UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Tese

Uma Abordagem Automatizada para Testar Ferramentas de Refatoramento

Gustavo Araújo Soares

Campina Grande, Paraíba, Brasil

©Gustavo Araujo Soares, março de 2014

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem Automatizada para Testar Ferramentas de Refatoramento

Gustavo Araújo Soares

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Rohit Gheyi

(Orientador)

Campina Grande, Paraíba, Brasil

©Gustavo Araújo Soares, 10/03/2014

S676a Soares, Gustavo Araújo.
Uma abordagem automatizada para testar ferramentas de refatoramento / Gustavo Araújo Soares. - Campina Grande, 2014.
133 f.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.
Referências.
"Orientação : Prof. Dr. Rohit Gheyi".

1. Engenharia de Software. 2. Refatoramento. 3. Testes. 4. Tese - Ciência da Computação. I. Gheyi, Rohit. II. Universidade Federal de Campina Grande - Campina Grande (PB). III. Título

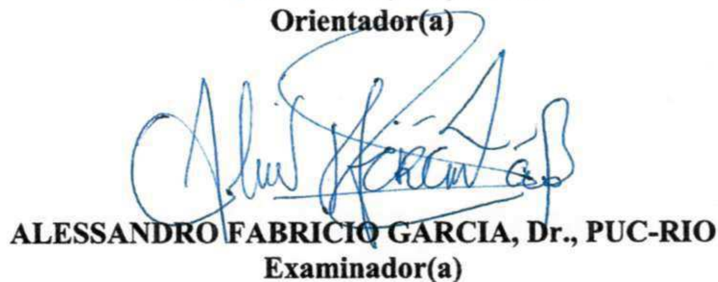
CDU 004.41(043)

"UMA ABORDAGEM AUTOMATIZADA PARA TESTAR FERRAMENTAS DE REFATORAMENTO"

GUSTAVO ARAUJO SOARES

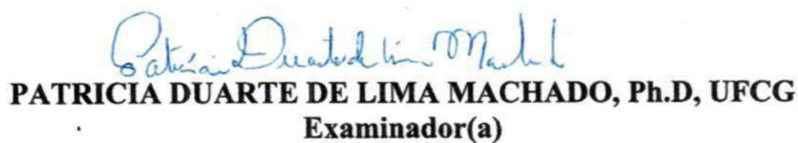
TESE APROVADA COM DISTINÇÃO EM 27/02/2014

ROHIT GHEYI, Dr., UFCG
Orientador(a)

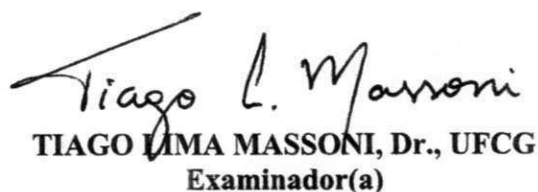


ALESSANDRO FABRICIO GARCIA, Dr., PUC-RIO
Examinador(a)

PAULO HENRIQUE MONTEIRO BORBA, Ph.D, UFPE
Examinador(a)



PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Examinador(a)



TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Refatoramento é uma transformação aplicada a um programa para melhorar suas qualidades internas sem alterar seu comportamento observável. Apesar de trazer benefícios, como facilitar a manutenção, refatorar também envolver riscos, como introduzir erros de compilação ou mudanças comportamentais. Para ajudar o desenvolvedor nesse processo, surgiram as ferramentas de refatoramento. Elas checam condições necessárias para garantir a preservação do comportamento, e quando estas condições são satisfeitas, aplicam a transformação. No entanto, é difícil identificar o conjunto mínimo e completo de condições para cada refatoramento. Se uma condição não é implementada, a ferramenta pode alterar o comportamento do programa. Por outro lado, desenvolvedores podem implementar condições que não só previnem mudanças comportamentais, mas também impedem a aplicação de transformações que preservam comportamento, diminuindo a aplicabilidade da ferramenta. Estas condições são conhecidas como condições muito fortes. Nesse trabalho, propomos uma técnica para testar ferramentas de refatoramento para Java com o objetivo de avaliar se o conjunto de condições implementadas é mínimo e completo. Primeiro, geramos automaticamente um conjunto de programas para serem refatorados. Para isso, propomos um gerador de programas Java, JDOLLY, que gera exaustivamente programas para um determinado escopo de elementos. Para cada programa gerado, aplicamos o refatoramento utilizando a ferramenta em teste. Para detectar falhas nas transformações, utilizamos o SAFEREFACTOR, uma ferramenta que propomos para detectar mudanças comportamentais. Por outro lado, quando as transformações são rejeitadas pela ferramenta, propomos uma abordagem de teste diferencial para detectar condições fortes. A técnica compara o resultado da ferramenta em teste com os resultados de outras ferramentas. Por fim, as falhas detectadas são classificadas em tipos distintos de faltas. Nós avaliamos a eficiência da nossa técnica testando três ferramentas: Eclipse 3.7, NetBeans 7.0.1, e duas versões do JastAdd Refactoring Tools (JRRTv1 e JRRTv2). Foram testados até 10 implementações de refatoramento em cada ferramenta. No total, nossa técnica identificou 34 faltas relacionadas a condições não implementadas no Eclipse, 51 faltas no NetBeans, 24 faltas no JRRTv1, e 11 faltas no JRRTv2. Além disso, foram encontradas 17 e 7 condições muito fortes no Eclipse e JRRTv1, respectivamente.

Abstract

Refactoring is a transformation applied to the program to improve its internal structure without changing its external behavior. Although it brings benefits, such as making it easier to maintain the code, it also involves risks, such as introducing compilation errors or behavioral changes. To help developers in this process, there are refactoring engines. They check conditions needed to guarantee behavioral preservation, and when these conditions are satisfied, they apply the desired transformation. However, identifying and implementing the complete and minimal set of conditions for each refactoring are non-trivial tasks. In practice, tool developers may not be aware of all conditions. When some condition is not implemented, the tool may change the program's behavior. On the other hand, they may also implement conditions that not only prevent behavioral changes, but also prevent behavior-preserving transformations, reducing the applicability of these tools. In this case, we say they implemented an overly strong condition. In this work, we propose a technique for automated testing of Java refactoring engines to identify problems related to missing conditions and overly strong ones. First, we automatically generate programs to be refactored, as test inputs. To do so, we propose a Java program generator, JDOLLY, that exhaustively generates programs for a given scope of Java elements. Then, for each generated program, the desired refactoring is applied by using the engine under tests. To detect failures in the applied transformations, we use SAFEREFACTOR, a tool that we proposed for detecting behavioral changes. On the other hand, when the transformations are rejected by the engine, we propose an differential testing technique to identify overly strong conditions. It compares the results of the engine under tests with results of other engines. The final step of the technique is to classify the detected failures into distinct faults. We evaluated the effectiveness of the technique by testing up to 10 refactorings implemented by three tools: Eclipse 3.7, NetBeans 7.0.1, and two versions of JastAdd Refactoring Tools (JRRTv1 and JRRTv2). Our technique identified 34 faults related to missing conditions in Eclipse, 51 ones in NetBeans, 24 ones in JRRTv1, and 11 ones in JRRTv2. In addition, it detected 17 and 7 overly strong conditions in Eclipse and JRRTv1, respectively.

Agradecimentos

Gostaria de agradecer, primeiramente, a Deus por estar ao meu lado em todas as etapas da minha vida. Esse doutorado também não seria possível sem a contribuição de algumas pessoas. Em primeiro lugar, eu tive muita sorte de ter Rohit Gheyi como meu orientador nesses quatro anos de doutorado na UFCG. Mesmo com pouco tempo como professor, não acredito que eu poderia ter tido orientador melhor. Rohit tem ideias brilhantes e sempre acredita que o trabalho pode ser melhorado. Ele sempre buscou me ensinar todas as habilidades para se tornar um pesquisador completo, desde a escolha do problema à arte de escrever e apresentar bem. E isso não foi uma tarefa fácil, mas ele sempre acreditou no meu potencial e me incentivou a chegar a lugares que nem eu imaginaria que poderia chegar. Obrigado, Rohit, por tudo o que você fez por mim, e por ter se tornado um grande amigo!

Eu gostaria de agradecer também a Emerson Murphy-Hill, por ter me orientado durante meu doutorado sanduíche na North Carolina State University. Apesar do curto período, seus conselhos foram fundamentais não só para a pesquisa que eu desenvolvi lá mas também para me tornar um pesquisador mais completo. Emerson chegou a fazer um "pair writing" (algo como pair programming para artigo) comigo na época de escrita de artigo. Além disso, pude assistir suas aulas sobre "aspectos humanos na engenharia de software", o que me ajudou a abrir mais meu leque de conhecimentos. Agradeço também a todos os membros do seu grupo que me acolheram muito bem em Raleigh: Xi Ge, Jim Witschey, Brittany Johnson, e Yoonki Songo. Não posso deixar de agradecer também ao programa Ciência Sem Fronteiras por ter viabilizado os recursos financeiros para essa viagem, e a Sérgio Soares por ter me ajudado no processo para receber o recursos.

Agradeço a Sumit Gulwani, por ter me orientado no meu estágio em pesquisa na Microsoft Research. Foi um prazer trabalhar com uma pessoa brilhante como Sumit. Nunca vou me esquecer de sua paixão e dedicação pelo trabalho e a vontade de usar seu conhecimento em computação para ajudar as pessoas. Serei eternamente grato pela oportunidade que ele me deu e por tudo que ele conseguiu me ensinar em tão pouco tempo. Agradeço também aos amigos que fiz na Microsoft: João Moreira, Rafael Auler, Ana Riekstin, Dileep Kini, Vu Le, Rohit Sinha, Iury Dewar, Gabor Simkó, e Nóra Bálint. Foi uma experiência fantástica.

Gostaria de agradecer também aos membros da minha banca de doutorado pelas contribuições neste trabalho: Alessandro Garcia, Paulo Borba, Patrícia Machado, e Tiago Massoni. Agradeço ao Max Schaefer por disponibilizar o JRRT e ajudar na classificação dos bugs encontrados. Agradeço aos membros da banca do simpósio de doutorado do SPLASH 2012, pelos seus comentários sobre meu trabalho: Michel Ernst, Cristina V. Lopes, Matthew B. Dwyer, Hridesh Rajan, Milind Kulkarni, e Andrew P. Black.

Gostaria de agradecer também a todas as pessoas que fizeram da UFCG e de Campina Grande um lugar especial para mim. Agradeço aos meus amigos: João Arthur Brunet, Melina Mongiovi, Catuxe Varjão, Elthon Oliveira, Larissa Braz, Alan Moraes, Marco Rosner, Jemerson Damásio, Paulo Ditarso, Marcus Carvalho, Nazareno Andrade, Laerte Xavier, e todos os outros amigos que fiz aqui. Agradeço também a UFCG, aos professores e aos funcionários do DSC/COPIN. Agradeço a todos os membros do grupo Software Productivity Group, tanto na UFCG, quanto na UFPE, aonde participei várias vezes de workshops que ajudaram a melhorar meu trabalho. Agradeço ao CNPq, Capes, e ao Instituto Nacional de Engenharia de Software por apoiarem meu trabalho.

Agradeço a minha namorada, Yohanna Klafke, por estar ao meu lado nessa jornada, pela compreensão nesses últimos dois anos de muito trabalho, e por sempre ter me apoiado e incentivado a ir mais longe. Queria agradecer também a toda minha família. Especialmente aos meus pais, Ronaldo e Graça, e meus irmãos. Meus pais sempre foram para mim um modelo de dedicação, honestidade, e desejo pelo conhecimento. Me incentivaram na minha carreira acadêmica desde o primeiro momento que eu pensei em seguí-la. E de lá para cá, sempre me apoiaram. Nada disso seria possível sem eles. Eu dedico essa tese ao meus pais.

UFCG/BIBLIOTECA/BC

Contents

1 Introduction 1

1.1 Problem . . . 2

1.1.1 Missing conditions . . . 2

1.1.2 Overly strong conditions . . . 5

1.1.3 Research questions . . . 6

1.2 Solution . . . 7

1.3 Evaluation . . . 8

1.4 Summary of contributions . . . 9

1.5 Organization . . . 10

2 Background 11

2.1 Program refactoring . . . 11

2.1.1 Example . . . 11

2.1.2 Refactoring engines . . . 15

2.1.3 Behavioral preservation . . . 17

2.1.4 Refactoring verification . . . 19

2.2 Testing overview . . . 22

2.2.1 Test case . . . 22

2.2.2 Oracle . . . 23

2.2.3 Test coverage criteria . . . 24

2.2.4 Testing refactoring engines . . . 24

2.3 Alloy Overview . . . 32

2.4 Concluding remarks . . . 34

3 JDOLLY: A Java program generator **35**

3.1 Overview . . . 35

3.2 Java metamodel . . . 36

3.2.1 Abstract syntax . . . 36

3.2.2 Well-formedness rules . . . 38

3.3 Program generation . . . 39

3.4 Generating more specific programs . . . 40

3.5 Evaluation . . . 40

3.5.1 Definition . . . 40

3.5.2 Planning . . . 41

3.5.3 Operation . . . 44

3.5.4 Discussion . . . 45

3.5.5 Answers to the research questions . . . 46

3.5.6 Threats to validity . . . 47

3.6 Concluding remarks . . . 47

4 SAFEREFACTOR **49**

4.1 Overview . . . 49

4.2 Evaluation . . . 52

4.2.1 Compared techniques . . . 52

4.2.2 Definition . . . 53

4.2.3 Planning . . . 55

4.2.4 Operation . . . 56

4.2.5 Discussion . . . 61

4.2.6 Threats to validity . . . 67

4.3 Concluding remarks . . . 70

5 A technique for testing of refactoring engines **72**

5.1 Overview . . . 72

5.2 Test input generation . . . 73

5.3 Refactoring application . . . 73

5.4 Test oracles . . . 73

5.4.1 Missing conditions 74

5.4.2 Overly strong conditions 74

5.5 Failure classification 76

5.5.1 Missing conditions 76

5.5.2 Overly strong conditions 80

5.6 Evaluation: missing conditions 80

5.6.1 Planning 80

5.6.2 Operation 83

5.6.3 Discussion 86

5.6.4 Threats to Validity 89

5.7 Evaluation: overly strong conditions 91

5.7.1 Threats to Validity 98

5.8 Concluding remarks 99

6 Related Work **100**

6.1 Refactoring verification and testing 100

6.2 Automated Testing 103

6.3 Empirical studies on refactoring 104

6.4 Concluding remarks 106

7 Conclusions **108**

7.1 Future work 110

A Java metamodel specification in Alloy **125**

B Algorithms for checking refactoring scope and granularity **132**

List of Figures

2.1 Pull Up Field from Eclipse. (a) developer selects the desired refactoring; (b) developer configures additional parameters, and confirms the refactoring by pressing the Finish button. 16

2.2 Eclipse 3.7 preview of the desired refactoring. 16

2.3 Rule for applying a refactoring in ROOL [8]. 20

2.4 Test cases created by JRRT developers to evaluate the Pull Up Method refactoring implementation. 25

2.5 Programs representing Java Inheritance Graphs. 29

2.6 A UML class diagram and its representation in Alloy. 33

3.1 The Java metamodel specified in JDOLLY. 37

3.2 Translation of an Alloy solution to a Java program. (a) A solution of the Java metamodel generated by Alloy Analyzer; (b) the translation of the solution into a concrete Java program. 39

3.3 Programs representing the generation of Java Inheritance Graphs by UDITA and JDOLLY for the scope of two elements. 46

3.4 Isomorphic programs generated by JDOLLY. 46

4.1 Safe Refactor's technique; 1) The tool identifies the methods with same signature before and after the transformation; 2) It generates a test suite for the identified methods using Randoop; 3) It runs the tests on the source program; 4) It runs the tests on the target program; 5) Finally, Safe Refactor evaluates the results: if they are different, the tool reports a behavioral change. Otherwise, the developer can increase confidence that the programs have the same behavior. 50

5.1 Automated behavioral testing of refactoring engines. 73

List of Tables

2.1 Industrial Java refactoring engines [88]. 15
2.2 Refactorings supported by Eclipse. 17

3.1 Comparison of JDOLLY and UDITA; Prog.: Number of generated programs; Comp.: number of compilable programs; Isomor: number of isomorphic programs; Unique: number of unique programs; NG: number of unique programs that were not generated. 45

4.1 Results of analyzing 40 versions of JHotDraw; LOC = non-blank, non-comment lines of code before and after the changes; Granu.: granularity of the transformation; Scope: scope of the transformation; Refact. = Is it a refactoring?; #Tests = number of tests used to evaluate the transformation; Cov. (%) = statement coverage on the target program; MH = Murphy-Hill. . 58
4.2 Results of analyzing 20 versions of Apache Common Collections; LOC = non-blank, non-comment lines of code before and after the changes; Granu.: granularity of the transformation; Scope: scope of the transformation; Refact. = Is it a refactoring?; #Tests = number of tests used to evaluate the transformation; Cov. (%) = statement coverage on the target program; MH = Murphy-Hill. 59
4.3 Summary of false positives, false negatives, true positives, and true negatives. 60
4.4 False positives of SAFEREFACTOR; Problem = description of the reason of the false positive; Versions = ids of the versions related to the false positives. 62

5.1 Filters for classifying behavioral changes. 79

5.2 Summary of evaluated refactorings; Scope = Package (P) - Class (C) - Field (F) - Method (M). 81
5.3 Summary of the main constraints. 82
5.4 Summary of the additional constraints. 83
5.5 Summary of faults reported. 84
5.6 Overall experimental results; GP = number of generated programs; CP = number of compilable programs (%); Time = total time to test the refactoring in hours; Fail. = number of detected failures; Bug = number of identified faults. 85
5.7 Summary of evaluated refactoring implementations. 92
5.8 Summary of the experiment; Program = number of programs generated by JDolly; Rejected Transformation = number of transformations rejected by the implementation; Rejected B. Pres. Transformation = number of behavior-preserving transformations that were rejected; Overly strong condition = number of overly strong conditions classified by our technique. . . 94
5.9 Summary of overly strong conditions of Eclipse 3.7 and JRRTv1. 96

Chapter 1

Introduction

During the life cycle of a software, its maintenance and evolution are inevitable. After its release, clients demand new requirements and revealed faults need to be fixed. The more the software is modified, the more complex its code become, making it more difficult to be maintained. To avoid that, developers need to restructure the code, improving its internal structure, while preserving its external functionalities; a kind of maintenance known as perfective [1]. The process of changing the internal structure of a program to improve its internal qualities but preserving its external behavior is known as *refactoring*. This term was coined by Opdyke and Johnson [54; 53], and latter, popularized in practice by Fowler [19].

Fowler [19] proposes to perform refactorings by applying small changes intercalated with compilation checks and tests to guarantee successful compilation and behavioral preservation. While compilation checks guarantees the absence of compilation errors after the transformation, tests evaluate whether the behavior of the program is preserved. In other words, refactorings must not only produce well-formed programs, but also the versions of the programs before and after refactoring must have the same external behavior.

To help developers in this activity, Don Roberts proposed the first *refactoring tool*, Refactoring Browser, which automates a number of refactorings for Smalltalk [65]. A refactoring tool automates the process of checking *conditions* that must be satisfied in order to apply the transformation. For instance, to pull up a method `m` to a superclass, we must check whether `m` conflicts with the signature of other methods in that superclass. Currently, most Java Integrated Development Environments (IDEs), such as Eclipse [16], NetBeans [85], JBuilder [18], IntelliJ [35], automate some refactorings.

UFCG/BIBLIOTECA/BC

1.1 Problem

Defining and implementing the minimal set of conditions needed for each refactoring are non-trivial tasks. One can prove the correctness of this set for a language with a simple and formal semantics. For instance, Proietti and Pettorossi [58] propose a formal semantics for Prolog and prove some transformation rules. However, a number of popular languages, such as Java, C, and C#, have a complex semantics without a complete formal definition considering all elements of the language, which makes it difficult to prove refactoring correctness. In this work, we focus on problems for specifying and implementing refactorings for Java programs. Java is one of the most popular languages,[1] and was used by Fowler [19] to illustrate all refactorings presented in his catalog. Moreover, modern IDEs for Java, such as Eclipse and NetBeans, contain a number of automated refactorings.

1.1.1 Missing conditions

In practice, refactoring tool developers may not be aware of all refactoring conditions. If some condition is missing, the refactoring engine may perform transformations that introduce compilation errors or behavioral changes. For instance, consider the Java program illustrated in Listing 1.1. The method `B.test()` yields 1. If we use Eclipse 3.7 to perform the Pull Up Method refactoring on `m()`, the tool will move method `m` from class `B` to class `A`, and update `super` to `this`. This transformation introduced a behavioral change: `test` yields 2 instead of 1. Since `m` is invoked on an instance of `B`, the call to `k` using `this` is dispatched to the implementation of `k` in `B`.

Formal methods

Researches have tried to handle the problem of missing conditions by formally specifying refactorings considering a subset of the language [8; 13; 86; 75; 71; 74; 68; 84; 51]. They provide guidelines and techniques that can be useful for developing refactoring engines. Previous approaches include analyses of some of the various aspects of a language, such as: accessibility, types, name binding, data flow, and control flow. For instance, Borba et al. [8] propose a set of refactorings for a subset of Java with copy semantics, a language called

[1] `http://langpop.com/`

Refinement Object-Oriented Language (ROOL). For each refactoring, they propose a set of conditions that guarantee behavioral preservation. They prove the refactoring correctness with respect to a formal semantics for a subset of Java. However, they have not considered all Java constructs, such as overloading and field hiding. Considering the whole Java language, the proposed conditions may not be enough.

Listing 1.1: Pulling up `B.k()` by using Eclipse 3.7 or JRRTv1 changes program behavior.

```
1  public class A {
2    int k() {
3      return 1;
4    }
5  }
6  public class B extends A {
7    int k() {
8      return 2;
9    }
10   int m() {
11     return super.k();
12   }
13   public int test() {
14     return m();
15   }
16 }
```

Recently, Schäfer and Moor [68] specified and implemented a number of refactorings for Java, and proposed a tool called JastAdd Refactoring Tools (JRRT) [68]. For each refactoring, as correctness criteria, they proposed some invariants that should be preserved to guarantee behavioral preservation. For instance, the Rename Method refactoring should preserve name binding. However, the same problem illustrated in Listing 1.1 occurs when we apply this transformation by using JRRTv1[2]. Proving refactoring correctness for the entire language constitutes a challenge [70].

[2]The JRRT version from May 18th, 2010

Testing

Although we cannot prove the absence of faults by using software tests, testing approaches have been useful in detecting faults in refactoring engines related to missing conditions. Daniel et al. [14] propose an approach of *bounded-exhaustive testing* [42] to automate this process. While manual testing requires manually identifying and writing each test case, bounded-exhaustive testing exhaustively tests all inputs for a given bound. They used a program generator (ASTGen) to generate programs as test inputs. To evaluate the engines' outputs, they implemented test oracles. These oracles check for compilation errors, and try to detect behavioral changes by applying static analysis. For instance, they apply the inverse refactoring to the output program and expect that the result be equal to the input program.

Although the approach proposed by Daniel et al. [14] identified a number of faults, we can point out some limitations in their program generator and test oracles. First, most of the faults that they identified are related to compilation errors in users' code. They identified only one fault related to behavioral changes. Second, ASTGen allows users to directly implement how the program will be generated. However, for some Java constructions, implementing how they will be combined does require some effort. Therefore, it may be difficult to generate a large variability of programs, potentially leaving many hidden faults. Later, Gligoric et al. [22] proposed (UDITA), a Java-like language that extends ASTGen allowing users to specify what is to be generated (instead of how to generate), and uses the Java Path Finder (JPF) model checker as a basis for searching for all possible combinations. By using UDITA, they found 4 new faults related to compilation errors in Eclipse.

In my Master's thesis [82], we propose SAFEREFACTOR. It analyzes a transformation, and generates tests for checking behavioral changes. We describe it along with the evaluation of 24 specific transformations applied to small examples and real open source projects (such as JHotDraw and JUnit). SAFEREFACTOR detected a number of behavioral changes. Additionally, we proposed an approach and its implementation (JDOLLY) for generating Java programs by using Alloy [32], a formal specification language, and ASTGen. It uses Alloy for generating the structural parts of the programs and ASTGen to generate the methods' bodies of the programs. We also proposed an approach for testing refactoring engines by using JDOLLY and SAFEREFACTOR. As a result, SAFEREFACTOR was useful for finding 50 faults in Eclipse 3.4.2 that lead to compilation errors and behavioral changes in users'

code.

By combining Alloy with ASTGen our technique increased the variability of generated programs, which was useful for finding more faults. However, it also required users to learn two technologies (Alloy and ASTGen) to specify the program generation. Users also need to synchronize the generation of the structural parts of the programs with the generation of the method bodies. Additionally, we lack evaluation to answer some questions about such a testing approach. First, can we generate programs with more expressivity to test refactorings? For example, the programs generated by JDOLLY do not contain packages, a common construct in Java programs. Steimann and Thies [84] show some faults in refactoring engines in the presence of packages. Second, is this testing approach good enough for finding faults in other refactoring engines? For example, JRRT developers used ASTGen to test their implementations but did not find any fault [71]. Finally, in spite of SAFEREFACTOR having being useful for catching a number of behavioral changes, we still need further evaluation to understand in which scenarios it can detect behavioral changes and in which ones it cannot.

These testing approaches may find a number of transformations that introduce compilation errors and behavioral changes. Some of these transformations may be related to the same fault in the refactoring engine. An important step is to analyze each one of these transformations to report the distinct faults found. Jagannath et al. [34] propose an approach to classify the faults related to compilation errors by the template of the compiler error message. However, there is no approach for classifying faults related to behavioral changes.

1.1.2 Overly strong conditions

So far, we have discussed about how difficult is to check whether the implemented conditions guarantee behavioral preservation. But we should also check whether these conditions not only avoid behavioral changes but prevent useful behavior preserving transformations. Due to the complexity of a large language as Java, developers may not realize that some condition will prevent some behavior-preserving transformation, reducing the applicability of the tool. Additionally, some conditions may be too difficult to implement, which may lead developers to implement less precise approximations. When a condition prevent behavior preserving transformations, we call it as a *overly strong condition*.

For example, consider the Java program in Listing 1.2. The class `A` declares the method

`k(long)`, and the class `B` declares methods `n` and `test`. Suppose we would like to rename `n` to `k`. If we apply this transformation by using Eclipse 3.7, the tool will not apply the transformation showing a warning message. However, we can apply this transformation by using JRRTv1. It performs an additional change to make the transformation behavior-preserving by adding a `super` access to the method invocation `k(2)` inside `test`.

Listing 1.2: Eclipse 3.7 prevents renaming B.n to B.k but JRRTv1 correctly applies the transformation.

```
1  public class A {
2    public long k(long l) {
3      return l;
4    }
5  }
6  public class B extends A {
7    public long n(int i) {
8      return 2;
9    }
10   public long test() {
11     return k(2);
12   }
13 }
```

To the best of our knowledge, there is no automated testing approach to detect and classify overly strong conditions.

