# Making Refactoring Safer through Impact Analysis

Melina Mongiovi[a,*], Rohit Gheyi[a], Gustavo Soares[a], Leopoldo Teixeira[b], Paulo Borba[b]

[a]*Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, 58429-900, Brazil.*
[b]*Informatics Center, Federal University of Pernambuco, Recife, PE, 50740-540, Brazil.*

**Abstract**

Currently most developers have to apply manual steps and use test suites to improve confidence that transformations applied to object-oriented (OO) and aspect-oriented (AO) programs are correct. However, it is not simple to do manual reasoning, due to the nontrivial semantics of OO and AO languages. Moreover, most refactoring implementations contain a number of bugs since it is difficult to establish all conditions required for a transformation to be behavior preserving. In this article, we propose a tool (SafeRefactorImpact) that analyzes the transformation and generates tests only for the methods impacted by a transformation identified by our change impact analyzer (Safira). We compare SafeRefactorImpact with our previous tool (SafeRefactor) with respect to correctness, performance, number of methods passed to the automatic test suite generator, change coverage, and number of relevant tests generated in 45 transformations. SafeRefactorImpact identifies behavioral changes undetected by SafeRefactor. Moreover, it reduces the number of methods passed to the test suite generator. Finally, SafeRefactorImpact has a better change coverage in larger subjects, and generates more relevant tests than SafeRefactor.

*Keywords:* refactoring, change impact analysis

*Corresponding author
*Email addresses:* `melina@copin.ufcg.edu.br` (Melina Mongiovi), `rohit@dsc.ufcg.edu.br` (Rohit Gheyi), `gsoares@dsc.ufcg.edu.br` (Gustavo Soares), `lmt@cin.ufpe.br` (Leopoldo Teixeira), `phmb@cin.ufpe.br` (Paulo Borba)

## 1. Introduction

Refactoring is the process of changing a program to improve its internal structure without changing its external behavior [1, 2, 3]. During software evolution, developers may apply refactorings to evolve the object-oriented (OO) or aspect-oriented (AO) code, or to extract part of the OO code into aspects to improve modularity and reduce complexity of existing software systems. AO programming aims at increasing modularity by allowing the separation of crosscutting concerns [4], such as persistence and exception handling. AspectJ [5] is a general purpose AO extension to the Java language. Existing Integrated Development Environments (IDEs), such as Eclipse and NetBeans, offer some support to refactor OO programs, but limited or no support to refactor AO programs.

Schäfer et al. [6] presented a number of Java refactoring implementations in a tool called JastAdd Refactoring Tools (JRRT). They translated a Java program to an enriched language that is easier to specify and check conditions, and apply the transformation. Monteiro and Fernandes [7] proposed 27 refactorings that we can use to introduce aspects and improve the design of AO programs. Cole and Borba [8] formally specified AO behavior-preserving transformations, and use them for deriving AO refactorings. Wloka et al. [9] proposed tool support for extending currently OO refactoring implementations for considering aspects. However, they may contain bugs since specifying and implementing refactorings is difficult. For instance, most of the current Java refactoring implementations do not check all preconditions, allowing non-behavior-preserving transformations [10, 11]. In fact, for complex languages such as Java, proving refactorings with respect to a formal semantics constitutes a challenge [12]. This problem is even worse with the presence of aspects (see Section 2). Moreover, a number of useful refactorings [13] implemented by Eclipse, such as *Extract Method*, do not consider aspects. In practice, developers have to apply manual steps and use test suites to guarantee behavior-preservation. However, Rachatasumrit and Kim [14] found that refactorings are not well tested. Their investigation identified that existing regression test cases cover only 22% of impacted entities. Moreover, they found that 38% of affected test cases are relevant for testing the refactorings. So, we need a more practical way to help developers during refactoring activities.

In this article, we propose a tool (SafeRefactorImpact) that analyzes a transformation applied to a Java or AspectJ program, and generates test cases for the methods impacted by it. Our change impact analysis identifies the methods impacted by a transformation by comparing two versions of a program, before and after the transformation. We decompose the coarse-grained transformation into smaller transformations, and analyze the impact of each of them separately. We formalize the impact of a number of small-grained transformations. We implemented this approach in a change impact analyzer called SAFIRA. The goal of the change impact analysis [15] step is to avoid the problems identified by Rachatasumrit and Kim [14]. We extend our previous work [16] by including the change impact analysis step in SAFEREFACTOR, and comparing both versions of the tool.

We evaluated SAFEREFACTOR and SAFEREFACTORIMPACT in 45 transformations. We compared the tools with respect to correctness (whether the tools identify the behavioral changes), time to analyze a transformation, number of methods considered for test generation, change coverage (the percentage of methods impacted that the test suite exercises), and relevant tests (the percentage of tests that exercises at least one impacted method). We also analyzed the influence of the time limit passed to the automatic test suite generator. First, we evaluated eight defective refactorings that change program's behavior in the presence of aspects, performed by Eclipse 4.2 with AJDT 2.2.3. We also evaluated 23 design patterns implemented in Java and AspectJ [17]. Then, we tested two JML compilers implemented using AspectJ [18, 19]. In this case, our tools compare the behavior of two JML programs as test inputs. Moreover, we evaluate four transformations that modularize exception handling in aspects, applied to two programs [20] (20 and 23 KLOC). Finally, we also evaluate eight transformations applied to different versions of JHotDraw (ranging from 28 to 79 KLOC) from its SVN repository. Some of these transformations introduce behavioral changes undetected by SAFEREFACTOR [21].

We found that SAFEREFACTORIMPACT detects behavioral changes that SAFEREFACTOR could not detect [21]. Moreover, SAFEREFACTORIMPACT is less dependent on the time limit passed to the automatic test suite generator than SAFEREFACTOR. Due to the change impact analysis, SAFEREFACTORIMPACT reduces the number of methods passed to the automatic test suite generator compared to SAFEREFACTOR. So, it has better results when analyzing transformations applied to larger programs. Furthermore, SAFEREFACTORIMPACT is faster than SAFEREFACTOR when analyzing transformations applied to small programs. For transformations applied to larger programs, the tools have similar performance. Finally, SAFEREFACTORIM-

PACT has a better change coverage than SafeRefactor in larger subjects, and most of the generated test cases in all transformations are relevant. In summary, the main contributions of this article are the following:

- extend SafeRefactor to generate test cases only for the methods impacted by the transformation (Section 3);
- compare SafeRefactor and SafeRefactorImpact with respect to correctness, time, number of methods considered for test generation, change coverage and relevant tests in 45 transformations (Section 4).

We organized this article as follows. Section 2 presents a motivating example. Section 3 proposes a technique for checking OO and AO programs based on change impact analysis. We evaluate this approach, comparing with SafeRefactor in 45 transformations (Section 4). Finally, we relate our work to others (Section 5), and present concluding remarks (Section 6).

## 2. Motivating Example

In this section, we present a defective refactoring performed by Eclipse 4.2 with AJDT 2.2.3 that introduces a behavioral change.

Consider the class $A$, its subclass $B$, and the aspect $AspectA$ presented in Listing 1. The class $C$ extends $B$, which declares the method $test$. Moreover, $AspectA$ declares the method $n$ in $B$ through an intertype declaration. By using Eclipse to apply the (aspect-aware) Rename Intertype Declaration refactoring to $B.n$, changing its name to $B.k$, we have as a result the program presented in Listing 2. Eclipse changed the intertype's name and updated its references. However, this transformation introduces a behavioral change: the $test$ method in the target program now yields 20 (Listing 2) instead of 10 (Listing 1). After the transformation, $test$ calls $B.k$, instead of the $A.k$ method.

Suppose that the developer has a test suite consisting of the test cases presented in Listing 3. It contains three test cases $test1$, $test2$, and $test3$ that call methods $A.k$, $B.test$, and $C.x$, respectively. As explained before, the transformation changed the behavior of method $B.test$. Then, $test2$ exposes the behavioral change in the modified program. However, the other tests ($test1$ and $test3$) are not relevant to test the transformation because the methods $A.k$ and $C.x$ are not impacted by the change.

Listing 1: Original program

```
class A {
  public int k() {
    return 10;
  }
}
class B extends A {
  public int test() {
    return k();
  }
}
class C extends B {
  public int x() {
    return 30;
  }
}
aspect AspectA {
  public int B.n() {
    return 20;
  }
}
```

Listing 2: Modified program

```
class A {
  public int k() {
    return 10;
  }
}
class B extends A {
  public int test() {
    return k();
  }
}
class C extends B {
  public int x() {
    return 30;
  }
}
aspect AspectA {
  public int B.k() {
    return 20;
  }
}
```

Figure 1: Applying the Rename Intertype Declaration refactoring of Eclipse 4.2 with AJDT 2.2.3 leads to a behavioral change.

Running all test cases may be time consuming, since only some test cases may be relevant to test the transformation. Rachatasumrit and Kim [14] found that existing regression tests exercise only 22% of refactored methods and fields and only 38% of tests are relevant to refactorings. In the previous example, the test suite only contains 33% of relevant tests. Furthermore, the tests may not exercise all entities impacted by the change. Therefore, to evaluate whether a transformation preserves the program behavior, it is important to test only the methods impacted by the transformation.

Listing 3: Test suite of the program presented in Listing 1.

```
public void test1() {
  A a = new A();
  long k = a.k();
```

```
    assertTrue(k == 10);
  }
  public void test2() {
    B b = new B();
    long i = b.test();
    assertTrue(i == 10);
  }
  public void test3() {
    C c = new C();
    long x = c.x();
    assertTrue(x == 30);
  }
```

## 3. SafeRefactorImpact

In this section, we present an overview of SafeRefactorImpact, whose objective is to detect behavioral changes during refactoring activities considering OO and AO constructs.

SafeRefactorImpact uses change impact analysis to generate tests only for the entities impacted by a transformation. By comparing two versions of a program, it identifies the methods impacted by the change (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 methods identified using an automatic test suite generator (Step 2). Since the tool focuses on identifying methods in common, 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.

In what follows, we describe the change impact analysis (Section 3.1) and test generation steps (Section 3.2) of SafeRefactorImpact. Then, we explain the approach using an example in Section 3.3. Finally, we describe a test data adequacy criteria [22] useful in the refactoring context, and define when a test case is relevant in Section 3.4.

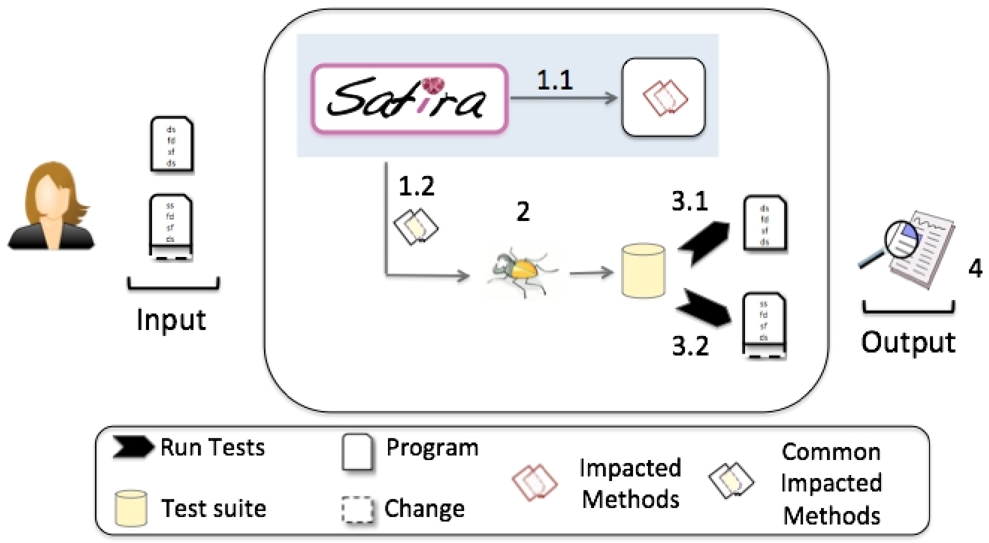


Figure 2: SafeRefactorImpact's technique.

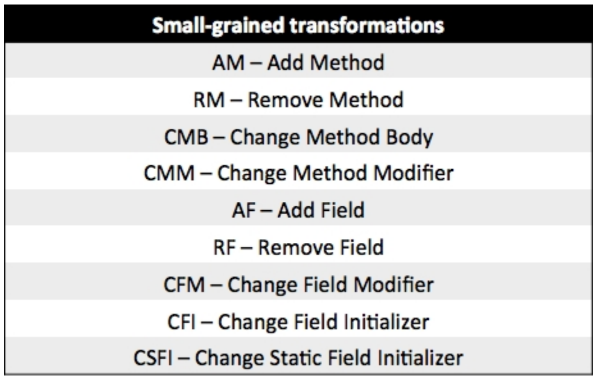
## 3.1. *Change Impact Analysis*

In this section, we explain the change impact analysis performed by SafeRefactorImpact. The goal is to analyze the original and modified programs, and yield the set of methods impacted by the change. First, we decompose a coarse-grained transformation into smaller transformations (Step 1). For each small-grained transformation, we identify the set of impacted methods. We formalized the impact of small-grained transformations in laws (Step 2). Then, we collect the union of the impacted methods set of each small-grained transformation (Step 3). Moreover, we also identify the methods that exercise an impacted method directly or indirectly (Step 4). Finally, we yield the set of impacted methods by the transformation, which is the union of directly and indirectly impacted methods (Step 5).

### 3.1.1. *Identifying Small-Grained Transformations*

We decompose the transformation into a set of small-grained transformations to analyze the impact of each one separately in the resulting program. We do so since it is simpler to analyze the impact of a small-grained transformation. Other change impact analyzers, such as Chianti [23] and FaultTracer [24], follow a similar approach.

