTORIMPACT as the oracle to help us in this activity.

We evaluate our oracles to identify transformation issues in eight refactoring implementations of Eclipse JDT and JRRT using DOLLY with the new constructs. We scale the new version of DOLLY to deal with a million Alloy instances. We use skip of 25 to reduce the costs and find 10 transformation issues in Eclipse and 8 in JRRT.

In summary, we propose a technique to scale testing of refactoring engines by reducing the costs and improving bug detection. The main contributions of this work [18], [19], [20] are the following:

- New features in the program generator, DOLLY (Section 3.1);
    - Extend it to generate C programs [18];
    - A skip mechanism to reduce the set of test inputs [18];
    - New Java constructs, such as abstract classes and methods, and interface;
- An oracle to identify overly strong preconditions in refactoring implementations by disabling some preconditions [19] (Section 3.3);
- Two oracles to identify transformation issues (Section 3.4);
- SAFEREFACTORIMPACT, an oracle to detect behavioral change transformations based on change impact analysis and test generation [20] (Section 3.2);
- An extensive evaluation of the proposed technique in 28 refactoring implementations of JRRT, Eclipse JDT (Java), and Eclipse CDT (C) (Section 4).

2 MOTIVATING EXAMPLES

In this section, we present some bugs related to overly strong preconditions and transformation issues in refactoring engines. First, we show 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*.

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% of them.

We also present a transformation applied by JRRT (02/03/13) in which the resulting program compiles and preserves behavior but it is not correct according to the refactoring definition. Consider a program that contains two packages: *p1* and *p2*. The first one contains two classes *A* and its subclass *B*. Class *A* declares a method *m*. Package *p2* contains class *B* that also extends class *A* (see Listing 3). By applying the Push Down Method refactoring in method *A.m* using JRRT, it removes a class from the program. The transformation moves the method to only one of its subclasses (*p2.B*) and removes the other subclass (*p1.B*). Listing 4 illustrates the resulting program.

The Push Down refactoring does not intend to remove classes from the program, this is neither the case of an overly strong precondition nor an overly weak precondition, but it is a transformation issue in the refactoring implementation. There are other similar scenarios that JRRT applies the transformations without removing entities. For example, JRRT would not remove class (*p1.B*) if it had a different name.

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% of them are concerned to overly strong preconditions, 10% to overly weak preconditions, 11% to transformation issues, 31% to overly weak preconditions and transformation issues, and 16% to other concerns. This way, we observe that Eclipse and JRRT developers are indeed concerned with identifying overly strong preconditions, overly weak preconditions, and transformation issues in their refactoring implementations. However, they may not seem to have a systematic way and automated oracles to create test cases to assess the refactoring implementations with respect to those kind of bugs.

3 TECHNIQUE

We propose a technique to scale testing of refactoring engines. First, it generates programs as test inputs using DOLLY [18], an automated program generator (Step 1). DOLLY receives as input the refactoring type, the language (Java or C), a skip number that may reduce the number of generated programs, and an Alloy specification, which includes specific constraints to a refactoring type and the program scope. Next, the refactoring is automatically applied to each generated program (Step 2). To evaluate the transformations correctness, our technique uses a set of oracles that can identify compilation errors, behavioral changes, overly strong preconditions, and transformation issues (Step 3). Finally, it automatically categorizes the detected failures into distinct bugs (Step 4). Figure 1 illustrates the main steps. Next, we explain DOLLY (Section 3.1) and the proposed oracles to detect behavioral changes (Section 3.2), overly strong preconditions (Section 3.3), and transformation issues (Section 3.4).

3.1 DOLLY

DOLLY is an automated and bounded-exhaustive Java (JDOLLY [13], [14] and C (CDOLLY [18]) program generator based on Alloy, a formal specification language [23]. DOLLY 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. DOLLY translates each instance found by the Alloy Analyzer to a Java or C program. It reuses the syntax tree available in Eclipse JDT for generating programs from those instances.

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

```
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 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 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
  }
}
```

Listing 2. Correct resulting program.

```
package p1;
public class A {
  public int m() {
    return 1;
  }
}
package p2;
import p1.*;
public class B extends A {}
package p1;
public class B extends A {}
```

Listing 3. Pushing down method A.m() using JRRT (02/03/13) removes a class from the program.

```
package p1;
public class A {}
package p2;
import p1.*;
public class B extends A {
  public int m() {
    return 1;
  }
}
```

Listing 4. Resulting program.

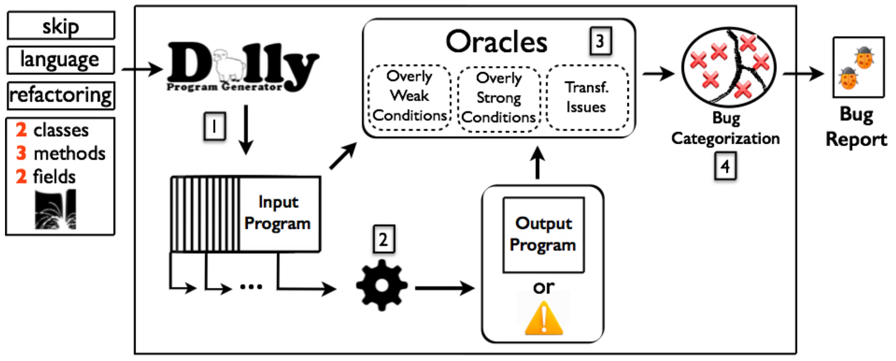


Fig. 1. A technique to scale testing of refactoring engines.

In this work, we extend DOLLY to generate C programs, add new features in DOLLY to increase its expressiveness, allow larger scopes, and reduce the cost to test the refactoring implementations [18]. We increase the expressiveness of DOLLY by generating programs considering more Java constructs, such as abstract classes and methods, and interface. We also add a new feature to skip some Alloy instances to reduce the number of generated programs and consequently the time to test the refactoring implementations.

3.1.1 *Generating C Programs*

We extend DOLLY to generate C programs. For this, we specify in Alloy a subset of the C meta-model. A C program may declare some functions. We specify the signature *Function* representing the functions of a program. A function can have parameters, a sequence of statements, and one return type.

```
sig Function {
  param: lone LocalVar,
  stmt: seq Stmt,
  returnType: one Type
}
```

For simplicity, all functions contain at most one parameter and we consider only the primitive types *int* and *float* in the specification. A function return type can be *void* or a primitive type. The statements of a function can be variable attributions (*VarAttrib*), a return statement (*Return*), local variable declarations (*LocalVarDecl*), or #ifdef declaration (*IfDef*). We have also considered programs with global variables and some C preprocessor directives such as *#define*, *#ifdef*, and *#endif*.

We specify well-formedness rules within Alloy facts to avoid C programs that do not compile. For example, the following fact specifies that if the return type of a function is not *void* the function must have a *Return* statement. The operator *&* denotes the set intersection operator.

```
fact WellFormednessRules {
  all f: Function |
    f·returnType ≠ Void ⇒
      #f·stmt·elems & Return = 1
  ...
}
```

We also specify some additional rules to cope with state explosion. For example, the predicate *optimization* does not allow functions with more than four statements. Moreover, all statements must be distinct. The relation *hasDups* yields whether there is some duplicate in the sequence.

```
pred optimization [] {
  all f: Function | #f·stmt < 5
  all f:Function | not f·stmt·hasDups
  ...
}
```

Similarly, we specify other elements of C's abstract syntax and other well-formedness rules. Notice that a sequence in Alloy may have a substantial impact in the number of Alloy instances generated by the Alloy Analyzer. Consider that a sequence may have at most *k* elements and *n* kinds of statements. The maximum number of valid sequences is $\sum_{i=0}^{k} n^i$. This number is multiplied by the number of all possible combinations of other elements of the specification.