1.1.3 Research questions

Given the problems shown in Sections 1.1.1 and 1.1.2, we focus on the following research question:

- **RQ1:** How can we automate Java program generation for generating test inputs useful for detecting faults in Java refactoring engines?
- **RQ2:** How can we automatically evaluate a refactoring engine output to detect faults related to overly weak and overly strong conditions?
- **RQ3:** What is the effectiveness of SAFEREFACTOR?

UFCG/BIBLIOTECA/BC

- **RQ4:** How can we classify transformations that lead to behavioral changes and overly strong conditions into distinct faults?

1.2 Solution

In this work, we propose a technique for automated testing of Java refactoring engines. Its goal is to identify missing conditions that lead to compilation errors or behavioral changes in sequential (non-concurrent) Java programs and overly strong conditions that prevent behavior-preserving transformations in sequential Java programs.

First, we automatically generate programs to be refactored, as test inputs. To do so, we propose a Java program generator called JDOLLY. It exhaustively generates programs for a given scope of Java declarations (packages, classes, fields, and methods). It contains a subset of the Java metamodel specified in Alloy [32]. It also employs the Alloy Analyzer [33], a tool for the analysis of Alloy models, to generate solutions for this metamodel. Each solution is translated into a Java program. Differently from our previous technique [82], which combines the Alloy Analyzer with ASTGen for generating programs, JDOLLY can generate entire programs using only the Alloy analyzer as enabling technology for finding all possible programs for a given scope. This difference avoids the need for developers to learn two different technologies to specify the program generation.

For each generated program, the desired refactoring is applied by using the refactoring engine under test. Then, the technique uses the following oracles to evaluate the output. To detect failures in the applied transformations, we use SAFEREFACTOR, a tool that we proposed for checking behavioral changes. On the other hand, when the transformations are rejected by the engine, we propose a differential testing technique based on SAFEREFACTOR to identify overly strong conditions. For the same input program, it compares the result of the engine under tests with results of other engines. Although, in my Master's thesis [82], we had already used SAFEREFACTOR for detecting faults related to missing conditions, here we combine it with differential testing to also detect faults related to overly strong conditions.

Manually inspecting all failures detect by our technique may require a lot of effort. The final step of the technique is to classify these failures into distinct faults. To classify failures related to compilation errors, we use an approach [34] that classifies the failures by using

the template of the compiler error message. We uses a similar approach to classify failures related to overly strong conditions, splitting them by the template of the warning message. On the other hand, to classify failures related to behavioral changes, we classify them based on the structural characteristics of the transformations. In contrast, the technique proposed in my Master's thesis [82] does not classify the failures into distinct faults.

1.3 Evaluation

We have conducted experiments[3] to evaluate our technique for testing of refactoring engines, and its components, JDOLLY and SAFEREFACTOR, with respect to our research questions.

We evaluated our technique with respect to effectiveness on finding faults due to missing conditions. We used it to test three refactoring engines: Eclipse JDT 3.7, NetBeans 7.0.1, and two versions of the JastAdd Refactoring Tools (JRRTv1 and JRRTv2) [71; 74; 68]. We tested up to 10 refactorings implemented by each engine. We assessed 153,444 transformations, and identified 57 faults related to compilation errors, and 63 faults related to behavioral changes. We reported all faults to the tools' developers, who have confirmed 90 out of 120 so far. Moreover, they have already fixed 35 faults reported by us.

We also conduct an experiment to evaluate the technique with respect to effectiveness in identifying overly strong conditions. We used the technique to test three Java refactoring engines (Eclipse JDT 3.7, NetBeans 7.0.1, and JRRTv1). For each engine, we tested up 10 refactoring implementations in a sample of 42,757 transformations. We found that 16% and 7% of transformations rejected by Eclipse and JRRT, respectively, are behavior-preserving. The implementations have overly strong conditions avoiding correct transformations to be applied. Our technique automatically categorized them in 17 and 7 kinds of overly strong conditions of Eclipse and JRRT, respectively. We reported all faults to the tools' developers. So far, they have accepted 11 faults and fixed 3 of them.

With respect to JDOLLY, we perform an experiment to compare JDOLLY and UDITA [22] with respect to effectiveness and efficiency in generating Java inheritance graphs. Our results shows that JDOLLY exhaustively generates solutions for a given scope.

[3]All experimental data are available at: `http://www.dsc.ufcg.edu.br/~gsoares/thesis-experiments.html`

On the other hand, UDITA failed to generate some solutions. Additionally, JDOLLY was faster than UDITA but generated more isomorphic (structurally equivalent) solutions, which is not desired since two or more programs with the same structure do not increase the change of finding new faults.

In regard to SAFEREFACTOR, we performed an empirical study to evaluate its effectiveness in detecting behavioral changes on a sample of 60 transformations gathered from two repositories of open source Java projects. We compared SAFEREFACTOR's results with the results of a manual analysis and the results of an automated approach for detecting refactorings by analyzing commit messages [61]. In this study, SAFEREFACTOR had 70% accuracy. In Section 4.2, we its discuss advantages and limitations when testing real Java programs.

1.4 Summary of contributions

The main contributions of this thesis can be summarized as follows:

- We propose an automated technique for testing of Java refactoring engines with respect to missing conditions and overly strong ones. We report on the results of experiments to show the effectiveness of our technique reporting 120 missing conditions and 24 overly strong ones to refactoring engine developers [77; 81; 79; 76; 83];

- We propose and implement a technique (JDOLLY) for generating Java programs that allows users to use Alloy constraints to guide the program generation. We show that JDOLLY is useful for generating test inputs for testing of refactoring engines. Our results also suggest that JDOLLY exhaustively generates programs for a given scope. On the other hand, UDITA [22] failed to generate all programs for a given scope [77; 76; 83];

- We report on the results of an experiment to show that SAFEREFACTOR has 70% accuracy in detecting transformations that preserve programs behavior and transformations that do not [78].

1.5 Organization

This thesis is organized as follows. In Chapter 2, we provide some background on program refactoring, testing, and Alloy. In Chapter 3, we present JDOLLY, our Java program generator, and its evaluation. In Chapter 4, we give an overview of SAFEREFACTOR, and present an evaluation of SAFEREFACTOR on 60 transformations of real Java programs. Then, in Section 5, we describe our technique for testing of Java refactoring engines. Moreover, we show its evaluation by testing real Java refactoring engines. Chapter 6 presents the related work, and Chapter 7 summarizes the contributions of the thesis and future work. Finally, Appendix B shows some algorithms used in the experiment shown in Section 4.2.

Chapter 2

Background

In this chapter, we present the background needed for the understanding of this work. First, we explain refactoring, and show an overview of the state-of-the-art in this area (Section 2.1). Then, we present some important concepts related to testing, and introduce testing of refactoring engines (Section 2.2). Finally, in Section 2.3, we give an overview of Alloy [32], a formal specification language, which we use to build JDOLLY, our program generator.

2.1 Program refactoring

The term refactoring was coined by Opdyke, in his PhD thesis [53]. Then, it was popularised by Fowler [19]. He defines refactoring as follows:

> *"It is a change made to the internal structure of a software to make it easier to understand and cheaper to modify without changing its observable behavior."*

Fowler also defines refactoring as a verb [19]:

> *"It is to restructure software by applying a series of refactorings without changing its observable behavior."*

2.1.1 Example

In this section, we give a refactoring example. First, we show the process of identifying which part of the code should be refactored, and then, we show the appropriated refac-

toring to be applied. To this example, consider superclass `Employee` and its subclasses `Engineer` and `Analyst` shown in Listing 2.1.

Listing 2.1: Program containing duplicated code.

```
1  public class Employee {
2    ...
3  }
4  public class Engineer extends Employee {
5    private double salary;
6    public double getSalary() {
7      return salary;
8    }
9    ...
10 }
11 public class Analyst extends Employee {
12   private double salary;
13   public double getSalary() {
14     return salary;
15   }
16   ...
17 }
```

Bad Smells

First, we should identify the code that should be refactored. To help the developer in this process, Beck [19] categorized 21 cases where there are points in the code indicating that it should be improved. Beck refer to these signs as *bad Smells*.

The first *bad smell* that he presents is the duplicated code. When the same code appears in different parts of the program, the maintenance of it may become difficult, since it is needed to apply the change to all duplications of the code. Therefore, it is better to find a way to remove duplicated code. For instance, by looking at the code shown in Listing 2.1, we notice that method `getSalary` and field `salary` are declared in the two subclasses. We thus should refactor that code to avoid this duplication.

Other examples of *bad smells* are: long methods, large classes, and long parameter list[19; 88].

Choosing and applying refactorings

Fowler [19; 62] defined a refactoring catalog. For each refactoring, he shows the motivation and the process to apply it. To remove the duplicated code of our example, we will apply two refactorings presented in Fowler's catalog.

First, we use the *Pull Up Field* refactoring to move the fields to the superclass. Fowler [19] defines the following steps to apply this refactoring:

1. Inspect the declaration of the candidate fields to assert that they are initialized in the same way;
2. If the fields do not have the same name, rename them so that they have the name you want;
3. Compile and test;
4. Create a new field in the super class. If the fields are private, you should declare it as `protected` so that the subclass can access it;
5. Remove the fields from the subclasses;
6. Compile and test.

We apply the refactorings by using small steps intercalated with compilation check and tests to guarantee that the transformation preserves the external behavior of the program.

Listing 2.2 shows the program after the performed refactoring. Notice that it was needed, as indicated in step 4 of Fowler's catalog, to change the access modifier of the field from `private` to `protected` to allow its access from the subclasses.

Listing 2.2: Program after applying the Pull Up Field refactoring.

```
1 public class Employee {
2   protected double salary;
3   ...
```

```
4  }
5  public class Engineer extends Employee {
6    public double getSalary() {
7      return salary;
8    }
9    ...
10 }
11 public class Analyst extends Employee {
12   public double getSalary() {
13     return salary;
14   }
15   ...
16 }
```

After removing the duplicated fields, we can apply the *Pull Up Method* refactoring [19] to move the `getSalary` methods to the superclass. We can apply this refactoring since both implementations of the method have the same behavior. Listing 2.3 shows the resulting program after applying the two refactorings.

Listing 2.3: Program after applying the Pull Up Method refactoring.

```
1 public class Employee {
2   protected double salary;
3   public double getSalary() {
4     return salary;
5   }
6   ...
7 }
8 public class Engineer extends Employee { ... }
9 public class Analyst extends Employee { ... }
```

Manually applying refactoring is time consuming and error prone. Fowler [19] suggests to use small steps intercalated with compilation checks and tests as a safer approach to apply refactorings. Besides that, there are tools that automate this process. In the next section, we show an overview of these tools.

2.1.2 Refactoring engines

The first *Refactoring engine*, Refactoring Browser [65], was proposed by Roberts in his PhD thesis. It implements a number of refactorings for the Smalltalk [23] language. Refactoring has become more popular, and so most of the current IDEs have implemented refactorings to support developers. Table 2.1 shows some IDEs that provide Java refactoring engines [88].

Tool	Company	Type	URL
CodeGuide	Omnicore	IDE	www.omnicore.com
Eclipse		IDE	www.eclipse.org
Idea	IntelliJ	IDE	www.intellij.com
JavaRefactor		Plugin for jEdit	plugins.jedit.org/plugins/?JavaRefactor
JBuilder	Borland	IDE	www.borland.com/jbuilder
JFactor	Instatiations	Plugin for Jbuilder and VisualAge	www.instatiations.com/jfactor
JRefactory		Plugin for Elixir, JBuilder and NetBeans	jrefactory.sourceforge.net
NetBeans	Sun Microsystems	IDE	www.netbeans.org
Transmogrify		Plugin for JBuilder and Forte4Java	transmogrify.sourceforge.net
XRefactory	Xref-Tech	Plugin for Emacs, jEdit and XEmacs	www.xref-tech.com

Table 2.1: Industrial Java refactoring engines [88].

A refactoring engine allows developers to select the refactoring to be applied and the parameters for configuration. The tool automatically checks the refactoring conditions to guarantee behavioral preservation. For instance, when we apply the *Rename Method*, the tool checks whether there are other methods with the same name of the refactored method. If all conditions are satisfied, the tool performs the desired transformation. To exemplify the process, we show the application of the *Pull Up Field* refactoring shown in Section 2.1.1 by using Eclipse. The developer selects the field that will be refactored, and choose *Pull Up* from the *Refactor* menu (Figure 2.1(a)). Eclipse shows a window where the developer can choose additional parameters to apply the refactoring (Figure 2.1(b)).

In addition, Eclipse allows the developer to see the preview of the transformation by pressing the *next* button (Figure 2.1(b)), which allows the developer to manually inspect the correctness of the transformation. Figure 2.2 shows the Eclipse preview containing the changes that will be applied.

Figure 2.1: Pull Up Field from Eclipse. (a) developer selects the desired refactoring; (b) developer configures additional parameters, and confirms the refactoring by pressing the Finish button.

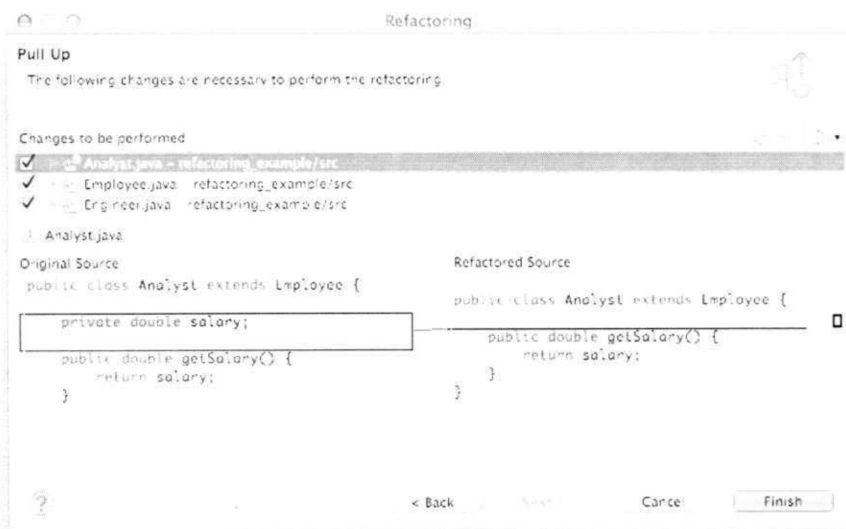


Figure 2.2: Eclipse 3.7 preview of the desired refactoring.

Eclipse was one of the first IDEs to implement refactorings. In its first version, released in the end of 2001, it included the following refactorings: *Rename*, *Move*, and *Extract Method* [20]. The refactorings implemented by Eclipse 3.7 can be seen in Table 2.2. Murphy et al. [48] conducted a survey on Java software development by using Eclipse. They analyzed the use of the Eclipse refactorings by 41 developers. The five most used refactorings were: *Rename*, *Move*, *Extract Method*, *Pull Up Method*, e *Add Parameter*.

Refactorings supported by Eclipse			
Rename (package, class, method, field)	Move (class, method)	Change Method Signature	Extract Method
Extract Local Variable	Extract Constant	Inline (method, variable)	Convert Anonymous Class to Nested
Convert Member Type to Top Level	Extract Superclass	Extract Interface	Use Super Type Where Possible
Push Down (method, field)	Pull Up (method, field)	Extract Class	Introduce Parameter Object
Introduce Indirection	Introduce Factory	Introduce Parameter	Encapsulate Field
Generalize Declared Type	Infer Generic Type Arguments		

Table 2.2: Refactorings supported by Eclipse.

2.1.3 Behavioral preservation

According to the refactoring definition shown in Section 2.1, two programs are equivalent when they have the same *external behavior*. In his PhD thesis, Opdyke formally specified 23 primitive refactorings and other three complex refactorings. Each primitive refactoring contains a set of conditions that guarantee behavioral preservation. For instance, Opdyke defines the following conditions to the *Pull Up Field* refactoring shown in Section 2.1.1:

1. The field should be defined in the same way in all subclasses;

2. The field should not be defined in the superclass.

Notice that if the second condition is violated, it will produce a program with a compilation error due to name conflicts. On the other hand, if the first condition is violated, the program will still compile but it may have different behavior since the value of one of the fields will be changed.

The conditions proposed by Opdyke are based on seven properties defined by him. According to him, these properties assure the correctness of the transformations. They are:

1. *Unique superclass.* Each class in the resulting program must have at most one superclass;
2. *Distinct class names.* All classes in the resulting program must have distinct names;
3. *Distinct member names.* Each class in the resulting program must have distinct variables and function names;
4. *Inherited member variables not redefined.* A member variable inherited from a superclass is not redefined in any of its subclasses;
5. *Compatible signatures in member function redefinition.* Redefinitions of methods have the same signatures as the redefined method;
6. *Compatible signatures in member function redefinition.* In the resulting program, every expression that is assigned to a variable must have the same type or a subtype of the variable's type;
7. *Semantically equivalence references and operations.* The resulting program must have the same output set of the original program for a given set of inputs.

The first six properties are related to preservation of well-formedness of the programs. We can check that by compiling the program after the transformation: if there is any compilation error, it means that the transformation was not correctly applied.

On the other hand, the last property is related to semantics preservation of the program, and thus, compiling the program is not enough to check it. The program can still compile but with a different external behavior of the original one.

Opdyke [53] defines semantics equivalence between programs as follows: "*let the external interface of the program be the main function. If the main function is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same* (p. 40)". This definition of equivalence notion allows a refactoring to change the internal structure of the program as long as the mapping between input and

outputs of the main function be preserved. This definition can be seen as an application of the notion of data refinement [28; 30].

Another way to deal with behavioral preservation is through testing. Roberts [65] defines that a refactoring is correct if after the transformation, the program still is in conformance with its specification. His equivalence notion is based on testing. According to him, a refactoring is correct if a program that passes the tests still passes them after the transformation. Fowler [19] uses the same equivalence notion.

Additionally, in some application domain, guaranteeing that for a set of inputs, the program has the same outputs after the transformation is not enough to state the transformation preserved behavior [45]. For instance, in real-time systems, it should also be considered the time to execute the program as part of its behavior. Also, in embedded systems, the memory space and energy consumption may be used as part of the program's behavior.

2.1.4 Refactoring verification

As shown in Section 1.1, even small transformations may be incorrectly applied by refactoring engines. The ideal solution would be formally specify the conditions for each refactoring and prove them with respect to a formal semantics.

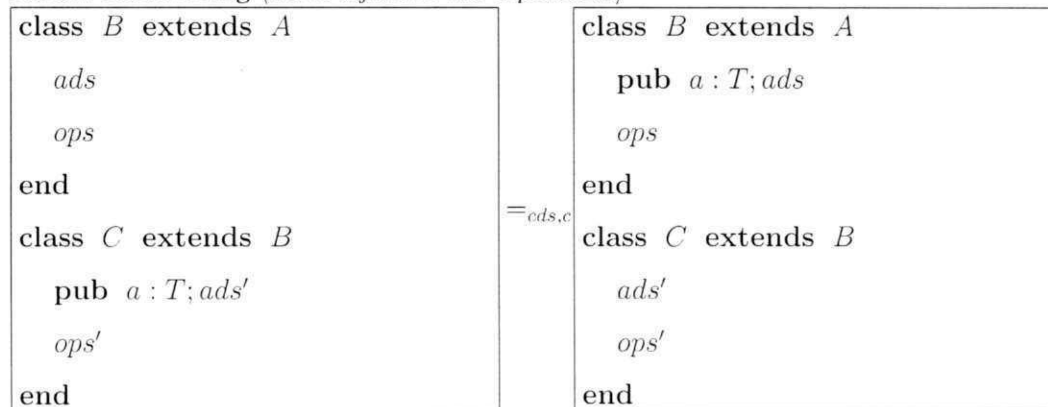
Proving refactoring correctness with respect to a formal semantics is a challenge [70]. Some approaches have contributed in this direction. Borba et al. [8] propose a set of refactorings for a subset of Java with copy semantics, a language called Refinement Object-Oriented Language (ROOL). They prove the refactoring correctness based on a formal semantics. To illustrate this process, next we show a refactoring proposed by them. The following rule formalizes the *Pull Up Field* refactoring when applied from the left hand side to the right hand side and *Push Down Field* when applied from the opposite direction (Figure 2.3). Each refactoring consists of two *templates* of ROOL programs. The refactoring can be applied as long as the programs match the templates, that is, if all variables in the templates can be assigned to the concrete values.

Each refactoring may also contain meta-variables. For instance, cds, ads, e ops are meta-variables that define sets of classes, fields, and operations, respectively. Moreover, the c meta-variable represents the main function. Their equivalence notion are based on comparing the main function with respect to the two versions of the program, in a similar

way of the notion proposed by Opdyke [53]. The ($\rightarrow$) arrow before the condition indicates that it is required when applying the rule from the left to the right. The ($\leftarrow$) arrow indicates a condition when applying it from the opposite direction. Additionally, the arrow ($\leftrightarrow$) indicates a condition needed when applying from both directions. In this example, we can see that to move a field to the superclass, there cannot be another field in the super class with the same name.

Figure 2.3: Rule for applying a refactoring in ROOL [8].

ROOL Refactoring ⟨*Move a field to the superclass*⟩



restrições

($\rightarrow$) The field with name a is not declared in the subclasses of B in cds;

($\leftarrow$) $D.a$, for any $D \leq B$ and $D \nleq C$, does not appear in cds, c, ops, or ops'.

Silva et al. [75] extended these previous laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each one 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.

Schäfer et al. [71] propose a Rename Class, Method and Field refactoring implementations. They state that a renaming must preserve name bindings, that is, each name should refer to the same entity before and after the transformation. Furthermore, Schäfer et al. [74; 68] present a number of Java refactoring implementations. They translate a Java program to an enriched language that is easier to specify and check conditions, and apply the transformation. As correctness criteria, besides using name binding preservation, they used other

invariants such as control flow and data flow preservation.

Steimann and Thies [84] show 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 conditions and computing the changes of access modifiers needed to preserve the program behavior.

Another approach for checking refactorings – generalization-related refactorings such as Extract Interface and Pull Up Method – is proposed by Tip et al. [86]. Their work proposes an approach that uses type constraints to verify conditions of those refactorings, determining which part of the code they may modify. Using type constraints, they also propose the refactoring Infer Generic Type Arguments [21], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [3]. These refactorings are implemented in the Eclipse JDT. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have conditions related to other constructs. Additionally, Schäfer et al. [69] propose refactorings for concurrent programs. They have proved the correctness of them with respect to some concurrency properties based on the Java memory model.

Dig and Johnson [15] analyzed refactorings in the context of software reuse. They analysed changes applied to three frameworks and one library largely used. As a result, they found that more than 80% of the changes made to API that lead to incompatibilities with clients are refactorings. Henkel e Diwan [29] proposed an approach and a tool for evolving an API by using refactorings. Their tool allows recording the applied refactorings to the API to automatically update the client code based on these refactorings.

Some studies have been contributing to popularize refactorings in aspect-oriented programming. Monteiro and Fernandes [47] proposed a catalog of 27 aspect-oriented refactorings. These refactorings aim at introducing aspects and improve the design of them. Cole and Borba [10] formally specify aspect-oriented programming laws (each law defines a bidirectional semantics-preserving transformation) for AspectJ. By composing them, they derive AspectJ refactorings. Each law formally states conditions. They proved one of them sound with respect to a formal semantics for a subset of Java and AspectJ [11]. They can be very useful for implementing aspect-aware refactoring tools. Wloka et al. [92] propose a tool sup-

port for extending currently OO refactoring implementations for considering aspects. They developed an impact analysis tool for detecting change effects on pointcuts to generate pointcut updates. Binkley et al. [7] present a human guided automated approach to refactor OO programs to the AO. Hannemann et al. [27] introduce a role-based refactoring approach to help programmers modularize crosscutting concerns in aspects. These works contribute for improving tool support for refactoring aspect-oriented programs.

2.2 Testing overview

Software testing is the primary method that industry uses to evaluate the software under development [2]. Testing can be defined as an evaluation of the software by observing its execution. There are three common concepts in software testing: *failure*, *fault*, and *error*. According to Binder [6], a fault is a static defect in the software; a system error is an incorrect internal state (the manifestation of some fault); and a failure is an external, incorrect behavior with respect to the expected behavior.

To specify a test, we can use two different techniques: black box testing and white box testing [2]. In the former, the goal is to evaluate whether the program satisfies some functional or non-functional requirement. We thus do not need the program's source code to specify a black box test. On the other hand, white box testing requires the source code in order to select parts of the code to be tested. This thesis focuses on black box testing in the sense that we do not need to look inside the refactoring engines' code to specify the tests. We just need the engine's API. The remainder of this section presents other software testing concepts that are important to the understanding of this thesis.

2.2.1 Test case

The main challenge on software testing is to determine a set of test cases (named test suite) for the software to be tested. A test case is composed of a set of inputs, expected results, and prefix and postfix values [2].

The inputs are values needed to complete some execution of the software under test. On the other hand, the expected result specifies the result that is expected to be produced after executing the test if the program satisfies the requirement. Prefix values are any inputs

needed to set up the software into the appropriate state to receive the inputs. And, postfix values are any inputs that needed to be sent to the software after the test.

For instance, test cases can be created by using JUnit [43], a *framework* for automating unit tests. The framework provides the `assertEquals` method, which compares the value returned by the method under test with the expected value. If the values are different, the test fails, and a red bar is shown in JUnit's GUI. On the other hand, if the values are equal, it shows a green bar. Listing 2.4 shows a unit test for the `getSalary()` method from class `Analyst`. In this test, we instantiate an object of type `Analyst`, set a value for field `salary`, and compare this value with the value returned by the `getSalary` method.

Listing 2.4: Unit test for method getSalary() from class Analyst.

```
1 public void testGeSalary() {
2   Analyst analyst = new Analyst();
3   analyst.setSalary(3000);
4   double expectedValue = 3000;
5   double value = analyst.getSalary();
6   assertEquals(expectedValue, value);
7 }
```

2.2.2 Oracle

A test case passes when the software under test produces the expected result. The pass/no pass evaluation is made by comparing the actual result with the expected one by a trusted mechanism, known as test oracle or just oracle [5; 91].

In many cases this oracle consists of a manual observation of the test input and output, which can be time consuming, tedious and error prone. However, it can also be automated, or partially automated. For instance, the comparison can be manually done by using the programmer's knowledge or automatically done by checking a formal specification. In Listing 2.4, the oracle is partially automated. The developer manually specifies the expected value, and it is automatically checked by using the JUnit framework.

2.2.3 Test coverage criteria

Usually, the number of inputs for a software is so large as to effectively infinite. For instance, potential inputs to a Java compiler are not just all Java programs, but all strings. The only limitation is the size of the file that can be read by the parser. Since we cannot test a software against all inputs, we use test coverage criteria to decide which inputs to use.

Test coverage criterion can be defined as a rule or a collection of rules that impose test requirements on a test set. A test requirement is a specific element of a software artifact that a test case must satisfy or cover [2]. To check how good a test suite is, we can measure it against a criterion in terms of coverage. Coverage is important because sometimes it is expensive or even infeasible to achieve some criteria, so we want at least achieve some test coverage level. There are many test coverage criteria that can be used to evaluate a test suite. For instance, for white box testing, we can measure: statement coverage, branch coverage, all-defs and all-uses coverage.

Test coverage criteria can be viewed as defining ways of splitting the input space according to test requirements, in the sense that any collection of value that satisfies the same test requirement will be equally useful [2]. Therefore, the input space is partitioned into regions that are assumed to contain equally useful inputs from a testing perspective.