As an example, if a transformation adds a method to a program, we consider it as the AM small-grained transformation. Another example is the CMB small-grained transformation, which modifies any part of a method body (adding, removing or changing a statement in a method body). Moreover, the CMM and CFM small-grained transformations add, remove or change a method and field modifier, respectively. Finally, the CFI and CSFI small-grained transformations add or remove field initializers or change the initialization value of instance and static fields, respectively. Table 1 describes all small-grained transformations considered by our approach.



| Small-grained transformations |
|---|
| AM – Add Method |
| RM – Remove Method |
| CMB – Change Method Body |
| CMM – Change Method Modifier |
| AF – Add Field |
| RF – Remove Field |
| CFM – Change Field Modifier |
| CFI – Change Field Initializer |
| CSFI – Change Static Field Initializer |

Table 1: Small-grained transformations considered by SAFIRA.

### *3.1.2. Identifying Impacted Methods*

After decomposing the coarse-grained transformation into smaller ones, we identify the impacted methods. We formalized the impact of each small-grained transformation described in Table 1. We grouped them into laws. Each law defines two small-grained transformations (from left to right and vice-versa) and declares two templates of programs. The meta-variables $cds$, $fds$ and $mds$ define a set of class, field and method declarations, respectively. Each law specifies how we obtain the set of impacted methods when applying it in a particular direction.

Next, 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, hence we use $\leftrightarrow$ to specify the impacted set for both directions. If the 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. For example, suppose that $C$ has a superclass different than *Object* implementing $m$, and has a subclass $D$ that does not implement $m$. Before the transformation, $D.m$ calls $B.m$. However, it calls $C.m$ after the transformation. So, $D.m$ may change its behavior. We consider as impacted all methods that inherit $m$ from $C$. We denote the subclass relation by $<$.

**Law 1.** ⟨Add/Remove Method⟩

```
cds
class C extends B {
   fds;
   mds;
}
```

⇔

```
cds′
class C extends B {
   fds′;
   mds′;
   m(...) {...}
}
```

$(\leftrightarrow)$ {n:Method | $\exists$ E:Class | (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 it redeclares $m$.

Law 2 adds the field $f$ in the class $C$ when applying it from left to right, and removes the field when applying it from right to left. If the class $B$ is *Object*, and $C$ does not have a subclass, the set of impacted methods is defined by all methods that call $C.f$. Otherwise, if there is a field in the hierarchy that inherits $f$ from $C$, the methods that use it are also impacted.

We specified other laws for the small-grained transformations presented in Table 1 similarly. After decomposing the coarse-grained transformation into smaller ones, we identify the impacted methods of each of them using our laws. The set of directly impacted methods is the union of the impacted set of each small-grained transformation. After that, we also identify the set of indirectly impacted methods that exercise an impacted method directly or indirectly. Finally, the union of directly and indirectly impacted methods defines the resulting set of impacted methods.

We implemented the change impact analyzer in a tool called Safira. It takes as input two Java or AspectJ programs (the original program and the program modified by a transformation) and reports the set of methods that

**Law 2.** ⟨Add/Remove Field⟩

$$\boxed{\begin{array}{l} cds \\ \textbf{class}\ C\ \textbf{extends}\ B\ \texttt{\{} \\ \quad fds; \\ \quad mds; \\ \texttt{\}} \\ \ \end{array}} \quad \Leftrightarrow \quad \boxed{\begin{array}{l} cds' \\ \textbf{class}\ C\ \textbf{extends}\ B\ \texttt{\{} \\ \quad fds'; \\ \quad mds'; \\ \quad T\ \ f; \\ \texttt{\}} \end{array}}$$

($\leftrightarrow$) {n:Method | $\exists$ E:Class | (F < E $\wedge$ E $\leq$ C) $\wedge$ (n $\in$ methods(*cds*$'$) $\cup$ *mds*$'$) $\wedge$ $E.f$ $\in$ commands(n)}, where $F$ is the closest subclass of $C$ such that it redeclares $f$.

can change behavior after the transformation. It uses ASM,[1] a framework to analyze and manipulate Java bytecode, to identify small-grained transformations and methods impacted. Since the tool analyzes the Java bytecode and the AspectJ compiler translates an AspectJ program to Java bytecode, we do not specify laws for AspectJ constructs.

## *3.2. Test Generation*

From the impacted methods set identified by Safira, we identify the public and common methods in both program versions. We pass them to an automatic test suite generator. Finally, we execute the same generated test suite before and after the transformation. If the results are different, we show a test case exposing the behavioral change. Otherwise, we improve confidence that the transformation is behavior preserving.

SafeRefactorImpact uses Randoop [25, 26] to automatically generate a test suite for Java programs. Randoop randomly generates unit tests for classes and methods within a time limit. A unit test typically consists of a sequence of method and constructor invocations that creates and mutates objects with random values, plus an assertion. Randoop executes the program to receive feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs. It creates method sequences incrementally, by randomly selecting a method call to apply and

[1]http://asm.ow2.org/

selecting arguments from previously constructed sequences. Then, it executes and checks each sequence against a set of contracts. For instance, a non-null object must be equal to itself. Our tool uses Randoop default contracts.

### *3.3. Example*

Consider the transformation presented in Figure 1. SafeRefactorImpact receives as input the programs shown in Listings 1 and 2. First, it decomposes the transformation into two small-grained transformations: AM (add method $k$ in class $B$) and RM (remove method $n$ from class $B$). Next, it identifies the methods impacted by each small-grained transformation. In AM, the impacted methods are the added method $B.k$ and the inherited method $C.k$. In RM, the impact methods are the removed method $B.n$ and the inherited method $C.n$. So, the set of impacted methods is $B.k$, $C.k$, $B.n$ and $C.n$. Moreover, we must also consider the indirectly impacted methods that exercise at least one impacted method. The method *B.test* exercises $B.k$, introduced by the aspect after the transformation. As *C.test* inherits of *B.test*, it is also impacted. So, the set of impacted methods identified by Safira is $B.k$, $C.k$, $B.n$, $C.n$, *B.test* and *C.test*.

Next, SafeRefactorImpact identifies the public and common impacted methods. Notice that $B.n$ and $C.n$ are not declared in the resulting program. So, our tool only generates tests for $B.k$, $C.k$, *B.test* and *C.test*. Using a time limit of one second, it generates 155 unit tests for these methods. Finally, it runs the test suite on both program versions, and evaluates the results. All tests pass in the original program but some of them do not pass in the resulting program. Listing 4 shows one of the generated test cases that reveal the behavioral change. The test case passes in the original program, since the value returned by *B.test* is 10, but fails in the modified program since the value returned by *B.test* is 20 in this version. Therefore, SafeRefactorImpact reports a behavioral change.

Listing 4: An unit test revealing a behavioral change in the transformation presented in Figure 1.

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

The main difference between SafeRefactor [16] and SafeRefactorImpact is the change impact analysis step. SafeRefactor only considers the common methods between both versions of the program. Some of them may not be impacted by the transformation. In the previous example, besides considering the same methods identified by SafeRefactorImpact to generate tests, SafeRefactor also generates test cases for the common methods $C.x$ and $A.k$, which are not impacted by the transformation. It generates test cases considering those methods (see Listing 5). However, such test cases are not relevant for analyzing the transformation since they only exercise methods that are not impacted. Executing SafeRefactor using a time limit of one second, we observe that only 66% of the test cases generated by SafeRefactor are relevant. The other test cases do not exercise an impacted method. On the other hand, due to the change impact analysis, SafeRefactorImpact does not generate such kind of non-relevant test cases.

Listing 5: A non-relevant unit test generated by SafeRefactor used to evaluate the transformation presented in Figure 1.

```
public void test() {
  C c = new C();
  long x = c.x();
  assertTrue(x == 30);
}
```

For transformations applied to small programs, this may not be a problem. However, for transformations applied to larger programs, SafeRefactor may need to increase the time limit to detect behavioral changes, since it can generate test cases that do not exercise the entities impacted by a transformation. Moreover, it might also face limitations of automatic test suite generators, if it passes a large number of methods to them.

SafeRefactor and SafeRefactorImpact check the observable behavior with respect to randomly generated sequences of method and constructor invocations. They only contain calls to methods in common. If the original and modified programs have different results for the same input, they do not have the same behavior. There are other equivalence notions. For instance, Opdyke [1] compares the observable behavior of two programs with respect to the *main* method (a method in common). If it is called twice (original and modified programs) with the same set of inputs, the resulting set of output values must be the same.

### *3.4. Change Coverage and Relevant Tests*

Based on Rachatasumrit and Kim [14] findings, refactorings are not well tested. They found that existing regression test suites may not cover the impacted entities, and a number of test cases may not be relevant for testing the refactorings. Based on this work, we define two metrics for evaluating the test suites generated by SafeRefactorImpact and SafeRefactor: Change Coverage and Relevant Tests.

The change coverage represents the percentage of impacted methods exercised by the test suite. We consider as impacted a method identified in the Safira's analysis. We define change coverage ($C$) as $C = \frac{\#E}{\#I}$, where $I$ is the set of impacted methods, and $E$ is the set of impacted methods exercised by the test suite.

We define a test case as relevant if and only if it successfully executes an impacted method identified by Safira. It is important to mention that if a test case throws an exception before or during the method execution, it is not considered relevant. We define the percentage of relevant test cases ($R$) as $R = \frac{T}{S}$, where $S$ is the number of test cases, and $T$ is the number of test cases that successfully execute at least an impacted method.

Considering the transformation presented in Figure 1, suppose that the test suite consists of the test cases presented in Listings 4 and 5. The first test case calls the method $B.test$, that calls $A.k$ in the original program and $B.k$ in the modified one. The second test case calls the method $C.x$. The set of impacted methods by this transformation is: $B.k$, $C.k$, $B.n$, $C.n$, $B.test$ and $C.test$. The test suite exercises two out of six impacted methods. So, the change coverage is: $C = \frac{2}{6} = 33\%$. Since the second test case does not exercise any impacted method, it is not relevant. So, the percentage of relevant tests in this example is: $R = \frac{1}{2} = 50\%$. Notice that some impacted methods do not belong to both programs, such as $B.n$ and $C.n$, and they are not called by other methods. Sometimes it is not possible to generate tests for them since SafeRefactorImpact generates a test suite that must execute in both versions of the program.

## 4. Evaluation

In this section, we present our experiment [27] to compare two approaches for identifying behavior-preserving transformations. First, we present the experiment definition (Section 4.1) and planning (Section 4.2). Then, Sections 4.3-4.6 describe the subjects and results. We describe some threats to validity in Section 4.7. Finally, Section 4.8 summarizes the main findings. All experimental data are available online.[2]

### *4.1. Definition*

We have structured the experiment definition using the goal, question, metric (GQM) approach in order to collect and analyze meaningful metrics to measure the proposed process. The goal of this experiment is to analyze two approaches (SafeRefactor and SafeRefactorImpact) for the purpose of evaluation with respect to identifying behavior preserving transformations from the point of view of researchers in the context of Java and AspectJ transformations. In particular, our experiment addresses the following research questions:

- **Q1**. Do SafeRefactorImpact and SafeRefactor detect the same behavioral changes?

  For each approach, we measure the number of behavioral changes detected in a given time limit.

- **Q2**. Is SafeRefactorImpact faster than SafeRefactor to evaluate a transformation?

  For each approach, we measure the total time to evaluate a transformation.

- **Q3**. Does SafeRefactorImpact consider less methods in common to generate tests than SafeRefactor?

  For each approach, we measure the number of methods in common passed to the automatic test suite generator to evaluate a transformation.

- **Q4**. Does SafeRefactorImpact generate a test suite with better change coverage than SafeRefactor?

  For each approach, we measure the change coverage of the test suite, that is, the percentage of methods impacted by the transformation identified by Safira that the test suite executes to evaluate the transformation.

[2]http://www.dsc.ufcg.edu.br/~spg/scp_experiments.html

- **Q5**. Does SafeRefactorImpact use a test suite to evaluate a transformation with more relevant test cases than SafeRefactor?

  For each approach, we measure the percentage of relevant test cases in a test suite to evaluate a transformation. A test case is relevant if and only if it successfully executes at least one method impacted by a transformation identified by Safira.

### *4.2. Planning*

In this section, we describe the hypothesis formulation, subjects used in the experiment, the experiment design, and its instrumentation.

#### *4.2.1. Hypothesis formulation*

In order to answer the research questions **Q2**, **Q3**, **Q4**, and **Q5** we formulate, respectively, the following statistical hypotheses:

- To answer **Q2**, concerning the time to evaluate a transformation:

$$H_0 : Time_{SRI} \geq Time_{SR} \tag{1}$$

$$H_1 : Time_{SRI} < Time_{SR} \tag{2}$$

- To answer **Q3**, concerning the number of methods identified to generate tests:

$$H_0 : NumberOfMethods_{SRI} \geq NumberOfMethods_{SR} \tag{3}$$

$$H_1 : NumberOfMethods_{SRI} < NumberOfMethods_{SR} \tag{4}$$

- To answer **Q4**, concerning the change coverage of the generated tests:

$$H_0 : ChangeCoverage_{SRI} \leq ChangeCoverage_{SR} \tag{5}$$

$$H_1 : ChangeCoverage_{SRI} > ChangeCoverage_{SR} \tag{6}$$

- To answer **Q5**, concerning the percentage of relevant tests:

$$H_0 : RelevantTests_{SRI} \leq RelevantTests_{SR} \tag{7}$$

$$H_1 : RelevantTests_{SRI} > RelevantTests_{SR} \tag{8}$$

We perform statistical analysis for each group of subjects that contains at least eight transformations. We use Shapiro-test [28] to analyze data normality because it is more adequate for small samples. Then, if the data are normal, we use T-test [29], otherwise we use Wilcoxon-test [30]. We use the level of significance 0.5.

### *4.2.2. Selection of subjects*

We evaluated SafeRefactor and SafeRefactorImpact in eight defective refactorings applied by Eclipse, 23 design patterns implemented in Java and AspectJ, in the bytecode generated by two Java Modeling Language (JML) [31] compilers for two programs, and 12 transformations applied to real OO and AO programs. In Sections 4.3-4.6, we give more details about them. Experienced developers and researchers in the OO and AO field applied the transformations, which have different granularities, to programs with different sizes (ranging from 10 LOC to 79 KLOC).