3.1.2 New Java program constructs

We add more Java constructs in DOLLY to increase its expressiveness and find more kinds of bugs. The first step is modifying the Java meta-model in Alloy used by DOLLY. To represent an abstract method in the Java meta-model used by DOLLY, we allow creating methods without body. We change the multiplicity of the relation *Method -> Body* from *one* to *lone* (see the following fragment of the specification).

```
sig Method {
  ...
  b: lone Body
}
```

Adding new Alloy signatures or relations may increase the number of Alloy instances for a given scope. We implement the new specification by focusing on minimizing this effect.

DOLLY considers interfaces as a special type of class because adding a new Alloy signature in the model may be costly. We add the relation *implement* in the *Class* signature to allow a class to implement an interface. The following specification fragment illustrates this relation.

```
sig Class extends Type {
  ...
  implement: lone Class
}
```

We add some well-formedness rules to reduce the number of uncompilable programs considering all the specified constructs without reducing the expressiveness of DOLLY. For example, an abstract method cannot be called. The following fact specifies that there is no abstract method related to a method invocation.

```
fact noAbstractMethodInvocation {
  no m: Method | some mi: MethodInvocation |
    mi.id = m.id && isAbstract[m]
}
```

We specify other well-formedness rules related to abstract classes and methods, and interface in the Java meta-model. We provide them in our website.

3.1.3 Skipping programs

By default, DOLLY exhaustively searches for all possible combinations yielded by the run command. Even for a small scope, DOLLY may generate thousands of programs. However, the Alloy Analyzer may generate a number of similar consecutive instances [25]. Inspired on a previous technique [21], we allow developers to guide the program generation by skipping some instances. By skipping some consecutive programs, we can reduce the number of failures related to the same bug. For a skip *n*, which *n* is a positive integer, DOLLY generates one program from an Alloy instance, and jumps the following *n-1* Alloy instances. It follows this approach until the Alloy Analyzer has no more instances to generate. We implement the skip mechanism by modifying the DOLLY's source code to discard the skipped Alloy instances instead of translating each one into a program.

3.2 SAFEREFACTORIMPACT

SAFEREFACTORIMPACT receives two versions of a program as input and yields whether they have the same behavior. It uses change impact analysis to generate tests only for the entities impacted by the transformation. By comparing two versions of a program, it identifies the methods impacted by the transformation (Step 1.1). We implemented a tool, called SAFIRA, to perform the change impact analysis, which identifies the public and common impacted methods in both program versions from the impacted set (Step 1.2). Next, SAFEREFACTORIMPACT generates a test suite for the previously identified impacted methods using Randoop [26], an automatic test suite generator (Step 2). Since the tool focuses on identifying common methods, it executes the same test suite before (Step 3.1) and after the transformation (Step 3.2). Finally, the tool evaluates the results after executing the test cases: 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 is behavior preserving (Step 4). Figure 2 illustrates the described process.

The goal of the change impact analysis step is to analyze the original and modified programs, and yield the set of methods impacted by the transformation. First, we decompose a coarse-grained transformation into smaller transformations. For each

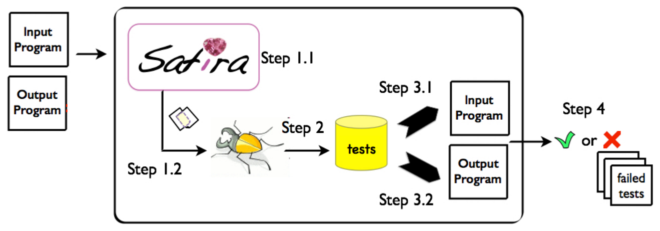


Fig. 2. SAFEREFACTORIMPACT's technique.

small-grained transformation, we identify the set of impacted methods. We formalized the impact of small-grained transformations in laws that specify the methods impacted by the transformation. For example, we specify the impact of adding or removing a method. Law 1 adds the method *m* in the class *C* when applying it from left to right, and removes the method when applying it from right to left. The set of impacted methods is the same in both directions. We use $\leftrightarrow$ to specify the impacted set for both directions. The transformation may change other program components but this law only identifies the impact of adding *m* method. If class *B* is *Object*, and *C* does not have a subclass, the set of impacted methods is *C.m*. Otherwise, other methods may be impacted due to overloading and overriding.

The next step consists of identifying the union of the set of impacted methods of each small-grained transformation. Moreover, we also identify the methods that exercise an impacted method directly or indirectly. Finally, we yield the set of impacted methods by the transformation, which is the union of directly and indirectly impacted methods.

3.3 Detecting overly strong preconditions

In this section, we explain our proposed technique to detect overly strong preconditions in refactoring implementations using the DP oracle. Our technique receives as input a refactoring implementation, the DP changes used to allow disabling the preconditions, and some parameters to configure DOLLY, 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. Figure 3 illustrates the main steps of our technique.

First, DOLLY 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 SAFEREFACTORIMPACT, 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 DOLLY that yield the same message. Algorithm 1 summarizes the main steps. Next, we explain in more details the process of disabling the preconditions.

3.3.1 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.

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 specify one DP change for JRRT and two for Eclipse in our evaluation presented in Section 4. 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

Law 1. ⟨Add/Remove Method⟩

$$\boxed{\begin{array}{l} cds \\ \texttt{class}\ C\ \texttt{extends}\ B\ \{ \\ \quad fds; \\ \quad mds; \\ \} \end{array}} \rightarrow \boxed{\begin{array}{l} cds' \\ \texttt{class}\ C\ \texttt{extends}\ B\ \{ \\ \quad fds'; \\ \quad mds'; \\ \quad m(\ldots)\ \{\ldots\} \\ \} \end{array}}$$

($\leftrightarrow$) {n:Method | $\exists$ E:Class $\cdot$ (F $<$ E $\wedge$ E $\leq$ C) $\wedge$ (n $\in$ methods(cds') $\cup$ mds') $\wedge$ n = $E.m$}, where F is the closest subclass of C such that overrides m.

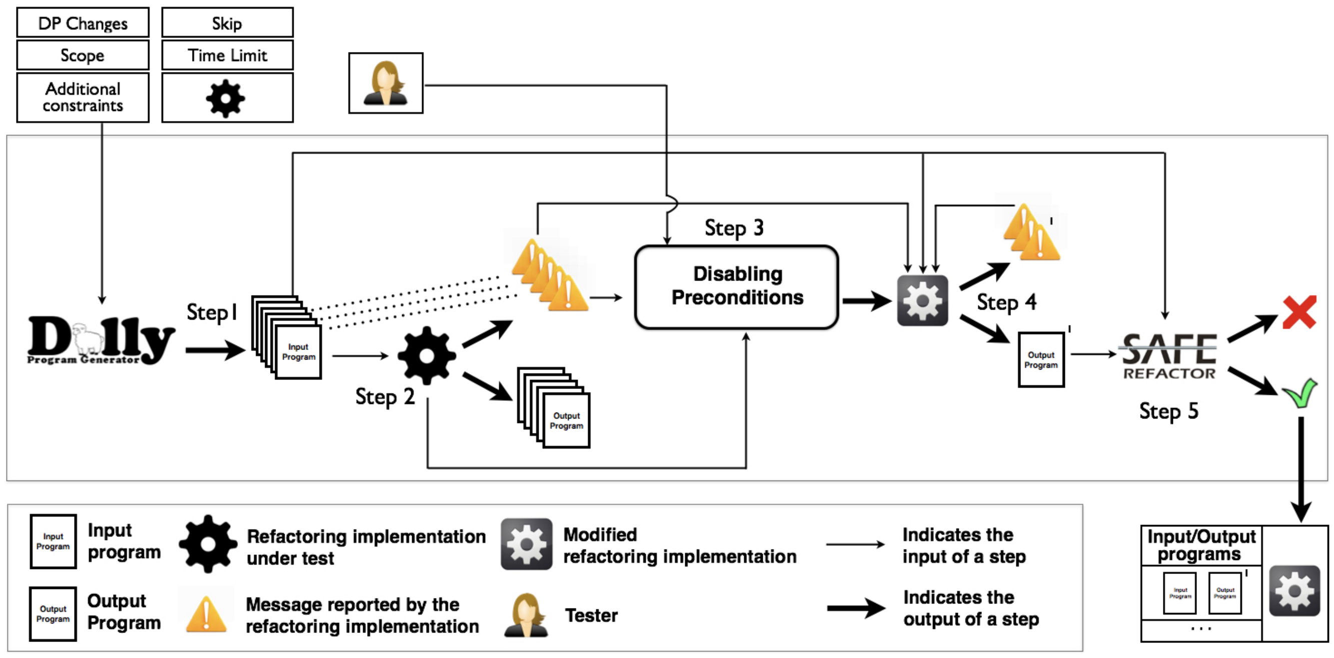


Fig. 3. A technique to detect overly strong preconditions.