We can use a syntactic description such as a grammar to model the input space, and define some criteria based on this description. For instance, we can define Java BNF grammar to describe the inputs for a Java compiler, and then generate valid (correct syntax) or invalid (incorrect syntax) programs to test the compiler. Additionally, there are coverage criteria with respect to syntactic descriptions that can be used to evaluate the test suite. For instance, considering a BNF grammar, a terminal symbol coverage evaluates the terminal symbols in the grammar that are covered by the test suite. Also, production coverage evaluates the productions in the grammar that are covered by the test suite.

2.2.4 Testing refactoring engines

A test case for a refactoring engine consists of an input program, as test input, and an expected output, which can be an output program, or an expected transformation rejection when some condition is violated.

For instance, Figure 2.4 shows two test cases created by JRRT developers to evaluate their Pull Up Method refactoring implementation. The first (`test1`) contains an input program with the classes `Super` and `A`, which extends `Super` and contains the `m` method. It also contains an expected output program contains the same classes `Super` and `A` but with the `m` method in the `Super` class. After performing this test, if the engine produces an output different from the expected one, the test will fail.

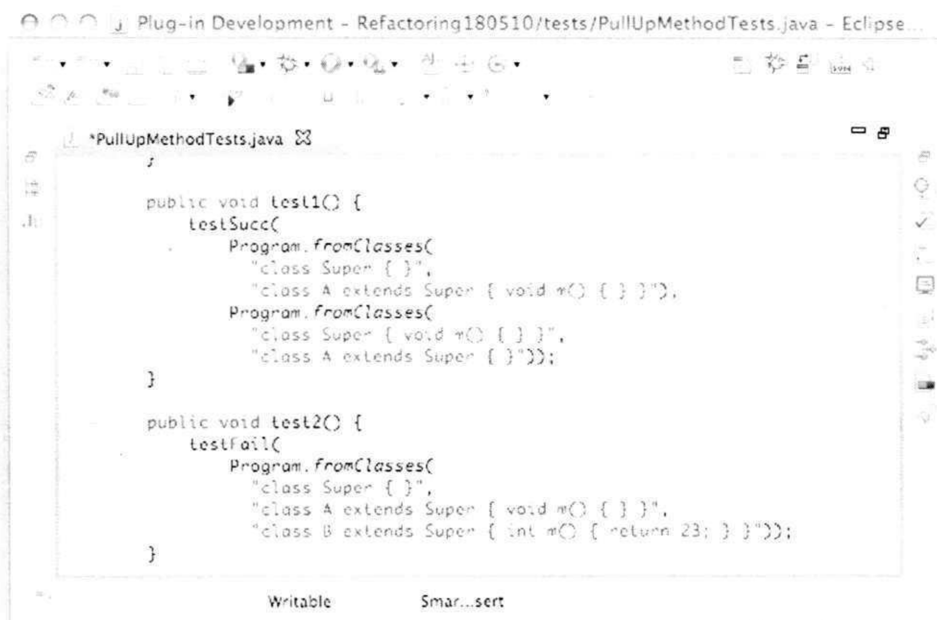


Figure 2.4: Test cases created by JRRT developers to evaluate the Pull Up Method refactoring implementation.

On the other hand, the second test case (`test2`) shows a situation where the refactoring engine should not apply the transformation. The input program has two subclasses, `A` and `B`. They contain a method `m`, but with different signatures and bodies. Therefore, the refactoring should not be applied.

Manually creating test cases for refactoring engines, besides time consuming, is diffi-

cult since developers need to create complex inputs (programs) and reason about behavioral preservation for creating expected outputs. This may lead to a test suite with a low level of production coverage, potentially leaving many hidden faults.

Daniel et al. [14] proposed an approach for automated testing of refactoring engines. They used a program generator (ASTGen) to generate programs as test inputs. ASTGen allows users to directly implement how the programs will be generated. To illustrate it, next we show how ASTGen generates Java fields. Suppose we want to generate field declarations for integers or booleans with any access modifier. To do so, ASTGen provides the `FieldDeclarationGen` generator (Listing 2.5).

Listing 2.5: Simplified version of the field generator of ASTGen.

```
1  class FieldDeclarationGen extends ASTNodeGenBase<FieldDeclaration>
     {
2    IGenerator<Modifier> modifierGen;
3    IGenerator<Type> typeGen;
4    IGenerator<Identifier> idGen;
5
6    ... (constructors and other methods)
7
8    FieldDeclaration generateCurrent() {
9      FieldDeclaration generated = new FieldDeclaration();
10     generated.setModifier(modifierGen.current());
11     generated.setType(typeGen.current());
12     generated.setIdentifier(idGen.current());
13     return generated;
14   }
15 }
```

The class `FieldDeclarationGen` extends `ASTNodeGenBase`, base class to create AST nodes. Each node is represented by using the Eclipse Core API[1]. The `FieldDeclaration` node has three child nodes: `Modifier` (access modi-

[1]Java Model Tutorial: `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_model.htm`

fier), `Type` (type declared by the field), `Identifier` (name of the field). The `FieldDeclarationGen` generator consists of three generators responsible for generating variations for these child nodes: `modifierGen`, `typeGen`, `idGen`. In each iteration, the `generateCurrent()` creates a field declaration by combining these three generators.

To initialize the `FieldDeclarationGen` generator, we need first to instantiate generators `modifierGen`, `typeGen`, `idGen`, as shown in Listing 2.6.

Listing 2.6: Instantiating generators that compose the FieldDeclarationGen generator.

```
1 IGenerator<Modifier> modifierGen = new Chain<Modifier>(public,
    private, protected, default);
2 IGenerator<Type> typeGen = new Chain<Type>(int, boolean);
3 IGenerator<Identifier> idGen = new Chain<Identifier>(x);
```

In this way, the declared field can have accessibility `public`, `private`, `protected`, or `default`. It will have the type `int` or `boolean`, and the name `x`. We pass these generators as parameters to instantiate the `FieldDeclarationGen` generator (see Listing 2.7). By using these parameters, the generator produces eight field declarations.

Listing 2.7: Instantiating the FieldDeclarationGen generator.

```
1 FieldDeclarationGen fieldDeclGen =
2   new FieldDeclarationGen(modifierGen, typeGen, idGen);
```

Besides using ASTGen, Daniel et al [14] implemented 6 test oracles to evaluate engine outputs:

- **DoesCrash.** It checks if the refactoring engine throws an uncaught exception during the test;
- **DoesNotCompile.** It checks if the program compiles after the transformation;
- **WarningStatus.** It checks if the refactoring engine throws a warning message as output. This oracle is useful when the tester intentionally creates programs that do not satisfies the refactoring conditions, and want to check if the engine correctly identifies and avoids these transformations;

- **InverseOracle.** In this oracle, they apply the refactoring under test, then they perform the inverse refactoring, that is, the opposite transformation, to the output program, and check if the resulting program is equal to the original one. For instance, to test if the engine correctly renames class A to B, they perform this transformation, then perform the inverse transformation, renaming B to A, and checks if the resulting program is equal to the original one. To compare the programs, they have implemented an AST comparator;

- **CustomOracle.** They have implemented some refactoring-specific oracles. These oracles checks properties of some refactorings. For instance, when you rename a field, the resulting program should not have the old field name anywhere in the AST;

- **DifferentialOracle.** This oracle performs the refactoring under testing by using two or more refactoring engines and compares the results. If they are different, a human inspect the two output programs to check whether the differences are related to some fault in one of the engines.

Although they have identified a number of faults in Eclipse and NetBeans that introduce compilation errors on the user's code, they have found only one fault related to behavioral change.

Additionally, writing ASTGen generators requires a considerable effort since the developers need to implement how the programs will be generated. Later, Gligoric et al. [22] proposed UDITA, which follows a *filtering approach*, that is, the generator automatically searches for all possible combinations of Java constructs to generate programs. Moreover, the tester can specify constraints to filter the program generation. The more constraints the tester specifies, the fewer programs it will generate. UDITA uses the Java Path Finder (JPF) model checker as a basis for searching for all possible combinations.

Gligoric et al. [22] previously specified a Java inheritance graph generation in UDITA. Figure 2.5 presents Java programs that illustrate different inheritance graphs that can be generated for a scope of two elements. Each inheritance graph needs to satisfy two invariants:

1. Directed Acyclic Graph (DAG). We cannot have directed cycles in Java inheritances;

2. A class has at most one supertype class, and all supertypes of an interface are interfaces.

#	Program
1	class A {} class B extends A {}
2	interface A {} class B implements A {}
3	interface A {} interface B extends A {}

Figure 2.5: Programs representing Java Inheritance Graphs.

UDITA allows users to specify the generation by using a Java language extended with non-deterministic choices. Next, we describe the inheritance graph specification presented by Gligoric et al. [22]. In Listing 2.8, we show the Java inheritance graph representation in UDITA. The class `IG` represents the graph, and contains fields that represent a list of nodes and the size of the graph. It also contains a class representing a node, which has an array of nodes as supertypes and a `boolean` flag to mark the node as a class (otherwise it is an interface). In Listings 2.9 and 2.10, we present invariants for the Java inheritance graph specified in UDITA. It returns true when these properties hold.

Listing 2.8: Java inheritance graph representation in UDITA

```
1 class IG {
2   Node[] nodes;
3   int size;
4   static class Node {
5     Node[] supertypes;
6     boolean isClass;
7   }
8 }
```

Listing 2.9: Java inheritance graph invariants in UDITA

```
1 boolean isDAG(IG ig) {
2   Set<Node> visited = new HashSet<Node>();
```

```
3   Set<Node> path = new HashSet<Node>();
4   if (ig.nodes == null || ig.size != ig.nodes.length)
5     return false;
6   for (Node n : ig.nodes)
7     if (!visited.contains(n))
8       if (!isAcyclic(n, path, visited)) return false;
9   return true;
10 }
11
12 boolean isAcyclic(Node node, Set<Node> path, Set<Node> visited){
13   if (path.contains(node)) return false;
14   path.add(node);
15   visited.add(node);
16   for (int i = 0; i < supertypes.length; i++) {
17     Node s = supertypes[i];
18     // two supertypes cannot be the same
19     for (int j = 0; j < i; j++)
20       if (s == supertypes[j]) return false;
21     // check property on every supertype of this node
22     if (!isAcyclic(s, path, visited)) return false;
23   }
24   path.remove(node);
25   return true;
26 }
```

Listing 2.10: Well-formedness rules for Java inheritance specified in UDITA

```
1 boolean isJavaInheritance(IG ig) {
2   for (Node n : ig.nodes) {
3     boolean doesExtend = false;
4     for (Node s : n.supertypes)
5       if (s.isClass) {
6       // interface must not extend any class
7       if (!n.isClass)
```

```
 8          return false;
 9        if (!doesExtend) {
10          doesExtend = true;
11        // class must not extend more than one class
12        } else {
13          return false;
14        }
15      }
16    }
17  }
```

To generate all graphs from predicates, we need to specify bounds on possible values for each elements in the graph representation, which are the array sizes, and the field `isClass`. UDITA uses non-deterministic choices based on JPF for this purpose. For example, when we run the command `k = getInt(1,N)`, JPF introduces N branches in a non-deterministic execution, where in the branch i (for $1 \leq i \leq N$) `k` has value i. JPF explores the combinations of all possible choices for primitive types. UDITA extends JPF, introducing new algorithms to explore combinations of choices for objects in a new *object pool* abstraction. Listing 2.11 presents the code to initialize the Java inheritance graph generation in UDITA. The method `initialize` performs 3 steps. First, it sets the graph size (the number of nodes). Then creates a pool of `Node` objects of this size, and finally iterates over all objects in the pool to initialize their supertypes pointing to other objects in the pool. The class `ObjectPool` has two methods: `getNew`, which returns a new object from the pool, and `getAny`, which returns an arbitrary object.

Listing 2.11: Initialization of Java inheritance graph generation in UDITA

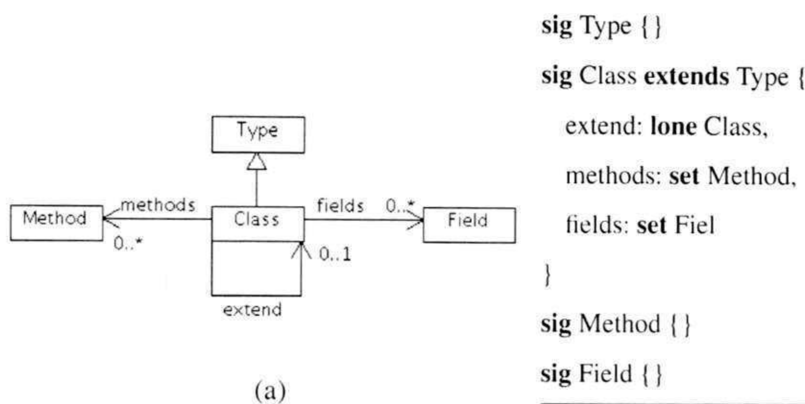
```
1 IG initialize(int N) {
2   IG ig = new IG();
3   ig.size = N;
4   ObjectPool<Node> pool = new ObjectPool<Node>(N);
5   ig.nodes = new Node[N];
6   for (int i = 0; i < N; i++) ig.nodes[i] = pool.getNew();
7     for (Node n : nodes) {
```

```
8         int num = getInt(0, N - 1);
9         n.supertypes = new Node[num];
10        for (int j = 0; j < num; j++)
11          n.supertypes[j] = pool.getAny();
12        n.isClass = getBoolean();
13      }
14    return ig;
15  }
16
17  static void mainFilt(int N) {
18    IG ig = initialize(N);
19    assume(isDAG(ig));
20    assume(isJavaInheritance(ig));
21    println(ig);
22  }
```

2.3 Alloy Overview

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures and constraints. Each *signature* denotes a set of objects associated to other objects by relations declared in the signatures. Each signature paragraph represents a type, and may declare a set of *relations* along with their types and other constraints on their included values.

We use as example part of the Java metamodel encoded in Alloy. A Java class is a type, and may extend another class. Additionally, it may declare fields and methods, as specified in the UML class diagram, as shown in Figure 2.6(a). Figure 2.6(b) presents its specification in Alloy. All classes and associations in the UML class diagram are analogous to the Alloy signatures and their relations, respectively. In `Class`, the **`set`** in relation `fields` and relation `methods` imposes no constraint on multiplicity. There are other multiplicity qualifiers, such as **`lone`**, denoting partial functions. If we omit the qualifier, the relation becomes a total function. In Alloy, one signature can extend another, establishing that the extended signature (subsignature) is a subset of the parent signature. For example, a `Class`



(a)

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

(b)

Figure 2.6: A UML class diagram and its representation in Alloy.

is a subsignature of `Type`.

A number of well-formedness constraints can be specified for Java. For instance, a class cannot extend itself. In Alloy, we can declare *facts* which package formulas that always hold. The `ClassCannotExtendItself` fact specifies this constraint.

```
1 fact ClassCannotExtendItself {
2   all c: Class | c ! in c.^extend
3 }
```

The **`all`** keyword represents the universal quantifier, and the **`in`** keyword denotes the set membership operator in the previous fragment. The operators ∧ and ! represent the transitive closure and negation operators, respectively. The dot operator (`.`) is a generalized definition of the relational join operator. For example, the expression `c.extend` yields the superclass of `c`.

In Alloy, predicates are used to package reusable formulas and specify operations. The following Alloy fragment declares the predicate `someClassHasNoField`, stating that there is a class without fields. The **`some`** keyword represents the existential quantifier. The **`no`** keyword, when applied to an expression, denotes that the expression is empty.

```
1 pred someClassHasNoField [] {
2   some c: Class | no c.field
3 }
```

UFCG/BIBLIOTECA/BC

The Alloy Analyzer tool [33] allows us to perform analysis on an Alloy specification; for example, in order to find a solution for a model in a pre-defined scope. A scope defines the maximum number of objects allowed for each signature during analysis, assigning a bound to the number of objects of each type. The simulations performed by the Alloy Analyzer tool are sound and complete, up to a given scope.

Alloy commands are used for analysis purposes. Next, we declare a **run** command that is applied to a predicate, specifying a scope for all declared signatures. For desired solutions containing as many as three of each type, class, field and method, and at least one of the classes with no fields, the Alloy Analyzer searches for all combinations that satisfy the signature and fact constraints, in addition to the `someClassHasNoField` predicate.

```
1 run someClassHasNoField for 3
```

2.4 Concluding remarks

In this chapter, we presented the theoretical basis needed for the understanding of this thesis. First, we showed an overview on program refactoring, along with the state-of-the-art approaches on refactoring verification.

Next, we introduced important concepts on software testing, such as test case, oracle, and coverage criteria. We also present the approach proposed by Daniel et al [] for testing of refactoring engines, and their program generator, ASTGen. We also presented UDITA an extension of ASTGen. In Chapter 3 we present a comparison between our program generator, JDOLLY, and UDITA. Finally, we presented an overview of Alloy and Alloy Analyzer, which we used to propose our program generator, JDOLLY.

Chapter 3

JDOLLY: A Java program generator

In this chapter, we present JDOLLY[1], a Java program generator that exhaustively generates programs, up to a given scope of Java constructs (e.g. packages, classes, methods, fields). The Alloy specification language (Section 2.3) is employed as the formal infrastructure for generating programs; a metamodel for Java is encoded in Alloy, and the Alloy Analyzer finds instances of this model, which are translated into programs by JDOLLY, for user-specified constraints.

Next we present an overview of the technique (Section 3.1). Then we show the encoding of a subset of the Java metamodel in Alloy. We then describe how to translate each Alloy solution to Java (Section 3.3), and explain how to use JDOLLY for generating more specific Java programs in Section 3.4. In Section 3.5, we describe an experiment to compare JDOLLY with another Java program generator, UDITA [22]. Finally, we present the concluding remarks (Section 3.6).

3.1 Overview

JDOLLY is a Java program generator. It contains a subset of the Java metamodel specified in Alloy [32]. It employs the Alloy Analyzer, a tool for analysis of Alloy models, to generate solutions (instances) for this metamodel. It then translates each solution into a Java program.

JDOLLY exhaustively generates all Java programs specified by its metamodel for a given scope. The user defines this scope by specifying the maximum number of elements for each

[1] It can be downloaded from: http://www.dsc.ufcg.edu.br/~spg/jdolly

Java construct presented in the metamodel. For instance, the user can specify the maximum number of classes to three. By doing so, JDOLLY will generate all programs with up to three classes. Furthermore, the user can specify specific constraints for the program generation. For example, when testing a refactoring that pulls up a method to a superclass, the input programs must contain at least a subclass declaring a method that is subject to be pulled up. The user can specify these constraints in Alloy.

3.2 Java metamodel

In this section, we describe the subset of the Java metamodel that we specified. If we consider the entire Java language, we can create a large number of different programs even for a small scope of elements, which may make it too expensive to exhaustive generate programs even for a small scope. Additionally, some Java constructs and well-formedness rules may require considerable effort to be specified in Alloy due to restrictions of the language. For instance, Alloy does not allow recursive predicates. Our goal is to specify a subset of the Java language that can be useful for finding real faults in refactoring engines. To do so, we studied faults previously identified by researchers [84; 72; 14] in order to understand which constructs are relevant to this context.

3.2.1 Abstract syntax

We illustrate a UML class diagram representing the subset of the Java metamodel encoded in Alloy in Figure 3.1. From Java, we have considered two primitive types: *int* and *long*. By using these primitive types, we can evaluate the refactoring engines in the presence of method overloading and implicit cast. We believe that if we have included other primitive types, such as *float*, it would not make much difference with respect to method overloading and implicit casting, but it would increase the number of programs, making it more expensive to generate all programs. A class is the only non-primitive type – currently, we do not consider interfaces. A Java class has an identifier, field and method declarations, and extends another class. Moreover, each class is located in a package. If a class is not explicitly related to a package, the default package is assumed.

Each field is associated with one identifier, one type, and at most one modifier, such as

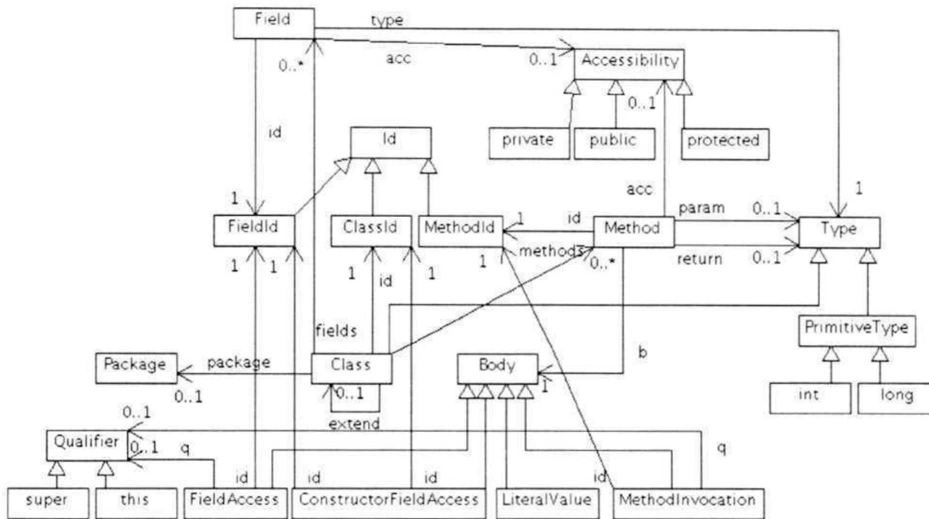


Figure 3.1: The Java metamodel specified in JDOLLY.

`public`, `protected`, or `private`. When it does not have a modifier, its accessibility is `package`. Similarly, a method declaration contains a return type, an identifier, a number of parameters, and a body. Moreover, it may contain an access modifier. We have considered methods with at most one parameter, which is useful for generating programs containing overloading. For instance, a method can have no parameter and another method with the same name can have one parameter, or both methods can have one parameter but with different types. Moreover, by generating methods with parameter we can generate programs to test refactorings that operate over parameters, such as the Remove Parameter refactoring. Although adding more parameters can be useful for finding more faults, it also would significantly increase the number of combinations for generating programs.

In Java, a method body contains a sequence of statements, whose last statement must be a *return* for every non-void method. Currently, a method body contains just a single return statement. So, the simplest return statement returns a literal value based on the return type. Return statements can also contain field accesses or method invocations. Field accesses include: `f`, `A.f`, `this.f`, `super.f` and `new A().f` – the latter is a `ConstructorFieldAccess`. `LiteralValue` represents the simplest kind of statement, extending the signature `Body`. `FieldAccess` and `MethodInvocation` contain the identifier of the accessed field and method with a single qualifier at most, respectively. If

a method with a single parameter is called JDOLLY always passes a constant value, such as 2, as argument to the call.

3.2.2 Well-formedness rules

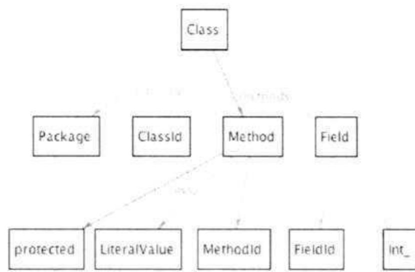
The Java language contains a number of well-formedness rules to evaluate whether a program is valid. We specified these rules within Alloy facts. For example a Java class cannot have two fields with the same identifier, as declared in the fact `noClassTwoFieldsSameId`.

```
1 fact noClassTwoFieldsSameId {
2   all c: Class | all f1,f2: c.fields |
3     f1 ! = f2 ⇒ f1.id ! = f2.id
4 }
```

Similarly, a Java class cannot contain two methods with the same name and parameter type, as presented in the fact `noClassTwoMethodsSameSignature`.

```
1 fact noClassTwoMethodsSameSignature {
2   all c: Class | all m1,m2: c.methods |
3     m1 ! = m2 ⇒
4       (m1.id ! = m2.id or m1.paramType ! = m2.paramType)
5 }
```

Some well-formedness may require a lot of effort to specify in Alloy. For example, we cannot have a method invocation to an undefined method. To analyze the binding between a method invocation and a method declaration, we may need to evaluate if the method declaration is in the same class, hierarchy, and package of the method invocation, its access modifier (public, protected, package, private), its parameters, and the kind of the method invocation (e.g. using super, this, qualified this). We could try to specify these rules exactly how they are, avoiding uncompilable programs, or specifying approximations that may result in uncompilable programs. Although the first option guarantees that all generated programs will compile, it requires more effort, and may lead to over constraining the model, leading the tool to miss some compilable programs. On the other hand, the second option requires less effort but produces uncompilable programs. We chose the second option because we can discard



(a)

```
1 package Package;
2 public ClassId {
3   int fieldId = 1;
4   protected int
      methodId() {
5     return 2;
6   }
7 }
```

(b)

Figure 3.2: Translation of an Alloy solution to a Java program. (a) A solution of the Java metamodel generated by Alloy Analyzer; (b) the translation of the solution into a concrete Java program.

the uncompilable input programs while testing a refactoring engine. Appendix A presents the complete specification of the abstract syntax and well-formedness rules for JDOLLY.

3.3 Program generation

The previous Alloy model is then used to generate Java programs. We specify the `run` command; specifically with the `generate` predicate. By default, the scope of at most three objects is used for each signature. Then we use the Alloy Analyzer API to execute the `run` command, generating all solutions for the given scope.

```
1 pred generate[] {}
2 run generate for 3
```

The Alloy Analyzer finds for solutions such as the instance depicted in Figure 3.2(a). The graph contains the `Class` object, which is associated with objects `Package`, `ClassId`, `Method`, and `Field`. Moreover, object `Field` is associated with `FieldId` and `Int_`, and `Method` is associated with `LiteralValue`, `MethodId`, `Protected`, and `Int_`. For simplicity, we distinguish class from field identifiers. For example, Figure 3.2(b) shows the counterpart in Java of the Alloy solution.

The Alloy Analyzer does not automatically convert an Alloy instance into a Java program. In fact, we use its API to generate *every* possible solution[2]. To complete the generation step, we reused the syntax tree available in Eclipse JDT [17] for generating programs from those solutions. For example, the Alloy objects `Class` and `Package` are mapped to a `TypeDeclaration` and a `PackageDeclaration`, respectively. The imports are automatically calculated from each Alloy instance generated; they are included in each program.

3.4 Generating more specific programs

With JDOLLY, we can specify different scopes to limit program generation. For instance, if we are not interested in fields, we can specify the scope of zero. Besides, the generation can be further constrained. In a context in which programs are needed with at least one class (`C2`) extending another one (`C1`), and `C2` declares at least a method (`M1`), the following Alloy fragment specifies `generate`. This particular specification is useful for testing the Pull Up Method refactoring, considering `M1`. For each instance, we pass the value given to `M1` to the refactoring engine.

```
1 one sig C1, C2 extends Class {}
2 one sig M1 extends Method {}
3 pred generate[] {
4   C1 in C2.extend
5   M1 in C2.methods
6 }
```

3.5 Evaluation

In this section, we present an experiment comparing JDOLLY against UDITA [22].

3.5.1 Definition

In previous work, Gligoric et al. [22] uses an Java inheritance graph generation to show that UDITA is more expressive and easier to use than ASTGen. In Section 2.2.4 we present an

[2]Accessing Alloy 4 using Java API: http://alloy.mit.edu/alloy4/api.html

UFCG/BIBLIOTECA/BC

UDITA specification to generate Java inheritance graphs.

