Making Refactoring Safer through Impact Analysis

Melina Mongiovi[1], **Rohit Gheyi (advisor)**[1]

[1]Department of Computing Systems – UFCG
58429-900 – Campina Grande – PB – Brazil

melina@copin.ufcg.edu.br, rohit@dsc.ufcg.edu.br

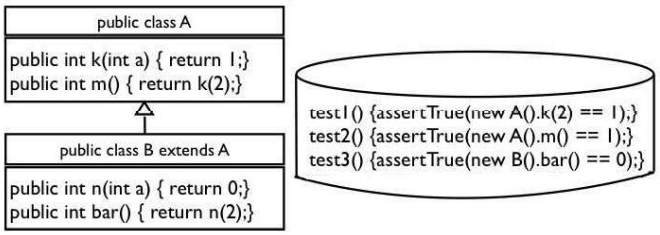
Abstract. *Developers use regression tests and refactoring tools to ensure that the refactoring preserves the behavior of the program. However, refactoring tools may have bugs. In addition, some studies revealed that in real projects the test suite may be inappropriate to test refactorings. We propose an approach for evaluating whether a transformation is behavior preserving by automatically generating tests for the entities impacted by the change. We evaluate it in a set of 45 transformations and to a technique to test refactoring implementations. We identify behavioral changes previously undetected in transformations applied to real systems (from 20 LOC to 79 KLOC). Furthermore, we reduce on average 60% the total time to test 18 refactoring implementations of Eclipse and JRRT.*

1. Introduction

Refactoring is the process of improving software internally without affecting external behavior [Opdyke 1992]. Developers can apply it manually or assisted by a tool with support to automatic refactorings such as Eclipse and NetBeans. However, refactoring tools may apply transformations that do not preserve the program behavior [Soares et al. 2013a]. In practice, developers have to apply manual steps and use test suites to guarantee behavior preservation in refactoring activities. However, the tests may not exercise the entities impacted by the change. Furthermore, executing all regression tests of the program may be time-consuming and unnecessary since only the test cases that exercise the change are relevant to analyze behavior preservation. Rachatasumrit and Kim [Rachatasumrit and Kim 2012] found that refactorings are not well tested in real projects. 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 transformation.

For example, we present a defective refactoring and a test suite that does not identify the behavioral change. In Figure 1 it shows a diagram representing a program and its test suite. The program contains the class *A* and its subclass *B*. Methods *A.m* and *B.m* (inherited from *A*) call *A.k(int)* yielding 1. Renaming method *B.n(int)* to *B.k(int)*, introduces a behavioral change: the inherited method *B.m* in the target program now yields *0* instead of *1*. The test suite contains three test cases $test1$, $test2$, and $test3$ that call methods *A.k(int)*, *A.m*, and *B.bar* respectively. The tests do not catch the behavioral change because the inherited method *B.m* is not exercised. Furthermore, only *test3* exercises an impacted method while other test cases are not relevant to test the transformation. Therefore, in a real program with a large test suite may be costly and unnecessary to execute all test cases. Moreover, the test suite may not exercise all entities impacted by the change.

Figure 1. Renaming method *B.n(int)* to *B.k(int)* changes program behavior.

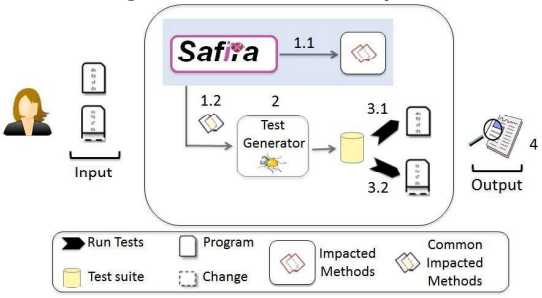


We propose an approach to analyze whether a transformation preserves the program behavior based on change impact analysis and test generation (Section 2). We implemented this approach in a tool called SAFEREFACTORIMPACT (SRI). The tool analyzes a transformation applied to a program and, within a time limit, it generates test cases for the impacted methods. We evaluated SRI in 45 transformations and compared it with SAFEREFACTOR (SR) [Soares et al. 2010], a similar tool that does not perform change impact analysis, and a manual inspection performed by experts [Murphy-Hill et al. 2012]. We also evaluated SRI in a technique to test refactoring implementations that found more than 100 bugs in the best academic (JRRT [Schäfer and de Moor 2010]) and commercial Java refactoring implementations (Eclipse and NetBeans). We found that SRI detects behavioral changes that the other approaches could not detect [Soares et al. 2013b]. Moreover, SRI reduced in 60% the time to test 18 refactoring implementations of Eclipse and JRRT while found the same bugs.