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. We automate the DP changes proposed for Eclipse and JRRT using aspect-oriented programming [27]. Next, we explain in more details the DP changes in JRRT and Eclipse, and the Aspect-Oriented implementation.

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. 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

Algorithm 1 A technique to detect overly strong preconditions.

```
Input: refactoring implementation R, skip, scope, constraints, timeLimit, DPChanges
Step 1. progs = DOLLY.generate(skip, scope, constraints);
progs' = ∅;                                   ▷ A set of pairs of programs and messages
msgs = ∅;                                     ▷ A set of all messages reported by R
Step 2. foreach prog ∈ progs do
    msg = R.canApplyRefactoring(prog);        ▷ canApplyRefactoring yields one message, for simplicity, if R cannot apply it
    if msg ≠ ∅ then
        progs'.add(⟨prog, msg⟩);
        msgs.add(msg);                        ▷ For simplicity, it does not show that it removes some names and keywords from msg
map = ∅;                                      ▷ 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 ∈ 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(⟨msg, cond⟩);                   ▷ 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 ∈ 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 ∈ places do
        R' = applyDPChange(DPChanges, R', place, msg, map);   ▷ Add if (ConditionsR.cond) {place}. Specific for each ref. engine
transformations = ∅;                          ▷ A set containing all transformations applied by R'
Step 4. foreach ⟨prog, msg⟩ ∈ 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 ∈ msgs then
        go to Step 4.2;
    else if msg = ∅ then
        transformations.add(⟨prog, R'.applyRefactoring(prog)⟩);   ▷ 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 = ∅;
Step 5. foreach t ∈ transformations do
    if SAFEREFACTORIMPACT(t.input, t.output, timeLimit).hasSameBehavior() then
        result.add(t);                        ▷ It saves a behavior preserving transformation applied by R'
Output: ⟨R', result⟩;        ▷ It returns R', and all transformations that yield a set of overly strong preconditions in R
```

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.

Aspect-Oriented Implementation

Aspect-Oriented Programming aims to increase modularity by allowing the separation of crosscutting concerns [28]. Disabling refactoring preconditions can be seen as a crosscutting concern of the refactoring engine. We implemented in AspectJ [29] all DP changes. The abstract aspect *DisablingPreconditions* (Listing 5), 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.

```
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);
  }
}
```

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

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

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

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

Listing 5. Abstract aspect to disable preconditions.

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 6 illustrates the aspect used to disable Eclipse preconditions. Similarly, we implement the aspect to disable JRRT preconditions.

3.4 Detecting Transformation Issues

Although there is no unique formal definition for each kind of refactoring, there are common characteristics among them that developers of refactoring engines should follow [1], [2], [6], [12]. For example, a transformation needs to move a field from its original class to a direct superclass to follow the Pull Up Field refactoring definition. Here, we focus on detecting transformations applied by refactoring engines that the resulting program compiles and preserves behavior, but do not follow its refactoring definition. Next, we explain the proposed oracles to detect transformation issues: DT and SCA.

3.4.1 Differential Testing Oracle

This oracle receives as input two pairs of programs resulting from transformations applied by refactoring engines for the same input and refactoring type. The same input is provided to both engines. First, the oracle executes SafeRefactorImpact to evaluate whether the transformations applied by both engines under test preserve the program behavior (Step 1). The following step is only executed if both transformations compile and preserve the program behavior. In Step 2, the technique compares the outputs generated by the engines. We implement a program to compare two programs concerning their AST (Abstract Syntax Tree). First, it executes a parser and creates the abstract syntactic tree of both programs using the Eclipse JDT API. The comparator checks if the programs contain the same set of packages, classes, interfaces, methods and fields. Next, it compares each pair of classes, methods and fields concerning their modifiers, types (fields and methods), parameters (methods), method bodies (methods), initialization (fields), and imports (classes). It yields the differences between the programs if they exist. If there are differences between the outputs, the oracle reports them and we check if there is a transformation issue. We manually inspect one pair of programs of each kind of difference to analyze if there is a transformation issue in both or one of the programs.

3.4.2 Structural Change Analysis Oracle

This oracle receives as input a transformation (pair of programs) applied by a refactoring engine and the refactoring type of the transformation. We implement a program to analyze the program structure of a transformation for each refactoring type. We analyze

```
public aspect DisablingPreconditionsEclipse 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();
    }
  }
}
```

Listing 6. Aspect to disable refactoring preconditions of Eclipse.

whether the transformation follows the refactoring definition. For example, for the Pull Up Method refactoring, the transformation must remove the method from its original class and add the removed method in the direct superclass of its original class. It also must pull up all methods that are in the direct subclasses of the target class (the methods must have the same name, return type, parameters, and body of the refactored method), and update all calls to the refactored methods. We also analyze additional transformations that must not be performed, such as removing an entity from the program. First, we check if the output program preserves the program behavior using SAFEREFACTORIMPACT (Step 1). If the transformation compiles and preserves the program behavior, we check if it is applied correctly by analyzing the structural changes of the modified program (Step 2).

4 EVALUATION

We evaluated our technique in 28 refactoring implementations of JRRT [6], Eclipse JDT (Java), and Eclipse CDT (C).[2] We generated 294,648 programs as test inputs by using DOLLY. Our technique found 119 bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, 35 overly strong preconditions (DP and DT oracles), and 18 transformation issues. Next, we explain in more details how we evaluated our technique to identify overly strong preconditions (Section 4.1) and transformation issues (Section 4.2). We also evaluated our technique using the input programs of Eclipse and JRRT refactoring test suites instead of the programs generated by DOLLY (Section 4.3).

4.1 Overly Strong Preconditions

We evaluate our technique using the oracle proposed in this work to detect overly strong preconditions in 10 refactoring implementations of Eclipse and 10 refactoring implementations of JRRT. First, we present the research questions (Section 4.1.1). Next, we present (Sections 4.1.2) and discuss (Section 4.1.3) the results. Finally, explain some threats to validity (Section 4.1.5) and summarize the main findings (Section 4.1.4).

2. All complete results and experimental data are available online at http://www.dsc.ufcg.edu.br/~spg/mongiovi_thesis.html

4.1.1 *Research Questions*

Our experiment has two goals. The first goal is to evaluate the DP oracle 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 oracle 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 oracle?
  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 oracle 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 oracles detect the same bugs?
  We measure the bugs detected by both techniques: DP and DT oracles.

