SAFIRA: A Tool for Evaluating Behavior Preservation

Melina Mongiovi

Abstract—To evaluate whether a program transformation is behavior preserving is required in several tasks, such as refactoring and mutation testing. However, in both areas it is not an easy task to formally establish conditions to preserve or not behavior for Java on account of its non-trivial semantics. For this reason, a number of refactoring and mutation testing tools are likely to present bugs. We propose a tool (SAFIRA) to improve confidence whether a transformation is behavior preserving through test generation for entities impacted by transformation. We propose an approach to test mutation testing and refactoring tools based on SAFIRA. We found 17 bugs in MuJava, and 62 bugs in refactorings implemented by Eclipse and JRRT.

Index Terms—behavior preservation, refactoring, mutation testing, automated testing

1 PROBLEM AND MOTIVATION

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

Defining and implementing refactorings is a non-trivial task which the literature has treated in different ways [6], [7], [8], [9], [10], [11], [12], [13]. 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 [14].

1.1 Motivating Example

In this section, we present a transformation – assumed to be refactoring – applied by Eclipse JDT 3.7 and JastAdd Refactoring Tools (JRRT) [10], [11], [12]. JRRT was proposed to improve the correctness of Eclipse JDT refactorings by using formal techniques. Some preconditions were formally stated for a number of refactorings. This transformation actually introduces a behavioral change.

Take class A and its subclass B as illustrated in Listing 1. A declares the k method, and B declares methods k, m, and $test$. The latter yields 1. Suppose we want to apply the Pull Up Method refactoring to move m from B to A. This method contains a reference to $A.k$ using the $super$ access. The use of either Eclipse JDT 3.7 or JRRT to perform this refactoring will produce the program presented in Listing 2. Method m is moved from B to A, and $super$ is updated to $this$; a compilation error is avoided with this change. Nevertheless, a behavioral change was introduced: $test$ yields 2 instead of 1. Since m is invoked on to an instance of B, the call to k using $this$ is dispatched on to the implementation of k in B. Refactoring tools may also introduce compilation errors.

Understanding inheritance, references to this/super, accessibility and other Java constructs in isolation may be simple, but nontrivial when considering them in conjunction [15]. So, it is difficult to find sufficient conditions for a refactoring to preserve behavior bearing in mind the complete Java language specification. For example, a simple transformation changing the access modifier may have an impact on a number of Java constructs [13]. Because it is time consuming and difficult to prove all refactorings sound with respect to a formal semantics, a less costly method to evaluate the correctness of refactorings is needed.

A number of behavioral preserving program transformations are also applied to other areas, such as those of mutation testing and compilers. For example, a dual problem occurs in the mutation testing area. Tools must introduce behavioral changes on to programs. Every transformation must contain a set of conditions stating when behavior must be preserved. However, it is not an easy task to formally establish conditions whether to preserve or not behavior for Java on account of its non-trivial semantics. Because of this, a number of refactoring and mutation testing tools are likely to present bugs.

In this paper, we propose a tool called SAFIRA to improve confidence that a transformation preserves

• *Melina Mongiovi is affiliated to the Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, 58429-900 Brazil. Her research supervisor is Rohit Gheyi (rohit@dsc.ufcg.edu.br). E-mail: melina@copin.ufcg.edu.br*

Listing 1: Before Refactoring

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

Listing 2: After Refactoring

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

behavior. It is a practical approach based on automatically generating tests for the entities impacted by a transformation. We have evaluated SAFIRA in the refactoring and mutation testing area.

2 BACKGROUND AND RELATED WORK

Since proving that a transformation is behavioral preserving for the entire language constitutes a challenge [14], in practice, refactoring engine developers use informal sets of preconditions as the basis for implementing refactorings. As a result, this may bring about differences between the implementations of the same refactoring [16], [17]. More importantly, incorrect implementations have been reported in those engines [16], [10], [13], [17]. While compilation errors introduced by the refactorings are easily detected in IDEs, behavioral changes may pass unnoticed (Section 1.1). Test suites are commonly seen as trustworthy resources for preventing such problems. However, these tests may also be affected by changes applied by tools, which may render them inappropriate for testing refactoring outcomes [3].