2. Approach

SRI 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. Then, we identify the public and common impacted methods in both program versions from the impacted set (Step 1.2). Next, SRI generates a test suite, within a time limit passed by the user, for the methods previously identified using Randoop [Pacheco et al. 2007], an automatic test suite generator (Step 2). It executes the same test suite before (Step 3.1) and after the transformation (Step 3.2). Finally, the tool evaluates the results: if the results of the tests are different, the tool reports a behavioral change. Otherwise, we improve confidence that the transformation is behavior preserving (Step 4). Figure 2 illustrates the described process.

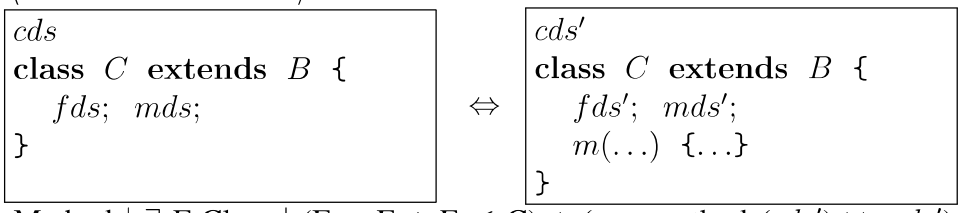
Figure 2. SRI's technique.



The goal of our change impact analysis is to analyze the original and modified programs, and yields the set of methods impacted by the change that is, the methods that may change the program behavior. First, we decompose a coarse-grained transformation into smaller transformations (Step 1). We consider nine small-grained transformations: Add and Remove Method, Change Method Body, Change Method Modifier, Add and Remove Field, Change Field Modifier, Change Field Initializer, and Change Static Field Initializer. For each small-grained transformation, we identify the set of impacted methods. We

formalized the impact of small-grained transformations in laws (Step 2). 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. 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. We specified other laws for the small-grained transformations. Next, 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, SAFIRA yields the set of impacted methods by the transformation, which is the union of directly and indirectly impacted methods (Step 5).

Law 1 ⟨Add/Remove Method⟩



(↔) {n:Method | ∃ E:Class | (F < E ∧ E ≤ C) ∧ (n ∈ methods(cds') ∪ mds') ∧ n = $E.m$}, where F is the closest subclass of C such that it redeclares m.

3. Evaluation

We evaluated SRI in 45 transformations applied to programs (Section 3.1) and in a technique to test refactoring implementations (Section 3.2). Experimental data are available online at `http://www.dsc.ufcg.edu.br/~spg/ctd/CTD/SafeRefactorImpact.html`.

3.1. Transformations Applied to Programs

The goal of this experiment is to analyze SRI for the purpose of comparing it with SR and a manual inspection with respect to fault detecting and performance from the point of view of researchers in the context of transformations applied to programs. We evaluated transformations applied to 23 design patterns implemented in Java and AspectJ, 2 programs compiled by two JML compilers, 8 defective refactorings of Eclipse, and 12 real programs. SRI correctly identifies behavioral changes in 5 out of the 23 design patterns implementations while SR identifies all of these behavioral changes but one (the Mediator design pattern). In the JML compilers evaluation, both tools correctly identify behavioral changes in both transformations. However, in the defective refactorings applied by Eclipse, SRI does not identify one behavioral change due to a limitation of SAFIRA. Comparing with SR, SRI reduces the time to evaluate these transformations while has higher percentage of relevant tests and a similar change coverage.

SRI correctly evaluates all transformations applied to the real programs but two (Subjects 5 and 6), while SR correctly evaluates seven transformations. Both tools used a time limit of 20s. In Subjects 5 and 6, the tools do not identify behavioral changes using this time limit because Randoop does not generate tests that exercise the impacted methods that change behavior. Different from SR, SRI 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. SR does not identify the behavioral changes using a

Table 1. Results of the evaluation of SR and SRI in the real subjects.