4.1.2 *Summary of the Results*

Concerning the JRRT evaluation, we identified 24 refactoring preconditions and found 15 (62%) overly strong preconditions in its refactoring implementations. The DP oracle did not detect 3 bugs using a skip of 25 in the Move Method and Push Down Field refactorings of JRRT. It took 0.89h to evaluate all JRRT refactoring implementations without skip to generate programs. Using skips of 10 and 25, the technique took 0.28h and 0.09h, 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%) different kinds of bugs in its refactoring implementations. The DP oracle did not detect 1 bug using skips of 10 and 25 in the Add Parameter refactoring of Eclipse. It took 35.72h to evaluate all Eclipse refactoring implementations without skip to generate programs. Using skips of 10 and 25, the technique took 4.22h and 1.75h, respectively. It took on average 17.41min to find the first failure using no skip. Using skips of 10 and 25, the technique took on average 2.35min and 1.01min 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.

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%, while for JRRT is 12.8%. 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 DP oracle with the DT oracle. The DP oracle found nine new bugs that the DT oracle 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 oracle detected in the refactoring implementations of Eclipse. Concerning the use of skips, the DP oracle did not detect four bugs using a skip of 25 and one bug using a skip of 10. The DT oracle 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 oracle without skip to find the missed bugs. Table 2 summarizes the evaluation results of the comparison between DP and DT oracles.

4.1.3 Discussion

Next, we discuss the results of our evaluation.

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. 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.

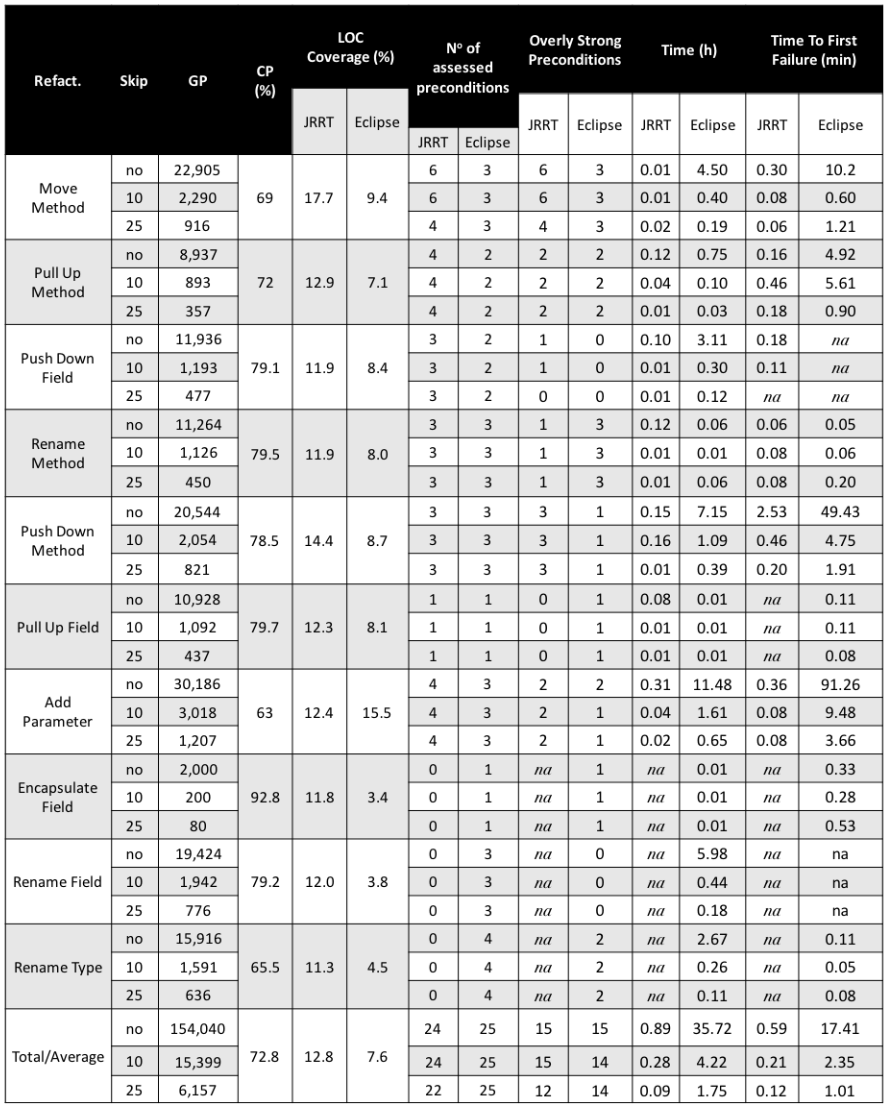
We identified some patterns followed by the developers to reject a transformation due to an unsatisfied precondition. In these cases, we propose DP changes, and we can use them to disable preconditions by preventing to report messages to the user. However, in some specific cases, we did not find a pattern 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.

Bugs Detected by DP oracle

Among the 49 assessed preconditions, we identified 30 overly strong preconditions (61%) in the Eclipse and JRRT refactoring implementations using the DP oracle. 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. We reported all detected bugs to the Eclipse developers. So far, they confirmed 47% of them (seven bugs), and did not answer for 27% (four bugs). The remaining four bugs were considered duplicates (13%) or invalid (13%). 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

TABLE 1
Summary of the DP oracle evaluation in the JRRT and Eclipse refactoring implementations; Refactoring = Kind of Refactoring; Skip = Skip value used by DOLLY to reduce the number of generated programs; GP = Number of Generated Programs by DOLLY; 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.



Refact.	Skip	GP	CP (%)	LOC Coverage (%)		Nº of assessed preconditions		Overly Strong Preconditions		Time (h)		Time To First Failure (min)	
				JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse	JRRT	Eclipse
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	0.59	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

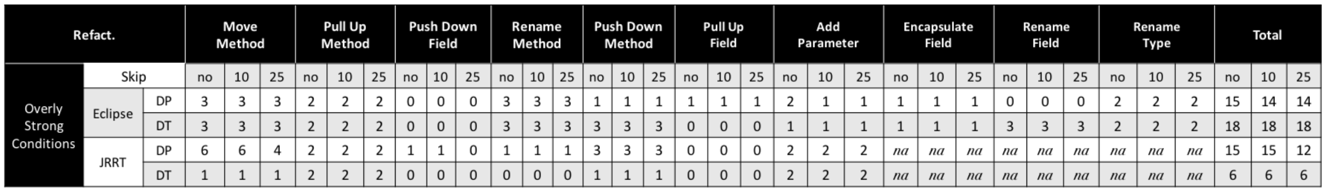
removing the overly strong preconditions found by our technique. By removing them, the refactoring implementation may apply incorrect transformations. Developers need to reason about the preconditions and choose the best strategy to slightly weaken them without making them overly weak. They can use the DP and DT oracles and our previous technique to detect overly weak preconditions [14] to reason about their preconditions.

Time

We computed the time for the automated steps of the DP oracle. 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 0.89h and 35.72h, respectively.