The transformations change OO (classes, methods, fields, inheritance, overloading, overriding, packages, accessibility) and AO (aspects, intertype declarations, pointcuts, advices) constructs. We analyzed local and global transformations. Some of them affect classes, aspects and method signatures, while others change blocks of code only within methods.

### *4.2.3. Experiment design*

In our experiment, we evaluate one factor (approaches for detecting behavior-preserving transformations) with two levels (SafeRefactor and SafeRefactorImpact). We choose a paired comparison design for the experiment, that is, we apply both treatments to all subjects. We evaluate the approaches on 45 transformations. The results can be "Yes" (behavior-preserving transformation) and "No" (non-behavior-preserving transformation).

### *4.2.4. Instrumentation*

We ran the experiment on a 2.7 GHz core i5 with 8 GB RAM and running Mac OS 10.8. We used the command line interfaces of SafeRefactor 1.1.4 and SafeRefactorImpact 1.0 using Java 1.6. They receive as parameters the original and the target program paths, and the time limit to generate tests. We used Safira 1.0, which uses ASM 3.0. We used a time limit of 0.2s, 0.5s and 0.2s to generate tests for subjects described in Sections 4.3, 4.4 and 4.5, respectively. These limits are enough to test transformations applied to small programs. We used a time limit of 20s for subjects described in Section 4.6. Both tools use Randoop 1.3.3, configured to avoid generating non-deterministic test cases.

Since we do not know beforehand which versions contain behavior-preserving transformations, the first and third authors of this article compared the results of all approaches in all transformations to establish a Baseline to check the results of each approach. For instance, if SafeRefactor yielded "Yes" and SafeRefactorImpact "No", the authors checked whether the test case showing the behavioral change reported by SafeRefactorImpact was correct. If so, the correct result was "No".

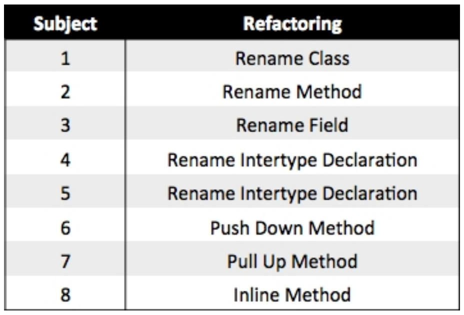
## *4.3. Defective Refactorings*

Next we evaluate SafeRefactor and SafeRefactorImpact in a number of non-behavior-preserving transformations applied by Eclipse refactorings to small toy examples created by us.

### *Selection of subjects*

Eclipse JDT is a popular Java IDE with a number of automated OO refactorings. It also offers refactoring support for AspectJ through the AspectJ Development Tools module (AJDT). For instance, since AJDT 2.1.0 delivery in 2010, the Rename refactorings of Eclipse are aspect-aware. Each subject contains a small set of classes, pointcuts, advices, and intertype declarations. Each subject contains one aspect and at most four classes. For instance, the example shown in Section 2 is similar to Subject 5 of our evaluation. We evaluate eight transformations applied by Eclipse 4.2 using AJDT 2.2.3 that introduce behavioral changes in AO programs. We found them based on our experience in finding bugs in OO refactoring tools [10]. Table 2 describes the transformations applied.

In Subjects 1-5, we apply Eclipse refactoring implementations that are aspect-aware. In some aspect-aware refactorings (Subjects 2 and 3), Eclipse does not update pointcuts, leading to behavioral changes. Pointcuts can use wildcards, which might impose additional challenges when checking preconditions. The behavioral changes in Subjects 4 and 5 are due to OO features, such as overloading and overriding. On the other hand, in Subjects 6-8, we apply different kinds of useful OO refactorings (Push Down Method, Pull Up Method, and Inline Method) performed by Eclipse that are unaware of aspects. In practice, refactoring tools have limited support for AO refactorings. So, developers may have to manually perform a transformation or use



| Subject | Refactoring |
|---|---|
| 1 | Rename Class |
| 2 | Rename Method |
| 3 | Rename Field |
| 4 | Rename Intertype Declaration |
| 5 | Rename Intertype Declaration |
| 6 | Push Down Method |
| 7 | Pull Up Method |
| 8 | Inline Method |

Table 2: A catalog of transformations performed by Eclipse that introduce behavioral changes in the presence of aspects.

an OO refactoring implementation to automate part of the transformation, and manually check whether it preserves behavior.

*Operation*

We compared SafeRefactor and SafeRefactorImpact using a time limit of 0.2s passed to Randoop. SafeRefactor and SafeRefactorImpact correctly identified all behavioral changes but one, Subject 2, that only SafeRefactor identified. SafeRefactorImpact evaluated the subjects faster than SafeRefactor. The change impact analysis is also useful to reduce the set of common methods passed to Randoop in all subjects by an average of 60%. Furthermore, both tools have almost the same change coverage in all subjects but Subject 4. Finally, all test cases generated by SafeRefactorImpact are relevant to test the change different from SafeRefactor. Table 3 summarizes the results.

*Discussion*

SafeRefactorImpact does not detect the behavioral change in Subject 2, since Safira does not perform data flow analysis. So, SafeRefactorImpact may not generate test cases containing some getter methods that may be useful to expose the behavioral change. SafeRefactorImpact has a parameter that, when enabled, allows us to consider all getter methods during the test suite generation. By enabling this parameter, SafeRefactorImpact correctly identifies the behavioral change in Subject 2. However, when using such option, the number of methods passed to the test suite generator may increase in some transformations.

Table 3: Results using a time limit of 0.2s. Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior-preserving.

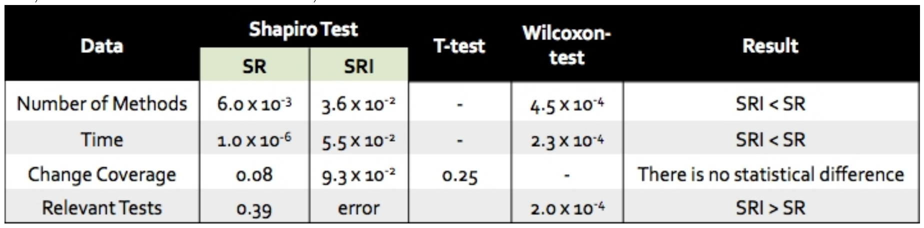
| Subject | Methods | | Time (s) | | Change Coverage (%) | | Relevant Tests (%) | | Result | | Baseline |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | SR | SRI | SR | SRI | SR | SRI | SR | SRI | SR | SRI | |
| 1 | 7 | 2 | 8 | 2 | 77 | 77 | 20 | 100 | No | No | No |
| 2 | 22 | 6 | 8 | 4 | 35 | 35 | 50 | 100 | No | Yes | No |
| 3 | 6 | 2 | 8 | 2 | 100 | 100 | 66.66 | 100 | No | No | No |
| 4 | 8 | 4 | 8 | 3 | 27 | 100 | 87.87 | 100 | No | No | No |
| 5 | 9 | 4 | 8 | 3 | 25 | 25 | 82.60 | 100 | No | No | No |
| 6 | 11 | 4 | 8 | 3 | 75 | 75 | 66.66 | 100 | No | No | No |
| 7 | 11 | 4 | 7 | 2 | 75 | 75 | 71.42 | 100 | No | No | No |
| 8 | 8 | 4 | 8 | 3 | 75 | 75 | 94.73 | 100 | No | No | No |

SafeRefactorImpact is faster than SafeRefactor, since it generates test cases considering less methods. SafeRefactorImpact uses ASM to perform analysis on the programs instead of reflection. Both tools achieved 100% change coverage in Subject 3. By inspecting the test cases, we observed that for some impacted methods, Randoop generated test cases that throw `IllegalArgumentException` when invoking them. Since the impacted methods are not executed in those test cases, SafeRefactorImpact cannot yield 100% of change coverage in some subjects. Finally, notice that SafeRefactor generates less relevant test cases than SafeRefactorImpact even for transformations applied to small programs. For example, in Subject 1, only 20% of the generated tests are relevant.

Table 4 describes the statistical analysis results. Column Shapiro Test indicates the Shapiro–Wilk test results. When we ran Shapiro-test in relevant tests data of SafeRefactorImpact an error occurred, because all data are equal (100% of relevant tests). Then, we consider it as non-normal. Notice that only the change coverage data of SafeRefactor and SafeRefactorImpact are normal. Columns T-test and Wilcoxon-test present the results of the tests to evaluate the hypothesis presented in Section 4.2.1.

Due to non-normality of data, we use Wilcoxon-test for number of methods, time, and percentage of relevant tests data. It reached small p-values to all of them: $4.5 \times 10^{-4}$, $2.3 \times 10^{-4}$ and $2.0 \times 10^{-4}$, respectively. The results give us evidence that SafeRefactorImpact reduces time, identifies less methods, and generates more relevant tests than SafeRefactor. To evaluate change coverage we use T-test due to normality of data. It reached a p-value of 0.25 which indicates that the change coverage of SafeRefactorImpact is less than or similar to the change coverage of SafeRefactor. Then, we execute another test (T-test) assuming a null hypothesis that the change coverage is equal for both tools. It reached a p-value of 0.51, which indicates that there is no statistical difference between the change coverage of SafeRefactor and SafeRefactorImpact.

Table 4: Statistical analysis for defective refactoring data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SafeRefactor; SRI = SafeRefactorImpact.



| Data | Shapiro Test | | T-test | Wilcoxon-test | Result |
|---|---|---|---|---|---|
| | SR | SRI | | | |
| Number of Methods | $6.0 \times 10^{-3}$ | $3.6 \times 10^{-2}$ | - | $4.5 \times 10^{-4}$ | SRI < SR |
| Time | $1.0 \times 10^{-6}$ | $5.5 \times 10^{-2}$ | - | $2.3 \times 10^{-4}$ | SRI < SR |
| Change Coverage | 0.08 | $9.3 \times 10^{-2}$ | 0.25 | - | There is no statistical difference |
| Relevant Tests | 0.39 | error | | $2.0 \times 10^{-4}$ | SRI > SR |

## *4.4. Design Patterns*

In this section, we evaluate SafeRefactor and SafeRefactorImpact to check the equivalence between OO implementations of design patterns and their correspondent AO versions.

*Selection of subjects*

Hannemann and Kiczales [17] implemented 23 design patterns [32] in Java. The same patterns are also implemented in AspectJ. They compared them with respect to locality, reusability, composability, and (un)pluggability. For Hannemann and Kiczales [17], the OO and AO implementations are equivalent. Table 5 describes the design patterns evaluated.

*Operation*

We compared SafeRefactor and SafeRefactorImpact using a time limit of 0.5s passed to Randoop. SafeRefactorImpact correctly identified behavioral changes in 5 out of 23 the design patterns implementations [17]. SafeRefactor identified all of these behavioral changes but one (the Mediator design pattern), which can only be detected using a time limit of three seconds. Hannemann and Kiczales [17] did not expect to introduce

Table 5: Design patterns implemented in Java and AspectJ.

| Subject | Design Pattern | Subject | Design Pattern |
|---|---|---|---|
| 9 | Abstract Factory | 21 | Iterator |
| 10 | Adapter | 22 | Mediator |
| 11 | Bridge | 23 | Memento |
| 12 | Builder | 24 | Observer |
| 13 | Chain of Responsibility | 25 | Prototype |
| 14 | Command | 26 | Proxy |
| 15 | Composite | 27 | Singleton |
| 16 | Decorator | 28 | State |
| 17 | Facade | 29 | Strategy |
| 18 | Factory Method | 30 | Template Method |
| 19 | Flyweight | 31 | Visitor |
| 20 | Interpreter | | |

behavioral changes in the Mediator, Prototype, State, Template and Visitor design patterns. SafeRefactorImpact evaluated the subjects faster than SafeRefactor. The change impact analysis reduces the set of common methods passed to Randoop in Subjects 13, 14 and 22. Both tools have almost the same change coverage except for Subjects 9, 18 and 30, but SafeRefactorImpact generated more relevant tests than SafeRefactor. Table 6 summarizes the results.

*Discussion*

Hannemann and Kiczales [17] implemented OO and AO versions of the Queue data structure to illustrate the State pattern (Subject 28). This pattern allows an object to behave differently according to its internal state. They implemented the `Queue` class to represent a queue and the abstract class `State` representing the queue states (`Empty`, `Normal`, and `Full`), as depicted by Figure 4.4. Each queue must contain at most three elements.