We propose a more practical approach to evaluate whether a transformation preserves behavior using SAFIRA. It evaluates all kinds of transformation, different from previous approaches which focus on formalizing some preconditions for some refactorings. We have improved SAFEREFACTOR [18], a similar tool for evaluating behavior preservation, by automatically generating tests only for the entities impacted by a transformation within a time limit. In general, SAFIRA generates a smaller test suite and time limit than SAFEREFACTOR to evaluate whether a transformation preserves behavior (see Section 4.2).

A dual problem does occur in the mutation testing area. Mutation testing can help developers to evaluate their test suite. It consists of introducing defects on to the code in order to alter its behavior. If a test suite fails to detect behavioral change (identify the introduced defect), it needs to be improved. There are a number of mutation testing tools, such as MuJava [19]. However, these tools may generate *equivalent mutants* — a mutant which is functionally equivalent to the original program. These mutants can never be killed, because they have the same behavior as that of the original program. They are syntactically different but functionally equivalent to the original program. Then, when a test suite does not kill a mutant, developers do not know whether the problem resides in their test suite or it is an equivalent mutant.

Some approaches suggest different ways of avoiding certain kinds of equivalent mutants. Based on previous work [13], Steimann and Thies proposed an approach to generate mutants based on negating the conditions required for a transformation to be a refactoring [20]. However, they did not formally prove the conditions were correct. Moreover, conditions are only related to accessibility constraints. Our technique is simple, and SAFIRA can be used to evaluate any kind of mutations.

Schuler and Zeller [21] proposed an approach to detect equivalent mutants based on changes in test coverage. They implemented a mutation testing tool called Javalanche. It specifies a number of program invariants. If a mutation does not violate these invariants, then it is more likely to be an equivalent mutant [21]. The tool implements four mutations. Analyzing changes in test coverage may yield false positives. SAFIRA generates a test suite to detect an equivalent mutant. When it finds a mutant, it yields a test case by exposing behavioral changes, which renders it different from all previous approaches. Also, we propose a technique to test mutation testing tools (see Section 4.1).

3 Approach and Uniqueness

In this paper, we propose an approach to assess whether a transformation is behavior preserving through change impact analysis [22] and automatic test generation [23]. We have implemented a tool called SAFIRA. It generates a test suite focusing on exercising only the entities impacted by the transformation. SAFIRA yields a test case whenever it detects a behavior change.

3.1 Example

To explain our approach, consider the transformation applied to the program in Listing 1. First, we identify the impacted entities by the transformation. Evaluating the impact of a large transformation may be difficult and time consuming. Therefore we split the transformation into smaller changes: we remove the method *B.m* and add the method *A.m*. We formalize a number of changes in order to identify the impacted entities (methods and constructors that may change behavior after the transformation). Removing the method *m* of the class *B* will impact itself, and *B.test* because it calls a removed method. Adding the method *A.m* will impact itself and *B.test*. Other methods calling *m* may be impacted due overloading and overriding in the class hierarchy. The set of impacted entities by the transformation is the union of the impacted sets by each change. We also include a set of methods that exercise directly or indirectly the impacted entities. In this example the impacted entities are the methods: *A.m*, *B.m* and *B.test*.

Then, SAFIRA generates a test suite. Since we intend to generate a unique test suite to execute in the source and target programs, then we will identify all public common methods and constructors (they belong to both programs) in the impacted set. In this example, the public common methods are *B.m* and *B.test*. Then, we automatically generate a test suite to exercise only these two methods. Finally, we execute the tests. If the results are different, SAFIRA will report the transformation has not preserved behavior. In the present example, this is not a refactoring, because in the source program the method *B.test* yields *1* while in the target program it yields *2*.

3.2 SAFIRA

The major steps performed by SAFIRA are explained as follows: To begin with, the source and target programs and the time limit are passed as parameters by developers. The change impact analyzer decomposes the transformation into smaller changes (Step 1.1). Then, for each change, SAFIRA identifies the entities impacted by it (Step 1.2). Finally, SAFIRA identifies a set of methods in common that exercises, directly or indirectly, impacted entities (Step 1.3). A method in common must have the same signature in the source and target programs. After this a test suite is generated only for the identified methods within a time limit (Step 2). The tests are then executed on both the source (Step 3) and on the target programs (Step 4). SAFIRA reveals a set of tests that passes on to the source program but fails on target program. If this set is empty, the developer increases confidence that the transformation is behavioral preserving; otherwise, the test cases will show behavior changes. Our approach is illustrated in Figure 1.