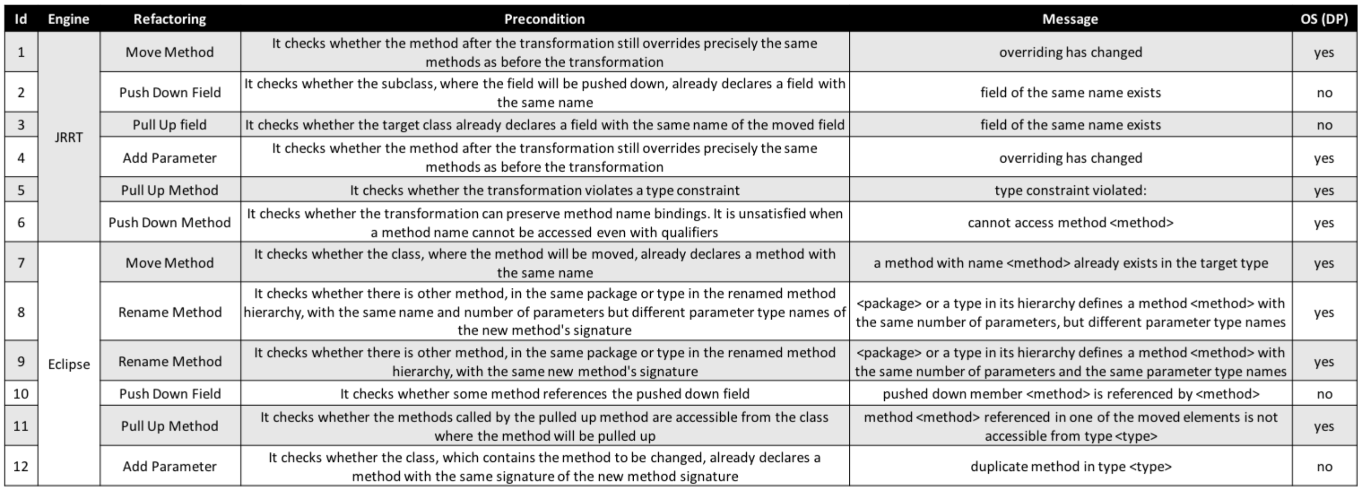
Eclipse took 11.48h and 7.15h to evaluate the Add Parameter and Push Down Method refactorings, respectively. These times were higher than the time to evaluate the other refactoring implementations. DOLLY generated more programs to evaluate these refactoring implementations (30,186 for Add Parameter and 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 (17.41min). 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

TABLE 2
Summary of the comparison between DP and DT oracles using input programs generated by DOLLY; Refactoring = Kind of Refactoring; Skip = Skip value used by DOLLY to reduce the number of generated programs; DP = DP oracle; DT = DT oracle; Overly Strong Preconditions = Number of detected overly strong preconditions in the refactoring implementations; "na" = not assessed.



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

TABLE 3
Subset of Eclipse and JRRT assessed preconditions. 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 oracle found this precondition as overly strong in this experiment, otherwise no.



Id	Engine	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		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		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		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		Pull Up Method	It checks whether the transformation violates a type constraint	type constraint violated:	yes
6		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 <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		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		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		Push Down Field	It checks whether some method references the pushed down field	pushed down member <method> is referenced by <method>	no
11		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		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

Parameter refactorings.

Comparison of DP and DT oracles

The oracles are complementary in terms of bug detection. The DP oracle detected 11 new bugs (37% of the bugs) that the DT oracle 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 oracle 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 7 presents a DOLLY 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 8 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 oracle. The DT oracle 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 oracle detected five bugs that the DP oracle cannot detect in the Eclipse Push Down Method and Rename Field refactorings. The DP oracle 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 9 presents another DOLLY 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 10 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 oracle. The DP oracle 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.

```
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 DPP{}
```

Listing 7. Pushing down field A.f to class C is rejected by JRRT due to overly strong preconditions. Bug detected by DP oracle and not detected by DT oracle.

```
public class A DPP{}

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

public class C extends A {
  private int f = 0;
}
```

Listing 8. A possible correct resulting program applied by JRRT.

```
public class B {
  protected int f1 = 0;
}

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

Listing 9. Renaming C.f0 to f1 is rejected by Eclipse JDT 4.5 due to overly strong preconditions. Bug detected by DT oracle and not detected by DP oracle.

```
public class B {
  protected int f1 = 0;
}

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

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

4.1.4 Answers to the Research Questions

Next, we answer our research questions.

- **Q1** Can the DP oracle 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%) 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 oracle?
  The technique can find the first bug in each JRRT refactoring implementation in 0.59min on average. Finding the first bug in the Eclipse evaluation took an average of 17min. 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 oracle among the set of assessed preconditions?
  In the Eclipse and JRRT refactoring implementations, 60% and 62% of the evaluated preconditions in this study are overly strong, respectively.
- **Q4** Do DP and DT oracles detect the same bugs?
  The oracles detect 19 bugs in common. The DT oracle cannot detect 11 bugs that the DP oracle 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 oracle fails to detect bugs. The DT oracle detected 5 bugs in Eclipse that the DP oracle cannot detect in the Push Down Method and Rename Field refactorings of Eclipse.

4.1.5 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% 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. 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. Finally, we specify in Table 3 some preconditions based on the available source code and documentation of JRRT and Eclipse [6], [7], [9], [11], [30], [31], [32], [33]. 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. 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. 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. We did not evaluate more refactoring implementations due to time constraints.

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 to test other refactoring implementations. 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.

We performed a feasibility study to evaluate some refactoring implementations of NetBeans 8.2. 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 11. It reports the following message: *Member "f" already exists in the target type*. By disabling this precondition, we can apply a behavior preserving transformation.

```
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;
  }
}
```

Listing 11. Pulling up field C.f to class B is rejected by NetBeans 8.2 due to an overly strong precondition.

4.2 Transformation Issues

We evaluate our technique using the oracles proposed in this work to detect transformation issues in eight refactoring implementations of Eclipse and JRRT. First, we present the research questions (Section 4.2.1). Next, we present (Sections 4.2.2) and discuss (Section 4.2.3) the results. Finally, explain some threats to validity (Section 4.2.4) and summarize the main findings (Section 4.2.5).

4.2.1 Research Questions

The goal of our experiment is to analyze our technique concerning the transformation issues detection in refactoring engines for the purpose of evaluating it with respect to issues related to transformations that do not follow the properties of each refactoring type. For this goal, we address the following research questions:

- **Q1** Can the proposed technique using the DT and SCA oracles detect transformation issues in the refactoring engines?
  We measure the number of transformation issues detected by the technique using the DT and SCA oracles for each kind of refactoring implementation.
- **Q2** Do the DT and SCA oracles detect the same issues?
  We analyze the transformation issues detected by both oracles: DT and SCA.
- **Q3** What is the time to test the refactoring implementation using the technique with DT and SCA oracles?
  We measure the total time to test each refactoring implementation using the technique with DT and SCA oracles.

4.2.2 Results

DOLLY used the Alloy Analyzer to generate up to 1,051,608 instances (Pull Up Field refactoring), which corresponded to 42,064 generated programs using skip of 25. The rate of compilable generated programs was at least 97% in each refactoring. For some kinds of refactorings, the number of rejected transformations was high. For example, in the Pull Up Field refactoring, both engines did not apply 41,721 transformations, among the set of 42,064 generated programs. On the other hand, JRRT applied all transformations in the Push Down Method refactoring.

The technique using the SCA oracle found 10 and 8 transformation issues in the refactoring implementations of Eclipse and JRRT, respectively. We found no issue in the Pull Up Field refactoring. It took 39.26h to evaluate the refactoring implementations of Eclipse and 36.91h to evaluate the refactoring implementations of JRRT. Table 4 illustrates the result of the technique using the SCA oracle.

The technique using DT oracle found two and three transformation issues in the refactoring implementations of Eclipse and JRRT, respectively. We also found no issue in the Pull Up Field refactoring. It took a total of 75.83h to evaluate all refactoring implementations. Table 5 illustrates the result of the technique using DT oracle.

4.2.3 Discussion

In this section, we discuss the results of our evaluation concerning the transformation issues detected, issue report, time to test the refactoring implementations, and the new version of DOLLY with abstract methods, abstract classes, and interface.

Transformation Issues Detected by Both Oracles

Since the DT oracle can only analyze transformations that both engines apply and the outputs compile and preserve the program behavior, we may miss some issues. Despite this, the detected issues assisted us to improve the refactoring definitions used to analyze the transformations in the SCA oracle. For example, we found a transformation applied by JRRT removes a class and from the program. Based on this detected issue, we added a refactoring definition that no transformation can remove an entity from the program. In addition to reading some proposed informal refactoring definitions, we suggest executing the technique using DT oracle before implementing the SCA oracle for each refactoring type. As we explained before, the detected issues can assist us to define the set of refactoring definitions used by the SCA oracle. We implemented the SCA and DT oracles in Java and executed the experiment using the new version of DOLLY with abstract methods, abstract classes, and interface.