In the OO version, the state transitions are performed in each class representing a possible state. For instance, the following code snippet shows the `insert` method from the `Empty` class, which changes the queue's state to normal, and adds an element.

```
public boolean insert(Queue queue, Object arg) {
  Normal nextState = new Normal();
  queue.setState(nextState);
  return nextState.insert(context, arg);
```

Table 6: Results using a time limit of 0.5s. Impacted Methods = number of methods identified by SAFIRA; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.

| Subject | Impacted Methods | Methods | | Time (s) | | Change Coverage (%) | | Relevant Tests (%) | | Result | | Baseline |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SR | SRI | SR | SRI | SR | SRI | SR | SRI | SR | SRI | |
| 9 | 107 | 315 | 10 | 16 | 4 | 2 | 7 | 33.33 | 100 | Yes | Yes | Yes |
| 10 | 95 | 4 | 4 | 8 | 3 | 1 | 1 | 97.56 | 100 | Yes | Yes | Yes |
| 11 | 122 | 14 | 16 | 8 | 3 | 12 | 14 | 97.05 | 98.86 | Yes | Yes | Yes |
| 12 | 115 | 11 | 12 | 12 | 3 | 11 | 13 | 98.87 | 100 | Yes | Yes | Yes |
| 13 | 130 | 998 | 7 | 18 | 3 | 0 | 0 | 0 | 0 | Yes | Yes | Yes |
| 14 | 116 | 384 | 5 | 13 | 6 | 6 | 6 | 9.09 | 100 | Yes | Yes | Yes |
| 15 | 131 | 7 | 7 | 9 | 5 | 2 | 2 | 69.52 | 91.08 | Yes | Yes | Yes |
| 16 | 108 | 4 | 4 | 10 | 4 | 2 | 2 | 97.67 | 100 | Yes | Yes | Yes |
| 17 | 96 | 14 | 14 | 8 | 3 | 8 | 8 | 95.29 | 100 | Yes | Yes | Yes |
| 18 | 113 | 8 | 10 | 15 | 7 | 7 | 12 | 91.66 | 100 | Yes | Yes | Yes |
| 19 | 99 | 6 | 6 | 10 | 3 | 2 | 2 | 98.18 | 100 | Yes | Yes | Yes |
| 20 | 135 | 38 | 27 | 8 | 3 | 6 | 9 | 66.67 | 84.62 | Yes | Yes | Yes |
| 21 | 109 | 7 | 7 | 10 | 3 | 3 | 3 | 100 | 100 | Yes | Yes | Yes |
| 22 | 99 | 714 | 5 | 17 | 3 | 0 | 3 | 0 | 50.00 | Yes | No | No |
| 23 | 97 | 5 | 5 | 8 | 3 | 2 | 2 | 98.36 | 100 | Yes | Yes | Yes |
| 24 | 134 | 11 | 11 | 8 | 5 | 2 | 2 | 96.67 | 100 | Yes | Yes | Yes |
| 25 | 103 | 10 | 10 | 9 | 3 | 5 | 5 | 88.61 | 88.78 | No | No | No |
| 26 | 144 | 6 | 6 | 10 | 3 | 6 | 6 | 100 | 100 | Yes | Yes | Yes |
| 27 | 115 | 1 | 2 | 10 | 3 | 2 | 5 | 100 | 100 | Yes | Yes | Yes |
| 28 | 123 | 9 | 9 | 9 | 3 | 21 | 20 | 96.67 | 100 | No | No | No |
| 29 | 107 | 6 | 6 | 9 | 3 | 1 | 1 | 90.91 | 100 | Yes | Yes | Yes |
| 30 | 104 | 10 | 11 | 12 | 3 | 8 | 15 | 98.73 | 100 | No | No | No |
| 31 | 115 | 11 | 11 | 9 | 4 | 4 | 4 | 88.89 | 91.67 | No | No | No |

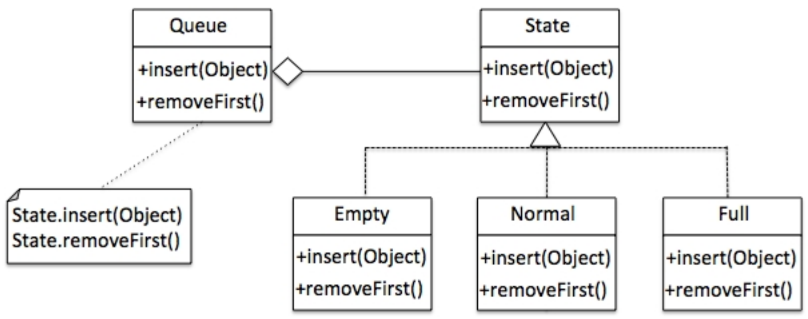


Figure 3: The class diagram of a Queue using the State design pattern.

```
}
```

On the other hand, they implemented the state transitions in an aspect in the AO version. The aspect declares the state objects (empty, normal, full), and an advice makes the state transition after the invocation of the `insert` method.

```
public aspect QueueStateAspect {
  protected Empty empty  = new Empty();
  protected Normal normal = new Normal();
  protected Full full = new Full(); ...
  after(Queue queue, State qs, Object arg):
      call(boolean State+.insert(Object)) && ... {
    if (qs == empty) {
      normal.insert(arg);
      queue.setState(normal);
    } ...
  }
}
```

Both tools detected a behavioral change. Listing 6 shows a test case generated by SafeRefactor that reveals a behavioral change. It instantiates the `q1` queue and adds one element to it. Next, it instantiates another queue `q2` and add three elements. The OO version correctly inserts all elements. However, the last element cannot be inserted into the queue in the AO version. The `r4` variable yields false. It states that the queue is full (it contains three elements).

Listing 6: A unit test revealing a behavioral change in the State pattern.

```
public void test() {
  Queue q1 = new Queue();
  boolean r1 = q1.insert(``element1'');
  Queue q2 = new Queue();
  boolean r2 = q2.insert(``element1'');
  boolean r3 = q2.insert(``element2'');
  boolean r4 = q2.insert(``element3'');
  assertTrue(r1 == true);
  assertTrue(r2 == true);
  assertTrue(r3 == true);
  assertTrue(r4 == true);
}
```

Aspects are singleton by default in AspectJ [5]. Notice that the fields of the `QueueStateAspect` aspect are only instantiated when the aspect is created. Therefore, all queues share the same state. `Normal` contains an array for storing three elements. When we insert an element in `q1`, it is inserted in this array. However, when we create `q2`, this array is not cleared. Therefore, we can only include two elements in `q2`. To avoid this problem, they could have instantiated an aspect for every new queue instance. AspectJ allows per-object aspects by using the `perthis` and `pertarget` keywords [5].

We also found a behavioral change in the Mediator pattern implementations. Developers implemented a GUI application and used the mediator pattern to deal with changes to GUI components that require updates. In the OO version, they implemented this pattern as a field of the component, which must be set by using a setter method.

Both tools generated test cases that instantiate the `Button` object, which contains a mediator field, and apply changes to the object, as shown in Listing 7.

Listing 7: A unit test revealing a behavioral change in the Mediator pattern.

```
public void test() {
  Button var1 = new Button(‘‘’’);
  var1.clicked();
}
```

The mediator field should handle the change performed by method `clicked`. However, the test case does not set a mediator to the `Button` object (it should be set by using the `setMediator` method). Therefore, it throws the exception `NullPointerException`. On the other hand, Hannemann and Kiczales [17] implemented the mediator as an aspect in the AO version, which has already been instantiated when the change is performed. Therefore, the tools successfully executed this test in the AO version.

Notice that SafeRefactor cannot detect this behavioral change. The time limit of 0.5s passed to Randoop is not enough to generate tests considering 714 methods. So, it does not generate relevant test cases and cover the change different from SafeRefactorImpact.

Finally, we found simple behavioral changes in three design patterns (Subjects 25, 30 and 31). Some methods yield different `String` messages.

Both tools have a low change coverage. The number of impacted methods (see Column Impacted Methods in Table 6) identified by Safira is larger

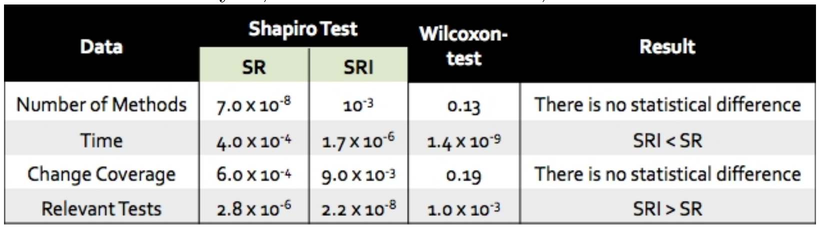
(90%) than the number of methods passed to Randoop by SafeRefactorImpact. The transformation adds or removes most impacted methods. Then, SafeRefactorImpact cannot pass them to Randoop because they do not belong to both versions of the program. As mentioned before, our goal is to generate a test suite to be executed before and after the transformation. Furthermore, some methods contain parameter types declared in external libraries, such as Java AWT, in some subjects. Randoop does not generate test inputs for them unless we pass them as parameters, or some method being tested yields an object of the library's type. In Subject 13, all test cases generated by SafeRefactorImpact throw exceptions before executing the impacted method. We may increase the time limit, or this may indicate a limitation of the test suite generator that cannot handle some kinds of Java constructions, such as GUI elements. So, the tool does not generate relevant tests to exercise the change in this subject. In Subjects 13 and 22, a similar scenario happens in SafeRefactor.

Notice that SafeRefactor identifies a number of common methods to generate tests. In Subjects 9, 13, 14 and 22, there are some classes that extend Java Swing and Java AWT classes. SafeRefactor generates tests for the inherited methods since they belong to both versions of the program. SafeRefactorImpact only generates tests for the methods impacted by the transformation. In some subjects, SafeRefactorImpact passed more methods in common to Randoop than SafeRefactor. Different from SafeRefactor, SafeRefactorImpact takes into consideration methods that are moved from a class to an aspect, that introduces it in the same class using an intertype declaration.

Table 7 describes the statistical analysis results. Column Shapiro Test consists of Shapiro-Wilk test results. The results indicate that all data are non-normal. Then, we use Wilcoxon-test for all of them. Column Wilcoxon-test presents the results of the test to evaluate the hypothesis presented in Section 4.2.1.

The tests reached small p-values to time and relevant tests data ($1.4 \times 10^{-9}$ and $1.0 \times 10^{-3}$, respectively). The results give us evidence that SafeRefactorImpact reduces time and generates more relevant tests than SafeRefactor. For number of methods and change coverage, the tests reached p-values of 0.13 and 0.19, respectively. The result indicates that SafeRefactorImpact identifies a number of methods greater than or similar to SafeRefactor and the change coverage of SafeRefactorImpact is less than or similar to the change coverage of SafeRefactor. Then, we execute another test (Wilcoxon-test) assuming a null hypothesis that the samples are equal to each metric. It reached a p-value of 0.27 for number of methods and 0.38 for change coverage. Then, we conclude that there is no statistical difference between the number of methods and change coverage of SafeRefactor and SafeRefactorImpact.

Table 7: Statistical analysis for design patterns data. Shapiro Test = analyze data normality; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final results of the statistical analysis; SR = SafeRefactor; SRI = SafeRefactorImpact.



| Data | Shapiro Test | | Wilcoxon-test | Result |
|---|---|---|---|---|
| | SR | SRI | | |
| Number of Methods | $7.0 \times 10^{-8}$ | $10^{-3}$ | 0.13 | There is no statistical difference |
| Time | $4.0 \times 10^{-4}$ | $1.7 \times 10^{-6}$ | $1.4 \times 10^{-9}$ | SRI < SR |
| Change Coverage | $6.0 \times 10^{-4}$ | $9.0 \times 10^{-3}$ | 0.19 | There is no statistical difference |
| Relevant Tests | $2.8 \times 10^{-6}$ | $2.2 \times 10^{-8}$ | $1.0 \times 10^{-3}$ | SRI > SR |

## *4.5. JML Compiler*

In this section, we evaluate SafeRefactor and SafeRefactorImpact by using them to test compilers. We check whether program compiled by two different compilers has the same behavior.

*Selection of subjects*

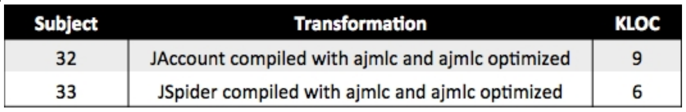
JML is a behavioral interface specification language used to specify contracts, such as pre and post conditions and invariants with annotations. The standard JML compiler (*jmlc*) reads a Java program annotated with JML and produces instrumented bytecode with additional code to check the program correctness against restrictions imposed by the JML specification.

Rêbelo et al. [18] propose a JML compiler (*ajmlc*) implemented using AspectJ to avoid using reflection, which was used in jmlc. In this way, they could use JML with Java ME applications, which do not support reflection. Later, they proposed an optimized version of this compiler (*ajmlc optimized*) [19]. They optimized the bytecode size and running time. Moreover, they used refactorings based on AspectJ programming laws [8] to reason about the compilation process.

We evaluated the JML compilers implemented in AspectJ (ajmlc 0.5 and ajmlc optimized 1.1) using SafeRefactor and SafeRefactorImpact. We use two Java programs annotated with JML (JAccounting and JSpider) as test inputs for the compilers. For each input, SafeRefactor and

SafeRefactorImpact compare the behavior of the programs yielded by these compilers. Rêbelo et al. [18] state that the *ajmlc* and *ajmlc optimized* are equivalent. Table 8 describes the subjects.

Table 8: Evaluation of two JML compilers. KLOC = non-blank, non-comment thousands of lines of code.



| Subject | Transformation | KLOC |
|---|---|---|
| 32 | JAccount compiled with ajmlc and ajmlc optimized | 9 |
| 33 | JSpider compiled with ajmlc and ajmlc optimized | 6 |

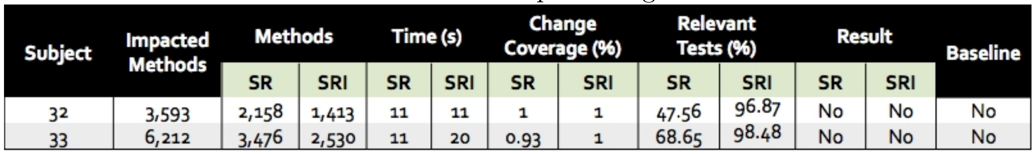
*Operation*

We compared SafeRefactor and SafeRefactorImpact using a time limit of 0.2s passed to Randoop. SafeRefactor and SafeRefactorImpact correctly identified behavioral changes in both transformations (Subjects 32 and 33). Both tools take almost the same time to evaluate the Subject 32. However, SafeRefactorImpact takes more time to evaluate Subject 33 since the change impact analysis is more expensive. Table 9 summarizes the results. Notice that the change impact analysis is useful to reduce at least 27% the set of methods passed to Randoop in SafeRefactorImpact. Moreover, both tools have similar low change coverage. Finally, SafeRefactorImpact generates more relevant test cases than SafeRefactor.