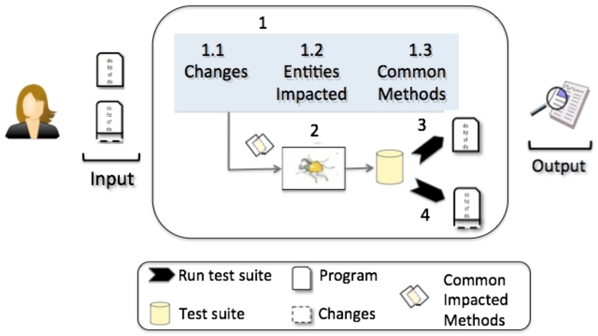


Fig. 1: SAFIRA architecture

3.2.1 Impact Analysis

SAFIRA analyzes a transformation and decomposes it into smaller changes. For each change, we formalize the set of impacted methods and constructors for eight changes: changing a method signature, modifying a method body, and adding and removing a class, method and field. An impacted method or constructor may change behavior in the target program. The impact of the whole transformation is the union of each change's impacted entities. Moreover, SAFIRA also includes all methods and constructors that exercise directly or indirectly this impacted set.

For example, Change 1 states the entities impacted when adding a method to a program. Each change consists of two templates (patterns) of Java programs, on left-hand and right-hand sides. Each change may declare some meta-variables. We used cds, mds and fds to denote a set of classes, methods and fields declarations, respectively. For each change, we define the set of entities impacted by a transformation. For instance, Change 1 adds a method m to a class C. The method m of C and inherited by its subclasses that do not redefine it (overload or override) will be impacted. Any method that calls them will also be impacted. The $<^*$ operator defines the class inheritance relationship. The other seven changes are specified similarly.

3.2.2 Test Generation

In this step SAFIRA generate tests only for the methods in common that exercise either directly or indirectly the impacted entities. It uses Randoop [24] to randomly generates unit tests for the classes and methods received as parameters within a time limit specified by the developer.

Change 1: ⟨adding method⟩

```
cds                      cds'
class C ext D {          class C ext D {
  fds               →      fds'
  mds                      mds'
}                          void m(...) {}
                         }
```

Impacted set. Let F be the closest child of C that redefines method m. The set of entities impacted is defined by: {n:Method | ∃ E:Class | (F $<^*$ E ∧ E $\leq^*$ C) ∧ (n ∈ methods(cds') ∪ ops') ∧ ((n = $E.m$) ∨ ($E.m$ ∈ commands(n)))}.

3.2.3 *Change Coverage*

Test data adequacy criteria provides measurements of test quality. It may also provide explicit rules to determine when it is appropriate to end the testing phase [25], [26]. There are a number of notions of test data adequacy. In our context, our notion evaluates whether the test suite covers all entities impacted by a transformation. SAFIRA can yield the percentage of impacted methods exercised by a test suite (change coverage).

4 RESULTS AND CONTRIBUTIONS

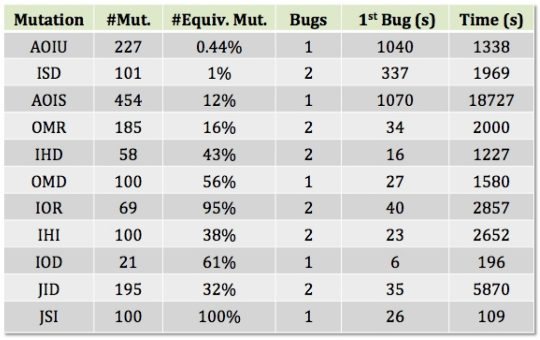
We have evaluated SAFIRA in the mutation testing and refactoring contexts, as explained next.

4.1 Mutation Testing

As explained in Section 2, it is difficult to detect whether a mutant is equivalent. We can use SAFIRA to evaluate a program and its mutant. If SAFIRA does not detect a behavioral change, developers will improve confidence that this is an equivalent mutant. However, if a behavioral change is found, SAFIRA will yield a test case by means of which developers can improve their test suite. This is a difference between the present work and previous ones [21], [20].