The technique using the SCA oracle detected all issues detected by the technique using DT oracle. The technique using the DT oracle did not detect some issues because we only analyze the transformations applied by both engines that compile and preserve the program behavior. Figure 4 shows an issue in the Encapsulate Field refactoring of Eclipse JDT 4.5 detected by the technique using the SCA oracle. The original program presented in Listing 12 contains class `A`, which declares field `f` and method `getF` returning 0. Applying the Encapsulate Field refactoring in field `f`, choosing the default *get/set* names, the transformation does not create the correct `getF` method because there is a method with the same signature in the class (see Listing 13). So, the field is not correctly encapsulated. The engine should ask the user if he would like to choose other *get/set* names or to cancel the transformation.

Issue Report

We reported all transformation issues detected in Eclipse. So far, they confirmed some issues and rejected or marked others as duplicate. We did not report the issues detected in JRRT since there is no one in charge of it. One of the issues that we reported to Eclipse is related to encapsulating a private field. According to Fowler, only public fields can be encapsulated [2]. We cited Fowler's book to Eclipse developers and they answered that this book is out of date. On the other hand, NetBeans suggests to its developers this book to understand the refactorings.[3] Another issue that we reported to Eclipse was rejected and after our argumentation they confirmed it. This kind of issue or anomaly introduced by the refactoring engine is still somewhat difficult to confirm by refactoring engine developers, because they implement the refactorings based on their own definitions.

3. http://wiki.netbeans.org/Refactoring

Time

The time to evaluate the technique using SCA and DT oracles in the refactoring implementations of Eclipse and JRRT was almost the same. So, the oracles have a similar cost to execute. For some kinds of refactorings, the time to evaluate one engine is higher than the time to evaluate the other engine. The time may be related to the number of transformations evaluated. For example, in the Push Down Method refactoring, Eclipse rejected 8,094 transformations, while JRRT applied all transformations. So, the time to evaluate the transformations applied by JRRT is higher than the time of Eclipse. In the Pull Up Field refactoring the engines rejected the same number of transformations and the time of Eclipse is higher than the JRRT's time. Testing the refactorings implementations of Eclipse is costlier than testing JRRT's ones because we need to create an Eclipse plugin application. Also, we find that the Eclipse API used to execute the experiment has memory leak, which can make slower its execution.

DOLLY with the New Constructs

To avoid state explosion, we adapted the scope for each kind of refactoring and added some new constraints. We specified the new constraints focusing on reducing the number of non-compilable inputs. For example, a class cannot implement another class. The following fact specifies this constraint.

```
fact aClassCannotImplementAnotherClass {
  no c1: Class | some c2: Class | !isInterface[
      c2] && c2 in c1·implement
}
```

The average rate of compilable programs in DOLLY 1.0 is 68.8% [14]. We added some constraints related to all constructs of the Java metamodel implemented in DOLLY to reduce this rate of uncompilable programs. After adding the new constraints, we have reached a rate of 99.5% of compilable programs generated by DOLLY with abstract methods, abstract classes, and interface. In the Encapsulate Field refactoring 100% of the generated programs compile. The lowest rate is 97.2% in the Pull Up Method refactoring.

Despite the fact that the new constraints have reduced the number of Alloy instances, the Pull Up Field specification has 1,051,608 instances using a scope of three classes and two methods, fields, and packages. This small scope coupled with a high number of Alloy instances indicates the expressiveness of DOLLY. In the previous technique, DOLLY 1.0 generated at most 30,186 Alloy instances to generate useful programs to find bugs in the refactoring implementations [14]. After the addition of the new constructs, DOLLY had to deal with a number of Alloy instances 30 times higher than DOLLY 1.0, which increased the cost to test the refactoring implementations. Furthermore, we have to deal with memory leaks in the Eclipse API. To alleviate these problems and reduce the costs to run the experiment we choose a skip of 25 to generate programs.

4.2.4 Threats to Validity

Next, we identify some threats to validity for the evaluation performed.

Construct Validity

Construct validity refers to whether the transformation issues that we have detected are indeed incorrect transformations performed

TABLE 4
Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using the SCA oracle; Refactoring = kind of refactoring; Scope = scope used by DOLLY to generate programs; P = package; C = class; M = method; F = field; Alloy Instances = number of Alloy instances generated by the Alloy Analyzer; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY; CP = compilable generated programs; Transformation Issues = number of different kinds of issues related to incorrect transformations; Time = total time to evaluate the refactoring implementations.

Refactoring	Scope (P-C-M-F)	Alloy Instances	GP (skip of 25)	CP (%)	Rejected Refactorings		Transformation Issues		Total Time (h)	
					Ecl.	JRRT	Ecl.	JRRT	Ecl.	JRRT
Encapsulate Field	2-3-3-1	550,550	22,022	100	698	12,900	3	2	16.78	12.71
Pull Up Field	2-3-2-2	1,051,608	42,064	99.96	41,721	41,721	0	0	10.72	3.90
Pull Up Method	2-3-3-0	332,370	13,294	97.24	356	366	5	1	8.72	9.30
Push Down Method	2-3-4-0	255,177	10,207	99.47	8,094	0	2	5	3.04	11.00
Total	-	2,189,705	87,587	99.50	50,869	54,987	10	8	39.26	36.91

TABLE 5
Summary of the evaluation results of Eclipse and JRRT refactoring implementations with our technique using DT oracle; GP (using skip of 25) = number of generated programs using skip of 25 in DOLLY; App. = applied transformations; BP = behavioral preserving applied transformations; Reject. by = rejected transformations; Different/Equal transf. applied = the outputs of the engines are different/equal; Transf. Issues = number of different kinds of issues related to incorrect transformations.

Refactoring	GP (skip of 25)	Eclipse			JRRT			Different transf. applied		Equal transf. applied		Transf. Issues		Total Time (h)
		App.	BP	Reject. by JRRT	App.	BP	Reject. by Ecl.	App.	BP	App.	BP	Ecl.	JRRT	
Encapsulate Field	22,022	21,324	17,718	12,463	9,122	9,122	261	8,861	7,060	0	0	0	1	29.44
Pull Up Field	42,064	322	322	0	322	322	0	0	0	322	322	0	0	14.65
Pull Up Method	13,294	12,571	8,733	4,337	8,413	8,413	179	6,789	6,453	1,445	1,445	1	1	17.75
Push down method	10,207	2,058	2,058	0	10,153	10,153	8,095	705	705	1,353	1,353	1	1	13.99
Total	87,153	36,275	28,831	16,800	28,010	28,010	8,535	16,335	14,218	3,120	3,120	2	3	75.83

```
public class A {
  int f;
  public int getF(){
    return 0;
  }
}
```

Listing 12. Original version.

```
public class A {
  private int f;
  public int getF(){
    return 0;
  }
  public void setF(int f) {
    this.f = f;
  }
}
```

Listing 13. Resulting program.

Fig. 4. Encapsulating field B.f using Eclipse JDT 4.5 does not implement a correct *get* method because there is a method with the same signature.

by the refactoring engines. The definition of a transformation issue is strictly related to the refactoring definition used by developers of refactoring engines. They are not only concerned with implementing the pure refactoring. Sometimes, they include features that are not presented in the refactoring definitions. For example, the developers of Eclipse agree on encapsulating a private field while Fowler asserts in his book that only public fields can be encapsulated [2].