We carried out a similar comparison on the differences between JDOLLY and UDITA. The goal of this experiment is to analyze two tools (JDOLLY and UDITA) for the purpose of evaluation with respect to test input generation from the point of view of researchers in the context of Java inheritance graph generation. For instance, In particular, our experiment addresses the following research questions:

- **Q1**. Do the tools exhaustively generate inheritance graphs for a given scope?

  Since we do not know all inheritance graphs that can be generated, we compare all graphs generated by JDOLLY against the ones generated by UDITA in order to detect missing graphs in each one of the tools' results.

- **Q2**. Do the tools generate isomorphic inheritance graphs?

  A tool may generate more than one structurally equivalent (isomorphic) solution. In the context of test input generation, generating isomorphic inputs does not increase the chances of finding new faults, and makes the test input generation slower. Therefore, the less isomorphic graphs generated by each approach, the better. We measure the number of isomorphic and non-isomorphic graphs for each tool.

3.5.2 Planning

Next, we describe how we selected the subjects and how we instrument the experiment.

Selection of subjects

To compare JDOLLY against UDITA, we chose to generate a Java inheritance graph by using both tools. We chose to use a Java inheritance graph because it has non-trivial invariants and it is directly related to generating Java programs. Additionally, it was previously used to describe UDITA and compare it with ASTGen [22]. Each inheritance graph needs to satisfy two invariants:

1. Directed Acyclic Graph (DAG). We cannot have directed cycles in Java inheritances;

2. A class has at most one supertype class, and all supertypes of an interface are interfaces.

Experiment Design

For each approach, we perform the Java inheritance graph generation by using scopes from 1 to 4. This scope is similar to the scope of previous programs that revealed faults in refactoring engines [84; 72; 14].

Instrumentation

To perform the UDITA generation, we downloaded the Java inheritance graph specification from UDITA website[3]. In Section 2.2.4, we present a simplified version of this specification.

We created a JDOLLY version containing the metamodel of the Java graph inheritance. Next, we describe this metamodel. First, we specified the signatures `IG` and `Node` to represent the inheritance graph as shown in Listing A.1.

Listing 3.1: Java inheritance graph representation in JDolly

```
1 sig IG {
2   nodes: set Node
3 }
4 abstract sig Node{
5   supertypes : set Node,
6   isClass : one Bool
7 }
```

Then, we specified Alloy facts that represent the invariants of the Java inheritance graph as shown Listing 3.5.

Listing 3.2: invariants for the Java inheritance graph in JDolly

```
1 fact DAG {
2   no n:Node | n in n.^supertypes
3 }
4 fact JavaInheritance {
5   all n:Node | isTrue[n·isClass] ⇒
6     lone n1:Node | n1 in n·supertypes and isTrue[n1·isClass]
7   all n:Node | isFalse[n·isClass] ⇒
```

[3]http://mir.cs.illinois.edu/udita/

```
8       no n1:Node | n1 in n·supertypes and isTrue[n1·isClass]
9   }
```

Finally, we initialize the generation by running the `Run` command on the `Show` predicate as illustrated in Listing 3.6. We specified a constraint in the `Show` predicate to specify that all generated nodes must be in the inheritance graph.

Listing 3.3: Running Java graph generation by using the Alloy Analyzer.

```
1 pred show[] {
2   Node in IG·nodes
3 }
4 run show for exactly 1 IG, exactly 4 Node
```

We implemented a graph comparator to compare the graphs generated by both tools. The comparator abstracts the name of the nodes, so that if two graphs have the same structure but different names, the comparator says that they are isomorphic.

To check whether the tools exhaustively generates solutions for a given scope, we check if each graph generated by JDOLLY was also generated by UDITA, and the other way around, by using our graph comparator. To check if the tools generate isomorphic graphs, we use our graph comparator to compare each graph generated by the tool against all the other graphs generated by it.

In JDOLLY, we specify the Java inheritance graph generation by using Alloy. First, we specified the signatures `IG` and `Node` to represent the inheritance graph as shown in Listing A.1.

Listing 3.4: Java inheritance graph representation in JDolly

```
1 sig IG {
2   nodes: set Node
3 }
4 abstract sig Node{
5   supertypes : set Node,
6   isClass : one Bool
7 }
```

Then, we specified Alloy facts that represent the invariants of the Java inheritance graph as shown Listing 3.5.

Listing 3.5: Java inheritance graph representation in JDolly

```
1 fact DAG {
2   no n:Node | n in n.^supertypes
3 }
4 fact JavaInheritance {
5   all n:Node | isTrue[n·isClass] ⇒
6     lone n1:Node | n1 in n·supertypes && isTrue[n1·isClass]
7   all n:Node | isFalse[n·isClass] ⇒
8     no n1:Node | n1 in n·supertypes && isTrue[n1·isClass]
9 }
```

Finally, we initialize the generation by running the `Run` command on the `Show` predicate as illustrated in Listing 3.6. We specified a constraint in the `Show` predicate to specify that all generated nodes must be in the inheritance graph.

Listing 3.6: Running Java graph generation by using the Alloy Analyzer.

```
1 pred show[] {
2   Node in IG·nodes
3 }
4 run show for exactly 1 IG, exactly 4 Node
```

3.5.3 Operation

We performed the Java inheritance graph generation on a MacBook Pro Intel Core i5 2.4GHz with 8GB of RAM. Table 3.1 summarizes the results of the experiment. In contrast with JDOLLY, UDITA did not generate 2, 7 and 37 non-isomorphic programs in scopes 2, 3 and 4, respectively. For example, Figure 3.3 shows the programs that represent the Java inheritance graphs generated by JDOLLY and UDITA for a scope of two elements. UDITA did not generate the program 5, which contains two classes, and program 6, which has two classes, one extending the other one. On the other hand, JDOLLY generated much more isomorphic programs than UDITA.

Table 3.1: Comparison of JDOLLY and UDITA; Prog.: Number of generated programs; Comp.: number of compilable programs; Isomor: number of isomorphic programs; Unique: number of unique programs; NG: number of unique programs that were not generated.

	JDolly				UDITA			
Scope	Prog.	Isomor.	Unique	NG	Prog.	Isomor.	Unique	NG
1	2	0	2	0	2	0	2	0
2	6	0	6	0	4	0	4	2
3	29	5	24	0	18	1	17	7
4	230	81	149	0	123	11	112	37

3.5.4 Discussion

One of the reasons why UDITA did not generate all programs may be an incorrect specification of the constraints for the Java inheritance. By looking at the code that we downloaded from UDITA website, we noticed slightly differences with respect to the simplified code presented in Listings 2.8, 2.9, 2.10, and 2.11. For instance, in the `isJavaInheritance` predicate (Listing 2.10), before checking if the node is a class (line 5), there is another statement `if (isClass);`. This statement would have no effect in Java because there is no command to be executed in this `if` statement. However, when we remove this statement, and run UDITA again, it generates all six programs for the scope of 2; in fact, it generates seven programs (one isomorphic program). We also evaluated to replace this `if` statement to `System.out.println(isClass)`. When we added this statement to print this variable, UDITA generated only four programs (the same ones that it generated in the original version), missing two graphs. This may be a fault in the current implementation of UDITA.

In our experiment, both tools generated isomorphic inheritance graphs. JDOLLY uses the Alloy Analyzer for generating programs, which uses SAT solvers for searching solutions for the Alloy models. These solvers contain algorithms for avoiding generating several isomorphic solutions. UDITA also implements an algorithm for this purpose. On the other hand, in ASTGen, the tester would be in charge of this task.

Figure 3.4 shows two programs representing isomorphic graphs generated by JDOLLY for a scope of three elements. These programs have the same structure but different identifiers. Although JDOLLY generated other four isomorphic programs for this scope, it avoided a number of other isomorphic programs. Notice that it generated 24 distinct programs (see Table 3.1). Each one of these programs has tree elements. By permuting the identifiers of

#	JDolly	UDITA
1	interface A {} class B {}	interface A {} class B {}
2	interface A {} class B implements A {}	interface A {} class B implements A {}
3	interface A {} interface B {}	interface A {} interface B {}
4	interface A {} interface B extends A{}	interface A {} interface B extends A{}
5	class A {} class B{}	
6	class A { class B extends A	

Figure 3.3: Programs representing the generation of Java Inheritance Graphs by UDITA and JDOLLY for the scope of two elements.

```
1 interface A {}
2 interface B extends A {}
3 interface C extends B {}
```

```
1 interface B {}
2 interface C extends B {}
3 interface A extends C {}
```

Figure 3.4: Isomorphic programs generated by JDOLLY.

these elements, we can have 6 programs with the same structure. Considering all 24 programs, JDOLLY could have generated 144 programs (120 isomorphic ones). It is important to avoid isomorphic programs because they do not increase the chances of finding faults in refactoring engines and slow the program generation.

Alloy logic presented, as expected, a higher level of abstraction than Java-like code of UDITA. For example, while we specified the DAG invariant in one line by using Alloy, Gligoric et al. [22] needed about 20 lines to specify it in UDITA.

3.5.5 Answers to the research questions

Next, we discuss these results with respect to our research questions.

Do the tools exhaustively generate inheritance graphs for a given scope?

No. JDOLLY generated all inheritance graphs, but UDITA failed to generate some graphs for the scope of two, three, and four. In bounded-exhaustive testing, failing to generate some test input may lead to leave some fault uncaught, reducing the effectiveness of the approach.

Do the tools generate isomorphic inheritance graphs?

Yes. In our experiment, both tools generated isomorphic inheritance graphs. JDOLLY, though, generated more than UDITA. For instance, with a scope of four, while 35% of the graphs generated by JDOLLY were isomorphic, in UDITA, only 9% of the graphs were isomorphic. Our results suggest that UDITA handles isomorphism better than JDOLLY.

3.5.6 Threats to validity

With respect to construct validity, we compare the results of both tools to evaluate whether they exhaustively generates inheritance graphs. Therefore, if none of the tools exhaustively generates these graphs, our results will be incorrect. Finally, we compare both tools with respect to Java inheritance graphs. Our results are not representative of all program generation allowed on both tools.

3.6 Concluding remarks

In this chapter, we presented JDOLLY, a Java program generator that uses Alloy and the Alloy Analyzer as basis for generating programs. It allows users to exhaustively generate Java programs by specifying the scope of the program generation and constraints on what programs should be generated. Our goal was to define a subset of the language expressive enough for finding faults in refactoring engines, but not too large to make it too complex and expensive. We studied previously faults in refactoring engines found in literature. We used this knowledge to specify a Java metamodel that includes relevant constructs to test refactoring engines.

We compared JDOLLY against a state-of-the-art program generator, UDITA. Our results suggest that while JDOLLY exhaustively generates programs, UDITA may fail to generate

some programs in a given scope. On the other hand, JDOLLY generates more isomorphic programs than UDITA, which may slow down the program generation. In our experiment, though, JDOLLY was faster than UDITA.

Chapter 4

SAFEREFACTOR

In this chapter, we present SAFEREFACTOR [80], a tool for checking behavioral changes in program transformations. In Section 4.1, we show its overview. Next, we show an empirical study to evaluate the effectiveness of SAFEREFACTOR 4.2. Finally, we show the concluding remarks (Section 4.3).

4.1 Overview

SAFEREFACTOR [80] checks whether a transformation introduce behavioral changes. First, the tool checks for compilation errors in the resulting program, and reports those errors; if no errors are found, it analyzes the transformation and generates a number of tests suited for detecting behavioral changes. SAFEREFACTOR identifies the methods with matching signature (methods with exactly the same modifier, return type, qualified name, parameter types and exceptions thrown) before and after the transformation. Next, it applies Randoop [56], a Java unit test generator, to produce a test suite for those methods. Randoop randomly generates tests for a set of methods given a time limit. Finally, it runs the tests before and after the transformation, and evaluates the results. If results are different, the tool reports a behavioral change, and displays the set of unsuccessful tests. Figure 4.1 illustrates this process.

To illustrate SAFEREFACTOR, take class `A` and its subclass `B` as illustrated in Listing 4.1. `A` declares the `k` method, and `B` declares methods `k`, `m`, and `target`. 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

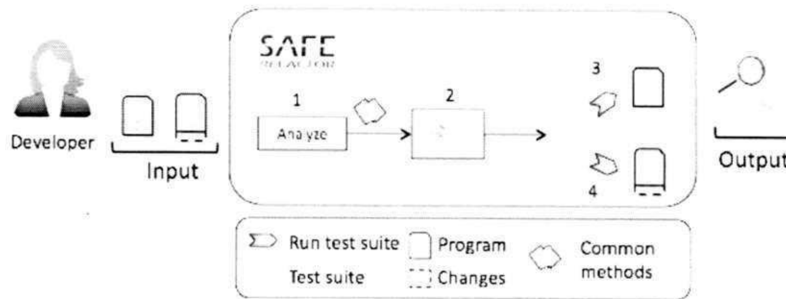


Figure 4.1: Safe Refactor's technique; 1) The tool identifies the methods with same signature before and after the transformation; 2) It generates a test suite for the identified methods using Randoop; 3) It runs the tests on the source program; 4) It runs the tests on the target program; 5) Finally, Safe Refactor evaluates the results: if they are different, the tool reports a behavioral change. Otherwise, the developer can increase confidence that the programs have the same behavior.

3.7 or JRRTv1 to perform this refactoring will produce the program presented in Listing 4.2[1]. 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: `target` yields 2 instead of 1. Since `m` is invoked on an instance of `B`, the call to `k` using `this` is dispatched on to the implementation of `k` in `B`.

Assuming the programs in Listings 4.1 and 4.2 as input, SAFEREFACTOR first identifies the methods with matching signatures on both versions: `A.k`, `B.k`, and `B.target`. Next, it generates 78 unit tests for these methods within a time limit of two seconds. Finally, it runs the test suite on both versions and evaluates the results. A number of tests (64) passed in the source program, but did not pass in the refactored program; so SAFEREFACTOR reports a behavioral change. Next, we show one of the generated tests that reveal behavioral changes. The test passes in the source program since the value returned by `B.target` is 1; however, it fails in the target program since the value returned by `B.target` is 2.

[1]The same problem happens when we omit the keyword **this**

Listing 4.1: Before Refactoring

```
1  public class A  {
2    int k() {
3      return 1;
4    }
5  }
6  public class B extends A {
7    int k() {
8      return 2;
9    }
10   int m() {
11     return super.k();
12   }
13   public int target() {
14     return m();
15   }
16 }
```

Listing 4.2: After Refactoring.Applying Pull Up Method in Eclipse JDT 3.7 or JRRTv1 leads to a behavioral change due to incorrect change of **super** to **this**.

```
1  public class A  {
2    int k() {
3      return 1;
4    }
5    int m() {
6      return this.k();
7    }
8  }
9  public class B extends A {
10   int k() {
11     return 2;
12   }
13   public int target() {
14     return m();
15   }
16 }
```

```
1 public void test() {
2   B b = new B();
3   int x = b.target();
4   assertTrue(x == 1);
5 }
```

4.2 Evaluation

Previously, we evaluated SAFEREFACTOR in 8 transformations applied to real Java programs [79]. Although these transformations were classified as refactorings by the developers that performed them, SAFEREFACTOR found that one of them changed program's behavior. These results suggest that SAFEREFACTOR can find behavioral changes in real software, but do not give evidences on how effective the tool is.

In this section, we evaluate[2] SAFEREFACTOR in 60 transformations gathered from source code repositories. We previously did not know whether these transformations are behavior preserving. To evaluate the correctness of SAFEREFACTOR's results, we compare it against other two approaches for identifying refactorings: a manual inspection proposed by Murphy-Hill et al. [49; 50]; and an approach based on commit-message analysis [61; 60].

The remaining of this section is organized as follows: the following subsection describes the approaches compared with SAFEREFACTOR(Section 4.2.1). Then, we present the experiment definition (Section 4.2.2), and show the experiment planning (Section 4.2.3). Next, we describe the experiment operation, and show the results (Section 4.2.4). Then, we interpret and discuss them in Section 4.2.5. Finally, we describe some threats to validity (Section 4.2.6).

4.2.1 Compared techniques

Manual Analyses Overview

The manual analysis is based on the methodology of Murphy-Hill et al. [49; 50], which compares the code before each commit against its counterpart after the commit. For brevity, we will simply call this approach 'Murphy-Hill'. For each commit, two evaluators sit together and use the standard Eclipse diff tool to compare files before the commit to the files after the commit. Reading through each file, the evaluators attempt to logically group fine-grained code changes together, classifying each change as either a refactoring (such as "Extract Method") or a non-refactoring (such as "Add null Check"). The evaluators also attempt to group together logical changes across files by re-comparing files as necessary. For exam-

[2]All experimental data are available at: http://www.dsc.ufcg.edu.br/~gsoares/thesis-experiments.html

UFCG/BIBLIOTECA/BC

ple, if the evaluators noticed that a change to one file deleted a piece of code, they would have initially classified that change as a non-refactoring, but if later the evaluators found that the code had actually been moved to another file, the evaluators would re-classify the two changes together as a single refactoring. If the two evaluators did not agree on whether a change was a refactoring, to reach agreement they would discuss under what circumstances it might possibly change the behavior of the program.

By assessing the transformations performed during a commit, this approach is able to determine whether a commit contained only refactorings, no refactorings, or a mix of refactorings and non-refactorings.[3]

Commit Message Analyses Overview

Ratzinger et al. [60; 61] proposed an approach to detect whether a transformation is a refactoring by analyzing a commit message. If the message contains a number of words that are related to refactoring activities, the transformation is considered a refactoring. We implemented their approach in Algorithm 1.

The implemented analyzer is based on Ratzinger et. al.'s algorithm [60; 61], which we will simply call 'Ratzinger'.

4.2.2 Definition

The goal of this experiment is to analyze three approaches (SAFEREFACTOR, Ratzinger, and Murphy-Hill) for the purpose of evaluation with respect to identifying bevahior-preserving transformations from the point of view of researchers in the context of open-source Java project repositories. In particular, our experiment addresses the following research questions:

- **Q1**. Do the approaches identify all behavior-preserving transformations?

  For each approach, we measure the true positive rate (also called *recall*). $tPos$ (true positive) and $fPos$ (false positive) represent the correctly and incorrectly behavior-

[3]One difference between the present study and the previous study [49] was that in the previous study they included a "pure whitespace" category; in the present study, we consider "pure whitespace", "Java comments changes", and "non-Java files changes" to be a refactoring, to maintain consistency with the definition of refactoring used by SAFEREFACTOR.

Algorithm 1 Ratzinger

Require: *message* $\Leftarrow$ commit message

Ensure: Indicates whether a transformation is a refactoring

```
keywords ⇐ {refactor, restruct, clean, not used, unused, reformat, import, remove, re-
moved, replace, split, reorg, rename, move}
if 'needs refactoring' ∈ message then
  return FALSE
end if
for k ∈ keywords do
  if k ∈ message then
    return TRUE
  end if
end for
return FALSE
```

preserving transformations, respectively. $tNeg$ (true negative) and $fNeg$ (false negative) represent correctly and incorrectly identified non-behavior-preserving transformations, respectively. Recall is defined as follows [52]:

$$recall = \frac{\#tPos}{\#tPos + \#fNeg} \quad (4.1)$$

- **Q2**. Do the approaches correctly identify behavior-preserving transformations?

  For each approach, we measure the false positive rate (*precision*). It is defined as follows [52]:

$$precision = \frac{\#tPos}{\#tPos + \#fPos} \quad (4.2)$$

- **Q3**. Are the overall results of the approaches correct?

  We measure the *accuracy* of each approach by dividing the total correctly identified behavior-preserving and non-behavior-preserving transformations by the total number of samples. It is defined as follows [52]:

$$accuracy = \frac{\#tPos + \#tNeg}{\#tPos + \#fPos + \#tNeg + \#fNeg} \quad (4.3)$$

4.2.3 Planning

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

Selection of subjects

We analyze two Java open-source projects. JHotDraw is a framework for development of graphical editors. Its SVN repository contains 650 versions. The second SVN repository is from the Apache Common Collections (we will simply call 'Collections'), which is an API build upon the JDK Collections Framework to provide new interfaces, implementations and utilities.

We randomly select 40 out of 650 versions from the JHotDraw repository (four developers were responsible for these changes) and 20 out of 466 versions from the Collections repository (six developers were responsible for these changes). For each randomly selected version, we take its previous version to analyze whether they have the same behavior. For instance, we evaluate Version 134 of JHotDraw and the previous one (133).

Tables 5.8 and 4.2 indicate the version analyzed, number of lines of code of the selected version and its previous version, and characterize the scope and granularity of the transformation. We evaluate transformations with different granularities (low and high level) and scope (local and global).

Experiment design

In our experiment, we evaluate one factor (approaches for detecting behavior-preserving transformations) with three treatments (SAFEREFACTOR, Murphy-Hill, Ratzinger). We choose a paired comparison design for the experiment, that is, the subjects are applied to all treatments. Therefore, we perform the approaches under evaluation in the 60 pairs of versions. The results can be "Yes" (behavior-preserving transformation) and "No" (non-behavior-preserving transformation).

Instrumentation

We invited Murphy-Hill and one of his collaborators to perform his approach. We automate the experiment for checking SAFEREFACTOR and Ratzinger results[4]. The Ratzinger approach was implemented in Algorithm 1.

We use SAFEREFACTOR 1.1.4 with default configuration but using a time limit of 120 seconds, and setting Randoop to avoid generating non-deterministic test cases. We chose the time limit based on previous experiences of Randoop in real subjects [79; 56; 66]. Additionally, SAFEREFACTOR may have different results each time it is executed due to the randomly generation of the test suite. So, we execute it up to three times in each version. If none of the executions finds a behavioral change, we classify the version as behavior-preserving transformation. Otherwise, we classify it as non-behavior-preserving transformation. We use Emma 2.0.5312[5] to collect the statement coverage of the test suite generated by SAFEREFACTOR in the resulting program. Additionally, we collect additional metrics for the subjects: non-blank, non-comment lines of code, scope, and granularity. The algorithms to collect refactoring scope and granularity are presented in B.

Since we previously do not know which versions contain behavior-preserving transformations, we the results of all approaches in all transformations to derive a *Baseline*. For instance, if the Murphy-Hill approach yielded "Yes" and SAFEREFACTOR returned "No", the first author would checked whether the test case showing the behavioral change reported by SAFEREFACTOR was correct. If so, the correct result was "No". So, we establish a *Baseline* to check the results of each approach, and calculate their recall, precision, and accuracy.

4.2.4 Operation

Before performing the experiment, we implemented a script to download 60 pairs of versions and log commit information: version_id, date, author, and commit message. We named each pair of versions with suffix _BEFORE and _AFTER to indicate the program before and after the change. The versions that were non-Eclipse projects were made Eclipse projects so that the Murphy-Hill approach could use the Eclipse diff tool. Murphy-Hill and his collaborators

[4]The automated experiment containing SAFEREFACTOR and Ratzinger approaches, and additional information are available at: http://www.dsc.ufcg.edu.br/~spg/jss_experiments.html

[5]http://emma.sourceforge.net/

scheduled two meetings to analyze the subjects following the Murphy-Hill approach. The automated analyses of SAFEREFACTOR and Ratzinger were performed on a MacBook Pro Core i5 2.4GHz and 4 GB RAM, running Mac OS 10.7.4.

Additionally, for SAFEREFACTOR we also downloaded all dependencies of JHotDraw. SAFEREFACTOR compiles each version and than generates tests to detect behavioral changes. We also manually create buildFiles to compile the JHotDraw subjects. As software evolves, it may modify the original build file due to changes in the project structure, compiler version or used libraries. For JHotDraw's subjects, we needed 4 buildFiles, and used JDK 1.5 and 1.6. We do not have information which JDK they used. For each subject, we used SAFEREFACTOR with a specific buildFile. The Apache Common Collections subjects were compiled with JDK 1.6. Moreover, we performed the test generation of Randoop, and the test execution using JDK 1.6 on both samples.

Tables 5.8 and 4.2 present the results of our evaluation for JHotDraw and Collections, respectively. Column *Version* indicates the version analyzed, and Column *Baseline* shows whether the pair is indeed a refactoring. This column was derived based on all results, as explained in Section 4.2.3. The following columns represent the results of each approach. In the bottom of the table, it is shown the precision, recall, and accuracy of each approach with respect to Column *Baseline*.

We have identified 14 and 11 refactorings (Baseline) in JHotDraw and Collections, respectively. In 17 out of 60 pairs, all approaches have the same result. While some versions fixed bugs, such as Versions 134, 176, and 518, or introduced new features, for instance Version 572952, others are refactorings (see Baseline of Tables 5.8 and 4.2). Some versions did not change any Java file (Versions 251, 274, 275, 300, 304, 405, 697, 609497, 923339, 1095934) or changed just Java comments (Versions 156, 814123, 814128, 966327, 1023771, 1023897, 1299210, 1300075). In this study, we regard them as refactorings (behavior-preserving transformations).

The Murphy-Hill approach detected all refactorings of JHotDraw and Collections, which means a recall of 1 on both samples. However, it classifies four uncompilable versions as refactoring: one in JHotDraw (Version 357) and three in Collections (Versions 814997, 815022, 815042). This is the main reason why the manual inspection performed by the Murphy-Hill approach is not considered as the Baseline alone. So, 14 out of the 15 detected

Version	LOC		Granu.	Scope	Baseline	Ratzinger	MH	SafeRefactor		
	Before	After			Refact.	Refact.	Refact.	Refact.	# Tests	Cov. (%)
134	20422	20422	Low	Local	No	No	No	Yes	449	24
151	28103	28108	Low	Local	No	No	No	No	4778	48
156	28121	28121	Low	Local	Yes	Yes	Yes	Yes	4778	48
173	28101	28052	Low	Global	No	Yes	No	Yes	454	20
174	28052	28053	Low	Global	Yes	No	Yes	Yes	599	23
176	28055	28055	Low	Local	No	No	No	No	436	33
179	28065	28065	Low	Local	Yes	No	Yes	Yes	3199	41
193	28291	28298	Low	Global	Yes	No	Yes	Yes	2108	39
251	28398	28398	Low	Local	Yes	No	Yes	Yes	5162	49
267	28398	28409	Low	Local	No	No	No	Yes	5433	48
274	32408	32408	Low	Local	Yes	No	Yes	Yes	418	4
275	32408	32408	Low	Local	Yes	No	Yes	Yes	476	3
294	39249	39081	High	Local	No	No	No	No	Compilation Error	
300	39161	39161	Low	Local	Yes	No	Yes	Yes	314	14
302	38993	39161	High	Local	No	No	No	No	Compilation Error	
304	39161	39161	Low	Local	Yes	No	Yes	Yes	286	15
318	39160	39173	Low	Local	No	No	No	Yes	2356	8
322	39377	39480	High	Local	No	No	No	Yes	802	26
324	39472	39551	High	Global	No	No	No	No	490	10
344	51339	51596	High	Global	No	No	No	Yes	1022	13
357	52991	52636	Low	Global	No	No	Yes	No	Compilation Error	
384	52594	52601	Low	Local	No	No	No	Yes	2167	24
405	53708	53708	Low	Local	Yes	No	Yes	Yes	1816	10
409	53712	53721	High	Global	No	No	No	Yes	1687	10
458	64939	64940	Low	Local	No	No	No	No	1549	12
501	69300	69404	High	Global	Yes	No	Yes	Yes	2600	29
503	69570	69566	High	Global	Yes	No	Yes	No	2084	21
518	71578	71979	High	Global	No	No	No	No	1114	9
526	72027	72053	High	Global	No	No	No	No	1942	7
549	72245	72286	Low	Global	No	No	No	No	1940	12
590	74235	71943	High	Local	Yes	No	Yes	Yes	255	7
596	72402	72553	High	Global	No	No	No	No	823	25
609	72752	72754	High	Global	No	No	No	Yes	2417	31
649	75664	75664	Low	Local	No	No	No	Yes	1752	27
650	75664	76220	High	Global	No	No	No	Yes	1755	27
660	76469	79135	High	Global	No	No	No	No	966	27
697	79708	79708	Low	Local	Yes	No	Yes	Yes	1418	21
700	79731	79741	Low	Global	No	No	No	No	1282	23
704	79746	79746	Low	Local	No	No	No	Yes	2334	28
743	80208	80213	Low	Local	No	No	No	Yes	1175	23
					Precision	0.50	0.93	0.50		
					Recall	0.07	1.00	0.93		
					Accuracy	0.65	0.98	0.65		

Table 4.1: Results of analyzing 40 versions of JHotDraw; LOC = non-blank, non-comment lines of code before and after the changes; Granu.: granularity of the transformation; Scope: scope of the transformation; Refact. = Is it a refactoring?; #Tests = number of tests used to evaluate the transformation; Cov. (%) = statement coverage on the target program; MH = Murphy-Hill.