*Discussion*

Different from what Rêbelo et al. [18] expected, the programs compiled using the standard JML compiler (*jmlc*) and *ajmlc* are not equivalent. They must check invariants after creating an object, and before and after a method

Table 9: Results using a time limit of 0.2s. Impacted Methods = number of methods identified by Safira; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.



| Subject | Impacted Methods | Methods | | Time (s) | | Change Coverage (%) | | Relevant Tests (%) | | Result | | Baseline |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SR | SRI | SR | SRI | SR | SRI | SR | SRI | SR | SRI | |
| 32 | 3,593 | 2,158 | 1,413 | 11 | 11 | 1 | 1 | 47.56 | 96.87 | No | No | No |
| 33 | 6,212 | 3,476 | 2,530 | 11 | 20 | 0.93 | 1 | 68.65 | 98.48 | No | No | No |

call. By analyzing the tests reported by our tools, we detected that *ajmlc* checks invariants before each constructor. This led to false invariant violation warnings. For example, consider the following class specifying a person.

```
public class Person {
  private /*@ spec_public @*/ int height;
  //@invariant height > 0;
  //@pre i > 0;
  public Person(int i) {
    this.height = i;
  } ...
}
```

The class `Person` contains the field `height`, and a constructor that sets the height of each person. An invariant states that each person must have a height greater than 0. Moreover, the constructor of `Person` has an precondition specifying that the `i` parameter must be greater than 0. Now, suppose we would like to instantiate this class.

```
Person x = new Person(178);
```

The previous code compiled by *ajmlc optimized* is normally executed. However, it throws a warning due to postcondition violation when compiled by *ajmlc*. By analyzing the code generated by the compilers to check the constructor precondition, we notice that *ajmlc* implements this check in an intertype declaration.

```
before (Person p, int i): execution(Person.new()) {
  boolean b = p.checkPrePerson(i); ...
}
boolean Person.checkPrePerson(int i) {
  return (i > 0);
}
```

An advice invokes the method `checkPrePerson` before the execution of the constructor. Notice that this method belongs to `Person`. Therefore, by calling it, the invariants of this class will also be checked. However, since the constructor was not initialized so far, the `height` attribute is still `0`, leading to an invariant warning.

On the other hand, the *ajmlc optimized* changes the previous checking code by applying the *Inline method intertype within before-execution* refactoring [19]. Next we show part of the resulting code.

```
before (Person p, int i): execution(Person.new()) {
  boolean b = (i > 0);   ...
}
```

Notice that they removed the intertype declaration. Therefore, *ajmlc* contains a bug.

Our tools also detected a behavioral change during a postcondition evaluation of a method declared in JAccount. In a test case generated by our tools, the code compiled with *ajmlc optimized* throws the `JMLEvaluationError` exception, as expected. However, the code compiled with *ajmlc* throws `JMLInternalExceptionalPostconditionError`. These exceptions have different meaning. The former occurs when it throws an exception, such as `NullPointerException`, during the postcondition evaluation. The latter notifies an internal exceptional postcondition violation.

Both tools have a low change coverage. Although we found behavioral changes in Subjects 32 and 33, we may exercise more impacted methods by increasing the time limit. However, it is also important to mention that the test suite cannot exercise most impacted methods detected by SAFIRA since they do not belong to both versions of the program. Finally, SAFEREFACTORIMPACT generates more relevant test cases than SAFEREFACTOR.

## *4.6. Larger Case Studies*

In this section, we evaluate SAFEREFACTOR and SAFEREFACTORIMPACT to check the correctness of refactorings applied to larger OO and AO programs than previous sections.

### *Selection of subjects*

Taveira et al. [20] present two approaches to modularize exception handling mechanisms. They change an OO version into two equivalent ones: OO' (a class modularizes the exception handling code) and AO (an aspect modularizes it), as depicted by Figure 4.

Eight programmers working in pairs performed the changes. They relied on refactoring tools, pair review, and unit tests to assure behavior preservation. They refactored JHotDraw and CheckStylePlugin 4.2 using the proposed approach. Taveira et al. [20] establish that the OO, OO' and AO versions are equivalent. Subjects 34-37 are the OO to AO versions, and the OO' to AO versions (see Figure 4) of JHotDraw and CheckStylePlugin, respectively.

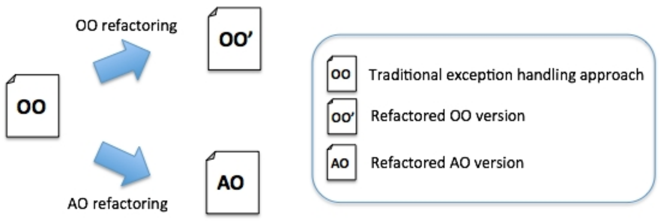
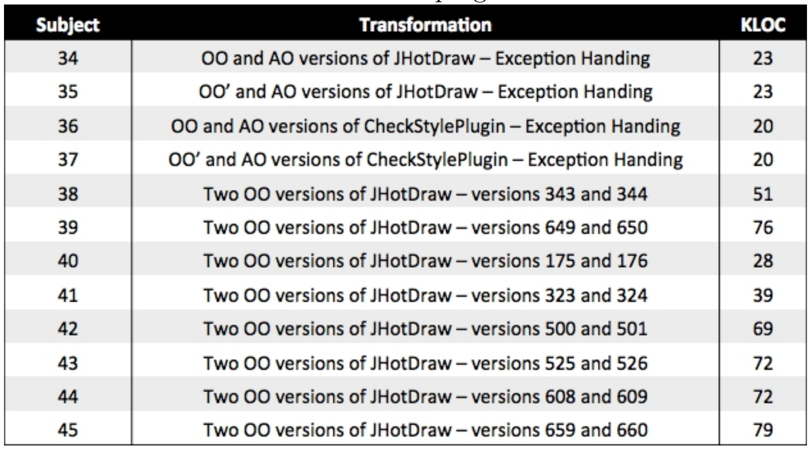

OO refactoring
OO'
OO
AO refactoring
AO
OO Traditional exception handling approach
OO' Refactored OO version
AO Refactored AO version

Figure 4: Two alternative refactorings to modularize exception handling code.

Soares et al. [21] used SafeRefactor to evaluate a number of transformations applied to JHotDraw from its repository. We randomly selected them. We evaluate some of them in Subjects 38-45. It is important to mention that we consider some transformations where SafeRefactor did not detect some behavioral changes identified by the manual inspection [33]. Table 10 describes the transformations evaluated.

Table 10: Evaluation of transformations applied to case studies. KLOC = non-blank, non-comment thousands of lines of code of the program before the transformation.



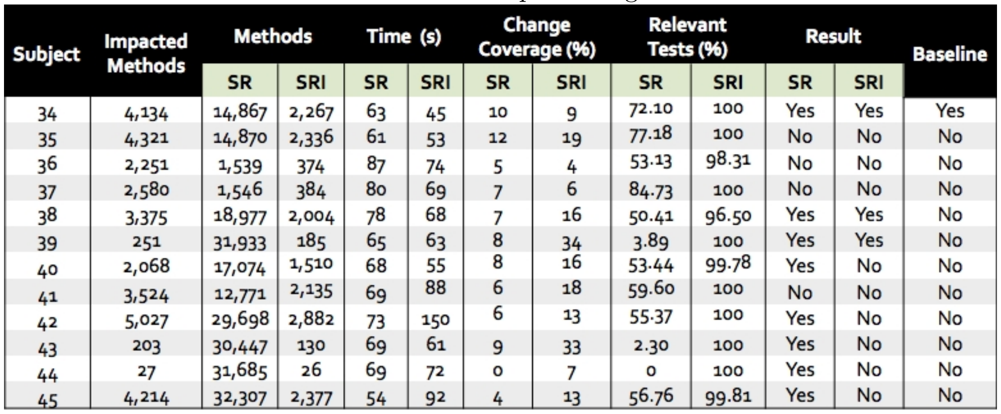
| Subject | Transformation | KLOC |
|---|---|---|
| 34 | OO and AO versions of JHotDraw – Exception Handing | 23 |
| 35 | OO' and AO versions of JHotDraw – Exception Handing | 23 |
| 36 | OO and AO versions of CheckStylePlugin – Exception Handing | 20 |
| 37 | OO' and AO versions of CheckStylePlugin – Exception Handing | 20 |
| 38 | Two OO versions of JHotDraw – versions 343 and 344 | 51 |
| 39 | Two OO versions of JHotDraw – versions 649 and 650 | 76 |
| 40 | Two OO versions of JHotDraw – versions 175 and 176 | 28 |
| 41 | Two OO versions of JHotDraw – versions 323 and 324 | 39 |
| 42 | Two OO versions of JHotDraw – versions 500 and 501 | 69 |
| 43 | Two OO versions of JHotDraw – versions 525 and 526 | 72 |
| 44 | Two OO versions of JHotDraw – versions 608 and 609 | 72 |
| 45 | Two OO versions of JHotDraw – versions 659 and 660 | 79 |

*Operation*

We compared SafeRefactor and SafeRefactorImpact using a time limit of 20s passed to Randoop. SafeRefactorImpact correctly evaluated all transformations but two (Subjects 38 and 39), while SafeRefactor correctly evaluated seven transformations. The change impact analysis is useful to reduce the set of methods passed to Randoop in SafeRefactorImpact.

The reduction ranges from 75% to 99% of the methods considered by SafeRefactor in our evaluation. Both tools took almost the same time to evaluate the subjects. As expected, SafeRefactorImpact has higher percentage of change coverage in nine subjects, since it focuses on testing the methods impacted by the change. In the other three subjects, they have almost the same change coverage. SafeRefactorImpact generates at least 95% of relevant tests. In Subjects 39, 43 and 44, SafeRefactor generates less than 10% of relevant tests since it passes more than 30,000 methods to Randoop generate tests. Table 11 summarizes the results.

Table 11: Results using a time limit of 20s. Impacted Methods = number of methods identified by Safira; Methods = number of methods passed to Randoop to generate tests; Time = the total time of the analysis in seconds; Change Coverage = the percentage of impacted methods covered; Relevant Tests = the percentage of relevant tests; Result = it states whether the transformation is behavior preserving.



| Subject | Impacted Methods | Methods | | Time (s) | | Change Coverage (%) | | Relevant Tests (%) | | Result | | Baseline |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SR | SRI | SR | SRI | SR | SRI | SR | SRI | SR | SRI | |
| 34 | 4,134 | 14,867 | 2,267 | 63 | 45 | 10 | 9 | 72.10 | 100 | Yes | Yes | Yes |
| 35 | 4,321 | 14,870 | 2,336 | 61 | 53 | 12 | 19 | 77.18 | 100 | No | No | No |
| 36 | 2,251 | 1,539 | 374 | 87 | 74 | 5 | 4 | 53.13 | 98.31 | No | No | No |
| 37 | 2,580 | 1,546 | 384 | 80 | 69 | 7 | 6 | 84.73 | 100 | No | No | No |
| 38 | 3,375 | 18,977 | 2,004 | 78 | 68 | 7 | 16 | 50.41 | 96.50 | Yes | Yes | No |
| 39 | 251 | 31,933 | 185 | 65 | 63 | 8 | 34 | 3.89 | 100 | Yes | Yes | No |
| 40 | 2,068 | 17,074 | 1,510 | 68 | 55 | 8 | 16 | 53.44 | 99.78 | Yes | No | No |
| 41 | 3,524 | 12,771 | 2,135 | 69 | 88 | 6 | 18 | 59.60 | 100 | No | No | No |
| 42 | 5,027 | 29,698 | 2,882 | 73 | 150 | 6 | 13 | 55.37 | 100 | Yes | No | No |
| 43 | 203 | 30,447 | 130 | 69 | 61 | 9 | 33 | 2.30 | 100 | Yes | No | No |
| 44 | 27 | 31,685 | 26 | 69 | 72 | 0 | 7 | 0 | 100 | Yes | No | No |
| 45 | 4,214 | 32,307 | 2,377 | 54 | 92 | 4 | 13 | 56.76 | 99.81 | Yes | No | No |

*Discussion*

In Subject 35, we evaluated the OO' and AO versions, and both tools also detected this behavioral change. In our previous work [11], we also found a behavioral change in the OO and OO' versions of JHotDraw. To perform the OO refactoring, developers extracted the code inside the `try`, `catch`, and `finally` blocks to methods in specific classes that handle exceptions. They refactored some classes that implement `Serializable`.

```
class A implements Serializable {
  Object clone() {
    try { ... }
    catch (IOException e) { ... }
  }
}
```

Developers changed the `clone` method and introduced the handler attribute to handle exceptions. However, they forgot to serialize this new attribute.

```
class A implements Serializable {
  ExceptionHander handler; ...
  Object clone() {
    try { ... }
    catch (IOException e) {
      handler.handle(e);
    }
  }
}
class ExceptionHandler { ... }
```

Thus, the program throws an exception when the method `clone` tries to serialize the object. Therefore, they introduced a bug in the code. On the other hand, developers extracted the exception handling code to aspects in the AO version. Since in this version there was no need to introduce new fields in the classes to handle exceptions, this problem did not happen. They used tools and a test suite to guarantee behavior preservation. However, as we previously mention, there is no good tool support for refactoring AO code, so we can introduce behavioral changes even when applying small changes. Hence, this may be the cause of the unintentional behavioral change introduced in the OO' version.

SafeRefactor and SafeRefactorImpact detect behavioral changes between the OO and AO versions of CheckStylePlugin (Subjects 36 and 37) using a time limit of 20s. Next, we describe the two behavioral changes found. In the class `FileMatchPattern`, the method `setMatchPattern` contains a `try-catch` block that catches a `PatternSyntaxException` and throw an `CheckStylePluginException`. Developers removed this `try-catch` block and added an aspect to transform this exception into a `SoftException`, a kind of `RuntimeException`. `SoftException` is then re-thrown as `CheckStylePluginException`. However, `PatternSyntaxException` is already a subclass of `RuntimeException`, and thus it is not softened by the aspects. In this way, after the transformation, `PatternSyntaxException` is not caught and re-thrown as `CheckStylePluginException`, changing the behavior of the program.

We found a second behavioral change in the class `ConfigurationType`. Developers removed `try-catch` blocks that catch `IOException` and re-throw `CheckStylePluginException`, and added an aspect to transform this exception into a `SoftException`. An aspect should catch this exception and re-throw as `CheckStylePluginException`. However, after the transformation, `SoftException` is not caught, changing the behavior of the program. It seems that developers forgot to implement the last part of the transformation (catch the `SoftException`). These results corroborate with the results found in a previous study [34], suggesting that exception handling code in AO systems without good tool support may be error-prone.

In Subjects 38 and 39, both tools do not identify behavioral changes using a time limit of 20s. Randoop does not generate tests that exercise the impacted methods that change behavior using this time limit. Different from SafeRefactor, SafeRefactorImpact identifies the behavioral changes in both subjects using a time limit of 120s, since it reduces by more than 90% the number of methods passed to Randoop to generate tests.