Internal Validity

Our set of conditions to define the refactoring engines is not complete. Our technique using the SCA oracle may hide possibly detectable transformation issues. The technique using DT oracle may hide some issues when one engine applies a transformation that contains an issue and the other engine does not apply or the output does not compile or preserve the program behavior. Our issue categorizer may also hide some issues. The set of conditions and the issue categorization rules can always evolve. The diversity of the generated programs is strictly related to the number of detected issues. The higher the diversity, more issues we can find. So, the scope, constraints, and skip used by DOLLY control the number of generated programs, and consequently may also hide possible issues.

External Validity

We evaluated eight refactoring implementations of Eclipse and JRRT. A survey carried out by Murphy et al. [34] shows that Java developers commonly use Pull Up refactoring. We evaluated Pull Up Field and Pull Up Method refactorings. We plan to evaluate more kinds of refactorings and other refactoring engines, such as NetBeans.

4.2.5 Answers to the Research Questions

Next, we answer our research questions.

- **Q1** Can the proposed technique using the DT and SCA oracles detect transformation issues in the refactoring engines?
  Yes. The technique using DT oracle found one issue related to transformation issue in the Pull Up Method of Eclipse, one in the Push Down Method of Eclipse, one in the Encapsulate Field of JRRT, one in the Pull Up Method of JRRT, and one in the Push Down Method of JRRT. The technique using the SCA oracle found three issues related to transformation issue in the Encapsulate Field of Eclipse, five in the Pull Up Method of Eclipse, two in the Push Down Method of Eclipse, two in the Encapsulate Field of JRRT, one in the Pull Up Method of JRRT, and five in the Push Down Method of JRRT.
- **Q2** Do the DT and SCA oracles detect the same issues?
  The technique using the SCA oracle detected all issues related to transformation issues detected by the technique using DT oracle. DT technique did not detect three issues in the Encapsulate Field of Eclipse, four in the Pull Up Method of Eclipse, one in the Push Down Method of Eclipse, one in the Encapsulate Field of JRRT, and four in the Push Down Method of JRRT.
- **Q3** What is the time to test the refactoring implementation using the technique with DT and SCA oracles?
  The oracles had a similar cost to test the refactoring implementations in this study by using the technique with DT and SCA oracles. The technique using DT oracle took 75.83h to evaluate the refactoring implementations of Eclipse and JRRT, while the technique using the SCA oracle took 76.17h to evaluate the same implementations.

4.3 Using the input programs of Eclipse and JRRT refactoring test suites

We manually analyzed the test assertions of Eclipse and JRRT to identify their concern. We found that 32% of them are related to overly strong conditions, 10% to overly weak conditions, 11% to transformation issues, 31% to overly weak conditions and transformation issues, and 16% to other concerns. We also performed another study, where we used programs from Eclipse and JRRT refactoring test suites as input for our technique instead of the DOLLY generated programs. We detected 23 overly strong conditions (17 of them were not detected using the programs generated by DOLLY), 6 bugs related to compilation errors, and 2 bugs related to behavioral changes previously undetected by the developers. The developers did not find these bugs because they may not have a systematic strategy to detect overly strong conditions, even with useful input programs in their test suite. Additionally, they may not have an automated oracle to check for behavior preservation. We use SAFEREFACTORIMPACT as the oracle to help us in this activity. Therefore, our technique can help them to avoid those kinds of bugs.

5 RELATED WORK

Daniel et al. [15] proposed an approach for automated testing refactoring engines. The technique is based on ASTGEN, a Java program generator, and a set of programmatic oracles. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. For instance, the oracles check for compilation errors and warning messages. There is one oracle that evaluates behavior preservation. It checks whether applying a refactoring to a program, its inverse refactoring to the target program yields the same initial program. If they are syntactically different, the refactoring engine developer has to manually check whether they have the same behavior. They used the oracles Differential Testing, Inverse Transformations, and Custom Oracles to identify transformation issues. The Custom oracle is aware of the structural changes that their corresponding refactorings should make and thus check that the refactored program exhibits the expected changes. Our SCA oracle is based on their Custom oracle. But they did not make available the refactoring definitions used to implement this oracle. They evaluated the technique by testing 42 refactoring implementations and found three transformation issues using Differential Testing and Inverse oracles, and only one bug using the Custom oracle. We evaluated 8 refactoring implementations and found 18 transformation issues (18 using SCA oracle and 5 using DT oracle). They identified a total of 21 bugs in Eclipse JDT and 24 in NetBeans. In Eclipse JDT, 17 bugs were related to compilation errors, 3 bugs were related to incomplete transformations (e.g. the Encapsulate field refactoring did not encapsulate all field accesses), and 1 bug was related to behavioral change. We found 17 bugs related to behavioral change in 18 refactoring implementations of JRRT and Eclipse.

Jagannath et at. [21] presented the STG technique to reduce the costs of bounded-exhaustive testing by skipping some test inputs. They randomly select a skip up to 20 after generating each program. They evaluated it using ASTGEN and found that the technique took some seconds to find the first failure related to compilation error or engine crash in the refactoring implementations using STG. We also included the skip parameter in DOLLY to reduce the time to test the refactoring implementations and to find the first failure. Different from them we use skips to identify overly strong preconditions and transformation issues. Also, we use a fixed skip that is set by the user while they use a random skip. As our results are deterministic, we can execute the tests again using the same skip to evaluate whether we have already fixed the bugs. Moreover, we can execute using a different skip to find some missed bugs. Finally, they did not measure the rate of missed bugs using skips to generate programs different from our work.

Later, Gligoric et al. [16] proposed UDITA, a Java-like language that extends ASTGEN allowing users to describe properties in UDITA using any desired mix of filtering and generating style in opposed to ASTGEN that uses a purely generating style. UDITA evolved ASTGEN to be more expressive and easier to use, usually resulting in faster program generation as well. They found four new bugs related to compilation errors in Eclipse in a few minutes. However, the technique requires substantial manual effort for writing test generators [17] since they are specified in a Java-like language. Soares et al. [14] found that UDITA does not generate some programs that JDOLLY generates using the same scope and without skipping.

More recently Gligoric et at. [17] used real systems to reduce the effort for writing test generators using the same oracles [16]. They found 141 bugs related to compilation errors in refactoring implementations for Java and C in 285 hours. However, the technique may be costly to apply the refactorings in large systems and to minimize the failure into a small program to categorize the bugs. Moreover, evaluating transformations on large real programs is time consuming, and it would produce less accurate results. We can use SAFEREFACTORIMPACT to automatically detect behavioral changes in their technique. SAFEREFACTORIMPACT detected behavioral transformations applied on real systems that even a well-defined manual inspection conducted by experts did not detect [20], [36].

Previously [13], [14], we proposed a technique to test refactoring engines by detecting bugs related to compilation errors, behavioral changes, and overly strong preconditions. It is based on JDOLLY, an exhaustive program generator, a set of automated oracles, such as SAFEREFACTOR [35], and differential testing to identify overly strong preconditions. As opposed to ASTGEN and UDITA that use a Java-like language, JDOLLY only needs to declaratively specify the structures of the programs. However, it may be costly to evaluate all test inputs. It took a total of 590 hours to detect 106 bugs related to compilation errors and behavioral changes in 39 refactoring implementations. Moreover, the technique does not test refactorings applied within method level. In this work, we optimize the technique to reduce the costs of testing. For example, using a skip of 25 in the program generator, it reduces in 96% the time to test the refactoring implementations while missing only 6% of the bugs.

Vakilian and Johnson [37] presented a technique to detect usability problems in refactoring engines. It is based on refactoring alternate paths. They adapt critical incident technique to refactoring tools and show that alternate refactoring paths are indicators of the usability problems of refactoring tools. Their technique manually found two usability problems related to overly strong preconditions. We use SAFEREFACTORIMPACT to evaluates whether the applied transformation is behavior preserving. Our technique automatically found 35 bugs related to overly strong preconditions in Eclipse JDT and JRRT.