Version	LOC		Granu.	Scope	Baseline	Ratzinger	MH	SafeRefactor		
	Before	After			Refact.	Refact.	Refact.	Refact.	# Tests	Cov. (%)
572952	26350	26428	High	Global	No	No	No	Yes	879	29
609497	26428	26428	Low	Local	Yes	No	Yes	Yes	2259	42
637489	26428	26454	High	Local	No	No	No	No	3158	44
656960	26501	26514	Low	Local	No	No	No	Yes	3487	47
711140	26536	26539	Low	Local	No	No	No	Yes	1247	36
814123	26558	26558	Low	Global	Yes	No	Yes	Yes	2972	44
814128	26558	26558	Low	Global	Yes	No	Yes	Yes	2741	44
814997	26558	26761	High	Global	No	No	Yes	No	Compilation Error	
815022	20221	20222	Low	Local	No	Yes	Yes	No	Compilation Error	
815042	20258	20255	Low	Local	No	Yes	Yes	No	Compilation Error	
923339	20901	20901	Low	Local	Yes	No	Yes	Yes	2712	49
956279	20901	20848	High	Local	No	No	No	No	2709	49
966327	20926	21513	Low	Global	Yes	No	Yes	Yes	2667	49
1023771	21551	21551	Low	Global	Yes	No	Yes	Yes	2201	44
1023897	21551	21551	Low	Global	Yes	No	Yes	Yes	2033	44
1095934	21608	21608	Low	Local	Yes	No	Yes	Yes	3180	51
1148801	21618	21628	High	Global	Yes	Yes	Yes	Yes	3237	50
1299210	21627	21627	Low	Global	Yes	Yes	Yes	Yes	1886	49
1300075	21632	21632	Low	Local	Yes	Yes	Yes	Yes	1813	48
1311904	21636	21893	High	Global	No	No	No	Yes	2072	48
					Precision	0.60	0.79	0.73		
					Recall	0.27	1.00	1.00		
					Accuracy	0.50	0.85	0.80		

Table 4.2: Results of analyzing 20 versions of Apache Common Collections; LOC = non-blank, non-comment lines of code before and after the changes; Granu.: granularity of the transformation; Scope: scope of the transformation; Refact. = Is it a refactoring?; #Tests = number of tests used to evaluate the transformation; Cov. (%) = statement coverage on the target program; MH = Murphy-Hill.

	Ratzinger	Murphy-Hill	SAFEREFACTOR
False Positive	3	4	17
False Negative	21	0	1
True Positive	4	25	24
True Negative	32	31	18
Total	60	60	60
Recall	0.16	1.00	0.96
Precision	0.57	0.86	0.59
Accuracy	0.60	0.93	0.70

Table 4.3: Summary of false positives, false negatives, true positives, and true negatives.

refactorings were correct in JHotDraw (precision of 0.93) and 11 out of the 14 detected refactorings in Collections were correct (precision of 0.79). The Murphy-Hill analysis correctly classified 39 out of 40 versions in JHotDraw and 17 out of 20 versions in Collections, leading to an accuracy of 0.98 and 0.85, respectively.

SAFEREFACTOR identified all refactorings but one (Version 503), leading to a recall of 0.93 in JHotDraw sample. However, it also classified 13 non-refactoring as refactoring, which gives it a precision of 0.5. SAFEREFACTOR correctly classified 26 out of the 40 pairs of JHotDraw (Accuracy of 0.65). On the other hand, it had an accuracy of 0.8 in Collections, which means that it was correct in 16 out of the 20 versions. SAFEREFACTOR identified 11 out of the 11 refactorings (recall of 1). However, it incorrectly classified 4 versions as refactoring (precision of 0.73).

Finally, the Ratzinger approach correctly classified 26 out of the 40 versions of JHotDraw (accuracy of 0.65) and 10 out of 20 versions of Collections (accuracy of 0.5). The approach detected 1 (Version 156) out of 14 refactorings in the JHotDraw sample, and 3 out of 11 refactorings in Collections, having recall values of 0.07 and 0.27, respectively. The approach also incorrectly classified three versions as refactoring: Version 173 of JHotDraw (precision of 0.5) and Versions 815022 and 815042 of Collections (precision of 0.6). Table 4.3 summarizes the approaches' results with respect to false positives, false negatives, true positives, and true negatives. It also shows the overall recall, precision, and accuracy of each approach.

Performing the evaluated approaches involves different time costs. The Murphy-Hill approach took around 15 minutes to evaluate each subject. However, in some subjects containing larger changes, the approach took up to 30 minutes and was not able to check all changed files. Ratzinger automatically evaluate the commit message in less than a second.

SAFEREFACTOR took around 4 minutes to analyze each subject.

4.2.5 Discussion

In this section, we interpret and discuss the results. First, we present the main advantages and disadvantages of each approach. Then, we summarize the answers of the research questions (Section 4.2.5).

Murphy-Hill

The manual analysis presented the best results in terms of accuracy, recall, and precision, in our evaluation. An evaluator can carefully review the code to understand the syntax and the semantic changes to check whether they preserve behavior. Although a manual process can be error-prone, the Murphy-Hill et al. approach [49; 50] double checked the results by using two experienced evaluators. Moreover, they systematically decompose the transformation in minor changes making it easier to understand them. They also used a diff tool to help them analyze the transformation.

On the other hand, it is time consuming to analyze all changes in large transformations. For instance, Collections Versions 1148801, 814997, 815042, and 966327 were so large that the reviewers could not inspect all the changes. Furthermore, it is not trivial to identify whether the code compiles by manually inspecting the transformation. The approach classified four versions that do not compile as refactoring.

In Version 357 of JHotDraw, among other changes, the `AbstractDocumentOrientedApplication` class was moved from folder `org/jhotdraw/app` to folder `org/jhotdraw/application`. Although this seems to be a move package refactoring, it fixes a compilation error because the class begins with the statement `package org.jhotdraw.application;` in both versions. Also, the commit message describes the transformation as fixing broken repository, which suggest that the transformation is not a refactoring. SAFEREFACTOR detected compilation errors in this version.

Finally, the manual analysis classified 15 versions as having a mix of refactorings and non-refactorings. The SAFEREFACTOR and Ratzinger approaches are not able to identify

#	Problem	Versions
1	Testing of GUI code	318, 322, 344, 384, 409, 609, 650, 704, 743
2	Tests do not cover impacted methods	173, 267, 322, 344, 649, 650
3	Tests do not cover impacted branches	134, 322, 711140
4	Weak JUnit assertions	650
5	Cannot apply regression testing	572952, 656960, 1311904

Table 4.4: False positives of SAFEREFACTOR; Problem = description of the reason of the false positive; Versions = ids of the versions related to the false positives.

which refactorings are applied.

SAFEREFACTOR

Although the manual analysis had the best results, it is a time-consuming activity to manually analyze all versions. It also depends on experienced evaluators. SAFEREFACTOR has the advantage of automating this process, making an entire repository analysis feasible. In this study, the main problem of SAFEREFACTOR was the high number of false positives in the JHotDraw sample, that is, non-refactorings that were classified as refactoring, which leaded to the precision of only 0.5. In the Collections sample, its precision was close to manual analysis (0.73 to 0.79), though. Next, we discuss about the false positives, false negatives, and also the true negatives of SAFEREFACTOR.

False Positives

SAFEREFACTOR had 13 and 4 false positives in the JHotDraw and Collections samples, respectively. We manually analyzed each one and classified them as shown in Table 4.4. Most of the false positives were related to testing of GUI code. Application code may interact with the user (such as creating a dialog box) in a variety of different situations. In JHotDraw, some generated tests needed manual intervention to cover the functionality under test. SAFEREFACTOR ignored them during evaluation. Moreover, Randoop did not generate tests for methods that require events from the Java AWT framework, for instance `MouseEvent`, since Randoop could not generate this type of dependence.

Recently, a new feature was added to Randoop to allow specifying a mapping from current method calls to a replacement call [66]. For instance,

the `javax.swing.JOptionPane.showMessageDialog` method, which usually presents a dialog box, can be replaced with a call that simply prints out the message and returns. In this way, it can be used to remove dialog boxes that require a response. We plan to incorporate this feature into SAFEREFACTOR's approach in the near future.

SAFEREFACTOR also generated false positives because the tests generated by Randoop within a time limit did not cover methods changed by the transformation. For instance, while in Versions 173, 267, 649, one changed method was not covered by the tests, in Versions 322 and 650, two and three changed methods were not covered, respectively. SAFEREFACTOR passes to Randoop the list of all methods in common for both versions of a pair. The time limit passed to Randoop to generate the tests may have been insufficient to produce a test for these methods. The average statement coverage of the tests was 22.68% and 45.12% in JHotDraw and Collections, respectively. As future work, we intend to improve SAFEREFACTOR by identifying the methods impacted by a transformation. In this way, we can focus on generating tests for those methods.

Moreover, Randoop uses primitive, String and return values as input to the called methods. Still, some methods may present additional dependencies. For instance, parameters from class libraries may not be tested by Randoop if the library is not also under test.

Additionally, in Versions 134, 322, and 711140, Randoop produced tests that call the changed methods, but the tests did not cover the branches affected by the change. In those cases, the arguments produced by Randoop to the methods under test were not sufficient to exercise every behavior possible. The Randoop team recently incorporated the option of using any constant that appears in the source code as input to the methods under test [66]. Moreover, it allows users to specify primitives or String values as input to specific methods. We plan to investigate whether applying them may reduce SAFEREFACTOR's false positives.

On the other hand, in Version 650 there were two changes that were covered by the tests, but the assertion established in the tests were not sufficient to detect the change. For instance, the `ComplexColorWheelImageProducer.getColorAt` method returns an array of floating-point values. Version 650 fixes the value returned by this method, but the test generated by Randoop only checks whether the value returned was not null. If Randoop could generate asserts to check the values of the array, the behavioral change would be detected. The other change affects one private attribute. Recently, Robinson et al. [66] introduced

an enhancement to Randoop that allows the user to define a set of observer methods to the attributes, and check their results – an observer method is a method with no side effects. Therefore, instead of having a single assertion at the end of a generated test, there may be many assertions at the end, one for each applicable observer method. As future work, we will investigate how to automatically compute the observer methods and pass to Randoop to check whether this option improves its effectiveness.

Finally, 3 out of the 4 false positives of Collections were due to addition or removal of methods not used in other parts of the program. If the transformation removes a method, it invalidates every unit test that directly calls the absent method. Likewise, if a method and its unit test is added, this unit test would not compile in the original version. Because of that, SAFEREFACTOR identifies the common methods of the program, and tests them in the two versions of the pair, comparing their results. The tests indirectly exercise the change cause by an added/removed method, as long as this method affects the common methods. Opdyke compares the observable behavior of two programs with respect to the `main` method (a method in common). If it is called twice (source and target programs) with the same set of inputs, the resulting set of output values must be the same [53]. SAFEREFACTOR checks the observable behavior with respect to randomly generated sequences of methods and constructor invocations. They only contain calls to methods in common. Therefore, SAFEREFACTOR can produce false positives due to different equivalence notion in the API context when features are removed or added, since their code may not be used in other parts of the program but only by clients of the API.

False Negatives

In Version 503 of JHotDraw, SAFEREFACTOR showed a false negative. By manually inspecting the results we identified that the behavioral change was due to a non-deterministic behavior of JHotDraw. The test generated by Randoop contained a statement `assertEquals` that indirectly checks the value returned by the `toString` method of an object of class `DrawingPageable`. This class does not implement `toString`. Therefore, it was returned the default value of `toString`, which prints a unique identifier based on the hashcode. The hashcode may change each time the program is executed, which was

the cause of the non-deterministic result.

Nondeterministic results tend to fall into simple patterns, such as the default return value of `toString`. To avoid that, Randoop has the option of executing the tests twice and removing the tests that return different results [66]. We also implemented this option in SAFEREFACTOR, which was used in the experiment. However, it was not sufficient to eliminate all cases of non-deterministic results, such as the one in Version 503.

True Negatives

In this section, we discuss some of the non-behavioral transformations detected by SAFEREFACTOR. In Version 637489 of the Collections API, an overridden method was changed, while Version 956279 changes a `toString` method. Any overridden method may have a very different behavior from the original, which favors its detection by SAFEREFACTOR.

In JHotDraw, Version 151 changes the field value inside a constructor, which is detected by an assertion generated by Randoop. In some transformations, the target program raised an exception. In Versions 176, 518 and 526, SAFEREFACTOR identified a `NullPointerException` in the target program inside a method body and constructors. In Version 324, the transformation removed an interface from a class. The resulting code yields a `ClassCastException` identified by SAFEREFACTOR. Version 596 removed a `System.exit` from a method body.

On the other hand, the behavioral changes found by SAFEREFACTOR in Versions 458, 549, 660, 700 were due to non-deterministic results of JHotDraw. JHotDraw contains global variables that lead to different results of the tests depending of the other that they are executed. SAFEREFACTOR currently executes the tests generated by Randoop in batch through an Ant script. As future work, we plan to implement in SAFEREFACTOR an option to execute the tests in the same order in the source and target versions to avoid non-deterministic results because of the order of the tests.

In our experiments, SAFEREFACTOR had better results evaluating a repository of a data structure library (Collections) than one of a GUI application (JHotDraw). The first one was easier to evaluate since it does not have GUI, does not produced non-deterministic results,

and require simpler arguments to exercise its behavior. On the order hand, APIs are less likely to have behavioral changes during its evolution [66].

Ratzinger

The Ratzinger approach has the advantage of being the simplest and fastest approach for identifying behavior-preserving transformations. However, in our experiment, many of the commit messages do not contain keywords related to refactoring, which led this approach to a recall of only 0.27 in the Collections sample and 0.07 in the JHotDraw sample. Only 4 out of 25 refactoring revisions in both repositories contain some of the refactoring keywords established by the approach.

Additionally, 3 out of 7 refactorings identified by the approach were false positives. In Version 173 of JHotDraw, the commit message indicates that developers removed unused imports and local variables, which suggests the commit was a refactoring. However, by manually inspecting the changes, we checked that one of the removed local variable assignments contains a method call that changes UI components. SAFEREFACTOR also classified this transformation as refactoring since the tests generated by Randoop did not detect this behavioral change in the GUI. This approach also classified Versions 815022 and 815042 as refactoring, but SAFEREFACTOR detected that these versions do not compile, so they cannot be classified as refactorings.

It is not simple to predict refactorings by just inspecting the commit message. The results confirm Murphy-Hill et al. findings [49; 50], which suggest that simply looking at commit messages is not a reliable way of identifying refactorings. Nevertheless, in some situations, if the company recommend strict patterns when writing a commit message, this approach may be useful.

Answers to the research questions

From the evaluation results, we make the following observations:

- **Q1**. Do the approaches identify all behavior-preserving transformations?

  We found evidence that Murphy-Hill approach is capable of detecting all behavior-preserving transformations since it achieved a recall of 1.0. With respect to the auto-

mated approaches, SAFEREFACTOR had an excellent recall of 0.96, but it may miss behavioral changes not detected by the tests or incorrectly detect behavioral changes in non-deterministic programs. On the other hand, our results show evidence that Ratzinger approach may miss a number of behavior-preserving transformations since it had an overall recall of only 0.16. Many of the evaluated behavior-preserving transformations were not documented in the commit messages in the way it is expected by this approach (see Section 4.2.5);

- **Q2**. Do the approaches correctly identify behavior-preserving transformations?

  No. Our results show evidence that the Murphy-Hill approach is the most precise among the evaluated approaches (precision of 0.86). However it may incorrectly classify transformations that contain compilation errors as behavior-preserving transformations. It is difficult to manually reason whether a program compiles. With respect to the automated approaches, the results indicate that SAFEREFACTOR (0.59) is slightly more precise than Ratzinger (0.57). Some of the non-behavior-preserving transformations evaluated contain commit messages related to refactorings that were applied among other changes, leading the Ratzinger approach to incorrectly classify them as behavior-preserving transformations;

- **Q3**. Are the overall results of the approaches correct?

  The results indicate the Murphy-Hill approach is very accurate. In our experiment, it only failed in 4 out of the 60 subjects (accuracy of 0.93). Also, the results show evidence that SAFEREFACTOR is more accurate (0.70) than Ratzinger's approach (0.60). Although close in terms of accuracy, SAFEREFACTOR and Ratzinger have different limitations. While the former had a total of 17 false positives, the latter had just 3. On the other hand, the former had just one false negative, while the latter had 21.

4.2.6 Threats to validity

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

Construct validity

To evaluate the correctness of the results of each approach, we created the baseline (see Column *Baseline* of Tables 5.8 and 4.2) by comparing the approaches' results since we did not previously know which versions contain behavior-preserving transformations. Therefore, if all approaches present incorrect results, our baseline may also be incorrect.

Another threat was our assumption that changes to non-Java files are refactorings. This may not be true in some cases, such as when a library that the code depends upon is upgraded. With respect to SAFEREFACTOR, it does not evaluate developer intention to refactor, but whether a transformation changes behavior.

Internal validity

The time limit used in SAFEREFACTOR for generating tests may have influence on the detection of non-refactorings. To determine this parameter in our experiment, we compared the test coverage achieved by different values of time limit. In general, achieving 100% test coverage in real applications is often an unreachable goal; SAFEREFACTOR only analyzes the methods in common of both programs. For each subject, we evaluated one of the selected pairs, and analyzed the statement coverage of the test suite generated by SAFEREFACTOR on the source and the target programs. After increasing the time limit to more than 120 seconds, the coverage did not present significant variation. So, the value of time limit chosen was 120 seconds. We follow the same approach used in previous evaluations on Randoop [66].

In 17 changes classified as refactoring by SAFEREFACTOR, our manual analysis showed different change classifications. Some of these changes were not covered by SAFEREFACTOR's test suite. In transformations that only modify a few methods, SAFEREFACTOR considers most methods in common. When this set is large the time limit given to Randoop (120s) may not be sufficient to generate a test case exposing the behavioral change. As a future work, we intend to improve SAFEREFACTOR by generating tests only for the methods impacted by the transformation [64]. In this way, we can use SAFEREFACTOR using a smaller time limit.

We used the default value for mostly Randoop parameters. By changing them, we may improve SAFEREFACTOR results. Moreover, since SAFEREFACTOR randomly generates a

test suite, there might be different results each time we run the tool. To improve the confidence, we ran SAFEREFACTOR three times to analyze each transformation. If SAFEREFACTOR does not find a behavioral change in all runs, we consider that the transformation to be behavior-preserving. Otherwise, it is classified as a non-behavior-preserving transformation. The tests generated by Randoop had coverage lower than 10% in some versions of JHotDraw. By manually inspecting the tests, we check that they contain calls to JHotDraw's methods that call `System.exit()`, which ends the test execution. As future work, we plan to improve the test execution by avoiding some method calls.

We manually created the buildFiles for JHotDraw, and downloaded the dependencies. We made sure the compilation errors found by SAFEREFACTOR were not related to any missed dependency. We do not have information on the SVN indicating the JDK version used to build the program. By changing the JDK, results may change. Moreover, we run tests using JDK 1.6.

The Murphy-Hill approach was performed by two experienced evaluators. One one them was the author of the approach. They also have an extensive background in refactoring. The accuracy of this approach may change according to the level of Java expertise of the inspectors.

External validity

We evaluated only two open-source Java projects (JHotDraw and Apache Collections) due to the costs of manual analyses. Our results, therefore, are not representative of all Java projects. To maximize the external validity we evaluated two kinds of software: a GUI application (JHotDraw) and an API (Apache Common Collections).

Randoop does not deal with concurrency. In those situations, SAFEREFACTOR may yield non-deterministic results. Also, SAFEREFACTOR does not take into account characteristics of some specific domains. For instance, currently, it does 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. These changes may be non-trivial to be tested by using JUnit tests.

Moreover, some changes (Versions 743 and 549) improve the robustness of JHotDraw. Randoop could not generate test cases that produce invalid conditions of JHotDraw to iden-

tify these behavioral changes. Also, it seems that some of the bug fixes need complex scenarios to expose behavioral changes. For instance, Version 267 introduces a work-around in one method to avoid a bug in the JDK. Since we may have tested it using a new JDK, probably, the transformation does not change program's behavior. In Version 700, developers change some instructions to assign a copy of the array instead of the array itself. Although this change fixed the array exposure, Randoop could not detect any behavioral change.

Similarly, the manual analysis presents a number of limitations as well. Manually inspecting code leaves room for human error. We only selected changes from two projects (JHotDraw and Collections), which may not be representative of other software projects. In other software domains, it may be harder to understand the logic of the software and define whether the change preserves behavior. Moreover, Java semantics is complex. Even formal refactoring tools may fail to identify whether a transformation preserves behavior [77]. We tried to mitigate this by having two experienced evaluators simultaneously analyzing the source code. Finally, during our manual analysis, we encountered six very large changes that we were unable to manually inspect completely; in these cases we spent about 30 minutes manually cataloging refactorings, but did not find any semantics changes in doing so. Had we spent significantly more time inspecting, we may have encountered some non-refactorings. This illustrates that manual inspection, while theoretically quite accurate, is practically difficult to perform thoroughly.

4.3 Concluding remarks

In this chapter, we presented SAFEREFACTOR, a tool for detecting behavioral changes. Its key idea is to compare the behavior of two versions of a program against the same tests. To do so, it identifies the methods in common before and after the transformation, generates tests for them, and run these tests against both programs. If the results are the same, it improves the confidence that both programs have same behavior. Otherwise, it detects a behavioral change.

We performed an experiment to compare SAFEREFACTOR and other two approaches (Murphy-Hill and Ratzinger) with respect to effectiveness in detecting behavioral changes. Our results suggest that SAFEREFACTOR has 70% accuracy. The evaluation in Section 4.2

shows some limitations of SAFEREFACTOR. For instance, it produced false positives when testing GUI code and false negatives when testing non-deterministic code. These limitations do not affect the use of SAFEREFACTOR in our technique for testing of refactoring engines since we use it against simple transformations that are deterministic and do not have GUI code.

Chapter 5

A technique for testing of refactoring engines

In this chapter, we present our technique for automated testing of Java Refactoring Engines. It focuses on identifying problems related with missing conditions and strong conditions. The key elements of the technique are JDOLLY (Chapter 3) and SAFEREFACTOR (Chapter 4).

The remainder of this chapter is organized as follows. Section 5.1 shows an overview of our technique. Then, each step of our technique is described from Section 5.2 to Section 5.5. Sections 5.6 and 5.7 describe our experiments to evaluate the technique. Finally, Section 5.8 shows the concluding remarks.

5.1 Overview

We propose an automated approach for testing of Java refactoring engines. The approach performs four major steps. First, a program generator automatically yields programs as test inputs for a refactoring (Section 5.2). Second, the refactoring under test is automatically applied to each generated program (Section 5.3). Then, the output is evaluated by test oracles in terms of missing conditions and overly strong conditions (Section 5.4). In the end, we may have detected a number of failures, which are categorized in Step 4 (Section 5.5). The whole process is depicted in Figure 5.1.

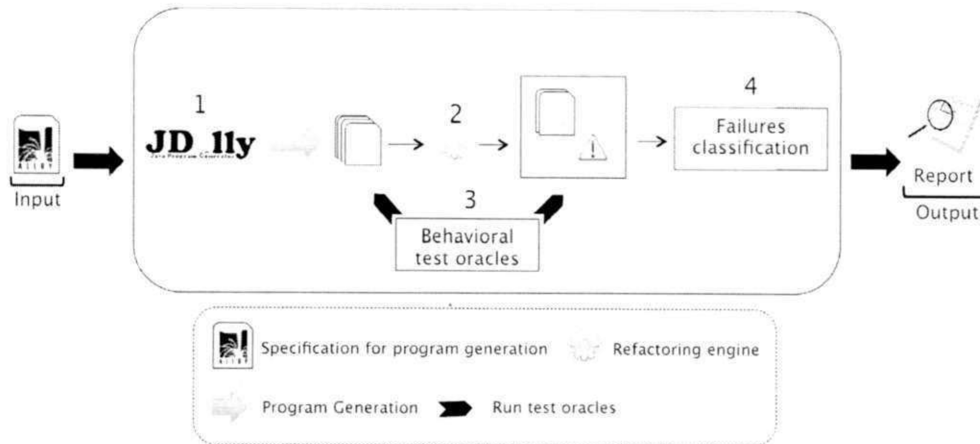


Figure 5.1: Automated behavioral testing of refactoring engines.

5.2 Test input generation

We automatically generate programs as test inputs for the refactoring engines. To perform the test input generation, we propose a Java program generator called JDOLLY. We show a detailed description of JDOLLY in Chapter 3.

5.3 Refactoring application

The second step of our technique is to apply the refactoring under test to each generated program. This step can be performed manually (by using the IDE directly) or by the use of an API offered by the IDE infrastructure. Each refactoring checks a set of conditions, and, given the fulfillment of these conditions, the transformation is applied; otherwise, the refactoring is rejected, and a warning message is shown.

5.4 Test oracles

An important problem in automated testing of refactoring engines is automated checking of outputs. In practice, developers manually write the expected output, which can be a refactored program or a warning message when a condition is violated. Next, we show our automated oracles to detect missing conditions and overly strong ones.

5.4.1 Missing conditions

We propose SAFEREFACTOR (Chapter 4), a tool for checking behavioral changes, as oracle for detecting missing conditions. For each pair of input and output programs produced by the technique, SAFEREFACTOR checks for behavioral changes. If it detects behavioral changes, we classify the transformation as a failure.

For instance, Listing 4.1 shows a Java program generated by JDOLLY, and Listing 4.2 shows the output program after applying a Pull Up Method refactoring by using Eclipse. Since SAFEREFACTOR detects behavioral changes in this transformation, we classify it as a failure. In Section 5.5.1 we show how to classify failures due to behavioral changes into distinct faults.