Subject	Impacted Methods	Methods		Time (s)		Change Coverage (%)		Relevant Tests (%)		Result		Baseline
		SR	SRI	SR	SRI	SR	SRI	SR	SRI	SR	SRI	
1	4,134	14,867	2,267	63	45	10	9	72.10	100	Yes	Yes	Yes
2	4,321	14,870	2,336	61	53	12	19	77.18	100	No	No	No
3	2,251	1,539	374	87	74	5	4	53.13	98.31	No	No	No
4	2,580	1,546	384	80	69	7	6	84.73	100	No	No	No
5	3,375	18,977	2,004	78	68	7	16	50.41	96.50	Yes	Yes	No
6	251	31,933	185	65	63	8	34	3.89	100	Yes	Yes	No
7	2,068	17,074	1,510	68	55	8	16	53.44	99.78	Yes	No	No
8	3,524	12,771	2,135	69	88	6	18	59.60	100	No	No	No
9	5,027	29,698	2,882	73	150	6	13	55.37	100	Yes	No	No
10	203	30,447	130	69	61	9	33	2.30	100	Yes	No	No
11	27	31,685	26	69	72	0	7	0	100	Yes	No	No
12	4,214	32,307	2,377	54	92	4	13	56.76	99.81	Yes	No	No

Table 2. Results of the evaluation of SR and SRI in the refactoring implementations.

Refactoring	Prog.	Failures		Bugs		Time Reduction (h)		Refactoring	Prog.	Failures		Bugs		Time Reduction (h)	
		Ecl.	JRRT	Ecl.	JRRT	Ecl.	JRRT			Ecl.	JRRT	Ecl.	JRRT	Ecl.	JRRT
rename class	15.322	145	0	1	0	1,43	5,44	pull up method	8.937	202	10	3	1	1,72	2,65
rename method	11.263	0	482	0	2	1,40	3,99	pull up field	10.927	546	0	4	0	1,76	0,27
rename field	19.424	0	0	0	0	7,08	6,08	encapsulate field	2.000	0	437	0	1	0,71	9,27
add parameter	30.186	2.238	0	2	0	6,58	1,55	move method	22.905	3.586	-	3	-	1,87	-
push down field	11.936	92	0	1	0	1,02	2,04	push down method	20.544	853	-	5	-	2,26	-

time limit of 20s in Subjects 7 and 9-12 different from SRI. However, SR detects three of them (Subjects 7, 10 and 12) using a time limit of 120s [Soares et al. 2013b]. Then, SR is more dependent to the time limit than SRI. SRI identified a previously undetected behavioral change in Subject 9. Both SR and a manual inspection performed by experts [Murphy-Hill et al. 2012] classified Subject 9 as behavior preserving. In real programs, the change impact analysis is useful to reduce (ranging from 75% to 99%) the set of methods passed to Randoop in SRI. Moreover, SRI has higher percentage of change coverage in 9 out of 12 subjects and generates at least 95% of relevant tests. Finally, both tools took almost the same time to evaluate these subjects. Table 1 summarizes the results of the evaluation on real programs using a time limit of 20s.

3.2. Technique to Test Refactoring Implementations

The goal of this experiment is to analyze SRI for the purpose of comparing it with SR in a technique to test refactoring implementations [Soares et al. 2013a] with respect to fault detecting and performance from the point of view of refactoring tools developers in the context of refactoring implementations. The technique generates automatically programs using JDolly, an automatic Java programs generator, and uses SR to evaluate the behavior preservation. In the technique SR used a time limit of one second to generate tests, which they consider appropriate for testing the small programs generated by JDolly [Soares et al. 2013a]. As SRI reduces the set of methods passed to Randoop generating tests due to change impact analysis, we used a time limit of 0.2 second in order to reduce the time to evaluate the transformations. We evaluated the refactoring implementations using SR and SRI to compare the time and failures detected for each tool.

The technique generated a total of 153,444 programs to evaluate all the refactoring implementations. SRI reduced in 64% and 54% the total time to evaluate the refactoring implementations of Eclipse and JRRT [Schäfer and de Moor 2010], which is equivalent to 25.83 and 31.29 hours, respectively. SRI and SR identified the same bugs in all refactoring implementations. We also evaluated SR in some refactoring implementations using a

time limit of 0.2 second and it did not catch some failures. Table 2 summarizes the results.