Rachatasumrit and Kim [38] studied the impact of a transformation on regression tests by using the version history of Java open source projects. Among the evaluated research questions, they investigated whether the regression tests are adequate for refactorings in practice. They found that refactoring changes are not well tested: regression test cases cover only 22% of impacted entities. Moreover, they found that 38% of affected test cases are relevant for testing the refactorings. We proposed SAFEREFACTORIMPACT that uses change impact analyses to guide the test suite generation for only testing the methods impacted by a transformation. Most of the tests generated by our tool are relevant for evaluating the transformations considered in our work.

Schäfer et al. [8] proposed refactorings for concurrent programs. They have proved the correctness based on the Java memory model. Currently, we do not deal with concurrency since SAFEREFACTORIMPACT can only evaluate sequential Java programs. However, they have demonstrated that some useful refactorings are not influenced by concurrency. In those situations, we can use SAFEREFACTORIMPACT. Later, they [6] implemented a number of Java refactoring implementations in JRRT. They aim to improve correctness of the refactoring implementations of Eclipse. We evaluated five JRRT implementations and found some bugs related to overly weak preconditions, overly strong preconditions, and transformation issues.

6 CONCLUSION

In this work, we propose a technique to scale testing of refactoring engines. It has a program generator, DOLLY [13], [14], [18], that automatically generates programs as test input. Our technique can detect bugs related to overly weak preconditions, overly strong preconditions, and transformation issues related to the refactoring definition. We propose a new version of DOLLY with two new features: skip parameter and new Java constructs. Moreover, we extend it to generate C programs. We present a strategy to reduce the time to test the refactoring implementations by skipping some consecutive test inputs [18]. Consecutive programs generated by DOLLY tend to be very similar, potentially detecting the same kind of bug. Thus, developers can set a parameter to skip some programs to reduce the time to test the refactoring implementations. By skipping those programs, we can reduce the Time to First Failure (TTFF), reducing the developer idle time. To improve the expressiveness of DOLLY we add new Java constructs, such as abstract classes and methods, and interface.

The previous techniques [13], [14] use a set of oracles to evaluate the correctness of the transformations related to overly strong preconditions, compilation errors, and behavioral changes. It uses Differential Testing (DT oracle) to identify faults related to overly strong preconditions. We propose an oracle to identify overly strong preconditions by disabling some preconditions [19]. We also propose an oracle to identify behavioral changes [20] based on change impact analysis and test generation. Finally, we present two oracles to identify a new kind of bug related to transformation issues in refactoring implementations. The oracles are based on Differential Testing (DT) and Structural Change Analysis (SCA).

We evaluated our technique to scale testing of refactoring engines in 28 kinds of refactoring implementations of JastAdd Refactoring Tools (JRRT) [6], Eclipse JDT (Java) and Eclipse CDT (C). We found 119 bugs in a total of 49 bugs related to compilation errors, 17 bugs related to behavioral changes, 35 bugs related to overly strong preconditions using DP and DT techniques, and 18 transformation issues using SCA and DT oracles. When using skips, the refactoring engine developer can detect a number of bugs in a few hours. The developer can run the technique again without skipping while fixing the detected bugs in order to find some missed bugs. Moreover, we can reduce even more the idle time of the developer. The technique finds the first failure in the refactoring implementations in a few seconds using a skip of 10 or 25. When there are many failures transformations in a refactoring implementation, the TTFF is similar even varying the skip to generate programs. So, the developer can find a bug in a few seconds, fix the bug, run it again to find another bug, and so on. By using this strategy, the bug categorization step is no longer needed since there is only one failure in each execution. Before a new release, the developer can run the technique without skip to improve confidence that the implementation is correct.

Each precondition avoids incorrect transformations. Therefore, all of them may be needed in the refactoring engines. Our technique reports to the developers a set of overly strong preconditions. After that, they can reason about their proposed preconditions to refine and slightly weak them. As a result, they improve the applicability of their refactoring implementations by avoiding the current scenario of only implementing the preconditions without evaluating them.

In summary, we scale a technique to test refactoring engines by improving limitations of previous techniques. These limitations are related to the kinds of bugs that can be detected (some techniques do not identify transformation issues [14] or overly strong preconditions [16], [17]), time consumption [13], [14], program generator (some techniques do not have an automated program generator to generate the test inputs [17], [37] or the program generator is not exhaustive [15], [16], has a costly setup [15], [16], or has a low expressiveness [14]), or some techniques need more than one refactoring engine to evaluate a refactoring implementation [13]. Our technique uses an automated and exhaustive program generator, DOLLY to generate the test inputs. We add some features in DOLLY to reduce the time to test the refactoring implementations by skipping some input programs, improve its expressiveness by adding more Java constructs, and extend it to generate C programs. We propose SAFEREFACTORIMPACT, an oracle to identify bugs related to behavioral changes, and two oracles to identify bugs related to transformations issues. Finally, we propose an oracle to identify overly strong preconditions by disabling some preconditions. The proposed oracle only needs one engine and although it has the manual step of disabling the preconditions, we automate this step by using aspects.

REFERENCES

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

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

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

[4] Eclipse.org, "Eclipse Project," 2016, at http://www.eclipse.org.

[5] NetBeans.org, "NetBeans IDE," 2016, at http://www.netbeans.org/.

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

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

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

[9] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java programs for flexible locking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE'11. ACM, 2011, pp. 71–80.

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

[11] R. Fuhrer, A. Kiezun, and M. Keller, "Refactoring in the Eclipse JDT: Past, present, and future," in *Workshop on Refactoring Tools at ECOOP*, ser. WRT'07, Berlin, Heidelberg, 2007, pp. 30–31.

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

[13] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ser. ICSM'11, 2011, pp. 173–182.

[14] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, pp. 147–162, 2013.

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

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

[17] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *Proceedings of the 27th European conference on Object-Oriented Programming*, ser. ECOOP '13. Springer-Verlag, 2013, pp. 629–653.

[18] M. Mongiovi, G. Mendes, R. Gheyi, G. Soares, and M. Ribeiro, "Scaling testing of refactoring engines," in *IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME'14. IEEE Computer Society, 2014, pp. 371–380.

[19] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira, "Detecting overly strong preconditions in refactoring engines," *IEEE Transactions on Software Engineering*, 2017, accepted for publication.

[20] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, "Making refactoring safer through impact analysis," *Science of Computer Programming*, vol. 93, pp. 39–64, 2014.

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

[22] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[23] D. Jackson, *Software Abstractions: Logic, Language, and Analysis. 2nd edition*. The MIT Press, 2012.

[24] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the Alloy constraint analyzer," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '00. IEEE Computer Society, 2000, pp. 730–733.

[25] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Proceedings of the 13th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '07. Wiley, 2007, pp. 632–647.

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

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, ser. ECOOP'97, vol. 1241. Springer-Verlag, 1997, pp. 220–242.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented programming*. Springer Berlin Heidelberg, 1997, pp. 220–242.

[29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

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

[31] M. Schäfer, A. Thies, F. Steimann, and F. Tip, "A comprehensive approach to naming and accessibility in refactoring Java programs," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1233–1257, 2012.

[32] M. Schäfer, T. Ekman, and O. Moor, "Sound and extensible renaming for Java," in *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA'08. ACM, 2008, pp. 277–294.

[33] M. Schäfer, "Specification, implementation and verification of refactorings," Ph.D. dissertation, University of Oxford, 2010.

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

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

[36] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing Approaches to Analyze Refactoring Activity on Software Repositories," *Journal of Systems and Software*, vol. 86, no. 4, pp. 1006–1022, 2013.

[37] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 1106–1116.

[38] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ser. ICSM'12. IEEE Computer Society, 2012.