In our previous work [21], we evaluate Subjects 40-45 using SafeRefactor and a manual inspection performed by experts [33]. SafeRefactor does not identify the behavioral changes using a time limit of 20s in Subjects 40, 42, 43, 44 and 45 different from SafeRefactorImpact. However, it detects three of them (Subjects 40, 43 and 45) using a time limit of 120s. Both manual inspection [33] and SafeRefactor classified Subject 42 as behavior preserving. However, SafeRefactorImpact identified a previously undetected behavioral change in Subject 42. Next we illustrate part of the original program of Subject 42. It specifies a class declaring the method `getAttribute`, which returns an object. Notice that if the required object does not exist in `attributes`, the method yields `null`.

```
class A {
  HashMap<AttributeKey, Object> attributes =
    new HashMap<AttributeKey,Object>();
  public Object getAttribute(AttributeKey name) {
    return attributes.get(name);
  }
}
```

In the modified program presented next, the method `getAttribute` calls a method `get` of the class *AttributeKey<T>* passing as a parameter the field `attributes`. Notice that the method checks if the required object exists in the Map. If it does not exist, the method yields a default value instead of `null` in the original program.

```
class A {
  HashMap<AttributeKey, Object> attributes =
    new HashMap<AttributeKey,Object>();
  public <T> T getAttribute(AttributeKey<T> key) {
    return key.get(attributes);
  }
}

class AttributeKey<T> {
  public T get(Map<AttributeKey, Object> a) {
    T value = (T) a.get(this);
    return (value == null && !isNullValueAllowed) ?
       defaultValue : value;
  }
}
```

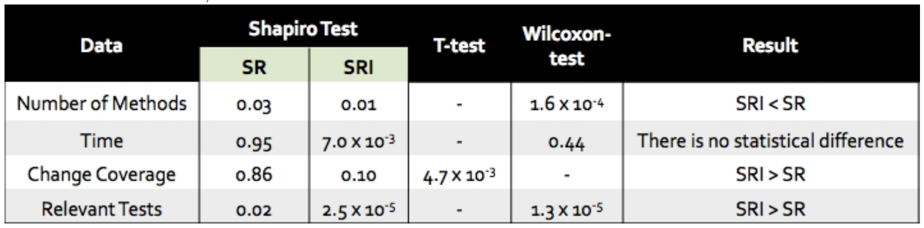
SafeRefactor does not identify this behavioral change because Randoop does not generate tests to expose them using the time limit of 120s, since the number of methods to test is much greater (90%) than in SafeRefactorImpact. It is also important to notice that finding behavioral changes is not an easy task, even when using a well defined manual inspection conducted by experts [33, 35]. It is a time consuming and error prone activity to manually evaluate whether a transformation is behavior preserving in larger programs.

Both tools have a low change coverage. Randoop does not generate test cases to many methods because they depend on classes from libraries that are not passed as parameter. Moreover, some methods have parameters, such as arrays, that Randoop does not handle well when generating tests. Finally, there are some added and removed methods that are not common to both versions of the program. In some subjects, SafeRefactorImpact does not yield 100% of relevant tests since it may throw an exception before or while executing the impacted method in a test case. Finally, SafeRefactorImpact is slower than or similar to SafeRefactor to evaluate these subjects, because the change impact analysis performed by SafeRefactorImpact is more expensive in larger programs than the analysis of SafeRefactor. However, it detects some behavioral changes undetected by SafeRefactor.

Table 12 describes the statistical analysis results. Column Shapiro Test indicates the Shapiro–Wilk test results. Notice that only the change coverage data of SafeRefactor and SafeRefactorImpact are normal. Columns T-test and Wilcoxon-test present the results of the tests to evaluate the hypothesis presented in Section 4.2.1.

Due to non-normality of data, we use Wilcoxon-test for number of methods, time, and percentage of relevant tests. We use T-test for change coverage due to data normality. The tests reached small p-values to number of methods, change coverage and relevant tests ($1.6 \times 10^{-4}$, $4.7 \times 10^{-3}$, and $1.3 \times 10^{-5}$, respectively) The results give us evidence that SafeRefactorImpact identifies less methods to generate tests, has a better change coverage, and generates more relevant tests than SafeRefactor. The test reached a p-value of 0.44 indicating that SafeRefactorImpact is slower than or similar to SafeRefactor. Then, we execute another test (Wilcoxon-test) assuming a null hypothesis that the time of both tools are equal. It reached a p-value of 0.88, which indicates that there is no statistical difference between the time to evaluate a transformation between SafeRefactor and SafeRefactorImpact.

Table 12: Statistical analysis for larger subjects data. Shapiro Test = analyze data normality; T-test = evaluate hypothesis test when data are normal; Wilcoxon-test = evaluate hypothesis test when data are non-normal; Result = final result of the statistical analysis; SR = SafeRefactor; SRI = SafeRefactorImpact.



| Data | Shapiro Test | | T-test | Wilcoxon-test | Result |
|---|---|---|---|---|---|
| | SR | SRI | | | |
| Number of Methods | 0.03 | 0.01 | - | $1.6 \times 10^{-4}$ | SRI < SR |
| Time | 0.95 | $7.0 \times 10^{-3}$ | - | 0.44 | There is no statistical difference |
| Change Coverage | 0.86 | 0.10 | $4.7 \times 10^{-3}$ | - | SRI > SR |
| Relevant Tests | 0.02 | $2.5 \times 10^{-5}$ | - | $1.3 \times 10^{-5}$ | SRI > SR |

## *4.7. Threats to Validity*

There are some limitations to this study. Next we describe some threats to the validity of our evaluation.

### *4.7.1. Construct validity*

We created the baseline by comparing the approaches' results, since we did not know beforehand which versions contain behavior-preserving transformations to evaluate the correctness of the results of each approach.

With respect to SafeRefactor and SafeRefactorImpact, they do not evaluate the developer's intention to refactor, but whether a transformation changes behavior. Moreover, in the closed world assumption, we have to use the test suite provided by the program that is being refactored. SafeRefactorImpact follows an open world assumption, in which every public method can be a potential target for the test suite generated by Randoop. Randoop may generate a test case that exposes a behavioral change. However, the test case may show an invalid scenario according to the software domain.

Our change coverage and the percentage of relevant test metrics are based on the impacted methods identified by Safira. However, Safira may fail to identify some impacted methods, or include a method that does not change behavior. For example, it may not include a method since it does not perform data flow analysis.

Safira does not analyze anonymous classes. It does not identify all impacted methods related to them. Moreover, Safira does not perform data flow analysis. Due to this limitation, it does not identify the behavioral change in Subject 2. Although it does not implement data flow analysis, SafeRefactorImpact has a parameter that allows us to include all common getter methods in the test generation. However, this may decrease its performance, and require to increase the time limit.

### *4.7.2. Internal validity*

Another threat is related to the time limit to generate the tests. The time limits used in SafeRefactor and SafeRefactorImpact may have influence on the detection of behavioral changes. We used the default values for most of Randoop parameters. By changing them, we may improve SafeRefactor and SafeRefactorImpact results. Moreover, since Randoop randomly generates a test suite, there might be different results each time we run the tool. We ran the experiment only once. Due to the randomness nature of the tests, different executions may have different results. As future work, we plan to execute the tools multiple times to improve the confidence on the results.

Finally, compilers may have introduced behavioral changes during the optimization process [36]. Since SafeRefactorImpact analyzes the Java bytecode, this may have an influence on the results if the compilers have bugs.

### *4.7.3. External validity*

To mitigate threats to external validity, we evaluated different kinds of software, such as a GUI application (JHotDraw) and an Eclipse Plugin (CheckStylePlugin), ranging from few lines of codes to thousands of lines of code. We also evaluate a number of different refactorings targeting different OO and AO constructs.

Randoop does not deal with concurrency. In those situations, SafeRefactor and SafeRefactorImpact may yield non-deterministic results. Also, they do not take into account characteristics of some specific domains. For instance, currently, they do not detect the difference in the standard output (*System.out.println*) message. Neither could the tool generate tests that exercise some changes related to the graphical interface (GUI) of JHotDraw.

## *4.8. Answer to the research questions*

From the evaluation results, we make the following observations:

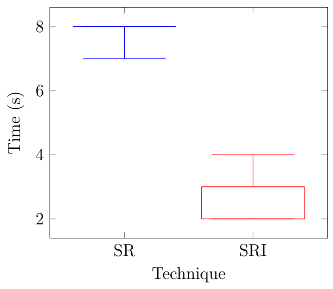
- **Q1**. Do SafeRefactorImpact and SafeRefactor detect the same behavioral changes?

  No. SafeRefactorImpact does not identify the behavioral change in Subject 2 due to a limitation in Safira. If we pass all getter methods as parameter in the test generation, SafeRefactorImpact detects it. Moreover, it does not detect behavioral changes in Subjects 38 and 39 using a time limit of 20s. If we increase the time limit to 120s, it detects the behavioral change different from SafeRefactor. On the other hand, SafeRefactorImpact detects behavioral changes in Subjects 22, 40, 42, 43, 44 and 45 that SafeRefactor does not identify them using a time limit of 20s. SafeRefactor detects the behavioral changes in Subjects 40, 43 and 45 using a time limit of 120s. SafeRefactorImpact finds a behavioral change in Subject 42 undetected by SafeRefactor and a well defined manual inspection conducted by experts.

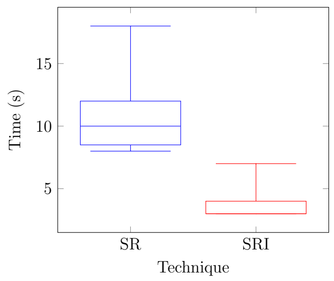
- **Q2**. Is SafeRefactorImpact faster than SafeRefactor to evaluate a transformation?

In the transformations applied to small programs, SafeRefactorImpact is faster than SafeRefactor. However, both tools take almost the same time to evaluate transformations applied to larger programs. Figure 5 illustrates the distribution of the total time to evaluate transformations by SafeRefactor and SafeRefactorImpact in the subjects of defective refactorings, designs patterns, and larger case studies.

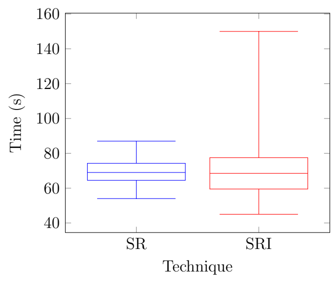
Figure 5: Distribution of the total time to evaluate transformations by SafeRefactor and SafeRefactorImpact.



(a) Defective Refactorings



(b) Design Patterns



(c) Real Subjects

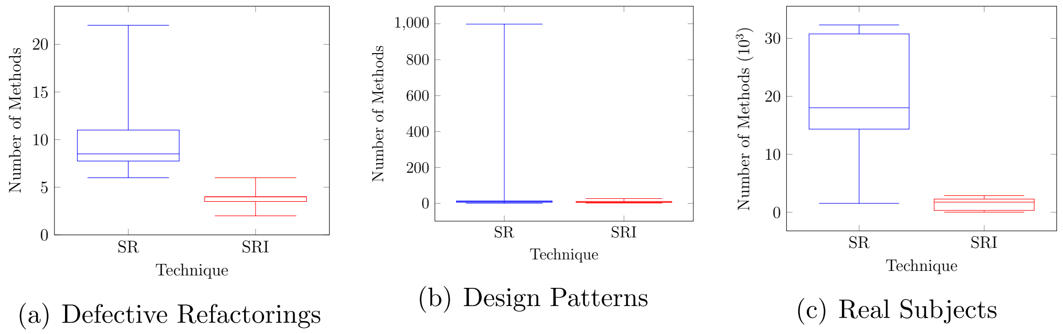
- **Q3**. Does SafeRefactorImpact consider less methods in common to generate tests than SafeRefactor?

  Yes. SafeRefactorImpact considers less methods in common to generate tests in all subjects except in some subjects of design patterns, because in these subjects SafeRefactor does not consider some impacted methods. In larger subjects, SafeRefactorImpact reduces at least 75% of the methods to test. Figure 6 illustrates the distribution of the number of methods identified by SafeRefactor and SafeRefactorImpact to generate tests, in the subjects of defective refactorings, designs patterns, and larger case studies.

- **Q4**. Does SafeRefactorImpact generate a test suite with better change coverage than SafeRefactor?

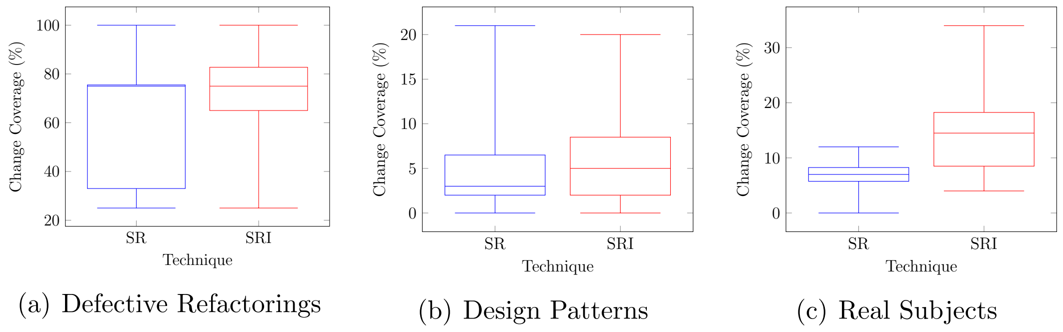
  The test cases generated by SafeRefactorImpact increase the change coverage in larger subjects. For small ones, there is no significant difference, but in most of the subjects, it is similar or better than

Figure 6: Distribution of the number of methods identified by SafeRefactor and SafeRefactorImpact to generate tests.



SafeRefactor. Figure 7 illustrates the distribution of the change coverage of the tests generated by SafeRefactor and SafeRefactorImpact in the subjects of defective refactorings, designs patterns, and larger case studies.