5.4.2 Overly strong conditions

We propose an oracle to detect overly strong conditions based on differential testing [81]. When the refactoring implementation under test rejects a transformation, we apply the same transformation by using one or more other refactoring implementations. If one implementation applies the transformation, and SAFEREFACTOR does not find behavioral changes, we establish that the implementation under test contains an overly strong condition since it rejected a behavior-preserving transformation.

For example, consider the `A` class and its subclass `B` in Listing 5.1. `A` declares the `k(long)` method, and `B` declares methods `n` and `test`. Suppose we would like to rename `n` to `k`. If we apply this transformation using Eclipse, it shows the warning message: *Method "A.k(long)" will be shadowed by the renamed declaration "B.k(int)".*

Eclipse has a functionality that allows us to preview the transformation. In the previous example, Listing 5.2 presents the preview of the resulting program. Notice that after the transformation, the `test` method yields `20`, but in the original version it yields `10`. This transformation does not preserve behavior. This is the reason why Eclipse showed a warning message.

However, we can apply this transformation using JRRT. The resulting program is presented in Listing 5.3. Notice that this transformation is different from Eclipse. JRRT performs an additional change to make the transformation behavior-preserving. JRRT identifies

that the call to `k` inside `test` must refer to `A.k` instead of `B.k` after the transformation. So, it adds a `super` access to the method invocation `k(2)` inside `test`. Therefore, the resulting program in Listing 5.3 correctly refactors the original program in Listing 5.1. NetBeans can also perform the transformation. It yields a target program presented in Listing 5.2. However, the transformation performed by NetBeans does not preserve behavior.

Listing 5.1: Original version.

```
1  public class A {
2    public long k(long a) {
3      return 10;
4    }
5  }
6  public class B extends A {
7    public long n(int a) {
8      return 20;
9    }
10   public long test() {
11     return k(2);
12   }
13 }
```

Listing 5.2: NetBeans target version.

```
1  public class A {
2    public long k(long a) {
3      return 10;
4    }
5  }
6  public class B extends A {
7    public long k(int a) {
8      return 20;
9    }
10   public long test() {
11     return k(2);
12   }
13 }
```

Listing 5.3: JRRT target's version.

```
1  public class A {
2    public long k(long a) {
3      return 10;
4    }
5  }
6  public class B extends A {
7    public long n(int a) {
8      return 20;
9    }
10   public long test() {
11     return super.k(2);
12   }
13 }
```

We compare the results of Eclipse, NetBeans, and JRRT. While the former rejected the transformation, NetBeans and JRRT applied it. SAFEREFACTOR evaluates the transformations applied by JRRT and NetBeans. It does not find behavioral changes in the transformation applied by JRRT. We conclude that Eclipse rejected a behavior-preserving transformation due to an overly strong condition since JRRT was able to correctly apply it. Moreover, it detects a fault (missing condition) in the transformation applied by NetBeans.

5.5 Failure classification

Our technique may produce a large number of failures since it automatically produces a number of test inputs. The process to manually classify the failures into distinct faults may demand a considerable effort. In the following subsections, we present techniques to automate the classification of failures into distinct faults.

5.5.1 Missing conditions

Missing conditions may produce two main types of failures: the ones that introduce compilation errors in user's code; and the failures that introduce behavioral changes in user's code.

Compilation errors

Jagannath et al. [34] propose an approach to split failures based on oracle messages (Oracle-based Test Clustering - OTC). They used it to classify refactoring engine failures that introduce compilation errors in the output program. The failures are grouped by the template of the compiler error message, so that each group contains a distinct fault. We adopt the same approach to classify this kind of failure.

For instance, Listings 5.4 shows a program generated by JDOLLY. If we apply the Rename Field refactoring by using JRRTv1, the tool will produce the output program shown in Listing 5.5, which contains the compilation error: "The field A.k is not visible". Listings 5.6 shows another program generated by JDOLLY. The only different between it and the previous program (Listings 5.4) is the addition of the `C` class. If we apply the same Rename Field refactoring, JRRTv1 will produce an output program (Listings 5.7) with the same kind of compilation error. Our technique groups both transformations together by using the template of the compilation error: "The field [F] is not visible".

Listing 5.4: Before Refactoring

```
1  package p1;
2  public class A {
3    protected int n=1;
4  }
5
6  package p2;
7  import p1.*;
8  public class B extends A {
9    int k=2;
10   public long m(){
11     return this.n;
12   }
13 }
```

Listing 5.5: After Refactoring. Applying Rename Field in JRRTv1 leads to a compilation error.

```
1  package p1;
2  public class A {
3    protected int k=-31;
4  }
5
6  package p2;
7  import p1.*;
8  public class B extends A {
9    int k=17;
10   public long m(){
11     return ((A)this).k;
12   }
13 }
```

Listing 5.6: Before Refactoring

```
1  package p1;
2  public class A {
3    protected int n=1;
4  }
5
6  package p2;
7  import p1.*;
8  public class B extends A {
9    int k=2;
10   public long m(){
11     return this.n;
12   }
13 }
14
15 package p2;
16 public class C {
17 }
```

Listing 5.7: After Refactoring. Applying Rename Field in JRRTv1 leads to a compilation error.

```
1  package p1;
2  public class A {
3    protected int k=-31;
4  }
5
6  package p2;
7  import p1.*;
8  public class B extends A {
9    int k=17;
10   public long m(){
11     return ((A)this).k;
12   }
13 }
14
15 package p2;
16 public class C {
17 }
```

Table 5.1: Filters for classifying behavioral changes.

Filter	Description
Enables/disables overriding	After a refactoring, a method comes to be (or no longer is) overridden
Enables/disables overloading	After a refactoring, a method comes to be (or no longer is) overloaded
Enables/disables field hiding	After a refactoring, a field comes to be (or no longer is) hidden by another field declaration
Shadows class declaration	After a refactoring, a class declaration comes to be shadowed by another declaration
Changes **super**(**this or implicit this**) to **this or implicit this** (**super**)	If a method call or field access has **this or implicit this** (**super**) as target, and after a refactoring this reference is replaced by **super** (**this or implicit this**), in order to keep the link to the same previous object
Maintains **super** while changing hierarchy	A reference to **super** is moved up or down the hierarchy during refactoring
Changes accessibility	The refactoring changes the access modifier of a given field or method
The refactored program crashes	The original program is normally executed by the test suite but the refactored one throws some exceptions
Enables/disables implicit cast	After a refactoring, an implicit cast between primitive types is (or no longer is) applied where it did not take (or took) place originally

Behavioral changes

We do not use the OTC approach for classifying failures related to behavioral changes since we did not identify any information from our oracle (SAFEREFACTOR) that could be used to split the failures. We propose an approach to classify behavioral changes by splitting each detected change based on the characteristics of each pair of input and output programs. Our approach is based on a set of *filters*; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the output program, relatively to the input program. All filters are presented in Table 5.1. We defined these filters by analyzing faults found through the use of our approach, in addition to other reported faults.

The filters may be applied in any order. The fault category of a behavior-changing transformation is then designated by the filters matched by its input and output programs. When a transformation does not match any of these filters, conventional debugging is demanded from refactoring engine developers. For instance, the failure in the Pull Up Method on either Eclipse JDT 3.7 or JRRTv1 showed in Listing 5.2 matches the filter named "Changes **super**(**this**) to **this**(**super**)" from Table 5.1, in which a problem with replacing a reference to `super` with `this` is detected.

The set of filters is not complete. Currently, they focus on the Java constructs supported by JDOLLY. New filters can be proposed based on additional faults found by refactoring

engine developers. Currently, the classification of behavioral changing transformations is carried out manually. The process consists in analyzing each pair of programs, and testing every filter for matches.

5.5.2 Overly strong conditions

We also use OTC to categorize the overly strong condition failures. Our hypothesis is that each condition has a particular warning message. Therefore, to split the failures, we use the template of the warning message thrown by a refactoring engine when a condition is not satisfied.

For example, when we apply the Rename Method refactoring of Eclipse to the program shown in Listing 5.1, the tool yields the following warning messages, respectively: *Method "A.k(long)" will be shadowed by the renamed declaration "B.k(int)"*. Our approach ignores the parts inside quotes, which contain names of packages, classes, methods, and fields. If there is another message that has the same template, the rejected transformations are automatically classified in the same category of overly strong condition.

5.6 Evaluation: missing conditions

The goal of this experiment is to analyze our technique for the purpose of evaluation with respect to effectiveness in identifying faults related to missing conditions from the point of view of refactoring engine developers in the context of academic and industrial Java refactoring engines. In particular, our experiment addresses the following research question:

- **Q1**. Can the technique identify faults related to missing conditions?

To address our research questions, we assess the effects of each technique by using the following metric:

- Number of distinct faults correctly detected by the technique.

5.6.1 Planning

In the following subsections, we describe the subjects used in the experiment, the experiment design, and its instrumentation.

Table 5.2: Summary of evaluated refactorings; Scope = Package (P) - Class (C) - Field (F) - Method (M).

Refactoring	Scope P-C-F-M	Eclipse	JRRT	NetBeans
Rename class	2-3-0-3	x	x	x
Rename method	2-3-0-3	x	x	x
Rename field	2-3-2-1	x	x	x
Push down method	2-3-0-4	x	x	x
Push down field	2-3-2-1	x	x	x
Pull up method	2-3-0-4	x	x	x
Pull up field	2-3-2-1	x	x	x
Encapsulate field	2-3-1-3	x	x	x
Move method	2-3-1-3	x	x	-
Add parameter	2-3-0-3	x	x	x

Selection of subjects

We evaluated Java refactorings implemented by Eclipse JDT 3.7 (10 refactorings), JRRTv1 and JRRTv2[1] (10 refactorings), and NetBeans 7.0.1 (9 refactorings). Table 5.2 summarizes all evaluated refactorings.

Eclipse is the most used Java IDE [48], and contains a number of automated refactorings (currently, more than 25). The evaluated refactorings focus on a representative set of program structures. Moreover, a survey carried out by Murphy et al. [48] shows the Eclipse JDT refactorings that Java developers use most: Rename, Move Method, Extract Method, Pull Up Method, and Add Parameter. Four of these are evaluated in this experiment. NetBeans is also a popular Java IDE. The Move Method refactoring was not supported by NetBeans by the time that this experiment was performed. A number of related approaches [14; 84; 71] have studied the correctness of their transformations.

JRRT implements a number of refactorings [71; 74; 68]. They aim at outperforming the refactoring implementations of Eclipse in terms of overly strong and too weak conditions. Some refactorings may have invariants to be preserved. For instance, their Rename Method refactoring implementation is based on the name binding invariant: each name should refer to the same entity before and after the transformation. They proposed other invariants such as control flow and data flow preservation. To alleviate the problem of overly strong conditions, their implementations may also perform additional changes, such as the one presented in the

[1]The JRRT version from July 9th, 2011

Table 5.3: Summary of the main constraints.

Refactoring Implementation	Main Constraint	Additional Constraints
Rename Class	some Class	{1, 2, 3}
Rename Method	some Method	{2, 3, 8}
Rename Field	some Field	{2, 3, 9}
Push Down Method	some c:Class | someSubclass[c] and someMethod[c]	{1, 6}
Push Down Field	some c:Class | someSubclass[c] and someField[c]	{2, 4}
Pull Up Method	some c:Class | someParent[c] and someMethod[c]	{1, 6}
Pull Up Field	some c:Class | someParent[c] and someField[c]	{2, 4}
Encapsulate Field	some Field	{5, 6, 7}
Move Method	some c:Class | someTargetClassField[c] and someMethodToMove[c]	{1, 2}
Add Parameter	some Method	{1, 2, 3}

transformation from Listing 5.1 to Listing 5.3.

We evaluated two versions of JRRT [71; 74; 68]. First, we evaluated with our technique the refactorings implemented by JRRTv1. Later, a new version with improvements and bug fixes was released (which we call JRRTv2); this new version was also subject to our analysis in order to evaluate whether our technique could be useful for identifying new faults during the evolution of the tool. The same refactorings from Eclipse JDT were tested in both versions of JRRT.

Experiment design and instrumentation

The scope column in Table 5.2 indicates the maximum number of packages, classes, fields, and methods passed as parameter to JDOLLY. For each refactoring, we specified *main constraints* for guiding JDOLLY to generate programs with certain characteristics needed to apply the refactoring. Table 5.3 shows these constraints; they prevent the generation of programs to which the refactoring under test is not applicable. For each refactoring, we used the same set of generated programs to evaluate Eclipse JDT, JRRTv1, JRRTv2, and NetBeans.

Exhaustively generating programs, even for a given scope, often causes state space explosion. In order to minimize the number of generated programs to a small, focused set, we have also defined *additional constraints*. These constraints were built on data about refactoring faults gathered in the literature, enforcing properties such as overriding, overloading, inheritance, field hiding, and accessibility. For each refactoring (column Additional Constraints in Table 5.3), we declare Alloy facts with additional constraints. These are fully described in Table 5.4. If a developer has the available resources to analyze the entire scope, then it will not be required to specify additional constraints.

Table 5.4: Summary of the additional constraints.

Id	Additional Constraint	Description
1	someOverriding[] or someOverloading[Int,Int]	overriding or overloading (number of parameters passed as argument)
2	someCaller[]	At least one method body calling a method or accessing a field
3	someInheritance[]	At least one case of inheritance
4	someFieldHiding[]	At least one case of field hiding
5	someGetter[]	At least one getter method
6	someTesterMethod[]	At least one method body with a simple call to a specific method
7	somePublicField[]	At least one public field
8	someMethodsWithSameNumParameter[]	At least two methods with the same number of parameters
9	somePrimitiveFields[]	At least two primitive fields

Each refactoring may possibly include parameters. For instance, a method can be renamed, or a field may be encapsulated. In those cases, we declare a singleton subsignature for each parameter, similar to what we have done with `C1`, `C2` in Section 3.4, and use it in both the main and the additional constraints.

5.6.2 Operation

We performed the evaluation on a 2.5 GHz dual-core PC with 1 GB of RAM. We used the SAFEREFACTOR command-line version with a time limit of one second, which is enough for testing the small generated programs. Cobertura[2] was used to collect the statement coverage of the test suite as generated by SAFEREFACTOR in the resulting program.

JDOLLY generated 153,444 programs to evaluate all refactorings. Even though Eclipse JDT, JRRT and NetBeans have their own test suites, our technique identified 120 (likely) *distinct* faults related to missing conditions. Table 5.5 summarizes the faults reported to Eclipse JDT, NetBeans and JRRT.

From our catalog, most faults were accepted (87). Some faults have not been dealt with by Eclipse JDT and NetBeans developers prior to this writing (22). All faults accepted by JRRT developers in JRRTv1 (20) were fixed in JRRTv2. We have also evaluated their new version (JRRTv2) after fixing the faults from JRRTv1, and reported 11 faults. They did not consider 4 faults due to the closed world assumption (CWA) adopted by them, as we discuss in Section 5.7.1. More importantly, they incorporated our test cases into their test suite[3].

[2] http://cobertura.sourceforge.net

[3] http://code.google.com/p/jrrt/source/checkout

Table 5.5: Summary of faults reported.

	Submitted	Accepted	Duplicated	Not a Fault	Not Answered	Fixed
Eclipse	34	34	16	0	0	2
JRRTv1	24	20	0	4	0	20
JRRTv2	11	6	0	5	0	6
NetBeans	51	27	0	2	22	7
Total	**120**	**87**	**16**	**11**	**22**	**35**

Eclipse JDT and NetBeans teams have fixed 2 and 7 faults[4], respectively, which should be included in the next version of the IDEs. Developers have already confirmed 34 and 27 faults in Eclipse JDT and NetBeans, respectively. However, 16 faults were considered duplicated in Eclipse JDT.

It took from 1h36m to 50h24m to evaluate each refactoring. This includes the time required to generate and compile the input programs, apply the transformations, compile the resulting programs, run SAFEREFACTOR, and collect the statement coverage. The required amount of time depends not only on the number of programs to be refactored, but also on the number of transformations to be carried out. For example, it took 6h54m to test the Rename Method refactoring on Eclipse JDT, whereas it took 13h36m to test the same refactoring in JRRTv2, with the same inputs. Time also depends on the static analysis performed by each refactoring to check conditions. Table 5.8 summarizes the experimental results.

The results include the number of programs generated by JDOLLY, the percentage of compilable programs, the time for testing, and the number of detected failures (encompassing compilation errors and behavioral changes). It also shows the number of faults identified by our approach in each refactoring. Table 5.8 indicates, for each refactoring, the mean value of the statement coverage from the refactored program.

Compilation Errors

Our technique detected 16 faults in Eclipse JDT, 11 faults in JRRTv1, 1 fault in JRRTv2, and 29 faults in NetBeans; all related to compilation errors. Our technique for classifying failures (Section 5.5.1) takes a few seconds to automatically classify all compilation error failures of a refactoring. For instance, our technique detected 1,267 compilation failures in the Push Down Method refactoring implementation of Eclipse JDT. The described approach

[4]The id of all faults are available at: http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.html

Table 5.6: Overall experimental results; GP = number of generated programs; CP = number of compilable programs (%); Time = total time to test the refactoring in hours; Fail. = number of detected failures; Bug = number of identified faults.

Refactoring	GP	CP (%)	Total Time (h)				Compilation Error								Behavioral Change								Statement Coverage (%)			
			Ecl.	JRRTv1	JRRTv2	NetB	Eclipse		JRRTv1		JRRTv2		NetBeans		Eclipse		JRRTv1		JRRTv2		NetBeans		Ecl.	JRRTv1	JRRTv2	NetB
							Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug	Fail.	Bug				
Rename class	15,322	74.3	6.7	23	18.3	22.84	1,016	3	0	0	0	0	3,352	4	145	1	0	0	0	0	15	1	54	63	67	56
Rename method	11,263	79.5	6.9	8.7	13.6	23.2	559	1	0	0	0	0	1,731	2	0	0	0	0	482	2	1,231	2	83	84	90	86
Rename field	19,424	79.2	29.3	22.4	30.4	50.41	48	1	520	2	0	0	326	3	0	0	167	1	0	0	1,667	1	100	100	100	100
Push down method	20,544	78.5	11.9	11.6	16.9	31.9	1,267	2	1,989	2	0	0	10,323	4	853	5	258	4	716	3	1,485	6	90	90	93	90
Push down field	11,936	79.1	6	3.7	4.6	13.7	342	1	0	0	0	0	6889	4	92	1	0	0	0	0	270	2	100	100	100	100
Pull up method	8,937	72	7.3	6.3	6.9	13.5	269	2	549	2	0	0	3,049	3	202	3	78	2	10	1	1,073	5	90	90	92	89
Pull up field	10,927	79.7	8.6	5.3	7.7	12.1	518	1	80	2	0	0	1,128	4	546	4	0	0	0	0	239	2	100	99	100	100
Encapsulate field	2,000	92.8	2.5	1.6	2.3	5.7	238	1	0	0	0	0	234	1	0	0	344	1	437	1	439	1	66	81	86	76
Move method	22,905	69	10.3	4.5	5.9	-	214	2	1,098	3	9	1	-	-	3,586	3	1,759	3	6,944	3	-	-	82	82	86	-
Add parameter	30,186	63	34.69	24.61	25.05	50.36	1,663	2	0	0	0	0	5,824	4	2,238	2	378	2	0	0	2,186	2	87	87	90	87
Total	**153,444**	**68.8**	**124.2**	**112.15**	**131.65**	**223.7**	**6,134**	**16**	**4,236**	**11**	**9**	**1**	**32,856**	**29**	**7,662**	**18**	**2,984**	**13**	**8,152**	**10**	**8,605**	**22**				

classified them into two groups: some transformations produced the message *"The method [M] from the type [T] is not visible"*, while others produced the message *"No enclosing instance of the type [T] is accessible in scope"*. Consequently, two faults were catalogued.

Even though all evaluated refactorings implemented by Eclipse JDT and NetBeans contain at least one fault related to compilation errors, our approach did not find faults related to compilation errors in 50% and 90% of the refactorings of JRRTv1 and JRRTv2, respectively. In Eclipse JDT, the Rename Class refactoring contains three faults; from JRRTv1 and JRRTv2, the Move Method refactoring showed more faults than the other refactorings. In NetBeans, three refactorings contain four faults each. Notice that the Rename Field, Pull Up Field and Move Method implemented by JRRTv1 have more faults than the similar implementation of Eclipse JDT. After fixing them, JRRTv2 presented fewer faults than Eclipse JDT.

Behavioral Changes

We identified 18, 13, 10 and 22 faults in Eclipse JDT, JRRTv1, JRRTv2 and NetBeans, respectively, all related to behavioral changes. We manually classified these faults by using our proposed set of filters (Section 5.5.1). For each refactoring type, it took approximately two hours to manually classify behavioral changes. As future work, we intend to implement tools to automate this process. For instance, Listings 5.1 and 5.2 show a fault of the Pull Up Method refactoring implemented in the Eclipse JDT, categorized as *"Change super to this"*.

5.6.3 Discussion

Next we discuss some issues related to compilation error, behavior preservation and JDOLLY.

Compilation Errors

Changing the name, location, or accessibility of a declaration can lead to compilation errors. All engines but JRRTv2 produced transformations that reduced the accessibility of an inherited method, which is not allowed in Java. Most compilation errors were due to dereferences of inaccessible or nonexistent declarations. For example, in Listing 5.8, `m` accesses the `f` field of its super class. If we apply the Pull Up Field refactoring of Eclipse JDT 3.7 to `B.f`, it yields the uncompilable program presented in Listing 5.9. After the transformation, `B.f` hides `A.f`, and since it is private, it cannot be accessed from `C`. To prevent such errors, JRRT statically checks whether every identifier refers to the same declaration as before. In that case, however, JRRTv1 introduced another compilation error by re-qualifying field access `super.f` to `((A)super).f`, which has a syntax error. We reported this fault to JRRT developers, and they fixed it. JRRTv2 correctly applies the transformation by re-qualifying the `super.f` field access to `((A)this).f`.

Moreover, JRRT refactorings translate the programs into a richer language, which provides a more straightforward specification. After this, the programs are translated back into Java. Although the implementation of the refactoring itself becomes simpler, it does require some effort to translate the program back from the enriched language into the base language. Our technique detected some failures in JRRTv1 that may be related to this step. For instance, some of the refactored programs presented compilation errors due to method invocations for non existing declarations, such as `unknown()`.

Although we only evaluated 9 refactorings from NetBeans, those refactorings contained more faults related to compilation errors than Eclipse JDT and JRRT. It seems that NetBeans does not implement a number of expected conditions. Since its refactorings present a lower rate of rejections, it takes, in general, more time to evaluate NetBeans than the other tools.

Listing 5.8: Before Refactoring.

```
1
2
3
4 public class A {
5   long f = 1;
6 }
7 public class B extends A {
8 }
9 public class C extends B {
10   private long f = 2;
11   public long m(){
12     return super.f;
13   }
14 }
```

Listing 5.9: After Refactoring. Pull Up Field implemented by Eclipse JDT 3.7 introduces a compilation error due to an invisible field.

```
1 public class A {
2   long f = 1;
3 }
4 public class B extends A {
5   private long f = 2;
6 }
7 public class C extends B {
8   public long m(){
9     return super.f;
10  }
11 }
```

Behavioral Changes

Some faults related to overloading and overriding have been known by Eclipse JDT developers for years. For instance, a fault related to the Add parameter refactoring has demanded the inclusion of additional conditions since 2004[5]. Nevertheless, it is difficult to establish and check conditions to avoid these faults. While the Add Parameter fault is still open, Eclipse JDT developers implemented simpler conditions for Rename Method, checking whether there are other methods in the hierarchy with the same signature as that of the refactored method. If so, the engine warns the user that the transformation may introduce behavioral changes. In this case, it is up to the user to analyze whether the transformation is safe.

For each refactoring, we analyzed the statement coverage of the random test suite used by SAFEREFACTOR over the program after refactoring; from these, we calculated the mean value of the statement coverage (see Table 5.8). The minimum mean value of the statement coverage of Eclipse JDT, JRRTv1, JRRTv2, and NetBeans in our evaluation was 54%, 63%,

[5]See Eclipse JDT Bug 58616

67%, and 56%, respectively, for the Rename Class refactoring. These numbers can be partially explained by the tests generated only for methods in common. Additionally, most of the programs generated by JDOLLY contain at most four methods, and fewer than 15 LOC. If a class or a method is renamed, and they are not referred to by methods with unchanged signatures, the statement coverage decreases significantly. Since refactorings engines may allow different transformations, and the test suite is randomly generated in SAFEREFACTOR, the mean value of the statement coverage may be different between engines.

The detected faults can be fixed either by modifying conditions or changing the transformation itself. For instance, one fault reported to JRRT generates a program with the following code fragment: `((A)super)`. This is an invalid Java expression. We can fix this fault by modifying the transformation applied by JRRT, which rewrites a command with the incorrect fragment. However, fixing faults may not be as straightforward as it appears to be. For example, consider the transformation showed in Listings 5.8 and 5.9. We can fix this fault by adding a condition avoiding this kind of transformation. However, adding conditions may avoid useful behavior-preserving transformations. JRRTv1 can apply this transformation, and yet preserve program behavior by replacing the super field access to a qualified this field access, `((A)this).f`.

JDOLLY

During evaluation, we specified the scope of the program generation in JDOLLY based on previous examples of faults in refactorings. For instance, we used the scope of two packages since Steimann and Thies [84] show accessibility problems when moving elements between packages. Schäfer et al. [73] show non-behavior-preserving transformations in programs with up to three classes and four methods/fields. Since JDOLLY exhaustively generates programs for a given scope, this approach has been useful for detecting faults that have not been detected so far.

JDOLLY generated uncompilable programs. The lowest percentage of compilable programs was in the Add Parameter (63%), and the highest was in the Encapsulate Field (92.8%). Considering all generated programs, the percentage of compilable programs was 68.8%. For future work, we intend to specify more well-formedness constraints so as to minimize uncompilable programs.

Our Java metamodel does not include constructs such as the **static** modifier, inner classes, interfaces, and richer method bodies. Therefore, the currently implementation of JDOLLY cannot reveal some previously identified faults in manual experiments [73]. We aim at improving the expressiveness of the programs generated by JDOLLY by adding more constructs to our model. This will increase the state space for the Alloy Analyzer to find solutions and, consequently, the number of programs generated by JDOLLY, which will take longer to evaluate all transformations. We plan to investigate the possibility of generating a greater range of programs, specifying as well a time limit, or limiting the number of generated programs. As a result, we will be able to evaluate refactorings by means of more sophisticated programs, though without considering the entire solution space.

Test data adequacy criteria provide measurements of test quality. Moreover, it may provide explicit rules to determine when it is appropriate to end the testing phase [24; 93]. There are a number of notions of test data adequacy. For instance, test data adequacy can be defined in terms of covering all productions in grammar-based testing. In our work, we have used a similar test data adequacy criterion. JDOLLY generates *every* possible program, for a subset of the Java metamodel, within a given scope of constructs. As such, the generator covers every terminal symbol and nonterminal production rule from the metamodel, which are represented by signatures and relations from the underlying Alloy specification. In the evaluation of the refactorings (Table 5.2), JDOLLY generated programs covering from 71% to 85% of the 41 signatures and relations of the metamodel. Some signatures and relations were not covered because we had specified a scope of 0 for `Field`. In other cases, some additional constraints implied that some relations could not have values.