4. Related Work

Schäfer et al. [Schäfer and de Moor 2010] presented a number of Java refactoring implementations and implemented them in a tool called JRRT. The tool was designed to improve correctness of refactorings implemented on Eclipse. We evaluated eight refactoring implementations of JRRT and found bugs related to missing conditions. Steimann and Thies [Steimann and Thies 2009] proposed a constraint-based approach to specify Java accessibility. It is useful for detecting bugs regarding accessibility-related properties. However, they have not proven the specifications according a formal semantic of the language. Borba et al. [Borba et al. 2004] proposed a set of refactorings for a subset of Java with copy semantics. Later, Silva et al. [Silva et al. 2008] presented a set of behavior preserving transformation laws for a sequential object-oriented language with reference semantics. They proved the refactoring correctness based on a formal semantics. However, they have not considered all Java constructs. SRI may be useful when their work may not be applied. Soares et al. [Soares et al. 2013a] introduced a technique to test refactoring implementations. This approach is based on a program generator (JDolly) and SR. We optimized this technique by using SRI instead of SR. Then, we reduced in 60% the time to test the refactoring implementations.

Chianti [Ren et al. 2004] 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. Zhang et al. [Zhang et al. 2012] 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 Chianti and FaultTracer, SAFIRA calculates the impact based on defined laws for each small-grained transformation. The other approaches use rules to formalize the dependencies between the atomic changes and then, identify the tests impacted by the change. Moreover, Chianti and FaultTracer depend on a test suite to assess the change impact while SRI automatically generates test cases for the methods impacted by a transformation.

5. Conclusions

We propose a tool (SAFEREFACTORIMPACT) to check whether a transformation is behavior preserving. We found that SAFEREFACTORIMPACT detects some non-behavior-preserving transformations applied to real programs (ranging from 20 LOC to 79 KLOC), undetected by SAFEREFACTOR and manual inspection performed by experts. Moreover, it has a better change coverage in larger subjects and generates much more relevant tests. Finally, it reduced in 60% the time to test the refactoring implementations.

Publications and Awards

This work was ranked the second best work in the ACM Student Research Competition [Mongiovi 2011]. It was published in ICSM [Soares et al. 2011] (Qualis A2) and it was accepted for publication in Science of Computer Programming Journal [Mongiovi et al. 2013] (Qualis B1).

Acknowledgment

We gratefully thank G. Soares, T. Massoni, P. Borba, A. Cajueiro, R. Coelho, J. Candido, and the anonymous referees for useful suggestions. This work was partially supported by INES funded by CNPq grants 573964/2008-4, 306610/2013-2, and 477943/2013-6.

References

[Borba et al. 2004] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic reasoning for object-oriented programming. *SCP*, 52:53–100.

[Mongiovi 2011] Mongiovi, M. (2011). Safira: A tool for evaluating behavior preservation. Student Research Competition at SPLASH, pages 213–214.

[Mongiovi et al. 2013] Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., and Borba, P. (2013). Making refactoring safer through impact analysis. *SCP*. In press.

[Murphy-Hill et al. 2012] Murphy-Hill, E., Parnin, C., and Black, A. (2012). How we refactor, and how we know it. *IEEE TSE*, 38(1):5–18.

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

[Pacheco et al. 2007] Pacheco, C., Lahiri, S. K., Ernst, M., and Ball, T. (2007). Feedback-directed random test generation. ICSE, pages 75–84.

[Rachatasumrit and Kim 2012] Rachatasumrit, N. and Kim, M. (2012). An empirical investigation into the impact of refactoring on regression testing. ICSM, pages 357–366.

[Ren et al. 2004] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: a tool for change impact analysis of Java programs. OOPSLA, pages 432–448.

[Schäfer and de Moor 2010] Schäfer, M. and de Moor, O. (2010). Specifying and implementing refactorings. OOPSLA, pages 286–301.

[Silva et al. 2008] Silva, L., Sampaio, A., and Liu, Z. (2008). Laws of object-orientation with reference semantics. SEFM, pages 217–226.

[Soares et al. 2013a] Soares, G., Gheyi, R., and Massoni, T. (2013a). Automated behavioral testing of refactoring engines. *IEEE TSE*, 39:147–162.

[Soares et al. 2013b] Soares, G., Gheyi, R., Murphy-Hill, E., and Johnson, B. (2013b). Comparing approaches to analyze refactoring activity on software repositories. *JSS*, 86:1006–1022.

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

[Soares et al. 2011] Soares, G., Mongiovi, M., and Gheyi, R. (2011). Identifying overly strong conditions in refactoring implementations. ICSM, pages 173–182.

[Steimann and Thies 2009] Steimann, F. and Thies, A. (2009). From public to private to absent: Refactoring Java programs under constrained accessibility. ECOOP, pages 419–443.

[Zhang et al. 2012] Zhang, L., Kim, M., and Khurshid, S. (2012). FaultTracer: a change impact and regression fault analysis tool for evolving Java programs. FSE, pages 40:1–40:4.