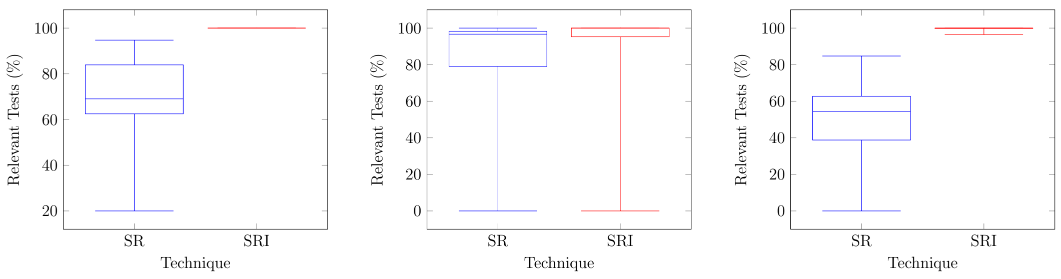
Figure 7: Distribution of the change coverage of the tests generated by SafeRefactor and SafeRefactorImpact.



- **Q5**. Does SafeRefactorImpact use a test suite to evaluate a transformation with more relevant test cases than SafeRefactor?

  Yes. SafeRefactorImpact generates more relevant tests in all subjects. Almost 90% of test cases generated by SafeRefactorImpact are relevant to evaluate the change. It only generates test cases that exercise an impacted method. Some test cases are not relevant because they throw an exception before or while executing an impacted method. Figure 8 illustrates the distribution of the percentage of relevant tests generated by SafeRefactor and SafeRefactorImpact in the subjects of defective refactorings, designs patterns, and larger case studies.

Figure 8: Distribution of the percentage of relevant tests generated by SafeRefactor and SafeRefactorImpact.



(a) Defective Refactorings (b) Design Patterns (c) Real Subjects

## 5. Related Work

In this section, we relate our work to a number of approaches proposed for refactoring OO (Section 5.1) and AO programs (Section 5.2), change impact analysis (Section 5.3) and detecting behavioral changes (Section 5.4).

### *5.1. Refactoring Object-Oriented Programs*

Preconditions are a key concept of research studies on the correctness of refactorings. Opdyke [1] proposed a number of refactoring preconditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these preconditions. In fact, later, Tokuda and Batory [37] showed that Opdyke's preconditions were not sufficient to ensure behavior preservation. Roberts [38] automated the basic refactorings proposed by Opdyke.

Kim et al. [39] conducted surveys, interviews, and quantitative analysis to evaluate refactoring challenges and benefits at Microsoft. Although participants of the survey mentioned that refactorings help on improving maintainability, 77% of them mentioned regression bugs as risks for applying refactorings. Also, except for the rename refactoring, most of the participants mentioned that they manually perform refactorings, despite the awareness of automated tools. This study indicates that tool support for refactoring should go beyond automated transformations. For example, they need to use a better tool support for checking behavior preservation correctness, as we propose in this article.

Rachatasumrit and Kim [14] 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 investigate 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. Although our tool has a low change coverage in larger subjects, it focuses only on generating tests to run on both versions of the program. There are a number of added or removed methods that are not exercised indirectly. So, it cannot generate tests for them.

Steimann and Thies [40] showed that by changing access modifiers (`public`, `protected`, `package`, `private`) in Java one can introduce compilation errors and behavioral changes. They propose a constraint-based approach to specify Java accessibility, which favors checking refactoring preconditions and computing the changes of access modifiers needed to preserve the program behavior. Such specialized approach is useful for detecting bugs regarding accessibility-related properties. On the other hand, our approach is general enough for detecting bugs with respect to other OO and AO constructs.

Tip et al. [41] proposed an approach that uses type constraints to verify preconditions of those refactorings, determining which part of the code they may modify. Using type constraints, they also proposed the refactoring Infer Generic Type Arguments [42], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [43]. Eclipse implemented these refactorings. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have preconditions related to other constructs. Our tool may be helpful in those situations.

Borba et al. [44] proposed a set of refactorings for a subset of Java with copy semantics (ROOL). They prove the refactoring correctness based on a formal semantics. Silva et al. [45] proposed a set of behavior preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. Yet, they have not considered all Java constructs, such as overloading and field hiding. SafeRefactorImpact may be useful when their work may not be applied.

Schäfer et al. [46] 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.

Overbey and Johnson [47] proposed a technique to check for behavior preservation. They implement it in a library containing preconditions for the most common refactorings. Refactoring engines for different languages can use their library to check refactoring preconditions. The preservation-checking algorithm is based on exploiting an isomorphism between graph nodes and textual intervals. They evaluate their technique for 18 refactorings in refactoring engines for Fortran 95, PHP 5 and BC. In our approach, we use SafeRefactorImpact to evaluate whether any transformation is behavior-preserving. Proving refactorings with respect to a formal semantics constitutes a challenge [12].

Soares et al. [48] proposed a technique to identify overly strong conditions based on differential testing [49]. If a tool correctly applies a refactoring according to SafeRefactor and another tool rejects the same transformation, the latter has an overly strong condition. In a sample of 42,774 programs generated by JDolly, they evaluated 27 refactorings of Eclipse, NetBeans and JastAdd Refactoring Tools (JRRT) [6], and found 17 and 7 types of overly strong conditions in Eclipse and JRRT, respectively. This approach is useful for detecting whether the set of refactoring preconditions is minimal. Later, Soares et al. [10] introduced a technique to test refactoring tools and found more than 100 bugs in the best academic (JRRT) and commercial Java refactoring implementations (Eclipse and NetBeans). This approach is based on a program generator (JDolly) and SafeRefactor. In this work, we extend SafeRefactor to consider AO constructs, and use change impact analysis to generate tests only for the methods impacted by a transformation. As a future work, we intend to use SafeRefactorImpact in this approach.

## *5.2. Refactoring Aspect-Oriented Programs*

Monteiro and Fernandes [7] proposed a catalog of 27 AO refactorings [2]. They can be useful for implementing aspect-aware refactoring tools. However, they do not prove their soundness. We can apply their refactorings and use SafeRefactorImpact to improve confidence that the transformation is correct.

Wloka et al. [9] proposed a tool support for extending currently OO refactoring implementations for considering aspects. They employ change impact analysis to identify pointcuts impacted by a transformation that can change the program behavior. The tool can change pointcuts to preserve program behavior in some cases. SafeRefactorImpact does not apply a transformation to a program. It only evaluates whether a transformation preserves behavior. Safira also considers aspects during the analysis. Moreover, SafeRefactorImpact evaluates any kind of transformation, while their tool evaluates only some Java refactorings, such as rename, move, extract and inline.

Binkley et al. [50, 51] presented a human guided automated approach to refactor OO to AO program. They implement six kinds of refactorings. Each refactoring defines a set of preconditions to guarantee behavior preservation. They refactored four OO real systems to modularize it in aspects (JHotDraw, PetStore, JSpider and JAccouting). Hannemann et al. [52] introduced a role-based refactoring approach to help programmers modularize crosscutting concerns into aspects. Malta and Valente [53] presented a collection of transformations used to enable the extraction of crosscutting statements to aspects. Each refactoring defines a set of preconditions. Their work may contribute for improving tool support for applying refactorings to AO programs. However, they do not prove them sound with respect to a formal semantics. Developers can use our tool together with their approaches to improve confidence that the transformation preserves behavior. Moreover, SafeRefactorImpact can evaluate any kind of transformation.

Yokomori et al. [54] analyzed two software applications that have been refactored into aspects (JHotDraw and Berkeley DB) to determine circumstances when such activities are effective at reducing component relationships and when they are not. They found that AO refactoring is successful in improving the modularity and complexity of the base code. In our work, we propose a tool based on change impact analysis to improve confidence that a transformation preserves behavior. SafeRefactorImpact does not evaluate whether the resulting program improves the quality of the original program.

Hannemann and Kiczales [17] implemented 23 design patterns [32] in Java and AspectJ. The study concludes that some patterns are better implemented using OO constructs and others using AO constructs. Taveira et al. [20] modularized exception handling in OO and AO code by using test suite and pair programming. The study indicates that the AO version promotes reuse of exception handling code. We used SafeRefactorImpact to analyze some transformations they evaluated, and found some behavioral changes that developers were unaware. SafeRefactorImpact does not evaluate whether the resulting program improves the quality of the original program.

Van Deursen et al. [55] used an existing well-designed open-source system (JHotDraw) and modified it to an equivalent AO version (AJHotDraw). In this article, we analyzed some transformations applied to JHotDraw collected from its SVN repository history and from studies that aimed to modularize the exception handling mechanism.

Cole and Borba [8] formally specified AO programming laws (each law defines a bidirectional semantics-preserving transformation) for AspectJ. By composing them, they derived AspectJ refactorings. Each law formally states preconditions. They proved one of them sound with respect to a formal semantics for a subset of Java and AspectJ [56]. They can be useful for implementing aspect-aware refactoring tools. However, they did not consider all AspectJ constructs and their catalog is incomplete. In those situations, we can use our tool.

## *5.3. Change Impact Analysis*

Law and Rothermel [57] proposed an approach based on static and dynamic partitioning and recursive algorithms of calls graphs to identify methods impacted by a change. Different from Safira, the analysis estimates the change impact before applying the transformation. Our change impact analyzer performs static analysis in any kind of transformation applied to Java or AspectJ programs. In addition, it does not need additional information to evaluate a transformation.

Chianti [23] is a change impact analyzer tool for Java. Based on a test suite and the changes applied to a program, it decomposes the change into atomic changes and generates a dependency graph. The tool indicates the test cases that are impacted by the change. Only these test cases need to be executed again. Zhang et al. [24] proposed a change impact analyzer tool (FaultTracer) that improves Chianti by refining the dependencies between the atomic changes, and adding more rules to calculate the change impact. Both tools receive two program versions as parameters, and decompose the change into small-grained transformations, similar to SAFIRA. However, different from SAFIRA, Chianti and FaultTracer depend on a test suite to assess the change impact. They execute the test cases, and identify the impacted test cases that must be executed again based on the call graphs. SAFEREFACTORIMPACT automatically generates test cases for the methods impacted by a transformation.

Kung et al. [58] presented an approach to identify impacted classes due to structural changes in library classes of OO languages. It is based on a reverse engineering approach that extracts information from the library classes and their relationships. This information is represented in dependency graphs used to automatically identify changes and their effects. Li and Offut [59] conducted a study to evaluate how changes applied to OO programs can affect program classes. They proposed an algorithm that computes the transitive closure of the program dependency graph. They analyze changes in a program to identify impacted classes. SAFIRA also identifies the methods impacted by a change.

Wloka et al. [60] proposed a tool called JUnitMX. It uses a change impact analysis tool to yield all entities impacted by a transformation. After executing a test suite, it indicates whether the test suite exercises all entities impacted by a transformation. If all test cases pass but they do not cover all entities impacted by a transformation, the tool yields a yellow bar. The tool yields a green bar if and only if the test cases pass and exercise all entities impacted by a transformation. Otherwise, it yields a red bar. As a future work, we intend to include this functionality in SAFEREFACTORIMPACT.

## *5.4. Detecting Behavioral Changes*

Some tools statically check whether a transformation preserves behavior. For instance, Eclipse JDT and NetBeans implement a number of refactorings. Each refactoring may contain a number of preconditions to ensure behavioral preservation [1]. Later, JastAdd Refactoring Tools (JRRT) [61, 62, 6] implemented a number of refactorings by using formal techniques. We evaluated 29 refactoring implementations of Eclipse JDT, NetBeans and JRRT using SAFEREFACTOR and found 63 bugs related to behavioral changes [10]. Defining refactoring preconditions is a nontrivial task, which the literature has treated in different ways [44, 63, 41, 45, 61, 62, 6, 40]. These include analyses of some of the various aspects of a language, such as: accessibility, types, name binding, data flow, and control flow. However, proving refactoring correctness for the entire language constitutes a challenge [12]. In our approach, instead of static analysis, we use dynamic analysis to evaluate whether a transformation preserves behavior. Moreover, we evaluate any kind of transformation different from previous approaches.

Daniel et al. [64] proposed an approach for automated testing refactoring engines using an automatic program generator (ASTGen). 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, and the 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. For example, consider the classes $A$, $B$ (subclass of $A$) and $C$ (subclass of $B$) presented in Listing 8. The class $A$ declares the field $k$, which is initialized with 10. The class $C$ has the field $k$ hiding $A.k$, which is initialized with 20, and the method *test* calling *super.k*. This method yields 10. By using Eclipse JDT 4.3 to apply the Pull Up Field refactoring to $C.k$ moving it to class $B$, it yields the program presented in Listing 9. This transformation introduces a behavioral change: the method *test* now calls $B.k$ yielding 20 instead of 10 in the initial program. Applying the Push Down Field refactoring to $B.k$ in the modified program presented in Listing 9, the resulted program is equals to the initial program presented in Listing 8. So, their oracle does not detect this behavioral change, different from SafeRefactorImpact.

They evaluated the technique by testing 21 refactorings, and identified 21 bugs in Eclipse JDT and 24 in NetBeans [64]. 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. Moreover, Gligoric et al. [65] evolved ASTGen and proposed UDITA. They found four new compilation error bugs in six refactorings (two in Eclipse JDT and two in NetBeans). Later, Gligoric et al. [66] evolved the technique and found a number of bugs in Java and C refactorings of Eclipse JDT and CDT, NetBeans and IntelliJ. However, they did not find bugs related to behavioral preservation. We proposed a similar approach to test refactoring engines using

Figure 9: Pulling up a field introduces a behavioral change in Eclipse.

Listing 8: Original Program

```
public class A {
  public int k = 10;
}
public class B extends A {
}
public class C extends B {
  public int k = 20;
  public int test() {
    return super.k;
  }
}
```

Listing 9: Modified Program

```
public class A {
  public int k = 10;
}
public class B extends A {
  public int k = 20;
}
public class C extends B {
  public int test() {
    return super.k;
  }
}
```

SafeRefactor as an oracle to detect behavioral changes [10]. While the oracles of previous approaches can only syntactically compare the programs to detect behavioral changes, SafeRefactor generates tests that compare program behavior. We automatically found 63 bugs related to behavioral changes in Eclipse JDT, NetBeans and JRRT.