5.6.4 Threats to Validity

Next we identify some threats to validity from the evaluation performed.

Construct Validity

Some tool developers follow a closed world assumption (CWA) to evaluate the correctness of the transformation. CWA means that what is not currently known by the refactoring engine does not exist. Since we generate tests after the refactoring, the tool does not consider the

generated tests when checking the refactoring conditions. In few cases, JRRT developers did not accept the faults because the tool would have detected the behavioral change if the tests existed by the time of the refactoring.

Despite the different criteria, many other reported faults were accepted by JRRT, Eclipse and NetBeans developers (see Table 5.5). Although our technique may produce false positives, it was considered useful by those developers in practice. In particular, the feedback given by the JRRT team shows evidence that our technique is convenient in detecting faults under both CWA and OWA criteria.

Internal Validity

Concerning JDOLLY generation with Alloy, additional constraints may hide possibly detectable faults. These constraints can be too restrictive with respect to the programs that can be generated by JDOLLY, which shows that one must be cautious when creating constraints for JDOLLY.

The results provided by SAFEREFACTOR deserve closer analysis. If, out of the programs generated by JDOLLY no compilation error or behavior change is detected, no definitive conclusion can be drawn from the refactoring under test. Our technique cannot, based on the absence of behavior changes, claim that a refactoring is correct. Nevertheless, developers have stronger evidence that the refactoring is correctly implemented, in practice; we use a test suite to evaluate the transformation.

SAFEREFACTOR only generates test suites that exercise methods with unchanged signatures. Methods with changed signatures may be called by the unchanged methods, which exercise a potential change of behavior. Otherwise, methods not called by others are not considered, in our approach, part of the overall behavior of the system under test; changes in these methods will not affect the system behavior. A stronger notion of equivalence could be used: testing every changed method of the system and creating a mapping between two versions of the modified versions, for comparing their results. We believe that this approach would add considerable costs with limited benefits to testing refactoring engines.

External Validity

We believe that other refactoring engines can be tested as well with our technique. This exercise can be accomplished by applying a test generator for the target language (a substitute for Randoop) and adaptations to SAFEREFACTOR. Also, the target language's metamodel must be provided to JDOLLY; or else we can use a different program generator. Therefore, refactoring engines targeted at other object-oriented programming languages can benefit from our technique.

Regarding some refactoring transformations other than the ones evaluated in this experiment, we have showed that our technique is applicable to any transformation, because it does not rely on specific properties of the transformation. In order to generate programs that exercise a specific refactoring, we may have to change the Alloy specification in JDOLLY.

The technique for classifying behavioral change failures described in Section 5.4 is limited, since the classification is not complete. We have only considered a subset of Java. Still, it is non-trivial to pinpoint a fault in a refactoring. Each refactoring engine may incorporate different design choices. Our fault categorizer is an approximation, and it may help refactoring engine developers with this task. For example, our approach may classify two distinct faults under the same category. After fixing the identified faults, the developer should re-run the technique to catch possibly missed faults. Moreover, our approach may identify two distinct faults that are, in fact, just one. Developers can easily detect whether two different test cases are related to the same fault by fixing each fault and running all faults again after. In spite of that, the technique reduced from thousands of failing test cases to 120 unique faults to be checked by refactoring engine developers. This classification was useful when reporting a number of faults in refactorings in Eclipse JDT, NetBeans and JRRT. Tool developers accepted a number of those faults.

5.7 Evaluation: overly strong conditions

The goal of this experiment is to analyze our technique for the purpose of evaluation with respect to effectiveness in identifying overly strong conditions from the point of view of refactoring engine developers in the context of academic and industrial Java refactoring engines. In particular, our experiment addresses the following research question. **Q1**: Can the

UFCG/BIBLIOTECA/BC

Table 5.7: Summary of evaluated refactoring implementations.

Refactoring	Eclipse	JRRT	Netbeans
Rename class	X	X	X
Rename method	X	X	X
Rename field	X	X	X
Push down method	X	X	X
Push down field	X	X	X
Pull up method	X	X	X
Pull up field	X	X	X
Encapsulate field	X	X	
Move method	X	X	
Add parameter	X	X	

technique identify overly strong conditions in real Java refactoring engines?

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

Selection of subjects

We tested 27 refactorings implementations for Java of three tools: Eclipse 3.7 (10 refactorings), JRRTv1[6] (10 refactorings), and NetBeans 7.0.1 (7 refactorings). Table 5.7 summarizes all evaluated refactorings.

In our experiment, we evaluate 10 refactoring types (Table 5.7). We tested only 7 refactoring types in NetBeans. The Move Method refactoring is not supported. As future work, we plan evaluate the Encapsulate Field and Add Parameter refactorings of NetBeans.

Experiment design and instrumentation

We used the SAFEREFACTOR command line version using the time limit of 1 second to generate tests, which is enough for testing the small programs generated by JDOLLY. We also used the JDOLLY command line version. For each generated input by JDolly, we compare the outputs of these three tools.

[6]The JRRT version from May 18th, 2010

Operation

We performed the experiment on a 2,7 GHz dual-core PC with 4 GB of RAM running Ubuntu 10.04. We evaluated Eclipse 3.4.2, NetBeans 6.9.1 and JRRT 1.0.

Our technique evaluated 27 refactoring implementations of Eclipse, NetBeans, and JRRT. Based on the scope and the additional constraints used for each refactoring, JDOLLY generated 42,774 programs[7]. Eclipse and JRRT did not apply a number of transformations, from which 32% and 16% were behavior-preserving, respectively. They reject them due to overly strong conditions. We automatically classified these transformations in categories. As a result, we identified 17 and 7 kinds of overly strong conditions in Eclipse and JRRT, respectively. We did not find overly strong conditions in the refactorings implemented by NetBeans.

Table 3 summarizes the experiment results. For each refactoring, we show the results of each implementation (Eclipse, NetBeans, and JRRT). The number of programs generated by JDOLLY is shown in Column *Program*. Column *Rejected Transformation* shows the number of transformations that were rejected by each implementation for not satisfying refactoring conditions. The number of behavior-preserving transformations that were rejected due to an overly strong condition of the implementation is shown in Column *Rejected B. Pres. Transformation*. Finally, Column *Overly Strong Condition* shows the number of overly strong conditions that were categorized by our technique.

Most transformations can be applied in NetBeans. It did not reject transformations except for the Rename Class refactoring. All transformations rejected by it were also rejected by Eclipse and JRRT. Therefore, we did not find problems related to overly strong conditions in NetBeans. However, it performed a number of non-behavior-preserving transformations that were rejected by Eclipse and JRRT. NetBeans contains a number of faults (missing conditions), as we shown in Section 5.6. The focus of this experiment is not on identifying missing conditions but in identifying overly strong conditions. Since NetBeans does not contain some conditions, it allows not only non-behavior-transformations, but also a number of behavior-preserving transformations that cannot be applied by other tools. Since the oracle of our technique is based on differential testing (Section 5.4), performing almost all transformations using NetBeans was useful for identifying whether transformations rejected

[7] All experiment data are available at: http://dsc.ufcg.edu.br/~spg/papers.html

Table 5.8: Summary of the experiment; Program = number of programs generated by JDolly; Rejected Transformation = number of transformations rejected by the implementation; Rejected B. Pres. Transformation = number of behavior-preserving transformations that were rejected; Overly strong condition = number of overly strong conditions classified by our technique.

Refactoring	Program	Rejected Transformation			Rejected B. Pres. Transformation			Overly Strong Condition		
		Eclipse	Netbeans	JRRT	Eclipse	Netbeans	JRRT	Eclipse	Netbeans	JRRT
Rename class	2037	1658	1212	1212	446	0	0	2	0	0
Rename method	6830	5995	0	1666	4802	0	419	4	0	1
Rename field	2647	324	0	0	200	0	0	2	0	0
Push down method	3822	2056	0	2065	59	0	40	1	0	1
Push down field	3043	1551	0	1551	0	0	0	0	0	0
Pull up method	5201	2907	0	3065	251	0	398	2	0	2
Pull up field	4151	976	0	912	744	0	584	1	0	1
Encapsulate field	3754	472	-	2736	176	-	1536	1	-	1
Move method	6316	5083	-	4757	367	-	135	2	-	1
Add parameter	4973	737	-	1189	79	-	0	2	-	0
Total	42774	21759	1212	19153	7124	0	3112	17	0	7

by Eclipse and JRRT could, in fact, be applied.

Eclipse was the tool that rejected more transformations. It rejected 21,759 transformations, from which 32% are behavior-preserving. We found overly strong conditions in all Eclipse's implementation but the Push Down Field refactoring. For instance, its Rename Method refactoring implementation rejected 5,995 out of 6,830 transformations but 4,802 of them could be applied without changing programs' behavior.

Renaming a method in the presence of features such as overloading and overriding may lead to behavioral changes in some situations due to changes in name bindings [71]. Eclipse developers may have implemented overly strong conditions for simplicity in order to avoid non-behavior-preserving transformations. However, this overly strong condition also rejected a number of useful behavior-preserving transformations since overloading and overriding are commonly used by Java developers.

JRRT rejected 19,153 transformations. In 16% of them, the program's behavior could be preserved. We found overly strong conditions in 6 out of 10 refactorings evaluated: Rename Method, Push Down Method, Pull Up Method, Pull Up Field, Encapsulate Field, and Move Method.

Discussion

Manually analyzing and classifying overly strong conditions in thousands of rejected transformations is time-consuming and error-prone. To avoid that, our technique automatically classifies them according to the template of the message shown by the implementation when a transformation is rejected. We analyze all warning messages in transformations that are behavior-preserving in at least another refactoring implementation. Our technique categorized 17 kinds of overly strong conditions in Eclipse, and 7 ones in JRRT. Table 5.9 shows the overly strong conditions identified in Eclipse and JRRT, respectively. Each line in the table contains a warning message template. The brackets abstract the names of packages, classes, methods, and fields, as described in Section 5.5.1.

We manually checked the overly strong conditions we found by randomly selecting a sample of 10 transformations for each kind of overly strong conditions, and we did not find false positives (a transformation that does not represent an overly strong condtion) or false negatives (the same template of warning message representing different overly strong conditions).

In five refactorings, we found less overly strong conditions in JRRT than Eclipse: Rename Class, Rename Method, Rename Field, Move Method, and Add parameter. Moreover, in four refactorings (Push Down Method, Pull Up Method, Pull Up Field, and Encapsulate Field), we found the same number of overly strong conditions in both tools. Finally, only in the Push Down Field refactoring, we did not find overly strong conditions.

Our technique identified 8 kinds of overly strong conditions in Rename Class, Method, and Field implementations of Eclipse, and only one in JRRT implementations. JRRT checks whether name bindings are preserved. Each name should refer to the same entity before and after the transformation [71]. Moreover, JRRT implementations may also check whether it is possible to re-qualify a name in order to preserve the name binding. This approach alleviates the problem of overly strong conditions. Listing 5.3 shows an example in which JRRT re-qualifies a name adding a `super` access to avoid name binding changes. Eclipse follows a different approach.

The overly strong condition found in the Rename Method refactoring of JRRT is related to overriding. This implementation has the invariant that overriding must be preserved. We also detected a condition in Eclipse related to that but NetBeans successfully applied a num-

Table 5.9: Summary of overly strong conditions of Eclipse 3.7 and JRRTv1.

Overly strong conditions of Eclipse	
Rename Class	**Rename Field**
Name conflict with type [] in []	Problem in [] The reference to [] will be shadowed by a renamed declaration
Another type named [] is referenced in []	Problem in [] Another name will shadow access to the renamed element
Rename Method	**Push Down Method**
[] or a type in its hierarchy defines a method [] with the same number of parameters, but different parameter type names.	The visibility of method [] will be changed to public
Problem in [] the reference to [] will be shadowed by a renamed declaration	**Pull Up Method**
Code modification may not be accurate as affected resource [] has compile errors	The visibility of method [] will be changed to public.
[] or a type in its hierarchy defines a method [] with the same number of parameters and the same parameter type names.	Method [] referenced in one of the moved elements is not accessible from type []
Pull Up Field	**Encapsulate Field**
Field [] declared in type [] has a different type than its moved counterpart	New method [] overrides an existing method in type []
Move Method	**Add Parameter**
The method invocations to [] cannot be updated, since the original method is used polymorphically.	The method [] from the type [] is not visible
The visibility of method [] will be changed to public.	The selected method overrides method [] declared in type []
Overly strong conditions of JRRT	
Rename Method	Push Down Method
overriding has changed	cannot access method
Pull Up Method	**Pull Up Field**
method is used	cannot access variable
cannot access method	**Move Method**
Encapsulate Field	cannot inline ambiguous method call
cannot insert method here	

ber of transformations that change overriding yet preserving program's behavior. In other refactorings, such as Move Method and Add Parameter, JRRT does not check conditions related to overriding.

Furthermore, we identified overly strong conditions related to accessibility. Both, Eclipse and JRRT, rejected transformations in the Push Down Method and Pull Up Method refactorings due to inaccessible methods. However, these transformations were performed by NetBeans. Changing access modifiers is not simple. It may change the name binding leading to behavioral changes [84]. Making these changes in ad-hoc way may be error-prone. Steimann and Thies [84] propose a number of conditions for applying refactorings with respect to Java accessibility. They show that these conditions are less strong than the ones implemented in Eclipse. While Eclipse implements some heuristics for that, Schäfer and de Moor [68] intend to integrate these conditions to JRRT.

Eclipse and NetBeans contain test suites for evaluating their refactoring implementations. For instance, the Eclipse test suite contains more than 2,600 unit tests. JRRT has a different test suite. Schäfer and de Moor [68] also evaluated JRRT over more than 1,000 unit tests of Eclipse's test suite. They used them not only for evaluating correctness, but also for identifying overly strong conditions [68]. Schäfer and de Moor checked whether all rejected transformations of Eclipse could be applied by JRRT. They identified some overly strong conditions in Eclipse. However, they also identified overly strong conditions in JRRT in the Add Parameter, the Move Method, and the Push Down refactorings. The overly strong conditions were related to visibility adjustment. However, they do not propose an approach to evaluate whether refactoring implementations have overly strong conditions. We can do it by using JDOLLY and SAFEREFACTOR.

In our evaluation, JDOLLY generated small programs (up to 15 LOC) with up to two packages, three classes, four methods, and two fields. These programs contain some common features of Java such as inheritance, overloading, and overriding. Although simple, they were useful for identifying 24 kinds of overly strong conditions in Eclipse and JRRT. The test suite of Eclipse and JRRT also contain small programs. However, the programs have some Java constructs such as interface, abstract classes and generics, that are currently not supported by JDolly. By improving the expressivity of JDolly, our technique can be useful for identifying other overly strong conditions.

5.7.1 Threats to Validity

Next we identify some threats to validity from the evaluation performed.

Construct Validity

Construct validity refers to whether the overly strong conditions that we have detected are indeed overly strong conditions in the refactoring engines. JRRT developers confirmed the overly strong conditions that we found. We have not received feedback from Eclipse developers yet. Some conditions that we found may not be overly strong with respect to the notion adopted by the developers. For instance, we found the "overriding has changed" overly strong condition in the rename method from JRRT. Its developers follow a closed world assumption. If they followed an open world assumption, this condition could not be considered overly strong since changing overriding could produce a behavioral change in some client code.

Additionally, different refactoring engines may use different refactoring templates. Therefore, comparing their outputs may not reveal overly strong conditions, just different notions of a refactoring.

Internal Validity

Concerning JDOLLY generation with Alloy, additional constraints may hide possibly detectable overly strong conditions. These constraints can be too restrictive with respect to the programs that can be generated by JDOLLY, which shows that one must be cautious when creating constraints for JDOLLY.

External Validity

We believe that other refactoring engines can be tested as well with our technique. Regarding some refactoring transformations other than the ones evaluated in this experiment, we have showed that our technique is applicable to any transformation, because it does not rely on specific properties of the transformation. In order to generate programs that exercise a specific refactoring, we may have to change the Alloy specification in JDOLLY. Additionally, it is needed to be at least two implementations of a same refactoring.

5.8 Concluding remarks

In this chapter, we presented our technique for automated testing of refactoring engines. Its goal is to identify missing conditions and overly strong ones. The technique has two main components: JDOLLY and SAFEREFACTOR.

We report on the results of an experiment to show the effectiveness of our technique. By using the technique, we tested up to 10 refactoring implementations from 3 refactoring engines: Eclipse JDT, NetBeans, and JRRT. As a result, we found 120 missing conditions and 24 overly strong ones. We reported them to the tools' developers, who have already fixed a number of them.

JRRT presented fewer faults than Eclipse and NetBeans, which suggests that the formal techniques used can improve the correctness of refactoring engines. Even so, our technique was useful for finding faults not only in its first version (JRRTv1) but also in its second version (JRRTv2) when they had fixed the faults of JRRTv1. We believe that our technique can be used to systematically evaluate these tools during their life cycle.

Chapter 6

Related Work

In this chapter, we relate our work to a number of approaches for verifying and testing refactorings (Section 6.1), approaches for automated testing (Section 6.2), and some empirical studies on refactorings (Section 6.3).

6.1 Refactoring verification and testing

Conditions are a key concept of research studies on the correctness of refactorings. Opdyke [53] proposes a number of refactoring conditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these conditions. In fact, later, Tokuda and Batory [87] showed that Opdyke's conditions were not sufficient to ensure preservation of behavior. Proving refactorings with respect to a formal semantics is a challenge [70]. Some approaches have been contributing in this direction. Borba et al. [8] propose 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. [75] propose a set of behavior-preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each one 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. Our testing approach still applies formal verification techniques (first-order logic and Alloy Analyzer) that are combined for a practical and less costly solution for increasing confidence when refactoring Java programs.

Steimann and Thies [84] show 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 conditions and computing the changes of access modifiers needed to preserve the program behavior. We have also detected new faults related to the Java access modifiers. Both approaches can be complementary for checking refactorings that affect accessibility constraints.

Another specialized approach for specifying refactorings – generalization-related refactorings such as Extract Interface and Pull Up Method – is proposed by Tip et al. [86]. Their work proposes an approach that uses type constraints to verify conditions of those refactorings, determining which part of the code they may modify. Using type constraints, they also propose the refactoring Infer Generic Type Arguments [21], which adapts a program to use the Generics feature of Java 5, and a refactoring to migration of legacy library classes [3]. These refactorings are implemented in the Eclipse JDT. Their technique allows sound refactorings with respect to type constraints. However, a refactoring may have conditions related to other constructs. Our general-purpose testing approach evaluates a refactoring independently of program structures being affected by the refactoring. The faults detected by our approach are related to missing conditions and overly strong ones.

Mens et al. [44] use graph rewriting for formalizing program refactorings. Two refactorings are specified for a subset of Java, and the authors propose a static semantics for Java, which is preserved by the two refactoring specifications. Graph-based verification is more ambitious than testing, aiming at full structural analysis, although presenting limited scalability. They have recognized that some refactorings, such as Move Method, which may deal with nested structures, require complex graph manipulation. Such analysis becomes considerably costly, which limits its results, in comparison with a more lightweight testing approach.

Overbey and Johnson [55] propose a technique to check for behavior preservation. They implement it in a library containing conditions for the most common refactorings. Refactoring engines for different languages can use their library to check refactoring conditions. 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

UFCG/BIBLIOTECA/BC

engines for Fortran 95, PHP 5 and BC. They do not evaluate them in terms of correctness but in terms of expressivity and performance. Our approach can be useful for evaluating their approach in terms of correctness and overly strong conditions.

Reichenbach et al. [63] propose the program metamorphosis approach for program refactoring. It breaks a coarse-grained transformation into small transformations. Although these small transformations may not preserve behavior individually, they guarantee that the coarse-grained transformation preserves behavior. Our approach can be used to increase confidence that the set of small transformations, applied in sequence, indeed preserve behavior.

Daniel et al. [14] propose an approach for automated testing refactoring engines. They used ASTGen to generate programs as input to refactoring engines. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. For instance, one of them checks for compilation errors, while another applies the inverse refactoring to the target program, and compares the result with the source program. If they were syntactically different, the refactoring engine developer would manually check whether they have the same behavior. They evaluated the technique by testing 21 refactorings, and identified 21 faults in Eclipse JDT and 24 in NetBeans. In Eclipse JDT, 17 faults were related to compilation errors, 3 were related to incomplete transformations (e.g. the Encapsulate field did not encapsulate all field accesses), and one was related to behavioral change. Later, Gligoric et at. [22] used the same approach to evaluate UDITA. They found 4 new compilation error faults in 6 refactorings (2 in Eclipse JDT and 2 in NetBeans). While the oracles of previous approaches can only syntactically compare the programs to detect behavioral changes, SAFEREFACTOR generates tests that do compare program behavior. We found 63 faults related to behavioral changes. Moreover, both techniques found a similar number of faults related to compilation errors.

Li and Thompson [41] propose an approach to test refactorings for Erlang using a tool called Quvid QuickCheck. They evaluate a number of implementations of the Wrangler refactoring engine. For each refactoring, they state a number of properties that it must satisfy, which is still a challenge. If a refactoring applies a transformation but does not satisfy a property, they indicate a fault in the implementation. We evaluate behavior preservation by using SAFEREFACTOR. We propose a similar approach for testing refactorings for Java. Their approach applies refactorings to a number of real case studies and toy examples. In

contrast, we apply refactorings to a number of programs generated by JDOLLY.

6.2 Automated Testing

Grammar-Based Test Generation (GBTB) is a well-known technique for automatically generating programs based on a formal grammar definition [9]. Using this technique, a generator is capable of building valid (or intentionally invalid) sentences of the target language. GBTB has been successfully used, for instance, to generate programs for testing the correctness and error messages in compilers [9; 4]. JDOLLY, by comparison, uses Alloy to specify the Java metamodel using signatures and relations. By performing analysis using the Alloy Analyzer, each Alloy solution is translated into a Java program. Moreover, we can guide JDOLLY to generate programs with properties that are specific to a given target domain (Section 3.4). In contrast, context-free grammars are somewhat limited for this purpose, being usually extended by operational definitions or even by code snippets for adapting generation to the desired class of test cases.

Recently, GBTB has been mixed with other advanced combinatorial techniques for generating programs of a language grammar. An approach that is very related to JDOLLY's generation technique has been described by Hoffman et al. [31]. It uses grammars instrumented by tags and code snippets written in Python that further constrain the generated test cases. In the referred tool, YouGen, tags inject parameters to the generation. For instance, parameters adjust the depth of a generation tree, limiting the derivation over recursive production rules. This feature is analogous to JDOLLY's scope. Also, while JDOLLY makes use of Alloy Analyzer's exhaustive search to generate a comprehensive set of programs, YouGen uses combinatorial techniques, such as mixed-strength covering arrays. In both cases, they evaluate all possible combinations. Their application contexts are in essence different, however: YouGen has been used for testing XML-based tools and network protocols, whereas JDOLLY is tailored for testing refactoring engines, using SAFEREFACTOR as a test oracle. Still, both tools can be adapted for diverse application cases.

Korel and Yami [40] propose an approach to automated regression test generation [26]. They use TESTGEN, a test data generation system for Pascal programs. Similarly, a component of our approach, SAFEREFACTOR, tests evaluate whether a transformation preserves

behavior. It uses the Randoop test generator. They test the parts of the programs whose functionality is unchanged after modifications. SAFEREFACTOR automatically detects the methods with unchanged signatures and generates tests for them. We are concerned with testing refactorings in this article.

Concerning automated regression testing, a more recent contribution is provided by BERT [36], a tool that focuses on detection of state changes from one version of a given class to its next version, considering transformations of any category (not only refactoring). The main distinction between the two approaches is their test oracle. SAFEREFACTOR uses a simple oracle that compares outputs of methods with unchanged signatures for the same input. If any changed behavior is, directly or indirectly, exercised by one of these methods, there is a high probability that the test goes wrong, and a behavior change is detected. BERT, on the other hand, does not consider changes in method signatures. It can be used only when all signatures are preserved. They focus on identifying differences in several structural aspects of the target program: return values of all methods, field values, and even output (textual) results. If a change is detected, there is an indication of a regression fault, although this may be not the case (false positives). Since they evaluate any kind of transformation, developers have to analyze whether the behavioral changes have been intentionally introduced.

Marinov and Khurshid [42; 12] propose TestEra, a framework for automated specification-based testing of Java programs. It uses Alloy to specify the pre and post-conditions of a method under test. Using this specification, it automatically generates the test inputs and checks post-conditions. This approach is similar to JDOLLY for generating test inputs, but we generate programs as test inputs.

6.3 Empirical studies on refactoring

A number of studies have investigated refactoring tasks in software projects. Ratzinger et al. [61] analyzed the relationship between refactoring and software defects. They proposed an approach to automatically identify refactorings based on commit messages, which we describe in Section 4.2.1. Using evolution algorithms, they confirmed the hypothesis that the number of software defects in the period decreases if the number of refactorings increases as

overall change type. To evaluate the effectiveness of the commit message analysis, they randomly sampled 500 versions from 5 projects, and analyzed whether their analysis correctly classify each version. In their experiment, the commit message analysis had only 4 false positives 10 false negatives in 500 software versions, leading to a high precision and recall.

Murphy-Hill et al. [50; 49] evaluated nine hypotheses about refactoring activities. They used data automatically retrieved from users through Mylyn Monitor and Eclipse Usage Collector. That data allowed Murphy-Hill et al. to identify the frequency of each automated refactoring. The most frequently applied refactorings are: Rename, Extract local variable, Move, Extract method, and Change method signature. They confirmed assumptions such as the fact that refactorings are frequent. Data gathered from Mylyn showed that 41% of the programming sessions contained refactorings.

Additionally, they evaluated the Ratzinger analysis. By using Ratzinger algorithm, they classified the refactoring versions from Eclipse CVS repository. Then, they randomly selected 20 versions from each set of refactoring versions and non-refactoring versions identified by Ratzinger, and applied to these 40 versions the manual inspection proposed by them, which we describe in Section 4.2.1. From the 20 versions labeled as refactoring by Ratzinger, only 7 could be classified as refactoring versions. The others include non-refactoring changes. On the other hand, the 20 versions classified as non-refactoring by Ratzinger were correct. In this thesis, we compared the results of these two techniques (Ratzinger and Murphy-Hill) with SAFEREFACTOR's results (Section 4.2). The Murphy-Hill approach was the most accurate among the refactoring technique we evaluated. However, it incorrectly classified versions containing compilation errors as refactoring versions. Differently from the original work, our results show a low recall and precision of Ratzinger approach, which we discuss in Section 4.2.

Kim et al. [37] investigate the relationship of API-level refactorings and bug fixes in three open source projects. They use a tool [38] to infer systematic declaration changes as rules and determine method-level matches (a previous version of Ref-Finder [57] that identifies 11 refactorings). They found that the number of bug fixes increases after API-level refactorings while the time taken to fix them decreases after refactorings. Moreover, the study indicated that refactorings are performed more often before major releases than after the releases.