Moreover, besides analyzing a transformation, we propose a technique to test mutation testing tools, such as MuJava, based on SAFIRA. To test mutation testing tools is nontrivial, since one needs structurally complex inputs, such as programs. Our technique consists of four major steps. First, a program generator automatically yields programs as test inputs for a mutation; we employ JDOLLY [27] in this step. JDOLLY automatically generates Java programs for a given number of declarations (packages, classes, methods, and fields). The program presented in Listing 1 is generated by JDOLLY. It contains a subset of the Java metamodel specified in Alloy, which is a formal specification language [28]. It also employs the Alloy Analyzer [29], a tool for the analysis of Alloy models to generate solutions for this metamodel. Second, the mutation is automatically applied to each generated program using MuJava (Step 2). Then, the transformation is evaluated in terms of behavior preservation, using SAFIRA. In the end, we may have detected a number of transformations that preserve behavior (equivalent mutants). In Step 4, the detected problems are then categorized.

TABLE 1: Testing MuJava mutations using SAFIRA



Mutation	#Mut.	#Equiv. Mut.	Bugs	1st Bug (s)	Time (s)
AOIU	227	0.44%	1	1040	1338
ISD	101	1%	2	337	1969
AOIS	454	12%	1	1070	18727
OMR	185	16%	2	34	2000
IHD	58	43%	2	16	1227
OMD	100	56%	1	27	1580
IOR	69	95%	2	40	2857
IHI	100	38%	2	23	2652
IOD	21	61%	1	6	196
JID	195	32%	2	35	5870
JSI	100	100%	1	26	109

For instance, let us consider the IHD mutation of MuJava [19]. It deletes a variable in a subclass that has the same name and type as a variable in the parent class. In order to generate a mutant for IHD, we will generate a program with an overridden field. We can guide JDOLLY to generate programs with this property by passing the following parameter specified in Alloy using the Java metamodel specified in JDOLLY.

$$\exists A, B : Class,\ f1, f2 : Field \mid A \in B.extends \wedge$$
$$f1 \in A.fields \wedge f2 \in B.fields \wedge$$
$$f2.name = f1.name \wedge f2.type = f1.type$$

For each mutation, we generated 100 programs by using JDOLLY. For instance, our approach found 25 equivalent mutants for the IHD mutation Some of them are related to the same bug. We manually categorized them in 2 distinct bugs. The first equivalent mutant was detected in 16 seconds. We evaluated 11 mutations implemented by MuJava and tested all of them by generating small programs with up to 4 classes, methods and fields. We found 17 bugs. All bugs were reported to MuJava developers. Table 1 summarizes the results.

4.2 Refactoring

SAFIRA was also used in refactoring context to assist developers in refactoring activities. We used SAFIRA in 3 real Java applications up to 23KLOC (JHotDraw, JUnit and VPoker) that were refactored by their developers. The transformations intended to be refactorings and were performed manually or using refactoring tools. Developers also used a test suite to evaluate behavioral preservation. We evaluated all transformations using SAFIRA. It found a behavior

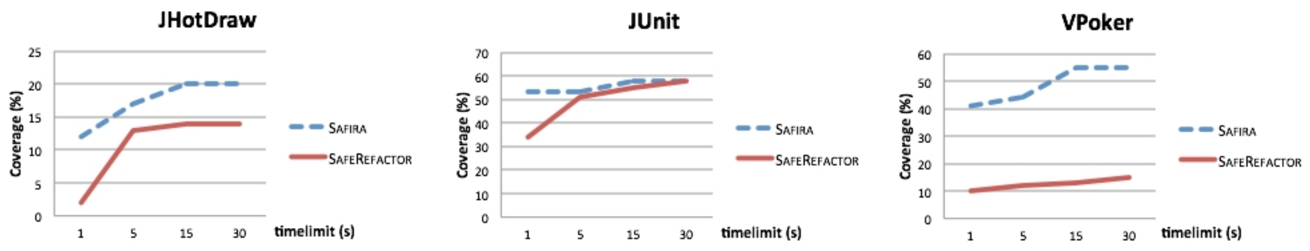


Fig. 2: Change Coverage of SAFIRA and SAFEREFACTOR in the refactorings applied to real case studies

TABLE 2: Testing refactorings applied to real case studies using SAFIRA and SAFEREFACTOR

Program	KLOC	Refactoring	Methods passed to Randoop		Generated Tests		Total Time (s)	
			SAFIRA	SR	SAFIRA	SR	SAFIRA	SR
JHotDraw	23	Extract Exception Handler	146	14565	64	2245	83	148
JUnit	3	Infer Generic Type	80	657	548	1127	26	99
VPoker	4	Infer Generic Type	41	1156	98	466	25	109

change in the refactoring applied to JHotDraw. Some classes that implemented Serializable were refactored. Developers extracted a class and forgot to serialize the new attribute. We did not find behavioral changes in the other transformations (Table 2).