Lahiri et al. [67] proposed a tool (SymDiff) for identifying behavioral changes. The tool translates the program to an intermediate language (Boogie). For each pair of procedures (before and after the change), it statically checks partial equivalence by using a program verifier for Boogie that exploits Satisfiability Modulo Theories solver Z3. They use the Z3 theorem prover to verify loop-free and call-free fragments. The precision of the tools relies on the soundness of the translator that translates the target language to Boogie. They have a front-end for C programs. In our work, we automatically generate a test suite to compare the behavior of two Java and AspectJ programs. We only generate tests for the entities impacted by a transformation. When SymDiff cannot prove two procedures as equivalent, it generates a counterexample describing the program trace. In our tool, we yield a test case to the user exposing the behavioral change.

Raghavan et al. [68] presented an automated tool called Dex for analyzing syntactic and semantic changes in C programs. It creates an abstract semantic graph (ASG) representation of each program version, and then applies a graph differencing algorithm to the resulting pair of ASGs. It consists of describing how to convert the ASG for the original version into the ASG for the modified version by matching, inserting, deleting, updating, or moving nodes. Safira decomposes a coarse-grained transformation into smaller ones, and calculates the impacted methods. Then, SafeRefactorImpact generates tests only for the entities impacted by the transformation.

Person et al. [69] presented a symbolic execution technique to characterize the impact of program changes in terms of behavioral changes. They defined two equivalence notions for programs: functional equivalence (the versions have the same black-box behavior), and partition-effects equivalence (the versions have corresponding sets of paths through their implementations). SafeRefactorImpact identifies a behavioral change when at least one test case passes in a program version but it fails in the other one.

Lahiri et al. [70] proposed an approach to statically compare different versions of a program with respect to a set of assertions. They use previous versions of a program to reduce the cost of program analysis. They include this approach in SymDiff and evaluate it. SafeRefactorImpact automatically generates a test suite for the entities impacted by the transformation to compare behavior of two Java and AspectJ programs. It does not use previous versions of the program to perform analysis.

## 6. Conclusions

In this article, we propose a tool (SafeRefactorImpact) for checking whether an OO or AO transformation is behavior preserving (Section 3). Moreover, it generates a test suite only for the methods impacted by the transformation. It performs a change impact analysis using Safira to identify the impacted methods. We compared SafeRefactor and SafeRefactorImpact in 45 transformations applied to programs with different sizes (10 LOC to 79 KLOC). The transformations change a number of OO and AO constructs (classes, methods, fields, inheritance, overloading, overriding, aspects, intertype declarations, pointcuts, advices). We found that SafeRefactorImpact detects behavioral changes undetected by SafeRefactor. It has a better performance when analyzing transformations applied to small programs. Moreover, it significantly reduces the number of methods passed to Randoop. So, it is less dependent to the time limit passed to Randoop to generate tests. Finally, it has a better change coverage in larger subjects and generates more relevant tests.

The goal of SafeRefactorImpact is to exercise only the entities impacted by the transformation to avoid the problems found by Rachatasumrit and Kim [14], which state that refactorings are not well tested. SafeRefactorImpact generates more relevant tests in all subjects. Although the change coverage is low in some subjects, it is important to remember that SafeRefactorImpact only generates tests for the impacted methods in common for both versions. Our goal is to compare two program versions with respect to the same test suite, differently from Rachatasumrit and Kim [14]. A number of impacted methods do not belong to both program versions and are not indirectly exercised in the subjects evaluated by our work.

SafeRefactorImpact detects some non-behavior-preserving transformations that SafeRefactor does not detect. In some cases, even developers were not aware of the behavioral changes. Developers used refactoring tools and test suite to improve confidence that the transformations were correct. However, the Java/AspectJ semantics is nontrivial, which imposes challenges in checking and performing refactorings. For instance, pointcuts may use wildcards making difficult to check preconditions. A small transformation can have an impact on a number of different parts of the program. Therefore, it is not simple to apply them without a good tool support.

The change impact analysis is useful because it reduced the number of methods passed for Randoop in most of the subjects. In some cases, some methods dealing with user interface and file manipulation are not passed to Randoop. The current version of Randoop does not work well with them [21]. By handling less methods and focusing on the impacted ones, Randoop generates some tests that showed behavioral changes previously undetected by SafeRefactor.

As future work, we plan to evaluate our tool with more case studies. Moreover, we intend to improve the analysis performance and include a data flow analysis in Safira. Additionally, we aim at defining other small-grained transformations to reduce the set of impacted methods identified by Safira. We are also interested in evaluating other automatic test suite generators in SafeRefactorImpact, such as EvoSuite [71] and Testful [72]. Finally, we intend to create an Eclipse plugin for SafeRefactorImpact.

**Acknowledgment**

We gratefully thank Tiago Massoni, Sérgio Soares, Márcio Ribeiro, Henrique Rêbelo, Fernando Castor, Roberta Coelho, Christiano Braga, José Fiadeiro, and the anonymous referees from the Brazilian Symposium on Programming Languages and Science of Computer Programming for their useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 304470/2010-4, 480160/2011-2, 484860/2011-9, and 477943/2013-6.

## References

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

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

[3] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2004) 126–139.

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

[5] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Company, 2003.

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

[7] M. Monteiro, J. Fernandes, Towards a catalog of Aspect-Oriented refactorings, in: Proceedings of the 4th Aspect-Oriented Software Development, AOSD '05, ACM, New York, NY, USA, 2005, pp. 111–122.

[8] L. Cole, P. Borba, Deriving refactorings for AspectJ, in: Proceedings of the 4th Aspect-Oriented Software Development, AOSD '05, ACM, New York, NY, USA, 2005, pp. 123–134.

[9] J. Wloka, R. Hirschfeld, J. Hänsel, Tool-supported refactoring of Aspect-Oriented programs, in: Proceedings of the 7th Aspect-Oriented Software Development, AOSD '08, ACM, New York, NY, USA, 2008, pp. 132–143.

[10] G. Soares, R. Gheyi, T. Massoni, Automated behavioral testing of refactoring engines, IEEE Transactions on Software Engineering 39 (2) (2013) 147–162.

[11] G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (2010) 52–57.

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

[13] G. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Software 23 (2006) 76–83.

[14] N. Rachatasumrit, M. Kim, An empirical investigation into the impact of refactoring on regression testing, in: Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM '12, IEEE Computer Society, Washington, USA, 2012, pp. 357–366.

[15] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Software Testing, Verification and Reliability, 2012.

[16] G. Soares, D. Cavalcanti, R. Gheyi, Making Aspect-Oriented refactoring safer, in: Proceedings of the 15th Brazilian Symposium on Programming Languages, SBLP '11, 2011, pp. 91–105.

[17] J. Hannemann, G. Kiczales, Design Pattern implementation in Java and Aspectj, in: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '02, ACM, New York, NY, USA, 2002, pp. 161–173.

[18] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, M. Cornélio, Implementing Java modeling language contracts with AspectJ, in: Proceedings of the 23rd Annual ACM Symposium on Applied Computing, SAC '08, ACM, New York, NY, USA, 2008, pp. 228–233.

[19] H. Rebêlo, R. Lima, M. Cornélio, G. T. Leavens, A. C. Mota, C. Oliveira, Optimizing JML features compilation in Ajmlc using Aspect-Oriented refactorings, in: Proceedings of the 13rd Brazilian Symposium on Programming Languages, SBLP '09, Brazilian Computer Society, 2009, pp. 117–130.

[20] J. Taveira, C. Queiroz, R. Lima, J. Saraiva, S. Soares, H. Oliveira, N. Temudo, A. Araújo, J. Amorim, F. Castor, E. Barreiros, Assessing intra-application exception handling reuse with Aspects, in: Proceedings of the 23rd Brazilian Symposium on Software Engineering, SBES '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 22–31.

[21] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006–1022.

[22] J. Goodenough, S. Gerhart, Toward a theory of test data selection, SIGPLAN Notes 10 (1975) 493–510.

[23] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of Java programs, in: Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '04, ACM, New York, USA, 2004, pp. 432–448.

[24] L. Zhang, M. Kim, S. Khurshid, FaultTracer: a change impact and regression fault analysis tool for evolving Java programs, in: Proceedings of the 20th ACM SIGSOFT Foundations of Software Engineering, FSE '12, ACM, New York, USA, 2012, pp. 40:1–40:4.

[25] B. Robinson, M. Ernst, J. Perkins, V. Augustine, N. Li, Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Washington, USA, 2011, pp. 23–32.

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

[27] V. Basili, R. Selby, D. Hutchens, Experimentation in software engineering, IEEE Transactions on Software Engineering 12 (7) (1986) 733–743.

[28] S. Shapiro, M. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3/4) (1965) 591–611.

[29] J. Box, Guinness, gosset, fisher, and small samples, Statistical Science 2 (1) (1987) 45–52.

[30] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bulletin 1 (6) (1945) 80–83.

[31] G. Leavens, A. Baker, C. Ruby, Preliminary design of JML: a behavioral interface specification language for Java, SIGSOFT Software Engineering Notes 31 (2006) 1–38.

[32] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 2005.

[33] E. Murphy-Hill, C. Parnin, A. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18.

[34] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa, C. Lucena, Assessing the impact of aspects on exception flows: An exploratory study, in: Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 207–234.

[35] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 287–296.

[36] X. Yang, Y. Chen, E. Eide, J. Regehr, Finding and understanding bugs in C compilers, in: Proceedings of the 32nd ACM SIGPLAN Programming Language Design and Implementation, PLDI '11, ACM, New York, USA, 2011, pp. 283–294.

[37] L. Tokuda, D. Batory, Evolving Object-Oriented designs with refactorings, Automated Software Engineering 8 (2001) 89–120.

[38] D. Roberts, Practical Analysis for Refactoring, Ph.D. thesis, University of Illinois at Urbana-Champaign (1999).

[39] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: Proceedings of the ACM SIGSOFT 20th Foundations of Software Engineering, FSE '12, ACM, New York, USA, 2012, pp. 50:1–50:11.

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

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

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

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

[44] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornélio, Algebraic reasoning for Object-Oriented programming, Science of Computer Programming 52 (2004) 53–100.

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

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

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

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

[49] W. Mckeeman, Differential testing for software, Digital Technical Journal 10 (1) (1998) 100–107.

[50] D. Binkley, M. Ceccato, M. Harman, F. Ricca, P. Tonella, Automated refactoring of Object-Oriented code into Aspects, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM '05, IEEE Computer Society, 2005, pp. 27–36.

[51] D. Binkley, M. Ceccato, M. Harman, F. Ricca, P. Tonella, Tool-supported refactoring of existing Object-Oriented code into Aspects, IEEE Transactions on Software Engineering 32 (9) (2006) 698–717.

[52] J. Hannemann, G. Murphy, G. Kiczales, Role-based refactoring of cross-cutting concerns, in: Proceedings of the 4th Aspect-Oriented Software Development, AOSD '05, ACM, New York, NY, USA, 2005, pp. 135–146.

[53] M. Malta, M. Valente, Object-Oriented transformations for extracting Aspects, Information and Software Technology 51 (1) (2009) 138–149.

[54] R. Yokomori, H. Siy, N. Yoshida, M. Noro, K. Inoue, Measuring the effects of Aspect-Oriented refactoring on component relationships: two case studies, in: Proceedings of the 10th Aspect-Oriented Software Development, AOSD '11, ACM, New York, USA, 2011, pp. 215–226.

[55] A. van Deursen, M. Marin, L. Moonen, AJHotDraw: A showcase for refactoring to aspects, in: Proceedings of the Workshop on Linking Aspect Technology and Evolution, LATE '05, 2005.

[56] L. Cole, P. Borba, A. Mota, Proving Aspect-Oriented programming laws, in: Proceedings of the 4th Foundations of Aspect-Oriented Languages,

FOAL '05, Technical Report, Department of Computer Science, Iowa State University, 2005, pp. 1–10.

[57] J. Law, G. Rothermel, Whole program path-based dynamic impact analysis, in: Proceedings of the 19th International Conference on Software Maintenance, ICSM '03, IEEE Computer Society, Washington, USA, 2003, pp. 308–318.

[58] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in Object-Oriented software maintenance, in: Proceedings of the International Conference on Software Maintenance, ICSM '94, IEEE Computer Society, British Columbia, Canada, 1994, pp. 202–211.

[59] L. Li, A. J. Offutt, Algorithmic analysis of the impact of changes to Object-Oriented software, in: Proceedings of the International Conference on Software Maintenance, ICSM '96, IEEE Computer Society, Washington, USA, 1996, pp. 171–184.

[60] J. Wloka, E. W. Host, B. G. Ryder, Tool support for change-centric test development, IEEE Software (2010) 66–71.

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

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

[63] M. Cornélio, Refactorings as Formal Refinements, Ph.D. thesis, Federal University of Pernambuco (2004).

[64] B. Daniel, D. Dig, K. Garcia, D. Marinov, Automated testing of refactoring engines, in: Proceedings of the 15th Foundations of Software Engineering, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 185–194.

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

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

[67] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, H. Rebêlo, SYMDIFF: a language-agnostic semantic diff tool for imperative programs, in: Proceedings of the 24th Computer Aided Verification, CAV '12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 712–717.

[68] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, V. Augustine, Dex: A semantic-graph differencing tool for studying changes in large code bases, in: Proceedings of the 20th International Conference on Software Maintenance, ICSM '04, IEEE Computer Society, Washington, USA, 2004, pp. 188–197.

[69] S. Person, M. Dwyer, S. Elbaum, C. Păsăreanu, Differential symbolic execution, in: Proceedings of the 16th Foundations of Software Engineering, FSE '2008, ACM, New York, NY, USA, 2008, pp. 226–237.

[70] S. Lahiri, K. McMillan, R. Sharma, C. Hawblitzel, Differential assertion checking, in: Proceedings of the 21th Foundations of Software Engineering, FSE '13, ACM, New York, USA, 2013.

[71] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for Object-Oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 416–419.

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