Prete et al. [57] propose Ref-Finder, a tool that can detect 63 refactoring types from

FowlerÕs catalog [57]. It can detect all refactorings of the previous works, and it can detect intra-method refactoring changes. A comprehensive comparison can be found in Prete et al. [57]. Rachatasumrit and Kim [59] analyze the relationship between the types and locations of refactorings identified by Ref-Finder and the affecting changes and affected tests identified by a change impact analyzer (FaultTracer). They evaluate their approach in three open source projects (Meter, XMLSecurity, and ANT) and found that refactoring changes are not very well tested. By selecting the test cases that only exercise the changes, we may decrease the regression test cost.

Kim et al. [39] interview a subset of engineers who led the Windows refactoring effort and analyzed Windows 7 version history data. They found that in practice developers may allow non-behavior-preserving program transformations during refactoring activities. Moreover, developers indicate that refactoring involves substantial cost and risks. By analyzing Windows 7 version history, the study indicated that refactored modules experienced higher reduction in the number of inter-module dependencies and post-release defects than other changed modules.

Görg and Weißgerber [25] proposed a technique to identify and rank refactoring candidates using names, signatures, and clone detection results. Later, Weißgerber and Diehl [90] evolved and evaluated this tool. Weißgerber and Diehl [89] analyzed the version histories of JEdit, JUnit, and ArgoUML and reconstructed the refactorings performed using the tool proposed before [25]. They also obtained bug reports from various sources. They related the percentage of refactorings per day to the ratio of bugs opened within the next five days. They found that the high ratio of refactoring is sometimes followed by an increasing ratio of bug reports.

6.4 Concluding remarks

In this section, we presented the works most related to this thesis. With respect to approaches based on formal methods, we propose a more practical approach that was useful for finding a number of faults in real refactoring engines. The main novelties of our technique with respect to other testing approach for refactoring engines are: (1) generating input programs by using Alloy; (2) detecting behavioral changes with SAFEREFACTOR, (4) identifying overly strong

conditions; and (3) classifying behavioral changes and overly strong conditions.

Chapter 7

Conclusions

In this work, we propose a technique for testing of Java refactoring engines. Its main components are JDOLLY (Chapter 3), a Java program generator, and a test system for refactorings, SAFEREFACTOR (Chapter 4). For each refactoring, the technique generates a number of Java programs, followed by the application of the refactoring, with these programs as target. It uses behavioral oracles to evaluate the outputs. If the engine produces an output program, it uses SAFEREFACTOR for detecting behavioral changes between the input and the output programs. On the other hand, if the engine rejects the transformation, it applies the same refactoring by using other engines and compares the results of the executions. Finally, the technique classifies failures into distinct faults. The failing transformations are classified by kind of behavioral change or compilation error introduced by them, and rejected behavior-preserving transformations are classified by kind of overly strong conditions. We propose a Java program generator (JDOLLY) to run the program generation step of our technique. It uses Alloy [32] and the Alloy Analyzer [33] to create programs for a given scope of elements (packages, classes, fields, and methods). We have evaluated our technique by testing three refactoring engines: Eclipse JDT 3.7, NetBeans 7.0.1, and two versions of JRRT (JRRTv1 and JRRTv2). For each refactoring engine, we tested up to 10 refactoring implementations, and found 57 and 63 faults related to compilation errors and behavioral changes, respectively, and 24 overly strong conditions.

Specifying the set of conditions needed for each refactorings is not simple. Even refactoring engines written with correctness in mind, such as JRRT, still have faults and overly strong conditions. We have shown some corner cases automatically detected by our technique. With

these results, we have demonstrated how the combination of JDOLLY and SAFEREFACTOR is powerful to detect missing conditions and overly strong ones. In the absence of formal proofs, our technique can be useful for the improvement of previous solutions. We have reported all faults to Eclipse JDT, NetBeans and JRRT, and a number of them have already been accepted. Moreover, Eclipse JDT and NetBeans developers have fixed some of them, and JRRT developers have already fixed all accepted faults. They have also included 21 test cases that we generated in their test suite[1].

We show that our technique is general enough to test different kinds of refactorings from different Java refactoring engines. We tested up to 10 refactoring implementations from Eclipse JDT, NetBeans, and JRRT. These refactoring implementations target declarations of classes, fields, and methods. Other refactorings that target these constructs, such as Remove Parameter or Change Access Modifier, can be tested by using the current implementation of the technique. On the other hand, we cannot test refactorings that target Java constructs not specified in the current implementation of JDOLLY. For instance, we cannot test the Rename Local Variable and the Extract Method refactorings because the method bodies generated by JDOLLY contain only a return statement. The metamodel implemented in JDOLLY also restricted the input programs that were generated to evaluate the tested refactoring implementations. For instance, we could not test the Rename Field refactoring in the presence of local variables since we did not generate programs containing local variable declarations.

To reduce these limitations, one can extend JDOLLY increasing the expressivity of the programs generated by it. It will be necessary to specify new Java constructs and well-formedness rules in Alloy. So far, it was possible to specify the current implementation of the Java metamodel with reasonable effort. However, we cannot generalize these results. Some Java constructs and well-formedness rules may be difficult to specify in Alloy due to some restrictions of the language. For instance, Alloy does not allow recursive functions directly. Therefore, we believe that we need further studies to evaluate the effort to extend JDOLLY. Even so, a version of JDOLLY for C/CC++ (CDolly) has been used for finding faults in refactoring engines for the C/C++ language[2]. This work gives evidences that our

[1]SVN path for our test cases included in JRRT test suite: `http://jrrt.googlecode.com/svn/trunk/tests/BrazilianTests.java`

[2]CAutomaticTester website: : `http://www.dsc.ufcg.edu.br/~spg/cautomatictester/`

approach is also useful for testing refactorings that target method statements. It also show that the approach can be applied not only for Java but also for other languages, such as C.

Additionally, the more signatures and relations are added to the Alloy specification, the more combinations can be generated by Alloy Analyzer, increasing the state space of solutions. As a result, the time required for using our approach can increase from hours to days. To make it more efficient, one can use optimization techniques that prune the program generation. For example, Jagannath et al. [34] suggest that we can make small jumps in the sequence of automated generated programs without losing effectiveness of the test suite.

With respect to SAFEREFACTOR, we evaluated its effectiveness in 60 transformations applied to real software. In our experiments, SAFEREFACTOR had 70% accuracy. It produced false positives when testing GUI code and false negative when testing non-deterministic code. Additionally, in some transformations that affected only few methods, the time limit used for generating tests was not enough to generate tests for these methods because SAFEREFACTOR identified a large set of common methods to test. To handle this limitation, Mongiovi et al. [46] extend SAFEREFACTOR including an impact analysis technique, which identifies the methods impacted by the change. By doing so, SAFEREFACTOR generates tests not for all common methods but just for the ones impacted by the change. These limitations of SAFEREFACTOR did not affect the use of it in our technique for testing of refactoring engines since we use it against simple transformations that are deterministic and do not have GUI code.

Finally, even generating just small programs, containing few classes, methods, and fields, our technique identified more than 100 faults in refactoring engines. These results corroborate with the small scope hypothesis [32], which believes that, in practice, any failure is likely to manifest itself on some small input, and thus testing all small inputs is enough to reveal any failure.

7.1 Future work

We plan to evaluate our technique on other refactorings, such as Extract Method. To do so, we need to extend JDOLLY to generate programs containing richer method bodies. For instance, we could change the relation of the `Method` signature. Now, `b` must contain a

UFCG/BIBLIOTECA/BC

sequence of statements.

```
1 sig Method {
2   b: seq[Body]
3 }
```

Moreover, we can extend `Body` to represent other kinds of statements. For example, we can create the following signature representing method invocation. Notice that we need a new kind of `Id` to represent the variable name that invokes a method id.

```
1 sig InstanceMethodInvocation extends Body {
2   id: one VarId,
3   method: one MethodId
4 }
```

In this way, JDOLLY can generate more elaborated method bodies, such as the one presented next.

```
1 public void m() {
2   A a = new A();
3   a.z(2);
4   a.y();
5 }
```

Currently, we manually classify the failures related to behavioral change into distinct faults. This process is done by checking each transformation that introduces behavioral change against a number of proposed filters (see Table 5.1). It may demand a considerable effort to perform this task when there are a lot of failures. We plan to automate this step by developing a static analysis to evaluate the non-behavior-preserving transformations against the proposed filters.

Additionally, it is time-consuming to test the refactoring implementation with respect to each test input generated. For instance, in a previous experiment (See Section 5.6), it was needed around 12 hours to test the Push Down Method implementation of JRRTv1[3] by using 15,322 input programs generated by a program generator called JDOLLY. From these 15,322 input programs, 2,247 were useful for producing test cases that expose faults. In

[3] The JRRT version from May 18th, 2010

such cases, by reordering the test cases, we may increase the rate of fault detection, reducing the time spent to find faults. In this way, developers can have an earlier feedback to start debugging and fixing the faults. Test case prioritization techniques [67; 34] schedule test cases in order to achieve some goal. Techniques for automated testing of refactoring engines iteratively test the refactoring implementation against each input program generated by a program generator. The order of the test cases is the order that the programs are generated.

Therefore, we can prioritize the test cases by changing the order that the input programs are used by the technique to run the test cases. We have observed that failures detected by using these programs can be classified into distinct faults based on characteristics of the input programs, such as overriding, overloading, field hiding. Therefore, having used an input program produced by the program generator and covered certain characteristics, we may be gained in subsequent input programs by covering characteristics that have not been covered yet. We thus can prioritize test cases based on the characteristic coverage of the input programs.

Besides missing conditions and overly strong ones, refactoring engines may also have faults related to incorrect or incomplete transformations. Daniel et al. [14] have implemented oracles to check whether an implementation of the Encapsulate Field refactoring encapsulates all accesses of a target field. They have found a fault in Eclipse related to that. Consider class `A` shown in Listing 7.1. If we apply the Encapsulate Field refactoring by using Eclipse 3.7, the tool will produce the output program shown in Listing 7.2. Notice that `setF(f)` should be `setF(getF())`.

We plan to investigate the use of structural oracles to check whether the performed transformation was incorrect or incomplete. In the same way Daniel et al. [14] implemented checks for the Encapsulate Field Refactoring, we can implement some checks for other refactorings. Another approach would be to use a tool such as Ref-Finder [57]. It performs static analysis on both input and output programs, in order to discover the application of complex refactorings. The tool identifies 63 refactorings presented by Fowler [19]. Each refactoring is represented by a set of logic predicates (a template), and the matching between program and template is accomplished by a logic programming engine. By using Ref-Finder against the transformations generated by our technique, we could identify transformations that do not match a specific refactoring template.

Listing 7.1: Before Refactoring.

```
1 class A {
2   int f;
3   void m(){
4     f=f;
5   }
6 }
```

Listing 7.2: After Refactoring. Encapsulated Field by Eclipse JDT 3.7 does not encapsulate the field read.

```
1  class A {
2    private int f;
3    void m(){
4      setF(f);
5    }
6    void setF(int f) {
7      this.f = f;
8    }
9    int getF() {
10     return f;
11   }
12 }
```

We also plan to perform a user study to compare JDOLLY against UDITA in the context of testing of refactoring engines. Our research question is: Is specifying program generation in JDOLLY easier than in UDITA? By performing a human study, we can measure the time to specify the program generation, the size, and its correctness. Additionally, we can further investigate both generators with respect to isomorphic programs and exhaustiveness.

Finally, SAFEREFACTOR produced false positives and negatives due to limitations of Randoop, its test generator. We plan to compare Randoop against other tests generators with respect their effectiveness. We also plan to investigate how to generate test cases in the context of concurrence, so that we can extend our approach for concurrent programs.

Bibliography

[1] ISO/IEC 14764:1999. Software engineering - software maintenance. ISO and IEC, 1999.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[3] Ittai Balaban, Frank Tip, and Robert 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, pages 265–279, New York, NY, USA, 2005. ACM.

[4] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, 8:343–353, July 1982.

[5] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools (ARP/AOD) 2 Vol. Set*. Addison-Wesley Professional, 1st edition, 2009.

[7] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 27–36. IEEE Computer Society, 2005.

[8] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, August 2004.

[9] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10(11):897–918, November 1980.

[10] Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *Proceedings of the 4th Aspect-Oriented Software Development*, AOSD '05, pages 123–134, New York, NY, USA, 2005. ACM.

[11] Leonardo Cole, Paulo Borba, and Alexandre Mota. Proving aspect-oriented programming laws. In *Proceedings of the 4th Foundations of Aspect-Oriented Languages*, FOAL '05, pages 1–10, 2005.

[12] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31:328–339, April 2005.

[13] Márcio Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Federal University of Pernambuco, 2004.

[14] Brett Daniel, Danny Dig, Kely Garcia, and Darko 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*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

[15] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Eclipse.org. Eclipse project. At http://www.eclipse.org, 2011.

[17] Eclipse.org. JDT core component. At http://www.eclipse.org/jdt/core/, 2011.

[18] Embarcadero Technologies. JBuilder. At http://www.codegear.com/br/products/jbuilder, 2011.

[19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[20] Robert Führer, Adam Kieżun, and Markus Keller. Refactoring in the eclipse JDT: Past, present, and future. In *Proceedings of the Workshop on Refactoring Tools*, 2007.

[21] Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP '05, pages 71–96, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[23] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[24] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Notes*, 10:493–510, April 1975.

[25] Carsten Görg and Peter Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 205–214, Washington, USA, 2005. IEEE Computer Society.

[26] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9:242–257, December 1970.

[27] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th Aspect-Oriented Software Development*, AOSD '05, pages 135–146, New York, NY, USA, 2005. ACM.

[28] J He, C A R Hoare, and J W Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[29] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[30] C. A. R. Hoare. Proof of correctness of data representations. pages 385–396, 2002.

[31] Daniel Malcolm Hoffman, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang. Grammar-based test generation with YouGen. *Software: Practice and Experience*, 41:427–447, April 2011.

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

[33] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 730–733, New York, NY, USA, 2000. ACM.

[34] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 171–185, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Jet Brains. IntelliJ Idea. At http://www.intellij.com/idea/, 2011.

[36] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *Proceedings of the 23rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 137–146, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[38] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the 20th Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. ACM.

[40] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 4th International Symposium on Software Testing and Analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[41] Huiqing Li and Simon Thompson. Testing Erlang Refactorings with QuickCheck. In *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2008.

[42] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 22–34, Washington, DC, USA, 2001. IEEE Computer Society.

[43] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[44] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proceedings of the 1st International Conference on Graph Transformation*, ICGT '02, pages 286–301, London, UK, 2002. Springer-Verlag.

[45] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, February 2004.

[46] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, (0):–, 2013.

[47] Miguel Monteiro and Joao Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th Aspect-Oriented Software Development*, AOSD '05, pages 111–122, New York, NY, USA, 2005. ACM.

[48] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23:76–83, July 2006.

[49] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, January-February 2012.

[50] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–296, Washington, DC, USA, 2009. IEEE Computer Society.

[51] David A. Naumann, Augusto Sampaio, and Leila Silva. Refactoring and representation independence for class hierarchies. *Theoretical Computer Science*, 433(0):60 – 97, 2012.

[52] David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. Springer, 2008.

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

[54] William Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Object-Oriented Programming emphasizing Practical Applications*, pages 145–160, 1990.

[55] Jeffrey L. Overbey and Ralph E. 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, pages 303–312, New York, NY, USA, 2011. ACM.

[56] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference*

on Software Engineering, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[57] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[58] Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for prolog. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '91, pages 274–284, New York, NY, USA, 1991. ACM.

[59] Napol Rachatasumrit and Miryung 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, Washington, USA, 2012. IEEE Computer Society.

[60] Jacek Ratzinger. *sPACE: Software Project Assessment in the Course of Evolution*. PhD thesis, Vienna University of Technology, 2007.

[61] Jacek Ratzinger, Thomas Sigmund, and Harald Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 5th Mining Software Repositories*, MSR '08, pages 35–38, 2008.

[62] Refactoring.com. Alpha list of refactorings. At http://refactoring.com/catalog/index.html, 2010.

[63] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program metamorphosis. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP '09, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia 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, pages 432–448, New York, NY, USA, 2004. ACM.

[65] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[66] Brian Robinson, Michael Ernst, Jeff Perkins, Vinay Augustine, and Nuo 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, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

[67] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–, Washington, DC, USA, 1999. IEEE Computer Society.

[68] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In *Proceedings of the 25th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM.

[69] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent Java code. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, pages 225–249, Berlin, Heidelberg, 2010. Springer-Verlag.

[70] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 67–72, New York, NY, USA, 2008. ACM.

[71] Max Schäfer, Torbjörn Ekman, and Oege 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, pages 277–294, New York, NY, USA, 2008. ACM.

[72] Max Schäfer, Torbjörn Ekman, and Oege 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, pages 277–294, New York, NY, USA, 2008. ACM.

[73] Max Schäfer, Torbjorn Ekman, Ran Ettinger, and Mathieu Verbaere. Refactoring bugs. At http://code.google.com/p/jrrt/wiki/RefactoringBugs, 2011.

[74] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. Stepping stones over the refactoring rubicon. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP '09, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[75] Leila Silva, Augusto Sampaio, and Zhiming Liu. Laws of object-orientation with reference semantics. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '08, pages 217–226, Washington, DC, USA, 2008. IEEE Computer Society.

[76] Gustavo Soares. Automated behavioral testing of refactoring engines. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 105–106, New York, NY, USA, 2012. ACM.

[77] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2012.

[78] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 2012. . To appear.

[79] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, July 2010.

[80] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, 2010.

[81] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 173–182, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Gustavo Araujo Soares. Uma abordagem para aumentar a segurança em refatoramentos de programas, 2010. Dissertação de Mestrado da Universidade Federal de Campina Grande.

[83] Gustavo Soares Soares. Automated behavioral testing of refactoring engines. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 49–52, New York, NY, USA, 2012. ACM.

[84] Friedrich Steimann and Andreas 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, pages 419–443, Berlin, Germany, 2009. Springer-Verlag.

[85] Sun Microsystems. NetBeans IDE. At http://www.netbeans.org/, 2011.

[86] Frank Tip, Adam Kieżun, and Dirk 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, pages 13–26, New York, NY, USA, 2003. ACM.

[87] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8:89–120, January 2001.

[88] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.

[89] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 3rd Mining Software Repositories*, MSR '06, pages 112–118, New York, NY, USA, 2006. ACM.

[90] Peter Weißgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[91] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, November 1982.

[92] Jan Wloka, Robert Hirschfeld, and Joachim Hänsel. Tool-supported refactoring of aspect-oriented programs. In *Proceedings of the 7th Aspect-Oriented Software Development*, AOSD '08, pages 132–143, New York, NY, USA, 2008. ACM.

[93] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29:366–427, December 1997.

Appendix A

Java metamodel specification in Alloy

Next, we present the complete specification of a subset of the Java language in Alloy, which was used by JDOLLY for generating Java programs.

Listing A.1: A subset of the Java language specified in Alloy

```
1  // ABSTRACT SYNTAX
2
3  abstract sig Id { }
4
5  sig Package{ }
6
7  sig ClassId, MethodId,FieldId extends Id { }
8
9  abstract sig Accessibility { }
10
11 one sig public, private_, protected extends Accessibility { }
12
13 abstract sig Type { }
14
15 abstract sig PrimitiveType extends Type { }
16
17 one sig Int_, Long_ extends PrimitiveType { }
18
19 sig Class extends Type {
```

```
20    package: one Package,
21    id: one ClassId,
22    extend: lone Class,
23    methods: set Method,
24    fields: set Field
25  }
26
27  fun classes[pack:Package]: set Class {
28    pack.~package
29  }
30
31  sig Field {
32      id : one FieldId,
33      type: one Type,
34      acc : lone Accessibility
35  }
36
37  sig Method {
38    id : one MethodId,
39      param: lone Type,
40      acc: lone Accessibility,
41      return: one Type,
42      b: one Body
43  }
44
45  abstract sig Body { }
46
47  sig LiteralValue extends Body { } // returns a literal value
48
49  abstract sig Qualifier { }
50
51  one sig qthis_, this_, super_ extends Qualifier { }
52
```

```
53
54 sig MethodInvocation extends Body {
55     id : one MethodId,
56     q: lone Qualifier
57 }
58 fact {
59 // call a declared method
60     all mi:MethodInvocation | some m:Method | mi·id = m·id
61 // avoid recursive calls
62     all m:Method | all mb: MethodInvocation | m·b = mb ⇒ mb·id ≠ m·id
63 }
64
65 // return new A()·k();
66 sig ConstructorMethodInvocation extends Body {
67     idClass : one ClassId,
68     idMethod: one MethodId
69 }
70 fact {
71 // calls a method declared in the class
72     all ci: ConstructorMethodInvocation |
73         some c:Class |
74             ci·idClass = c·id &&
75             (some m:Method | m in c·methods && m·id = ci·idMethod)
76
77 // avoid recursive calls
78     all m:Method | all mb: ConstructorMethodInvocation | m·b = mb ⇒ mb·idMethod ≠ m·id
79 }
80
81 fun classFromClassId[id1:ClassId]: set Class {
82   id1·~id
83 }
84
85 fun fieldFromFieldId[id1:FieldId]: set Field {
```

```
86    id1·~id
87  }
88
89  // return x;
90  // return this·x;
91  // return super·x;
92  // return A·this·x;
93  sig FieldInvocation extends Body {
94      idField : one FieldId,
95      qField: lone Qualifier
96  }
97
98  // return new A()·x;
99  sig ConstructorFieldInvocation extends Body {
100     idClass2 : one ClassId,
101     idField: one FieldId
102 }
103 fact {
104     //call field declared in the class
105     all ci: ConstructorFieldInvocation |
106         some c:Class |
107             ci·idClass2 = c·id &&
108            (some f:Field | f in c·fields && f·id = ci·idField)
109 }
110
111
112
113 // WELL−FORMED RULES
114 fact JavaWellFormedRules {
115   noPackageContainsTwoClassesWithSameId[]
116    noCalltoUndefinedField[]
117     noSuperCallToNotInheritedField[]
118
```

```
119      noClassExtendsItself[]
120      allFieldsBelongToAClass[]
121    noClassContainsTwoFieldsWithSameId []
122    noClassContainsTwoMethodsWithSameSignature[]
123    noClassExtendsAnotherWithSameId[]
124    allBodiesBelongToAMethod[]
125      allMethodsBelongToAClass[]
126      noSuperCallToNotInheritedMethod[]
127      noCalltoUndefinedMethod[]
128  }
129  pred noPackageContainsTwoClassesWithSameId[] {
130    all package: Package | all c1,c2:classes[package] | c1≠ c2 ⇒ c1·id ≠ c2·id
131  }
132
133  pred noClassExtendsItself[] {
134    no c:Class | c in c·^extend
135  }
136
137  pred noClassExtendsAnotherWithSameId[] {
138    all c1:Class | no c2: c1·^extend | c1·id = c2·id
139  }
140
141  pred noClassContainsTwoFieldsWithSameId [] {
142      no c:Class | some disj f1,f2:Field | f1·id = f2·id && f1 + f2 in c·fields
143  }
144
145  pred noCalltoUndefinedMethod[] {
146    all mi:MethodInvocation |
147        (#mi·q = 0 || mi·q = this_) ⇒
148            some c1,c2: Class, m1:c1·methods, m2:c2·methods | mi in m1·b && mi·id = m2·id &&
                 ((c1 = c2) || ((c2 in c1·^extend) && (m2·acc ≠ private_)))
149
150    all mi:MethodInvocation |
```

```
151        (mi·q = qthis_) ⇒
152            some c1:Class, m1,m2:c1·methods | mi in m1·b && mi·id = m2·id
153    }
154
155    pred noSuperCallToNotInheritedMethod[] {
156      all mi:MethodInvocation |
157         mi·q = super_ ⇒
158            some c1,c2: Class, m1:c1·methods, m2:c2·methods | mi in m1·b && mi·id = m2·id &&
                   c2 in c1·^extend && (m2·acc ≠ private_)
159    }
160
161    pred noSuperCallToNotInheritedField[] {
162      all fi:FieldInvocation |
163        fi·qField = super_ ⇒
164          some disj c1,c2: Class, m1:c1·methods, f:c2·fields | fi in m1·b && fi·idField = f·id && c2 in
                 c1·^extend && f·acc ≠ private_
165    }
166
167    pred noCalltoUndefinedField[] {
168     all mi:FieldInvocation |
169        ( mi·qField = this_) ⇒
170            some c1,c2: Class, m1:c1·methods, f:c2·fields | mi in m1·b && mi·idField = f·id && ((
                   c1 = c2) || ((c2 in c1·^extend) && (f·acc ≠ private_)))
171
172      all mi:FieldInvocation |
173        ( mi·qField = qthis_) ⇒
174            some c1,c2: Class, m1:c1·methods, f:c2·fields | mi in m1·b && mi·idField = f·id && ((
                   c1 = c2) || ((c2 in c1·^extend) && (f·acc ≠ private_)))
175
176      all mi:FieldInvocation |
177        ( #mi·qField = 0) ⇒
178            some c1,c2: Class, m1:c1·methods, f:c2·fields | mi in m1·b && mi·idField = f·id && ((
                   c1 = c2) || ((c2 in c1·^extend) && (f·acc ≠ private_)))
```

```
179 }
180
181 pred allFieldsBelongToAClass [] {
182     all f:Field | one c:Class | f in c·fields
183 }
184
185 pred noClassContainsTwoMethodsWithSameSignature[] {
186   all c: Class | all m1,m2:c·methods | m1≠ m2 ⇒ (m1·id ≠ m2·id or m1·param ≠ m2·param)
187 }
188
189 pred allMethodsBelongToAClass [] {
190     all m:Method | one c:Class | m in c·methods
191 }
192
193 pred allBodiesBelongToAMethod [] {
194    Body in Method·b
195 }
```

Appendix B

Algorithms for checking refactoring scope and granularity

Next we formalize some algorithms used to collect data from repository. Algorithm 2 indicates when a transformation is low or high-level. If a transformation only changes inside a method, it is considered low-level. Otherwise it is considered high-level. *methods* yields the set of methods of a program. *signature* yields the method signature of all methods received as parameter.

Algorithm 2 Refactoring Granularity

Require: source $\Leftarrow$ program before transformation

Require: target $\Leftarrow$ program after transformation

Ensure: Indicates whether a transformation is low or high-level

  mSource $\Leftarrow$ methods(*source*)

  mTarget $\Leftarrow$ methods(*target*)

  if signature(*mSource*) = signature(*mTarget*) **then**

    return LOW

  else

    return HIGH

  end if

Algorithm 3 establishes when a transformation is local or global. If a transformation only changes at most one package, it is considered local. Otherwise it is considered global.

packages yields the set of packages of a program. *name* yields the name of a package. *diff* is the shell command used to compare to directories.

Algorithm 3 Refactoring Scope

Require: *source* $\Leftarrow$ program before transformation

Require: *target* $\Leftarrow$ program after transformation

Ensure: Indicates whether a transformation is local or global

```
count ⇐ 0
for p ∈ packages(source) do
  pTarget ⇐ package(name(p),target)
  if diff(p,pTarget) ≠ ∅ then
    count++
  end if
end for
for p ∈ packages(target) do
  pSource ⇐ package(name(p),source)
  if diff(p,pSource) ≠ ∅ then
    count++
  end if
end for
if count ≤ 1 then
  return LOCAL
else
  return GLOBAL
end if
```