The same subjects were evaluated before by SAFEREFACTOR (SR) [18]. The difference between SAFIRA and SAFEREFACTOR is that SAFEREFACTOR does not analyze the entities impacted by a transformation. In the example presented in Section 2, tests are generated for the methods considered by SAFIRA, and also for *A.k* and *B.k* since they belong to both programs. SAFEREFACTOR requires a greater time limit for Randoop to detect the behavioral change than SAFIRA does. For instance, SAFIRA detects it by using a time limit of 1s in the transformation applied to JHotDraw, as different from SAFEREFACTOR that requires at least 10s. SAFEREFACTOR also takes more time to analyze the transformation because it generates a greater test suite. The problem is still greater when is done a small change in a large system. In this case, SAFEREFACTOR generates many unnecessary tests while SAFIRA focus on the change, generating tests only for the entities impacted by the transformation. It requires less time to generate a test suite to expose the behavioral change, far different from SAFEREFACTOR.

We have collected the change coverage of each tool (Figure 2) in the transformations presented in Table 2. We have noticed that SAFIRA has a greater change coverage than that of SAFEREFACTOR using the same time limit. For some subjects, we cannot have 100% of coverage because some methods are added and they are not exercised by other parts of the program. Moreover, some methods were deleted, and we cannot exercise them.

Besides analyzing a transformation, we can use

TABLE 3: Testing refactoring implementations of Eclipse JDT and JRRT using SAFIRA and SAFEREFACTOR. CE: Compilation Error, BC: Behavior Change

Refactoring	Prog	Bugs Eclipse		Bugs JRRT		Reduction of SAFIRA compared to SR		
		CE	BC	CE	BC	Tests (%)	Time (%)	Time (h)
rename class	2020	1	1	0	0	47.30	29.48	0.07
rename method	6823	1	0	0	0	74.58	29.50	0.20
rename field	2643	1	0	3	1	63.50	25.72	0.48
push downmethod	3842	2	6	2	2	79.19	34.64	0.46
push down field	3039	1	1	0	0	88.07	28.10	0.29
pull up method	5195	2	7	2	2	79.16	32.95	0.61
pull up field	4144	3	3	2	0	79.70	26.98	0.68
encapsulate field	3752	1	0	0	1	99.86	32.14	0.67
move method	6314	2	4	3	2	74.50	32.31	0.37
add parameter	4971	1	3	0	2	51.64	28.08	0.91
Total	42743	15	25	12	10	-	-	4.74
					Average	73.75	29.99	-

SAFIRA to test refactoring implementations. Soares et al. [17] proposed an approach based on JDOLLY and SAFEREFACTOR to test refactoring tools. They found a number of bugs in refactoring implementations of Eclipse JDT and JRRT [17]. We used their approach, but we replaced SAFEREFACTOR for SAFIRA, and compared them afterwards.

We evaluated 10 different kinds of refactorings. We detected 62 bugs. We decrease the test suite by 73.7% and the total time of the experiment by 29.9%. For the present experiment, we used a time limit of 0.4 second for SAFIRA and 1 second for SAFEREFACTOR. Both tools yielded the same result (see Table 3). We also evaluated some refactoring implementations by using a time limit of 0.4 seconds for both tools. In some transformations, SAFEREFACTOR did not detect any behavioral changes different from SAFIRA. We reported all bugs to Eclipse and JRRT developers.

ACKNOWLEDGMENTS

I would like to thank Rohit Gheyi for his guidance. I would also like to thank the referees of ACM Student Research Competition at SPLASH'11, Gustavo Soares and Tiago Massoni for their useful comments and assistance.

References

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

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

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

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

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

[6] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic reasoning for object-oriented programming," *Science of Computer Programming*, vol. 52, pp. 53–100, August 2004.

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

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

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

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

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

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

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

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

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

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

[17] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2012.

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

[19] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research articles," *Software Testing, Verification and Reliability*, vol. 15, pp. 97–133, June 2005.

[20] F. Steimann and A. Thies, "From behaviour preservation to behaviour modification: constraint-based mutant generation," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 425–434.

[21] D. Schuler and A. Zeller, "(Un-)covering equivalent mutants," in *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 45–54.

[22] X. Ren, F. Shah, F. Tip, B. G. Ryder, and 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*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 432–448.

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

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

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

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

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

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

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